PREREQUISITES:

Greedy algorithms, sorting (or binary search)

PROBLEM:

The beauty of an array equals the sum of pairwise products of its elements.
You're given an array A. At most K times, you can increase one of its elements by 1.
What's the maximum possible beauty of the final array?

EXPLANATION:

There are three parts to this problem:

1. Computing the beauty of an array faster than $\mathrm{O}(N^2)$.
2. Figuring out which elements to apply the operations to.
3. Actually applying these operations quickly, since $\mathrm{O}(K)$ would be too slow.

Let's go over these one at a time.

Computing beauty

We want to quickly compute the sum of pairwise products of elements.
Let $S = \sum_{i=1}^{N} \sum_{j=1}^{N} (A_i \cdot A_j)$ be this quantity. Then,
Note that

$$\begin{aligned}(A_1 + A_2 + \ldots + A_N)^2 &= (A_1^2 + A_2^2 + \ldots + A_N^2) + 2 \cdot \sum_{1 \le i < j \le N} A_i \cdot A_j \\ &= (A_1^2 + \ldots + A_N^2) + 2S\end{aligned}$$

So, S can easily be computed in $\mathrm{O}(N)$: we only need to know the square of the sum of the elements, and the sum of squares of the elements.

Now, let's look at what happens when we need to perform operations.
Suppose we had $K = 1$, i.e, only one operation needed to be performed. What would the optimal move be?
From the way we rewrote the cost above, it's obvious that we should choose the smallest A_i to increase by 1.

In fact, this greedy choice holds even for $K > 1$: for each move, choose the smallest element of A and increase it by one!

Proof

This will be a bit long to write out, though it's really just several applications of exchange arguments.

Let $A_1 \le A_2 \le \ldots \le A_N$.
Suppose we increase A_i exactly B_i times; so the final values are $C_i = A_i + B_i$.

Recall that the beauty is (half of) the square of the sum of C_i, minus the sum of squares of C_i.
Note that the sum of C_i is always a constant (and equals K plus the sum of A_i).
So, maximizing the beauty requires us to minimize the sum of the squares of C_i.

Let's call a *solution* an assignment of B_i values, and an *optimal solution* a solution that maximizes beauty.

Claim 1: There exists an optimal solution such that $C_i \le C_j$ when $i \le j$.
Proof: Trivial, if $C_i > C_j$ then since $A_i \le A_j$ it's always possible to shift operations from index i to index j and

essentially swap C_i and C_j.

From now on, we'll only consider optimal solutions with this structure.

Claim 2**:** In an optimal solution, if $i < j$ and $B_j > 0$, then $B_i > 0$.
Proof: If $B_i = 0$, reduce B_j by 1 and increase B_i by 1.
Work out the algebra to see that this gives us a strictly better beauty, which contradicts us starting with an optimal solution.

This tells us that we'll only operate on some *prefix* of indices.
Let m be the highest index operated on.

Claim 3**:** If $1 \leq i < j \leq m$, then $C_j - C_i \leq 1$
Proof: If C_i and C_j differ by two (or more), reduce one operation from the larger one and give it to the lower one; this improves beauty, which is once again a contradiction.

Note that these properties (almost) uniquely characterize an optimal solution.
In particular, there'll be an index m such that:

- The suffix starting from A_{m+1} is unchanged, i.e, $C_i = A_i$ for this suffix.
- There'll be some integer x such that every C_i upto m is either x or $x + 1$.

Note that if m is fixed, x is also uniquely fixed since we must perform exactly K moves.
Further, m itself is essentially fixed: we need to (at least) bring all the elements up to A_m, requiring $(A_m - A_1) + (A_m - A_2) + \ldots + (A_m - A_m)$ moves.
This is an increasing function of m and we want it to stay within K, so the breakpoint is essentially unique.

Finally, note that the greedy solution of "always increase the smallest element" achieves this exact structure as well; and hence is optimal.

Now, let's see how we can use this information to solve the problem at hand.

Let's sort A, so that $A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_N$.
Now, look at the process of performing operations:

- First, we increase only A_1 till it reaches A_2.
- Then, we alternate increasing A_1 and A_2 till they reach A_3.
- Then, we A_1, A_2, A_3 in turn till they all reach A_4.

  ⋮
- We increase $A_1, \ldots, A_{i-1}$ in turn till they all reach A_i.

This allows us to quickly simulate the process by combining operations.
Let's iterate from 1 to N.
When at index i:

- We've ensured that so far, $A_1 = A_2 = \ldots = A_i$.
  Note that they'll all be equal to (the initial value of) A_i.
- Next, we must perform operations till all these elements reach A_{i+1}.
  This takes exactly $i \cdot (A_{i+1} - A_i)$ operations.
- If K is at least this number, we know that all the operations can be performed, so we move to index $i + 1$.
- Otherwise, the process stops at this index.
  Since the operations are to be performed on the elements in turn, the exact values of all of $A_1, \ldots, A_i$ can be figured out; and $A_{i+1}, A_{i+2}, \ldots, A_N$ won't change.

We do $\mathrm{O}(1)$ work at each index; and $\mathrm{O}(N)$ work at one index, so this is overall linear.
Since we required sorting, the complexity is $\mathrm{O}(N \log N)$.

PROBLEM:

There are N empty boxes, and A_i balls of color i (for $1 \leq i \leq M$).
The balls must be distributed to the boxes such that:

- Every ball is placed in a box
- Each box contains balls of distinct colors

Find the minimum number of boxes that have balls of all M colors.

EXPLANATION:

Let's try to place the balls in the boxes one color at a time, i.e, from 1 to M.
A box will be called *bad* if it contains a ball of all colors **seen so far**, and *good* otherwise.
Our objective is to have as few bad boxes (or equivalently, as many good boxes) as possible in the end.

For each i from 1 to M, notice that:

- We can freely place balls of color i in any existing good boxes: they will still remain good.
  This is because a good box at this stage must be missing a color that's $< i$, which it will continue to not have in the future.
- If we place a ball of color i in a bad box, it will remain bad.
- If we *don't* place a ball of color i in a bad box, it will turn good after this step.

Given that our aim is to maximize the number of good boxes, clearly the optimal strategy is to place balls in good boxes first, and only then place them in bad boxes.
This ensures that we convert as many bad boxes to good, as possible.

In particular, we have the following algorithm:
Suppose there are currently x bad boxes and y good ones.
Initially, $x = N$ and $y = 0$.
For each i from 1 to M:

- We can place $\min(A_i, y)$ balls into good boxes; so let's do that first.
- The remaining $A_i - \min(A_i, y)$ balls must be placed into bad boxes; and only these boxes will remain bad.
- So, we can update $x \to A_i - \min(A_i, y)$ and $y \to N - x$.

Finally, print the value of x.

TIME COMPLEXITY

$\mathrm{O}(M)$ per testcase.

PROBLEM:

You have N songs saved on your Dotify app. The i-th song is M_i minutes long and in language L_i.
Find the longest playlist you can make if:

- Exactly K songs from these N should be chosen
- Each song should be of language L

EXPLANATION:

We only care about songs whose language is L.
So, let's create a list of the lengths of only these songs, say A.

