# Problem A. Card Game

Within the framework of the solution, we will assume that the additive strength increases are sorted in descending order $A_1 \geq A_2 \geq \cdots \geq A_N$.

In general, the size of the deck $M$ can be greater than $D + N$. In this case, your deck will contain $R = M - D - N$ cards that in no way affect your strength.

In the worst case, all these cards will come to you on the first turn, and therefore you will have to attack the monster using only $H = S - R$ cards. In the future, we will use $H$ to denote the taken strength-increasing cards in the worst case for any scenario.

**Statement 1**. If you draw at least $B = \lceil \log_2 X \rceil$ doubling strength cards from the deck, then you will definitely defeat the monster.

**Statement 2**. If $H \geq B$, then all available doubling strength cards must be added to the deck.

The remaining $N_H = M - D$ positions in the deck must be filled with cards $A_1 \dots A_{N_H}$.

In this case, the worst case will be taking cards $A_{N_H - H} \dots A_{N_H}$ and $D_H = H - N_H$ doubling cards.

The answer `YES` will only be in the case if it is possible to defeat the monster with this set of cards.

**Proof**. Let's denote the total strength gained by the additive cards as $P$. Then the final strength in the described case will be equal to $P \cdot 2^{D_H}$.

Remove one of the doubling strength cards from the deck, and in its place put a card with strength $C = A_{N_H+1}$. In this case, the new strength will be $(P + C) \cdot 2^{D_H - 1}$.

By construction, $C \leq P$, which means $C \cdot 2^{D_H-1} \leq P \cdot 2^{D_H-1}$. It follows that $(P + C) \cdot 2^{D_H-1} \leq (P + P) \cdot 2^{D_H-1} = P \cdot 2^{D_H}$.

**Statement 3**. If $H < B$, then drawing only doubling strength cards at the beginning of the battle is strictly disadvantageous.

In this case, it is necessary to make two nested iterations:

- iterate the number of doubling strength cards in the deck from 0 to $H - 1$;
- iterate the number of doubling strength cards drawn at the beginning of the battle.

Similarly to the scenario with $H \geq B$, the maximum cards are added among the additive cards in the deck, and at the beginning of the battle, the minimum of the added cards is drawn from the deck.

For each case of the outer iteration (the number of doubling strength cards in the deck), we will calculate the worst case among the inner iterations.

If we find at least one outer case where the monster is defeated in the worst case, the answer is `YES`.

**Final asymptotics**:

- For the case $H \geq B$, $O(1)$ checks are performed.
- For the case $H \leq B$, it is necessary to perform $O(\log^2 X)$ checks.

Each check is performed in $O(1)$ using precomputed prefix sums on the array $A$.

# Problem B. Symmetric Race

First of all, let's notice that the "blue" and "red" transition graphs are so-called functional graphs, that is, graphs in which transitions are defined by some function. Specifically, each vertex has exactly one outgoing edge. On such graphs, it is quite easy to calculate the distances from each vertex to a certain designated one: if the edge from vertex $u$ leads to vertex $v$, and it is necessary to reach vertex $t$, then

$$d_u = \begin{cases} 0 & \text{if } u = t \\ 1 + d_v & \text{otherwise} \end{cases}$$

Let's calculate the distances from all vertices of the blue and red graphs to the end points using this formula, for example, lazily with dynamic programming, in $\mathcal{O}(n)$ time. Then we will move on to the second phase of the solution.

Let $b_v$ be the distance from $v$ to $B$ in the blue graph, and $r_v$ be the distance from $v$ to $R$ in the red graph, and we consider some $v_1$, $v_2$, $v_3$ from 1 to $n$, for which both corresponding distances are defined. Then the condition for the time of the run is satisfied in one of the following cases:

- $b_{v_i} = r_{v_i}$ for all $i$ from 1 to 3;
- $b_{v_1} = r_{v_1}$ only for the first, and $b_{v_2} = r_{v_3}$ and $b_{v_3} = r_{v_2}$ (up to permutation of vertices);
- $b_{v_1} = r_{v_2}$, $b_{v_2} = r_{v_3}$, $b_{v_3} = r_{v_1}$ (up to permutation of vertices).

