Problem A. Birthday-Candies

Let us take a quantity $c_i = a_i - b_i$, which represents the net amount of candies Max will give to a friend. Then we can claim that in the worst-case scenario, where the minimum amount of candies he'll buy for a given order, is maximized when the order of arrival of friends is such that all friends with a positive c_i arrive before others, as after giving each of these friends candies, Max will be left with less than what he began with before giving. Even in this order, Max must either expect the person with the largest b_i (let's call this $i = x$)to come last among those with a positive c_i as this person will give Max the most candies back, or the person with maximum a_i (let's call this $i = y$) among those with a non-positive c_i to come first after all the friends with a positive c_i. If $b_x < a_y$, then the order to expect is the latter, else the order to expect is former. For cases with all positive or all non-positive c_i's, the solution becomes trivial.

Overall complexity: $O(n)$ per test case

Problem B. Birthday Gift

Denote the answer by ANS. There are two possible cases:

- $ANS \geq 0$ Note that if it is possible to achieve x as the maximal element of array using k coins, then $ANS \leq x$. So, we use binary search on $[0, MAX]$(MAX denotes the maximum element of array initially) to find ANS. Now, we just need to check for a given x, whether we can achieve the maximum element of array to be $\leq x$. Iterate over all the elements of array, and find the minimum coins to make all elements of array to be $\leq x$ and compare this value to k. We need to find the minimum number of coins to make any element a_i of the array to be $\leq x$. To do that, observe that $[\frac{a-1}{2}] \geq [\frac{a}{2}] - 1$. So, it is always better to apply operations of type 2 earlier. Iterate over number of times you want to apply 2^{nd} operation and find the cost in each case, and take minimum. Doing this for each a_i, we can find the minimum cost to achieve the max element to be $\leq x$, and hence find the answer.

- $ANS < 0$ First find the minimum value achievable in $[0, MAX]$ just like above case. If it comes out to be 0, then subtract the minimum cost to achieve this from k. Now, all elements are 0 and it is easy to find the answer. It is $-[\frac{k}{n \times c_1}]$ because operation 2 no more helps in decreasing the maximum of array.(k is obtained after the subtraction specified earlier.)

Overall complexity: $O(n * logMAX * logMAX)$

Problem C. AND OR XOR

Let's see how to generate a set S as answer with integers $a_1, a_2, \ldots, a_{|S|}$ if it exists. If $|S| = 1$ then $O = A = X$ and $a_1 = O$. Now assume $O > A$. Note that any element a_i satisfies $a_i \& A = A$. So, $X \& A = A$ or $X \& A = 0$. In the former case, $|S|$ must be odd, i.e. atleast 3. Take $a_1 = O$, $a_2 = A$ and $a_3 = O \oplus A \oplus X$. If an answer exists, then it is easy to check that these numbers are valid. In the latter case, $|S|$ must be even. If we need $|S| = 2$, then X must be equal to $A \oplus O$ because for any 2 integers x and y, $x \oplus y = (x \& y) \oplus (x|y)$. So, if $A \oplus O = X$, we have $a_1 = A, a_2 = O$ as an answer. If $X \neq A \oplus O$ then we need $|S| \geq 4$. Consider $|S| = 4$ and $a_1 = A, a_2 = A, a_3 = O, a_4 = O \oplus X$. If answer exists, then these numbers must be valid. So, we see $|S| \leq 4$ will always work, and we can always have one of the following sets: O,O, A,$O, A, O \oplus A \oplus X$, $O, A, A, O \oplus X$. Try each of them from left to right. If none of them work, print -1.

Problem D. Worthy Winner

Highly optimised brute force will do (iterate over all numbers from 0 to LCM).

But an elegant approach would be to use Chinese Remainder Theorem. The basic form

$N = a_1 \pmod{r_1}$

$N = a_2 \pmod{r_2}$

where r_1 and r_2 are coprime is very standard. You always have a guaranteed solution here. You can read about the proof https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html

Here r_1 and r_2 are non coprime. So if $\gcd(r_1, r_2)$ is not divisible by $abs(a_1 - a_2)$ then solution is not possible.

Now we calculate x and y from $r_1 x + r_2 y = \gcd(r_1, r_2)$ using Extended GCD algorithm https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/

and thus as a direct result of CRT we have one possible answer as $a_1 \times (\frac{r_2}{g}) \times y + a_2 \times (\frac{r_1}{g}) \times x$. (since $\frac{r_1}{g}$ and $\frac{r_2}{g}$ are coprime). Then we find the min positive ans and use AP formula to get the required result.

So here time complexity is $O(\log(\min(r_1, r_2))$ which passes the test cases.