We now want to choose K elements from A such that their sum is as large as possible.
So,

- If the length of A is less than K, choosing K songs is of course impossible.
  In this case, the answer is `-1`.
- Otherwise, clearly it's optimal to choose the largest K elements of A and find their sum.

A simple way to do this is to sort A and then take the sum of its last K elements.
However, the constraints here are small, so a solution in $\mathrm{O}(N \cdot K)$ that repeats "find the maximum element in A, add it to the sum, and then remove it from A", K times, will also get AC.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ or $\mathrm{O}(N \cdot K)$ per test case.

PREREQUISITES:

The process of **Huffman coding**

PROBLEM:

Alice and Bob play a game, independently, on an array of N elements. At each step, a player will take two elements x and y from the remaining set, delete them, and add $x+y$ to the set.
They also add $x+y$ to their score.

Alice has a well-defined strategy: (except for the very first move) she will always use the newly combined element and combine it again.
Bob, meanwhile, can choose his moves as he likes.

Bob wins if his score is strictly less than Alice's.
To achieve this, Bob can change elements of the array as he likes, before any moves are made.
Find the minimum number of changes so that Bob, with optimal play, can guarantee a win over Alice. Also find one such array.

EXPLANATION:

The first step to solving this problem is analyzing Alice's and Bob's best strategies, and seeing when exactly Bob can beat Alice.

Alice's strategy

Let $A_1 \leq A_2 \leq \ldots \leq A_N$.

Alice's best strategy is then as follows:

- Merge A_1 and A_2
- Merge $A_1 + A_2$ and A_3
- Merge $A_1 + A_2 + A_3$ and A_4

$\vdots$

Alice's best score is thus the sum of all prefix sums of (sorted) A, minus A_1.

Bob's strategy

Bob is free to make moves as he likes, so the optimal strategy is to always pick the two smallest remaining elements, Huffman-style. A proof of this optimality can be found **here** .

Now that we know their strategies, it's also not hard to see when exactly Bob can beat Alice: there should be a move when Alice *doesn't choose* the smallest two elements.
That is, in the sorted array A, there should exist an i such that $A_1 + A_2 + \ldots + A_i$ is *not* among the two smallest elements; with the other elements being $[A_{i+1}, A_{i+2}, \ldots, A_N]$.

In particular, notice that this can only happen when $A_1 + \ldots + A_i > A_{i+2}$.

So, Bob's aim is to reach an array where this is the case.

This leads us to a solution that deals with a few cases:

- If $N \leq 3$, Bob can never win.
- First, if this condition is already satisfied, nothing more needs to be done — Bob already wins.
- Otherwise, we need to see what we can modify.

First, let's check if one modification is enough.
If i is fixed, we want to see if $A_1 + \ldots + A_i > A_{i+2}$.

The best we can do is bring A_{i+2} down to A_{i+1} (since sorted order needs to be maintained), so if $A_1 + \ldots + A_i > A_{i+1}$, we're done: change A_{i+2} to A_{i+1}.

If this is not satisfied for any $1 \leq i < N-1$, it's always possible to achieve our aim in 2 moves by setting $A_2 = A_3 = A_4$, at which point $A_1 + A_2 > A_4$.

Note that we sorted A to see which changes need to be made, but the actual changes need to be made on the original positions of A; so make sure to keep track of indices as well.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Greedy algorithms, **Bitwise XOR**

PROBLEM:

Chef has 3 hidden numbers A, B, and C such that $0 \leq A, B, C \leq N$.

Let f be a function such that $f(i) = (A \oplus i) + (B \oplus i) + (C \oplus i)$. Here $\oplus$ denotes the **bitwise XOR** operation.

Given the values of $f(0), f(1), \ldots, f(N)$, determine the values of A, B, and C.

It is guaranteed that **at least** one tuple exists for the given input. If there are multiple valid tuples of A, B, C, print any one.

EXPLANATION:

$A, B, C \leq N \leq 10^5 \implies$ only first 18 bits can be set to 1 in A, B *and* C.
Also as only the sum $f(i) = (A \oplus i) + (B \oplus i) + (C \oplus i)$ is taken into consideration for each bit from 0^{th}(least significant) bit to the 17^{th} bit only the number of set bits is important.

only the number of set bits is important.

Let $countset_j$ represent the count of numbers in which the j^{th} bit is set to 1.

$f[i] = \sum_{j=0}^{17} 2^j * ((i \,\&\, 2^j)?(3 - countset_j) : countset_j)$. Thus only the number of set bits for each bit is important to calculate the answer.

Let set be the number of integers in which i^{th} bit is set to 1 and $unset$ be the number of integers in which i^{th} bit is set to 0.
Then $set + unset = 3$ and $set - unset = \frac{f[0]-f[2^i]}{2^i}$
The number of set bits for each bit i ($2^i <= N$) can be calculated as $\frac{(3+\frac{f[0]-f[2^i]}{2^i})}{2}$.

Initialize A, B and C as 0. Now iterate over bits from most significant to least significant bit and maintain the order of elements (ascending or descending), calculate the number of set bits for each bit using the above formula, add them to the the numbers in ascending order. For example if number of set bits for a bit is 1 add it to the smallest number, if it is 2 add it to the two smaller values and so on. This greedy approach produces one of the correct answers.

This greedy approach produces one of the correct answers.

Let i^{th} bit from right be the first bit which is not same in all A, B *and* C. If it is present in only one number, set it in A and then whenever we have a choice of setting the bits i.e. set bit count is 1 or 2 we set them in B and C as they can never exceed A. Since an answer always exists this produces a correct answer.
If it is present in two numbers set it in A and B and then whenever we have a choice of setting the bit in one number i.e. set bit count is 1, we set it in C as it can never exceed A. When we get a choice of setting a bit in two numbers for the first time we set it in A and C and later every time in B and C
Thus, this greedy approach always produces one of the correct answers.

For details of implementation please refer to the attached solutions.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Sorting

PROBLEM:

Chef wants to impress Chefina by giving her the maximum number of gifts possible.

Chef is in a gift shop having N items where the cost of the i^{th} item is equal to A_i.
Chef has K amount of money and a 50% off discount coupon that he can use for **at most one** of the items he buys.

If the cost of an item is equal to X, then, after applying the coupon on that item, Chef only has to pay $\lceil \frac{X}{2} \rceil$ (rounded up to the nearest integer) amount for that item.

Help Chef find the **maximum number** of items he can buy with K amount of money and a 50% discount coupon given that he can use the coupon on **at most one** item.

EXPLANATION:

Let M be the maximum number of items that Chef will buy, and let $\{i_1, i_2, \ldots i_M\}$ be the list of items that costs minimum among all possible list of items of size M. Also, let $S = \sum_{\alpha=1}^{M} A_{i_\alpha}$

Observation 1: Chef will always use the discount coupon on the most expensive item. This is because the amount of money that Chef will pay will be $S - discount$, and as the price increases, the discount increases.

Observation 2: The list of items will have the M lowest priced items. To prove this, let us introduce a new item j in the list of items and remove i_α such that $A_{i_\alpha} < A_j$. We can make 4 cases on {discount on A_{i_α}, discount on A_j} and in each case, we can see that the original list results in less overall cost.