Let's calculate the triples of the first kind, simply by remembering the number of vertices with $b_v = r_v$. Let it be $c_1$, then we need to add $\frac{c_1(c_1-1)(c_1-2)}{6}$ to the answer.

For triples of the second and third kinds, we will build a graph in which vertices correspond to **distance values** from 0 to $n$. For each $v$, we will draw an edge from vertex $b_v$ to vertex $r_v$. Then the number of triples of the second kind is equal to $c_1$ multiplied by the number of "paired" edges $x \to y$ and $y \to x$. Such edges can be counted using `unordered_map`.

And finally, triples of the third type are cycles of length 3 in such a graph. There is a standard algorithm for finding such cycles, which iterates through all triples in the graph in $\mathcal{O}(n\sqrt{n})$ time, and it remains only to count the number of multiple edges in each of its directions for each triangle and multiply them.

# Problem C. Array Partitioning

Count the number of prime divisors for each element of the array $a_i$. For this, create a prime divisor counter, iterate through all values $k$ from 2 to $\sqrt{a_i}$ inclusive. If $a_i \vdots k$, divide $a_i$ by $k$ and increase the counter while $a_i$ is divisible by $k$. Next, note that we can color numbers with an even number of prime divisors in color 1, and those with an odd number in color 2. Thus, if numbers differ from each other by a factor of $p$, where $p$ is prime, then the number of prime divisors of such numbers will differ by 1, thus they will have different parity, and this coloring will be correct. The final asymptotic complexity will be $O(n\sqrt{\max(a_1, a_2, \ldots, a_n)})$.

# Problem D. Ultra mex

Let's consider how our process works. Consider the binary trie of the binary representations of all numbers in the set (each number is considered from the most significant bit to the least significant). The depth of such a trie is $k$.

Note that if our trie is complete, that is, our set consists of all numbers and has size $2^k$, then the mex limit is also equal to $2^k$.

Let our trie be incomplete. Consider the left and right subtrees of the root of our trie.

- If the left subtree is not completely filled, then $mex = m$ will be $< 2^{k-1}$, hence we will perform the operation $\oplus(m-1)$. After such an operation, all elements that were in the left subtree of the trie will remain there, and all that were in the right subtree will also remain there. At the same time, at least one missing element will remain in the left subtree. Thus, in the right subtree of our trie, there can be any subset, we can only consider the left subtree and remove the most significant bit from all numbers; the mex limit will not change.
- Let the left subtree of the trie be completely filled. Consider the right subtree. Suppose the minimum element $2^{k-1}$ of the right subtree is in the set. Then $mex = m > 2^{k-1}$, hence we will perform the operation $\oplus(m-1)$ with the number $m-1 \geq 2^{k-1}$. Then note that after the operation, the subtrees will swap places, and we will get the first case. But then we will swap the subtrees before performing

the operation. Note that such a trie will satisfy the first case, and the mex limit of the set will not change after the swap.

- Let the left subtree of the trie be completely filled. Consider the right subtree. Suppose that the minimum element $2^{k-1}$ is not in the set. Then $mex = 2^{k-1}$ and we perform the operation $\oplus(2^{k-1}-1)$. Such an operation does not swap the subtrees of the trie, but flips them. Note that if the original set did not contain the number $2^k - 1$, then now the left subtree is filled again, while the right does not contain $2^{k-1}$. The process has cycled, and now all resulting $mex$ will be equal to $2^{k-1}$. We have obtained that the mex limit of our set is equal to $2^{k-1}$.
- The last case remains, when the left subtree of the trie is completely filled, the right subtree does not contain the number $2^{k-1}$, but contains the number $2^k - 1$. In this case, we will first perform the operation $\oplus(2^{k-1} - 1)$, flipping the subtrees, then we will get the same situation as in case 2. That is, the mex limit of the set is the same as the mex limit of the flipped right subtree.