Problem E. n-nice numbers

It can be shown that for a given value of n, the minimum possible length of binary representation of an n - nice pair is $2^n + n + 1$. For $n \geq 18$, the length will exceed 2×10^5 , hence no n - nice pairs are possible.

let (a, b) be an n - nice pair. Since a and b differ exactly by 2^n, the remainder obtained when 2^n divides a and b will be equal. Hence, the first n least significant bits of a and b will have to be same, which can be chosen arbitrarily. Without loss of generality let them be "$000\cdots$" (n times).

Now, a and b can be constructed as:

$a =$ "10" + "$1111\cdots$"($2^n - 1$ times) + "$000\cdots$"(n times)

$b =$ "11" + "$0000\cdots$"($2^n - 1$ times) + "$000\cdots$"(n times)

You can easily verify that a and b have 2^n mismatches. Also, b can be obtained if 2^n is added to a.

Problem F. Are you winning?

Since the permutation and numbers chosen by Bob's Dad were completely random, every possible sequence has an equal probability of being the correct sequence. Thus, it suffices to count the possible correct sequences based on the first guess.

To count the number of valid guesses, the following conditions need to be ensured (considering that there are g numbers coloured with G, y numbers coloured with Y and r numbers coloured with R):

1. The numbers marked with G will not be changed or reordered.

2. The numbers marked with R need to be replaced with numbers not present in the original guess. There are $\binom{n-k}{r}$ ways of choosing the numbers.

3. The numbers marked with Y need to be reordered so that none appears in the original place (since such a permutation **cannot** be a valid permutation).

4. Out of all permutations in the places marked with Y or R, those permutations need to be removed in which at least one of the y numbers retains its original position. The total number of valid permutations can be calculated using `Principle of Inclusion-Exclusion` as follows:

$$S = \sum_{i=0}^{y} (-1)^i \times (r + y - i)! \times \binom{y}{i}$$

Now, for every combination chosen in point 2, there are S valid permutations. Thus the total number of all valid permutations is:

$$T = S \times \binom{n-k}{r}$$

And thus, the maximum probability is $\frac{1}{T}$.

Problem G. Painting Tiles

The idea is to not use distinct colours for same number of tiles. If two distinct colours are use to paint two sets of 2^i tiles, it can be minimized by painting all $2 \times 2^i = 2^{i+1}$ tiles with a single colour. There is only one way to do this, which is the binary representation of n. Hence the minimum number of colours required will equal to the numbers of set bits in the binary representation of n.

Problem H. Good Sequence

The problem can be solved by using 3d-DP.

Let us consider the number of good sequences of length i, where the indices of $1s$ present in the binary representation of the last
(i^{th}) number are j and k ($0 \leq j < k < 32$), be stored in a 3d array dp[i][j][k].

Now for $i = 1$, dp[1][j][k] $= 1 \forall j, k \in [0, 32)$, as there is only one possible number with $1s$ present at indices j and k.

Now, any good sequence of length i is formed from good sequences of length $i - 1$. If the i^{th} number has ones at position j and k and the $i - 1^{th}$ number has ones at position J and K, then J and K satisfies the conditions: $0 \leq J < K < 32$, $j - m \leq J \leq j + m$ and $k - m \leq K \leq k + m$ Thus, dp[i][j][k] is the summation of dp[i][J][K] for all J and K satisfying the above conditions.

Since the value of n and m are small enough, we can loop over all the values of J and K.

Time Complexity: $O(nb^2m^2)$ where b is the maximum size of the binary representation of the numbers in the sequence (32 in this case).

Solution: `https://pastebin.com/qiC2sJif`

Problem I. Pamdalchemist

It is clear that we want the total number of increasing subsequences in the array of volatilities of length k.

First, we do a coordinate compression, to reduce the range of volatilities to $[0, U - 1]$, such that U is the number of unique elements.

This problem can be solved by DP (Dynamic Programming).

Let $vol[k]$ represent the volatility of chemical at index k. Define $dp[len][j]$ to be the number of subsequences of length $len + 1$ starting at index j. Then $dp[len][j]$ = Sum of number of subsequences of length len starting from any index k, such that $k > j$ and $volatility[k] > volatility[j]$. More formally:

$dp[len][j] = \sum dp[len - 1][k]$ such that $k > j$ and $vol[k] > vol[j]$

To quickly sum those $dp[len - 1][k]$ for which $vol[k] > vol[j]$ and $k > j$, for every index $0 \leq j \leq n - 1$, we use segment trees. Formally, we do k iterations, and calculate $dp[len][j]$ at the len^{th} iteration for every index j. We initialise the segment tree with $0's$.