So we can first sort the items in the increasing order of price, and for each i such that $1 \leq i \leq N$, check if we can take first i elements by applying discount on the i^{th} item. The largest feasible i will be our answer.

TIME COMPLEXITY:

$O(N \cdot \log N)$ for each test case.

PREREQUISITES:

Combinatorics

PROBLEM:

You are given a $N \times N$ chessboard. Calculate two things -

- The minimum number of pawns that need to be placed on the board so that they attack every square on the board except the squares on the first rank.
- The number of ways to arrange these pawns in order to achieve the goal modulo $10^9 + 7$.

QUICK EXPLANATION:

This is a casework problem. The following four cases arise -

$N \bmod 4$	Number of pawns needed	Number of ways of arranging them
0	$(N-1) \times \frac{N}{2}$	1
1	$(N-1) \times \frac{N+1}{2}$	$(\frac{N-1}{4})^{N-1}$
2	$(N-1) \times \frac{N+2}{2}$	$(\frac{N+2}{4})^{2(N-1)}$
3	$(N-1) \times \frac{N+1}{2}$	$(\frac{N+5}{4})^{N-1}$

EXPLANATION:

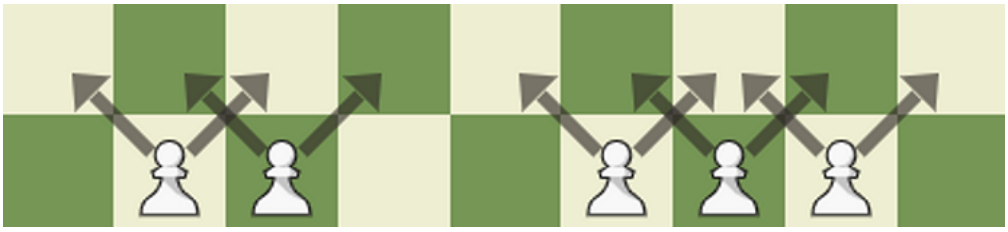
First, notice that a pawn on the i^{th} rank can only attack the squares on the $(i+1)^{th}$ rank. So, the arrangement of pawns in one rank cannot affect the arrangement in any other rank. Hence, the arrangement of pawns in different ranks are independent of each other and we can simply solve the problem for a single rank and then raise it to the power $N-1$ to get the answer for the whole board.

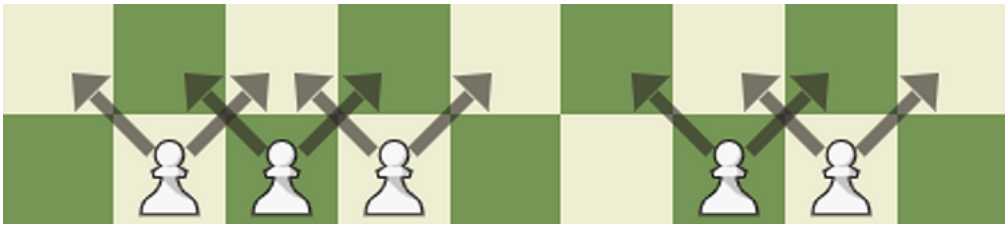
Second, note that a pawn can attack at most two squares having the same color as the square in which the pawn is placed. So, pawns placed on different colored squares are also independent of each other. This leads to the following 4 cases -

1. $N \bmod 4 = 0$: In this case, each rank has $\frac{N}{2}$ light squares and $\frac{N}{2}$ dark squares. So, we need at least $\frac{N}{4}$ pawns to attack the light squares and at least $\frac{N}{4}$ pawns to attack the dark squares. It turns out by construction that these many pawns are sufficient.
   It is also not hard to see that there is only 1 way to arrange them so that they attack all squares on the next rank. So, the second answer is simply 1.
   The following figure shows the arrangement for a single rank of an 8×8 board -

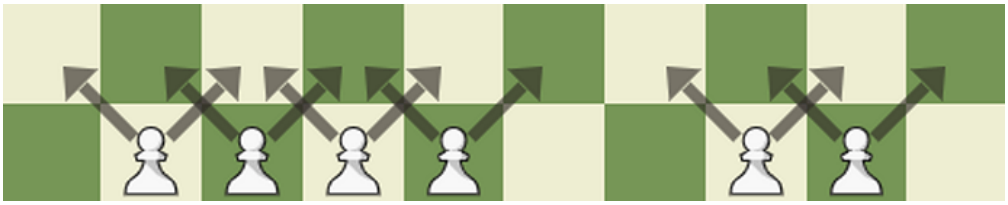
2. $N \bmod 4 = 1$: Without loss of generality, let's assume that there are $\frac{N+1}{2}$ light squares and $\frac{N-1}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 1$, there are an odd number of light squares and an even number of dark squares.
   For the pawns attacking the dark squares, there must be $\frac{N-1}{4}$ such pawns and they can be placed in 1 way only.

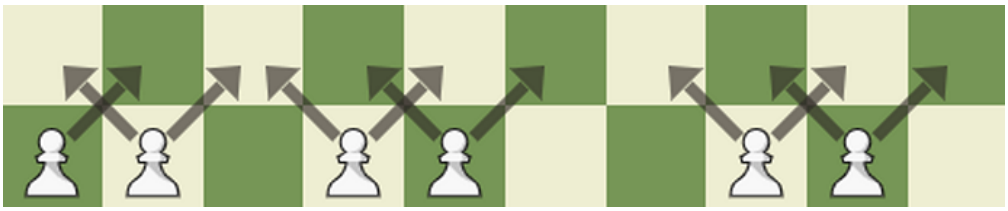
However, for the pawns attacking the light squares, there must be $\frac{N+3}{4}$ such pawns and there are multiple ways to arrange these pawns. It is not hard to see that all pair of consecutive pawns attacking the light squares must be placed at a distance of 4 files from each other except exactly one pair of pawns which must be placed at a distance of 2 from each other. There are $\frac{N+3}{4} - 1 = \frac{N-1}{4}$ choices for the pair.
So, the total number of arrangements for a single rank is $\frac{N-1}{4}$. The total number of arrangements is $(\frac{N-1}{4})^{N-1}$.
The following are the 2 possible arrangements for a single rank of a 9×9 chessboard -