Thus, in cases 1, 2, and 4, we obtain a trie of depth $k - 1$ with the same mex limit, in case 3, we obtain that the mex limit of the set is equal to $2^{k-1}$. Thus, the mex limit can only be a power of 2, and if $p$ is not a power of 2, then the answer is 0. From now on, we will assume that $p = 2^q$.

Fix $q$. We will write dynamic programming $dp[k][n][last]$, where $1 \le n \le 2^k$ and $last \in \{0, 1\}$. Here:

- $dp[k][n][0]$ is equal to the number of subsets of numbers from 0 to $2^k - 1$ containing 0, such that their size is $n$ and the mex limit is equal to $q$.
- $dp[k][n][1]$ is equal to the number of subsets of numbers from 0 to $2^k - 2$ containing 0, such that their size is $n$ and the mex limit is equal to $q$. In this case, the element $2^k - 1$ must not be in the subset.

Initialization (corresponds to case 3):

- $dp[q][2^q][0] = 1$, all other values with $k \le q$ are equal to 0
- $dp[q+1][2^q+n][0] = dp[q+1][2^q+n][1] = \binom{\max(2^q-2,0)}{n}$, all other values for $k > q+1$ are equal to 0

Computation formulas (for $k \ge q + 2$):

- $dp[k][n][0] += \sum_{r=0}^{2^{k-1}} dp[k-1][n-r][0] \cdot \binom{2^{k-1}}{r}$; $dp[k][n][1] += \sum_{r=0}^{2^{k-1}-1} dp[k-1][n-r][0] \cdot \binom{2^{k-1}-1}{r}$
  These transitions correspond to case 1, when the left subtree is not complete.
- $dp[k][2^{k-1}+n][0] += dp[k-1][n][0] + dp[k-1][n][1]$ (for $n \ge 1$)
  The first term corresponds to case 2, the second term corresponds to case 4.
- $dp[k][2^{k-1}+n][1] += dp[k-1][n][1]$ (for $n \ge 1$)
  The only term corresponds to case 2, case 4 with $last = 1$ is impossible since by definition the element $2^k - 1$ must not be in the set when $last = 1$.

The answer to the problem for the triplet $(k, n, 2^q)$ will be in the value $dp[k][n][0]$.

Thus, directly using the formulas presented above, we can compute all answers for a fixed $q$ in time $O(2^{2k})$. That is, the overall asymptotic is $O(k4^k)$.

The problem can be solved more efficiently for a single triplet. To do this, we will fix the path from the subtree of size $2^q$ to the root of the trie. There are $2^{k-q}$ of them. Note that all subtrees hanging on the left path must be complete. All subtrees hanging on the right path can contain arbitrary subsets, except for whether the elements $2^{i-1}$, $2^i - 1$ are included. Then we can distribute the remaining elements that

did not fall into the left subtrees among the right subtrees hanging on the path, so this value is equal to one binomial coefficient. This solution works in $O(k2^{k-q})$ for one test.

For a complete solution, we can speed up the computation of the initial dynamic programming using fast Fourier transform. We can note that the first type transitions during computation work in $O(2^{2k})$ if computed naively. Note that what needs to be computed is the product of two polynomials $P_i = dp[k-1][i][0]$ and $Q_i = \binom{2^{k-1}}{i}$. The product can be computed in $O(k2^k)$. Then the total time for a fixed $q$ will be $\sum_{i=q+2}^{k} i2^i = O(k2^k)$, so the overall asymptotic is $O(k^2 2^k)$.

# Problem E. Airport Codes

In this problem, we have a string, and we are given some queries: compare lexicographically minimum subsequences of given lengths for two given subsegments. Let's discuss how the lexicographically minimum subsequence looks like.

Let $t_1 t_2 \dots t_k$ be the lexmin subsequence of string $s$ of length $k$. If there exists a position $p$, such that $t_p > t_{p+1}$, let $p$ be the minimum such position. Note that if we remove $t_p$ we get a smaller string, but we cannot add one more character to the right, as otherwise we would have a smaller subsequence. Therefore, characters starting with $t_{p+1}$ form a suffix of our string.