At iteration len ($1 \leq len \leq k$), the segment tree's node representing the segment $[l, r]$, contains the total number of subsequences of length len starting from $l, l + 1, l + 2, ..., r$ ($0 \leq l \leq r \leq n - 1$). We start from the end of the array of volatilities and include $dp[len - 1][j]$ in the segment tree at the j^{th} sub-iteration, to ensure that only indices greater than j contribute to $dp[len][j]$. Before calculating $dp[len][j]$, we add $dp[len - 1][j]$ to the segment tree at the leaf $vol[j]$. Then $dp[len][j]$ = sum of segment $[vol[j] + 1, n]$.

Solution: `https://pastebin.com/atxud9y9`.

Note: Here we have used an iterative implementation of segment tree, by Al.Cash. To learn more, visit the blog `https://codeforces.com/blog/entry/18051`

Visit `https://cp-algorithms.com/` for more about recursive implementation of segment tree.

Problem K. Assignment Conundrum (Hard)

For ease of implementation, we number the assignments from 0 to $n-1$ and do the i-th assignment on days $i+1,\ldots,i+a_i$.

We solve the problem using dynamic programming.

Let $f[i]$ be the minimum number of days spent on assignment if an assignment was finished on day i. Let $g(i)$ be the minimum number of days that must be spent doing an assignment to reach day i. Let S_i be the set of all assignments ending on day i that is $S_i = \{j \mid j + a_j = i\}$.

Now the recurrence for f is given by,

$$f[i] = \min_{j \in S_i}(g(j) + a_j)$$

with the base case $f[0] = 0$ and $f[i] = \infty$ if S_i is empty. We get this recurrence by fixing the last assignment j and taking the minimum over all options. We must first do some assignments to reach the day j and then $j+1,\ldots,j+a_j = i$ were spent doing this assignment. $g(j)$ is the contribution of the former to the answer and a_j is the contribution of the latter.

The value of $g(i)$ is given by

$$g(i) = \begin{cases} 0 & i \le k \\ \min_{i-k \le j \le i} f[j] & \text{otherwise} \end{cases}.$$

Since an assignment must have been done on atleast one of the days $i-k,\ldots,i$.

The final answer is given by $\min_{i \ge n-k} f[i]$.

Let T be the time complexity of computing $g(i)$, thus the time complexity of the solution would be $O(\sum_{i=1}^{N}\sum_{j\in S_i} T) = O(T\sum_{i=1}^{N}|S_i|) = O(TN)$.

Implementing this naively leads to an $O(NK)$ solution which ACs for the easy version of the problem.

We can achieve $T = O(1)$ using a monotonic queue or $T = O(\log N)$ using a segment tree, both of which should comfortably pass the time limit.

Problem L. Square it up!

We may assume $n \le m \le n^2$, since otherwise no solution exists. Now we define the following DP to solve the problem, let $dp[n][m]$ be the minimum value of k for the given n, m if a solution exists, ∞ otherwise. The recurrence for the problem then becomes,

$$dp[n][m] = 1 + \min_{\substack{x \le n \\ x^2 \le m}} dp[n-x][m-x^2] \text{ for } n \le m \le n^2$$

with the base case, $dp[0][0] = 0$ and $dp[n][m] = \infty$ if $m < n$ or $m > n^2$.

The time complexity of computing this dp is $O(n^4)$, recomputing this for each testcase would cause the solution to TLE, so we can instead compute this DP once before all the testcases.

To construct the solution, we define an auxillary array $p[n][m]$, which stores the value of x for which $dp[n][m] = 1 + dp[n-x][m-x^2]$. We can compute this auxillary array while computing dp.

Now we simply print $x = p[n][m]$, then subtract x from n and x^2 from m. Repeating this until n becomes 0. The time complexity of this step is $O(k)$ per testcase.

The final time complexity of the solution is $O(n^4 + \sum_{\text{over all tests}} k)$.

Problem M. Awesome Strings

Firstly, observe that the operations performed on the even and odd substrings are mutually independent. So, we can treat the the two sub-problems independently, on a similar algorithm.

Now, consider the odd substring. In a greedy approach, it would be most optimum to change all the characters of the substring not belonging to the largest non-decreasing subsequence for this substring. It is trivial that this would guarantee the existence of a modified substring. Also, had it been possible to modify the substring using less number of steps, it would imply that at least one character not belonging to longest non-decreasing subsequence must remain unchanged. But this would rather imply the character to be present in the non decreasing subsequence: a contradiction!

The even substring could be dealt with similarly(hint: if all the elements of a non decreasing sequence are multiplied by -1, the resulting sequence is non increasing) To find the the longest non decreasing subsequence that fits well in the time constraint, an O(n log n) approach is feasible. Try out the patience algorithm!

Bonus: The program can be solved in O(n) time as well. Use dynamic programming and the fact that there are only 26 english alphabets.

Problem N. XYZ wants maximum money

We make a hash map containing the value of each coin along with its frequency. Therefore, the maximum amount that can be obtained is the maximum product of the value of each coin with its frequency of occurrence.