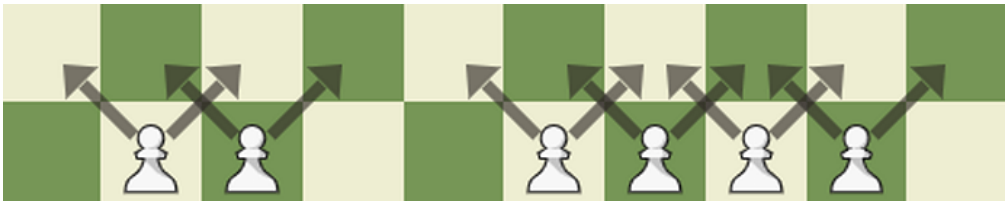


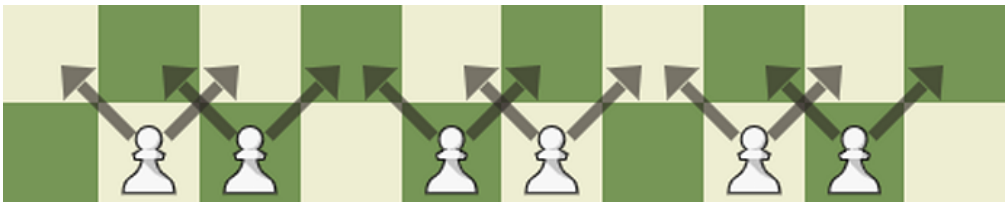


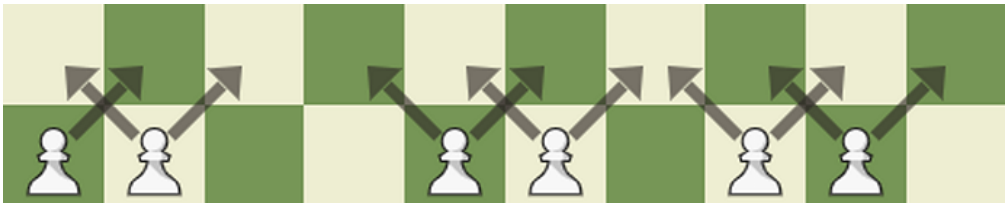
3. $N \bmod 4 = 2$: There are $\frac{N}{2}$ light squares and $\frac{N}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 2$, there are an odd number of light squares and an odd number of dark squares. We need $\frac{N+2}{4}$ pawns to attack squares of each type.
This time, all the pair of consecutive pawns attacking the light squares can be placed at a distance of 4 from each other except at most one pair of pawns which can be placed at a distance of 2 from each other. Same is true for the set of pawns attacking the dark squares. So, there are $\frac{N+2}{4}$ ways to arrange each of these set of pawns (there is 1 way in which no pair of consecutive pawns is at a distance of 2 from each other and $\frac{N+2}{4} - 1$ ways in which exactly one pair of consecutive pawns is at a distance of 2 from each other).
So, the total number of arrangements for a single rank is $(\frac{N+2}{4})^2$. The total number of arrangements for the board is $(\frac{N+2}{4})^{2(N-1)}$.
The following are the 9 possible arrangements for a single rank of a 10×10 chessboard -

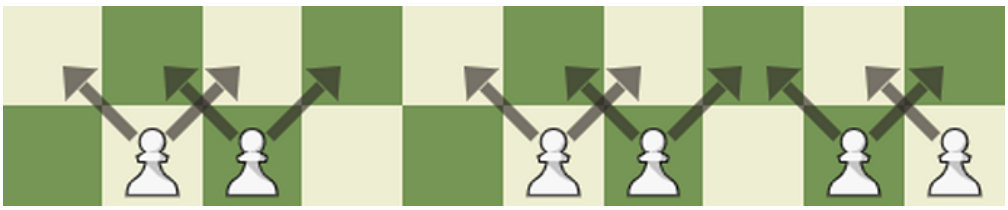


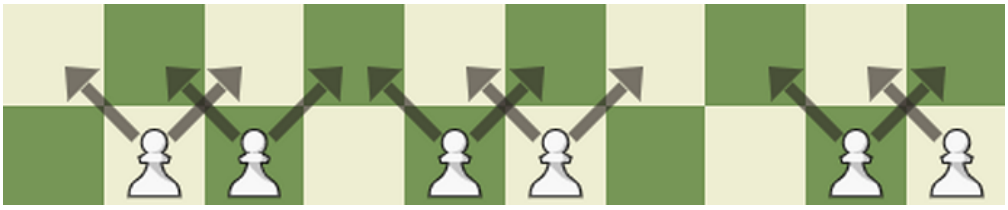


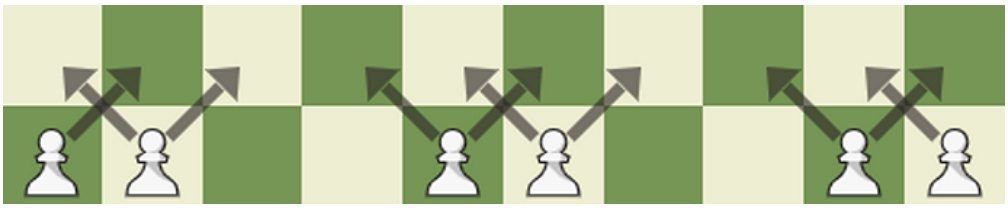




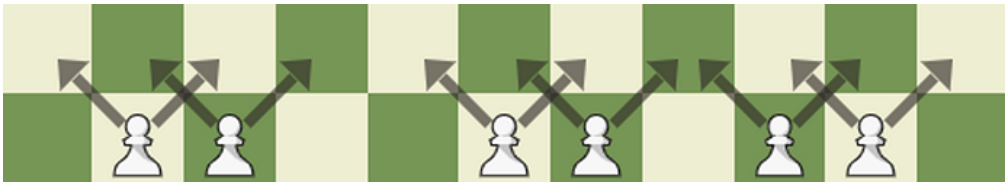


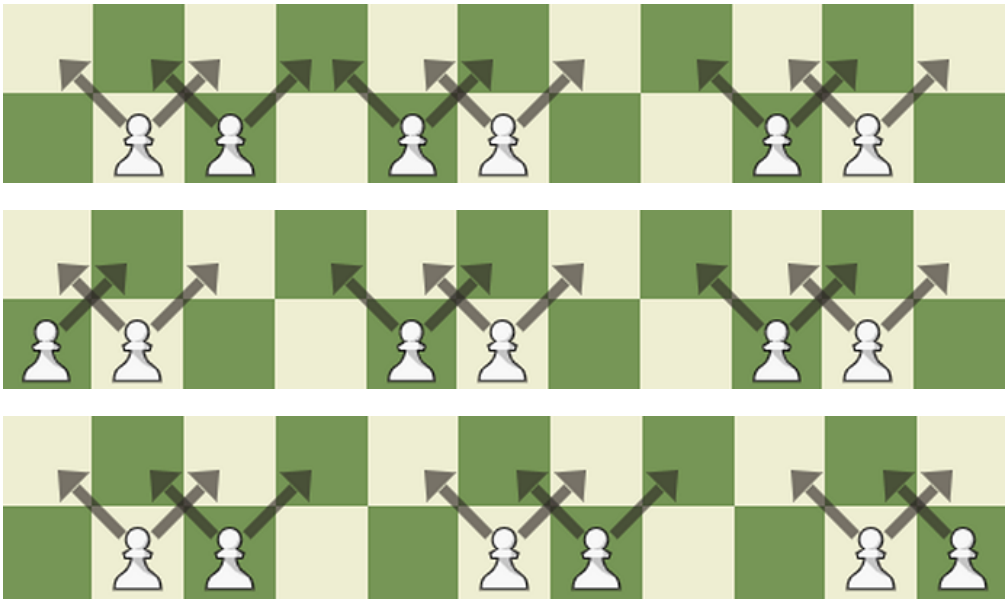






4. $N \bmod 4 = 3$: Without loss of generality, let's assume that there are $\frac{N+1}{2}$ light squares and $\frac{N-1}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 3$, there are an even number of light squares and an odd number of dark squares.
   For the pawns attacking the light squares, there must be $\frac{N+1}{4}$ such pawns and they can be placed in 1 way only.
   However, for the pawns attacking the dark cells, there must be $\frac{N+1}{4}$ such pawns and they can be arranged in $\frac{N+1}{4} + 1 = \frac{N+5}{4}$ ways (there are 2 ways in which no pair of consecutive pawns is at a distance of 2 from each other and $\frac{N+1}{4} - 1$ ways in which exactly one pair of consecutive pawns is at a distance of 2 from each other) .
   So, the total number of arrangements for a single rank is $\frac{N+5}{4}$. The total number of arrangements is $(\frac{N+5}{4})^{N-1}$.
   The following are the 4 possible arrangements for a single rank of a 11×11 chessboard -





TIME COMPLEXITY:

$O(T \log N)$ because of exponentiation.