So, lexmin subsequence of any length looks as follows: first, some characters are sorted, and next we have a suffix of our string. We can represent this by counting the number of occurrences of each character in the sorted part of the subsequence.

Now, we want to find such representation for a given subsegment and length, and compare two representations.

To find a representation, we do the following. We consider the characters in increasing order, (let the current character be $c$) and try to add as many letters $c$ as possible, so that it is possible to achieve a subsequence of a desired length. It can be done with a binary search on the precomputed array of occurrence positions. Once we added as many characters $c$ as possible, we might remove some of those, and stop. To do that, we consider the starting suffixes when we remove some of the letters $c$. Those suffix-starts form a range of our substring, and we need to check if there exists a smaller character. If it exists, we find the leftmost such character, and end with it. To do so, we precompute for each position and each character the next character, smaller than it.

So, we computed the representation in $O(\log n \cdot A)$, where $A = 26$ is the size of the alphabet.

Now we need to compare two such representations. First of all, while the characters in sorted parts in both representations exist, we take the minimum of those, check that they are equal, and subtract minimum count from both of them.

Now, one subsegment has suffix, and one subsegment has sorted part. We consider the left characters from its sorted part, and check that another subsegment has the same letters in its positions. To do so, we precompute the next occurrence of a different character for each position.

In the end, both subsegments consist of two suffixes, and we need to compare them. To do that, we compute the LCP (longest common prefix) by using suffix array or binary search with hashes.

In total, the solution works in $O(n(A + \log n) + q \log n \cdot A)$, if implemented with suffix array.

# Problem F. Bacteria

Notice that the function "number of bacteria over time" is non-decreasing. We will perform a binary search on the answer, setting the left boundary to 0 and the right boundary to $R = 2 \cdot 10^9 + 40$. The left boundary will correspond to the time when we accumulate less than $m$, and the right boundary will correspond to the time when we accumulate at least $m$.

To move the boundaries, we will count the total contribution from all bacteria. If it is less than $m$, we will move the left boundary; otherwise, we will move the right. The calculation of the contribution is done as

follows.

- If $t < a_i$, then the bacterium will not have time to enter the Petri dish;
- If $a_i \le t < a_i + t_i$, the bacterium will remain frozen and contribute 1;
- If $a_i + t_i - t > 40$, then the bacterium will produce more than $2^{40}$ bacteria, but $m \le 10^9 < 2^{40}$;
- Otherwise, the bacterium will contribute $2^{a_i+t_i-t+1}$.

Asymptotic complexity of the solution is $O(n \log R)$.

# Problem G. Yet Another Problem about a Grasshopper

Let's understand how the optimal route of the grasshopper from $l$ to $r$ is structured. If $r - l \le k$, then the grasshopper jumps to the right end in one move. Otherwise, consider the segment $[l+1; l+k]$. Let $p$ be the position of the maximum on this segment. Then we notice that the answer is not greater than $a_p$, since we must visit at least one column on this segment. Therefore, it is optimal to jump from $l$ to $p$ and continue the path to column $r$.

First, let's compute all maximum values on segments of length $k$. For this, we can use either a maximum queue or a "std::multiset". Let $b_i$ denote the maximum value on the segment $[i; i+k-1]$, or $10^9$ if $i + k - 1 > n$.

Then we notice that the answer for the segment $[l, r]$ is equal to $\min(\min\limits_{i=l}^{r} b_i, a_l, a_r)$. Indeed, the answer is not greater than $a_l$ and not greater than $a_r$, since we must visit these columns, and for any $l \le i \le r$, the answer is not greater than $b_i$, since either this segment fits entirely within ours, in which case we visit at least one cell on it, or it exceeds the boundary, but then the maximum value on it is not less than the value at the boundary of the segment. On the other hand, a greedy algorithm visits only the maximums on segments of length $k$ or the edges of the segment, so the answer is exactly that.

