List of Selected Problems along with links to Problem Statements:

No.	Problem ID	Title	Rating	Tags	Problem Link
1	2114F	Small Operations	2000	binary search, brute force, dfs and similar, dp, math, number theory, sortings	Link
2	2107F1	Cycling (Easy Version)	2300	binary search, brute force, dp, greedy	Link
3	2110F	Faculty	2400	brute force, greedy, math, number theory	Link
4	603A	Alternative Thinking	1600	dp, greedy, math	Link
5	2107C	Maximum Subarray Sum	1500	binary search, constructive algorithms, dp, implementation, math	Link
6	571A	Lengthening Sticks	2100	combinatorics, implementation, math	Link
7	2093C	Simple Repetition	1000	math, number theory	Link
8	2104C	Card Game	1100	brute force, constructive algorithms, games, greedy, math	Link
9	2094E	Boneca Ambalabu	1200	bitmasks	Link
10	2109B	Slice to Survive	1200	bitmasks, greedy	Link

Detailed Approach Explanation for above Problems:

Advanced Problem-Solving Techniques:

1. 2114F : Small Operations

- My solution approaches the problem by first breaking it into : "transforming 2 numbers (x and y) into their GCD through multiplication operations with factors $\leq$ k"
- First, it checks if x and y are equal, outputting 0 if true.
- For unequal numbers, it computes their GCD (g) and then determines the minimal steps required to reduce both x/g and y/g to 1, using valid factors.

- The calc function employs BFS on the divisors of the number, exploring valid transitions (multiplications) while ensuring that factors are $\leq$ k and the largest prime factor of the number is $\leq$ k.
- If either transformation is impossible, then it outputs -1; otherwise, it sums the steps for both transformations.

Techniques Utilized: *prime factorization, divisor enumeration, and BFS for shortest path calculation.*

2. 2107F1 : Cycling (Easy Version)

- My approach is to use DP to efficiently compute the minimum cost of processing elements in reverse order.
- The key observation is that for each position i, we need to find the next smallest element (p) in the remaining array, as this minimizes the cost of swaps and overtakes.
- The DP array dp[i] stores the minimum cost to process elements from i to n. For each i, we iterate through all possible positions j starting from p to n, calculating the cost of swapping (swapCost), overtaking (overCost), and backtracking (backtrack).
- The total cost for i is derived by combining these costs with the precomputed cost dp[j].
- This approach ensures optimal substructure and overlapping subproblems are handled efficiently, resulting in an O(n^2) solution per testcase.
- The main function reads input, processes each testcase, and outputs the result by calling **solveCase**.

3. 2110F : Faculty

- The approach is to maintain a running maximum (**mx**) and optimizing calculations by recognizing two key scenarios:
    1. When encountering an element at least twice as large as **mx**, it fully compares this element with all previous elements since it could yield higher values for **f(x, y)**.
    2. For smaller updates to **mx**, it simplifies the calculation knowing the maximum possible value of **f(x, y)** at that point is the new **mx** itself.
- This approach balances thoroughness with efficiency by minimizing unnecessary computations while ensuring all potential maxima are considered.

Dynamic Programming (DP):

4. 603A : Alternative Thinking

- In this question using the DP approach would be **less efficient** than the greedy approach that I found to be simpler.
- The DP approach would be O(n^2), whereas the greedy approach will only be O(n).
- The approach involves counting the initial number of distinct adjacent segments (**res**) in the string.

- For example, the string "0011" has 2 segments ("00" and "11"), so **res** = 2.
- We can see that each operation can reduce the number of segments by at most 2 (by merging adjacent segments).
- Thus, the maximum possible reduction is **res** + 2, but it cannot exceed the string length n.
- The final result is the minimum of these 2 values, ensuring efficiency with a single pass through the string.

5. 2107C : Maximum Subarray Sum

- The approach is to use Kadane's Algorithm to calculate the Maximum Subarray Sum, and check if it exceeds k or if adjustment isn't possible (no '0' in the string).
- If valid, then calculate the maximum possible sums to the left (R) and right (L) of the first '0' position.
- Then, the value at this position is set to **k - L - R**, ensuring the maximum subarray sum equals **k**.

Mathematics and Number Theory:

6. 571A : Lengthening Sticks

- The approach involves calculating the total possible ways to distribute **l** increments (combinations with repetitions) and subtracting the invalid cases where triangle inequality fails.
- The **count_invalid** function efficiently checks invalid combinations by iterating over possible increments for one side and calculating invalid distributions for the remaining sides using Arithmetic Progression.
- The main function aggregates invalid counts for all three sides and subtracts them from total possibilities, ensuring optimal performance with combinatorial mathematics.

7. 2093C : Simple Repetition

- The approach is to first check the edge cases: if **n** is 1, etc.
- For **k=1**, it simply checks if **n** itself is prime using a helper function **isPrime()**, which efficiently tests primality up to the square root of **n**.
- For other cases, the code quickly outputs "NO" if **n** is even (and not 2) or if both **n>1** and **k>1**, so that it is optimized for specific scenarios.
- The main logic in **solve()** handles targeted conditions, while the primality test ensures correctness for the base case when **k=1**.

Implementation and Greedy Strategies:

8. 2104C : Card Game

- The key insight is in checking if there exists any 'A' (Alice's move) that cannot be "beaten" by any 'B' (Bob's move) based on their positions.
- The **beats** function defines the winning condition: position **x** beats **y** if **x** is 0 and **y** is **n-1**, or **x** is **n-1** and **y** isn't 0, or **x** is greater than **y** in normal cases.
- The **solve** function iterates through each 'A' and checks if no 'B' can beat it.
- If such an 'A' exists, Alice wins; otherwise, Bob wins.

Bitmasking and Optimization:

9. 2094E : Boneca Ambalabu

- For each bit position (0 to 29), count how many numbers have that bit set (**cnt[j]**).
- For each element, calculate the sum of XORs with all other elements by leveraging these counts: if the bit is set, it contributes **(n - cnt[j])*(1 << j)** (since unset bits will produce a set bit in XOR), and if unset , it contributes **cnt[j]*(1 << j)**.
- The maximum of these sums across all elements is the answer.
- This approach efficiently reduces the O(n^2) brute-force problem to O(n*30) per test case by preprocessing bit counts.

10. 2109B : Slice to Survive

- The approach is to use two strategies: cutting rows first or cutting columns first.
- For each strategy, it computes the cost by considering the logarithmic ceiling of the remaining segments after each cut, which represents the number of binary cuts needed.
- The **compute_log_ceiling** function efficiently calculates this using bitwise operations (***_builtin_clz***).
- Then compare the costs of both strategies and selecting the minimum one.