PREREQUISITES:

Greedy

PROBLEM:

Given a binary string S, you can perform the following operations on it:

- Change a 0 to a 1
- Insert a 0 anywhere in the string

Find the maximum possible integer that can be formed after at most K operations.

EXPLANATION:

Let X be the decimal value of the string.
Let's see how the given operations affect the value of X.

- Suppose we change bit i from 0 to 1. This increases the value of X by 2^i.
- Suppose we insert a 0 after bit i. This doesn't change anything about bits $0, 1, 2, \ldots, i-1$, but shifts bits $i, i+1, \ldots$ to the left by one step, i.e, multiplies their values by 2.
  So, if Y is the total value of bits $\geq Y$, X will increase by exactly Y.

From this, it's easy to see that whenever we insert a 0, it's always optimal to do so at the end of the string, since it directly multiplies the value of X by 2.

So, we only need to decide when to insert a 0 and when to change a 0 into a 1.
In fact, this can be done greedily!

- If the leftmost character of S is 1, then insert a 0 at the end of S
- Otherwise, change the leftmost character of S to 1.

Each operation is easily performed in $\mathrm{O}(1)$, so the final string can be found in $\mathrm{O}(N+K)$.

TIME COMPLEXITY

$\mathrm{O}(N+K)$ per test case.

PREREQUISITES:

Frequency arrays

PROBLEM:

You have an array A, and an initially empty array B.
You can perform the following operation:

- Pick a non-empty subsequence of A.
- Append the mex of this subsequence to B.
- Delete the subsequence from A.

Find the lexicographically maximum possible value of B.

EXPLANATION:

Lexicographic minimization/maximization problems often require us to make greedy choices; and this one is no different.

In order to obtain the lexicographically largest array, the first step is to extract the maximum possible mex we can.
That is, find the largest possible integer k such that all of $0, 1, 2, \ldots, k$ exist in the array; giving us a mex of $k + 1$.

It's easy to see that this process can just be repeated:
While the array has at least one element, find the largest possible mex; and delete one instance of every element less than this mex.

We only need to implement this to run quickly enough.
That can be done with the help of a frequency array.

Let $\text{freq}[x]$ be the number of times x occurs in A.
Then, in each step we can do the following:

- Iterate k starting from 0. Stop when $\text{freq}[k] = 0$.
  k is the largest mex we can obtain.
- If $k > 0$, reduce $\text{freq}[x]$ by 1 for every $0 \leq x < k$, to simulate deleting the subsequence $\{0, 1, 2, \ldots, k - 1\}$ from the array.
- Otherwise, $k = 0$; meaning that 0 is no longer present in the array.
  This means all further mexes will only be 0.
  So, if there are m elements remaining in the array, append 0 to B, m times; then break out since we've processed everything we can.

The complexity of this is in fact $\mathrm{O}(N)$:

- Building the frequency array takes $\mathrm{O}(N)$ time, obviously.
- Whenever we find a mex of $k > 0$, we iterated $k + 1$ times (and deleted k elements) to do so.
  This means the total sum of non-zero mexes we obtain cannot exceed N; meaning the total number of iterations is also $\mathrm{O}(N)$.
- When $k = 0$, we append a bunch of elements to B and immediately broke out; obviously this is linear time as well.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

DFS/DSU, Observation

PROBLEM:

You're given a connected undirected weighted graph with N vertices and M edges.
Answer M **independent** queries on it:

- Add edge (u, v) to the graph with weight 0. Then, compute the weight of the minimum spanning tree of the resulting graph, where the total weight is the bitwise OR of chosen edges.

EXPLANATION:

Let Y be the value of the initial OR-MST weight.
Let's first see how to compute Y.

We'd like to minimize the bitwise OR of the chosen weights, so it's optimal to iterate across bits in decreasing order and see if we can avoid taking the current bit, even at the cost of all lower bits.

That leads us to the following algorithm:

- We'll iterate across bits in descending order from 29 down to 0.
  Let ans be the current answer; initially zero.
  Let E be the current active subset of edges; initially this equals all the edges.
- For a fixed bit b, remove any edge of E that has the b'th bit set. Let E' be the remaining set of edges.
  - if E' is a spanning set of edges for all N vertices, ans needn't have bit b set.
    In this case, replace E with E' and continue on.
  - Otherwise, ans must have the b'th bit set. So, increase ans by 2^b, but continue on to the next iteration without deleting any edges from E.

Checking whether a set of edges spans all vertices can be done in linear time using DFS/BFS; and almost linear time using DSU which.
We run this linear method once for each bit, which is good enough.
At the end of it all, ans is the answer we're looking for.

Now we need to deal with updates.
Of course, running the above algorithm afresh for each new edge would be way too slow.

Instead, let's analyze how a new edge $(u, v, 0)$ would change the answer.
For each bit b from 29 down to 0:

- If b isn't set in ans, then we don't need to use the new edge anyway.
- If b *is* set, then we'd like to check if it can improve our answer.
  - Recall that b is set in the answer only when we couldn't connect the graph without using an edge where b was set; i.e, the edges present divided the graph into > 1 connected components.
  - If there are ≥ 3 connected components, a single edge can't help us anyway so there's no change.
  - If there are exactly two connected components, and u and v are in different components, then this edge does help us reduce the total weight!
    Clearly, when we can do this it's optimal to do so.

This gives us an algorithm:

- Find the *highest* bit b such that the graph at this step of our initial algorithm had exactly two connected components, and u and v are in different components.

- At this stage, we're definitely using the edge $(u, v, 0)$ to connect these components. So, add it to the current edges (recall that we know which edges have been discarded already) and recompute the OR-MST.

Of course, running the entire OR-MST algorithm for each query is still too slow.
However, here is where we'll use the fact that queries are independent.

Notice that if we fix the bit b for which we're using the edge $(u, v, 0)$, the answer is in fact independent of what the actual values of u and v are.
The only thing that matters is that our initial OR-MST algorithm gave us exactly two components, and we compute the OR-MSTs of the union of these two components, along with some 0-weight edge joining them.

So, we only need to run the OR-MST algorithm (at most) once for each bit, making 31 runs in total including the initial one.
The results of these runs can be cached so that each query can be quickly answered later.

Putting everything together, our final solution is as follows:

- Run the OR-MST algorithm described initially.
- For each bit such that the resulting graph had exactly two components, run the OR-MST algorithm on the union of these two components (along with some 0-weight edge joining them) as well and store their answers.
- Then, for each query (u, v):
  - Find the highest bit b such that the graph for this bit has two components; and u and v are in different components.
  - The answer for this query is then simply the precomputed answer for this bit.
  - If no such bit exists, the answer is simply ans, the initial OR-MST value.

This requires us to quickly check if two vertices lie in the same component, but we can precompute components for each vertex corresponding to each bit, so this can be answered in $O(1)$ per bit as well.

TIME COMPLEXITY

$O(B^2 \cdot (N + M) + B \cdot Q)$ per test case, where $B = 30$ for this problem.

PREREQUISITES:

Greedy algorithms

PROBLEM:

You're given a large integer A in binary representation as a string of length N.
Find two binary strings B and C such that $A = B - C$, and the sum of counts of ones in B and C is minimized.

EXPLANATION:

For convenience, this explanation will treat A, B, C as if they were reversed and 0-indexed, so that A_i corresponds to whether 2^i is set in A or not and so on.
In terms of implementation, this means you can reverse A, follow the solution below, then reverse B and C before printing them.

The problem essentially asks us to write A using the smallest number of powers of two, except we're allowed to both add and subtract them (addition goes into B, subtraction goes into C).
However, there's one small catch: B and C have to be of length N each, which means we can't use *larger* powers.

Let's first solve this without the restriction on length N — in the end, we can see how it comes into play.

If we weren't allowed subtraction, the best we can do is obviously to just use all the bits set in the binary representation of A. So, let's take this as our starting point and see how to reduce it.

The main power of subtraction is that it allows us to reduce a block of ones into two ones.
That is, we have

$$2^x + 2^{x+1} + 2^{x+2} + \ldots + 2^y = 2^{y+1} - 2^x$$

so we can replace $y - x + 1$ additive powers of two with just two of them.
Of course, if $x = y$, this will replace one power of two with two of them, which is bad so we won't do it.

This gives us a natural greedy solution:

- Let $B = A$ and C be filled with zeros initially.
- Iterate from left to right, i.e from lower bits to higher bits; and find a maximum block of ones in B, say $[L, R]$. In particular, this means $B_{L-1} = B_{R+1} = 0$ must hold.
- If $L = R$, ignore it.
- Otherwise, set $B_{R+1} = 1$ and $C_L = 1$, since we're replacing $2^L + \ldots + 2^R$ by $2^{R+1} - 2^L$.

Note that iterating from lower to higher is important here because it allows us to further replace the bits we add to B.
For example, if $A = 110110$, going from low to high we'd get:

- Step 1: $B = 001110, C = 100000$
- Step 2: $B = 000001, C = 101000$
  whereas if we went high to low we'd get
- Step 1: $B = 110001, C = 000100$
- Step 2: $B = 001001, C = 100100$
  which is not optimal.

This almost solves our problem in $\mathrm{O}(N)$: the only thing that needs to be taken care of is the very end.
That is, notice that when replacing a block of ones, we need to use the next higher position.
However, if this block of ones forms a suffix of the string, the next position doesn't exist, so we can't perform this replacement.

It's not hard to see that if a block of ones forms a suffix like this, then we have no choice: this suffix of ones must be set in B and none of them can be set in C.
So, we can simply delete the suffix of ones, solve normally for the remaining string (now without having to worry about edgecases!) and finally add the ones back in.

This way, we've solved the problem in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Greedy algorithms

PROBLEM:

There's an $N \times M$ grid, with $A_{i,j} = i$.

A mountain is a set of integers $P, K, L_1, L_2, \ldots, L_K$, saying that you pick the first L_i integers from row $(P + i - 1)$.

Answer Q queries of the following form:

- Given S, find a mountain with sum S.

EXPLANATION:

This task can be solved greedily.

Suppose we want a sum of S.
Let's go from the first row to the last, each time taking as many numbers as possible till we first exceed the sum.
As soon as we exceed it, we can throw out (at most) one number to attain the exact sum we want.

That is, initialize a variable $\text{sum} = 0$ and set $P = 1$ since we're starting from the first row.
Then, for each i from 1 to N:

- If $\text{sum} + i \cdot M < S$, take all M elements from this row and continue. In other words, set $L_i = M$.
- Otherwise, let j be the smallest integer such that $\text{sum} + i \cdot j \geq S$. Take these j numbers into the sum, i.e, set $L_i = j$.
- Now, if $\text{sum} = S$ we're done.
- Otherwise, remove one element from the row $(\text{sum} - S)$, i.e, decrement $L_{\text{sum}-S}$ by one. Since we took elements in order from smallest to largest, it's guaranteed that the value of $\text{sum} - S$ is no larger than i, so this is always possible.

TIME COMPLEXITY

$\text{O}(N)$ or $\text{O}(N + M)$ per query.

PROBLEM:

Given N and K, find the sum of digits of S; where S denotes the smallest N-digit number such that it doesn't contain a palindromic substring of length $> K$.

EXPLANATION:

Let's try to greedily construct the smallest number we can that satisfies the palindrome property.

It's best if the first digit is 1, since we can't use 0.
After than, we can start using zeros - the next K digits can all be 0 without creating any palindromes of length $> K$.
However, the next character can't be 0; and it also can't be 1.

So, the best we can do is to place a 2.
Our number currently looks like

$$1\ \underbrace{000\ldots000}_{K\text{ times}}\ 2$$

Now, once again we can start placing zeros.
However, this time we can only place $\lfloor \frac{K-1}{2} \rfloor$ of them - otherwise we'd create a palindrome of the form $000\ldots00200\ldots000$.

So, we now have the string

$$1\ \underbrace{000\ldots000}_{K\text{ times}}\ 2\ \underbrace{00\ldots00}_{\lfloor \frac{K-1}{2} \rfloor\text{ times}}$$

After this, we can't place any more zeros; so we must place a 1.
Then, the pattern repeats — so the final string will look like

$$1\ \underbrace{000\ldots000}_{K\text{ times}}\ 2\ \underbrace{00\ldots00}_{\lfloor \frac{K-1}{2} \rfloor\text{ times}}\ 1\ \underbrace{000\ldots000}_{K\text{ times}}\ 2\ \underbrace{00\ldots00}_{\lfloor \frac{K-1}{2} \rfloor\text{ times}}\ 100\ldots$$

repeated upto length N.

Of course, what we're really interested in is the sum of this string.
The zeros don't contribute to it at all, so we just need to count the number of times 1 and 2 occur.
That's not too hard.
The length of the pattern is $L = 2 + K + \lfloor \frac{K-1}{2} \rfloor$.

- The ones appear at positions $1, L+1, 2L+1, 3L+1, \ldots$
  So, if x ones are to appear in the string upto position N, we'd need $(x-1)\cdot L + 1 \le N$.
  Using this, you can find the largest valid x either with math or just binary search it.
- The twos appear at positions $K+2, K+2+L, K+2+2L, \ldots$
  Similarly, for y twos to appear in the string, we'd need $K+2+L\cdot(y-1) \le N$; so the largest valid y can be found quickly.

Once x and y are known from above, the answer is just $x + 2y$.

PREREQUISITES:

Greedy algorithms, binary search

PROBLEM:

You're given a string S. In one move, you can pick and index i and replace S_i with either $S_i + 1$ (upwards move) or $S_i - 1$ (downwards move), where the values are cyclic from a to z.

You can perform at most P upwards moves and at most Q downwards moves.
What's the lexicographically smallest string you can obtain?

EXPLANATION:

Since our objective is lexicographic minimization, the obvious choice is to try and make the first character a, then make the second character a, and so on.

Let's try to find the largest prefix of S that can be turned into a's.

Suppose we want to check whether the first k characters of S can be turned into a.
First, compute the values up_i and $down_i$ for each $1 \leq i \leq k$, where up_i is the number of upward moves required to turn S_i into a, and $down_i$ is defined similarly.

Now, we want to choose up_i for some indices and $down_i$ for the rest, such that the sum of chosen up_i is at most P, and the sum of chosen $down_i$ for the rest is at most Q.

This can be done greedily!
That is, sort the pairs $(up_i, down_i)$ in ascending order of up_i value. Then, greedily keep taking the smallest up_i values as long as you don't exceed P, after which you take the $down_i$ values for everything else.
Finally, check if the sum of taken $down_i$ values is $\leq Q$.

Proof of correctness

The key to why this works is the fact that up_i and $down_i$ are complementary.
That is, $up_i + down_i = 26$ (unless $S_i = $ a, in which case $up_i = down_i = 0$ and we can just ignore this index).

This means that smaller values of up_i correspond to larger values of $down_i$, and vice versa.
So, if we sort the pairs and have $up_1 \leq up_2 \leq \ldots \leq up_k$, that also means we'll have $down_1 \geq down_2 \geq \ldots \geq down_k$.

Intuitively this already makes it obvious why taking a prefix of the up_i values is good, since it'll make us take a suffix of the $down_i$ values which is the best we can do there too.

It's also possible to prove this algebraically, using an exchange argument.
Suppose we choose up_x, but don't choose up_y where $y < x$ (in sorted order); meaning we chose $down_y$.

Suppose we replaced up_x with $down_x$ and $down_y$ with up_y.
Then, since $up_x \geq down_x$ and $down_y \geq up_y$, this reduces (or rather, doesn't increase) *both* the sum of up_i values and the sum of $down_i$ values, which is obviously good.
Repeating this replacement operation makes us end up with a prefix of up values and a suffix of $down$ values, as claimed.

So, we can now check whether the first k characters can be all turned into a, in $O(k \log k)$ time. It's possible to implement this in $O(k + 26)$ time, but unnecessary to get AC.
Our aim is to find the maximum such k.

Notice that if the first k characters can be turned into a, so can the first $k-1$ characters.
So, we can use binary search!
This allows us to find the largest valid k in $\mathrm{O}(N \log^2 N)$ time.

In addition, note that our method also tells us how many moves are remaining of each type, since it minimizes *both* the number of upward and downward moves.
We need to use these remaining moves to deal with the positions from $k+1$ onwards.

First, we know that S_{k+1} *cannot* be made into a, so the best we can do is to perform downward moves on it as much as possible. This uses up all our downward moves.

Finally, for each index i from $k+2$ to N, check whether S_i can be turned into a using the remaining upward moves; and if it can, do so.

It is also possible to implement this without binary search, by maintaining a set/priority queue of lowest up_i values and removing the largest of them whenever the sum exceeds P.

In fact, it's even possible to implement this solution in $\mathrm{O}(N)$; doing so is left as a challenge to the reader 😄

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

Difficulty: Simple

Pre-requisites: Bruteforce, Greedy

Problem:

Given a rectangular grid with **N** rows and **M** columns. Each its cell is either empty, contains an enemy, or contains a laser. Each laser can shoot in one of three directions: either left, right or up. When a laser shoots at some direction, it kills all the enemies on its way. There're **L** lasers on the grid.

You are to determine whether it's possible to kill all the enemies on the grid.

Explanation:

The key observations are that there're at most 16 lasers on the grid and there're only 3 directions to shoot.

So, let's just choose those lasers, who will shoot in the up direction(let's call them "vertical" lasers, those who are not chosen are "horizontal") There're at most 2^{16} variants, so we can simply iterate through them. After that, we can solve the problem independently for each row. How?

If there're no alive enimies left on a row(after the "vertical" lasers shot), then it's OK. Otherwise, if there're no "horizontal" laser in the row, then the current variant fails. If there're one "horizontal" laser on the row, then we should check if all the enemies lies to the one side of the laser. If there're one "horizontal" laser on the row, then it's OK anyway(we can make the most left one shoot to the right, the most right one shoot to the left - so all the row will be covered).

Total complexity is O($\mathbf{2^L NM}$) per testcase.

Please, check out Setter's and Tester's solutions for your better understanding.

PREREQUISITES:

Finding SCCs, binary search

PROBLEM:

You're given a directed graph with N vertices and M edges.
The *cost* of a subset of vertices is defined to be the number of pairs of vertices in the subset that aren't mutually reachable from each other.

Answer Q queries.
For each query, given K, find the maximum size of a subset whose cost doesn't exceed K.

EXPLANATION:

The cost of a subset S is the number of pairs of vertices in the subset that aren't mutually reachable from each other.
Turning that around, if S has size M and there are x pairs of vertices in S that *are* mutually reachable from each other, then $cost(S) = \binom{M}{2} - x$.

The latter condition is a bit simpler to deal with because it's more familiar.
In particular, we know that in a directed graph, two vertices are mutually reachable from each other *if and only if* they lie in the same **strongly connected component** (henceforth, SCC).
This hints to us that we should look at the SCCs of the given graph.

In particular, notice that we can take *all* the vertices from any single SCC, for zero cost.
However, if we have vertices from different SCCs, the cost will always be positive—intuitively, it seems better to not choose vertices from too many different SCCs, because the cost is just the number of pairs of vertices belonging to different SCCs.

Notice this in fact tells us the answer for $K = 0$: the size of the largest SCC.
Once again, this is a hint that the sizes of the SCCs are also important, not just the SCCs themselves.

So, let's first find all the SCCs of the given graph, and also their sizes.
This can be done in $O(N + M)$ time using some different algorithms: for example, **Kosaraju's algorithm** or **Tarjan's algorithm**.

Suppose there are r SCCs, denoted $C_1, C_2, \ldots, C_r$.
Let their sizes be $s_1, s_2, \ldots, s_r$; and for convenience, let's have $s_i \geq s_{i+1}$.

Our look at $K = 0$ above gives us a natural-sounding greedy algorithm:

- First, take all the vertices belonging to C_1, the largest SCC, for a cost of 0.
- Then, take as many vertices as we can from C_2, as long as the cost doesn't exceed K.
- Then, do the same to $C_3, C_4, \ldots$ in order.

That is, take vertices from SCCs in descending order of size.

It turns out that this greedy is indeed optimal!

Proof

The proof of this greedy relies on a couple of simpler claims.

Consider a solution set S.
We'll call an SCC *full* if all its vertices are in S, and *empty* if none of its vertices are.

Lemma: There exists a solution such that at most one SCC is neither full nor empty.
Proof: This is not hard to see.

Suppose there are two non-full, non-empty SCCs with respect to S.
Say one of them has x_1 vertices chosen from it, and the other has x_2, where $x_1 \leq x_2$.
Choosing $(x_1 - 1, x_2 + 1)$ vertices instead gives us a strictly smaller score for the same number of chosen vertices.
Repeat this process as long as you can; it will terminate after a finite number of steps.
Once the process terminates, we're left with at most one non-full, non-empty SCC, as claimed.

Lemma 2: The full SCCs will form a prefix of the SCC sorted by size (in descending order).
Proof: Once again, this isn't hard to prove.
Suppose an SCC with size s_1 is full and an SCC with size s_2 is not full, where $s_1 < s_2$.
Suppose $x_2 < s_2$ vertices are chosen from the second SCC.
Then, there are some cases:

- If $x_2 = 0$, move all chosen vertices to the second SCC instead.
  This doesn't change the answer, but does ensure that the set of full SCCs is closer to being a prefix.
- If $1 \leq x_2 \leq s_1$, just swap the number of vertices chosen from both SCCs.
  This doesn't change the answer at all; but now there are two non-full SCCs, and the construction from the previous lemma will make the smaller one empty first.
  In particular, once again we're closer to the full SCCs forming a prefix.
- If $x_2 > s_1$, it can be seen that moving one vertex from the first SCC to the second reduces the score, so keep doing this as long as possible.
  EIther the smaller SCC becomes empty, or the bigger one becomes full - either one is good for us.

Finally, we have some vertices from a non-full SCC.
It's obviously optimal to take them all from a single SCC; and it doesn't quite matter which one is chosen here so without loss of generality, we can just take them from the largest remaining one.

This proves the greedy's optimality.

Since there are multiple queries, we need to be able to perform this greedy algorithm quickly.
To do that, notice that our solution has a somewhat predictable structure:

- First, we take all vertices from SCCs $C_1, C_2, \ldots, C_l$ for some l.
- Then, we take some (but not all) vertices from C_{l+1}

Let's compute for each part separately.

First, we need to find the largest l so that the cost of taking the SCCs $C_1, C_2, \ldots, C_l$ doesn't exceed k.
The cost of this choice is the number of pairs of vertices belonging to different SCCs.
Since we know sizes already, this is easily seen to be $\sum_{i=1}^{l} \sum_{j=i+1}^{l} s_i s_j$.
This cost can in fact be precomputed using prefix sums, as follows:

- Let P_i denote the cost of taking the first i SCCs.
- Then, $P_i = P_{i-1} + s_i \cdot (s_1 + s_2 + \ldots + s_{i-1})$.
- So, by maintaining prefix sums of the P and s arrays, all costs can be computed in $\mathrm{O}(r)$ time.

Once costs are precomputed, finding the largest l is easy: just binary search on these costs to find the largest l such that $P_l \leq K$.

Next, we can still take some vertices from C_{l+1} (if it exists).
We've already incurred a cost of P_l so far, so we're left with $K - P_l$ to work with.
Note that each new vertex we take increases the cost by exactly $s_1 + s_2 + \ldots + s_l$.
So, the number of vertices we can take is exactly $\left\lfloor \frac{K - P_l}{s_1 + s_2 + \ldots + s_l} \right\rfloor$.

That's the entire solution!
After some precomputation, each query is answered in $\mathrm{O}(\log N)$ time since we only perform a single binary search.

PREREQUISITES:

Greedy/dynamic programming, careful implementation

PROBLEM:

You're given an array A. Find the minimum number of elements that need to be deleted from it so that the remaining array satisfies

$$A_1 + A_2 = A_3 + A_4 = A_5 + A_6 = \ldots = A_{2k-1} + A_{2k}$$

EXPLANATION:

The final array must be such that it can be partitioned into k adjacent pairs, each having the same sum.
So, let's fix what the final sum is, then try to find the longest array we can form that satisfies the condition.

Suppose we've fixed the sum to be S.
Let pairs_S be a list consisting of all pairs (i,j) such that $A_i + A_j = S$.
Note that pairs_S can be precomputed and stored for *all* S by simply iterating over all pairs of elements in A.

Suppose we choose to keep the pair (i,j) in the final array.
Then, notice that any other pair (i_2,j_2) must satisfy $i_2 > j$ **or** $j_2 < i$.
That is, we must have either $i_2 < j_2 < i < j$ or $i < j < i_2 < j_2$, since we're pairing i and j and hence can't have i_2 or j_2 lie between them.

Thinking of it another way, we can consider the pair (i,j) to be the *interval* $[i,j]$. The above condition then means that any intervals we pick must be disjoint.

Our problem thus transforms to: given a list of intervals, find the maximum number of disjoint intervals that can be picked from them.

This is a classical problem, and can be solved greedily as follows:

- Sort the intervals in increasing order of their **right endpoints**.
  That is, $[l_1, r_1]$ comes before $[l_2, r_2]$ if $r_1 < r_2$ (doesn't really matter how ties are broken).
- Let $R_{\max}$ denote the largest right endpoint we've encountered so far.
  Initially, $R_{\max} = -\infty$
- Then, for each interval $[l, r]$ in this sorted list:
  - If $l \leq R_{\max}$, ignore it
  - Otherwise, $[l, r]$ is disjoint from all the intervals seen so far, so take it into the answer and set $R_{\max} := r$

This greedy strategy is **provably optimal**.

If pairs_S has K intervals in it, this algorithm takes $O(K \log K)$ time, but that's only because we need to sort the intervals since the greedy part is linear.
In the specific case of this problem, we can in fact create the intervals in sorted form, eliminating the need for sorting entirely.

That can be done by the following pseudocode:

```
for j in 1...n:
    for i in 1...j-1:
        pairs[a[i] + a[j]].append((i, j))
```

That is, by iterating over the right endpoint in increasing order and then iterating over the left endpoint.

The time complexity of this approach is $\mathrm{O}\left(\sum_S len(\mathrm{pairs}_S)\right)$, which is just $\mathrm{O}(N^2)$ since each pair (i,j) appears in exactly one sum.

Since the values of $A_i + A_j$ can be as large as $2 \cdot 10^9$, pairs will likely need to be a map of some kind. Depending on the kind of map used, this adds an $\mathrm{O}(\log N)$ or (expected) $\mathrm{O}(1)$ to the time complexity. Hashmaps (`unordered_map` or `gp_hash_table` in C++, `dict` in Python) in particular pass well within the time limit.

There are other solutions to this problem, for example using dynamic programming.
However, you'll need to be reasonably careful when implementing such solutions, since too many map accesses or extra log factors will result in TLE.

TIME COMPLEXITY

$\mathrm{O}(N^2 \log N)$ or expected $\mathrm{O}(N^2)$ per test case.