Using this observation, we will compute the answer as follows. For each $i$, we will create two events: the event "activating $a_i$" and the event "activating $b_i$". Now we will process these events in descending order of their corresponding values.

After activating the next value, some segments $[l, r]$ become active: at the moment of activation of the last of the values $a_l, a_r, b_i (l \le i \le r)$. At this moment, the answer for the segment is equal to the current event value, and we need to update the answer by the number of such segments multiplied by the current value.

Let's consider the activation event of the value $b_i$ (events with the activation of $a_i$ are considered similarly). Let's find the nearest inactive value $b_L$ to the left and the nearest inactive value $b_R$ to the right. For this, we can maintain the current inactive positions in a "std::set". Now we are interested in segments that start in $[L+1; i]$ and end in $[i; R-1]$, for both boundaries of which the values of the array $a$ are already active. To find the number of suitable values of $a$ on the segments, we can maintain a Fenwick tree for the sum, or maintain a search tree with the ability to find the count of elements less than a given value. For example, the "tree" from "gnu pbds" will work, this structure is available in the "g++" compiler.

Asymptotics: $\mathcal{O}(n \log n)$.

# Problem H. Good arrays

Let the number of good arrays where the first number is $x$ be denoted as $f(x)$. The answer to the problem will be $f(1) + f(2) + \ldots + f(c)$. We will learn to find $f(p^k)$. First, there will be $c_k$ numbers of $p^k$, then $c_{k-1}$ numbers of $p^{k-1}$, ..., $c_0$ numbers of $p^0$. $p^k$ must appear at least once, while the others can appear zero times. Then $f(p^k)$ is the number of ways to partition the number $n$ into 1 positive and $k$ non-negative summands, which is $\binom{n+k-1}{k}$. We have $f(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) = f(p_1^{\alpha_1}) f(p_2^{\alpha_2} \ldots p_k^{\alpha_k})$. Thus, the array $a_1, a_2, \ldots, a_n$ corresponds to a pair of arrays $b_1, b_2, \ldots, b_n$ and $c_1, c_2, \ldots, c_n$, where

$a_i = b_i c_i$ and $b_i$ is $p_1$ raised to some power, and $c_i$ is the product of $p_2, \ldots, p_k$ raised to some powers. By applying this statement several times, we obtain $f(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) = f(p_1^{\alpha_1}) f(p_2^{\alpha_2}) \ldots f(p_k^{\alpha_k})$. Using the Sieve of Eratosthenes, we will find the smallest divisor for each number. Then, to find $f(x)$, we need to factor $x$ into its prime factors in $O(\log c)$, compute the answers for them, and multiply them together. To compute $\binom{n}{k}$, we can precompute the factorials and their inverses. The overall time complexity is $O(c \log c)$.

We can compute all $f(i)$ using the Sieve of Eratosthenes in $O(c)$. For each number $i$, we calculate its smallest divisor $p$ and the power of its occurrence in the factorization $d$. Then $f(i) = f(p^d) \cdot f(\frac{i}{p^d}) = \binom{n+d-1}{d} \cdot f(\frac{i}{p^d})$.

We do not need to compute all factorials and their inverses. It is sufficient to precompute all $\binom{n+d-1}{d}$ (each in $O(d)$) since $d$ can take at most $O(\log C)$ different values. The eventual time complexity is $O(c)$.

# Problem I. Plane stretching

Let's denote the convex hull of the original set of points as $A$. It is claimed that the distance between the two most distant points in this set is the maximum distance from the origin to a point lying in the set $B$ — the Minkowski sum of $A$ and $-A$. Note that $B$ also stretches by a factor of $\alpha$ when $A$ is stretched by $\alpha$. Thus, the problem reduces to the following: given $O(n)$ pairs of points $(x, y)$, we need to answer the query $\max_{(x,y)\in B} \sqrt{x^2 \cdot \alpha^2 + y^2} = \sqrt{\max_{(x,y)\in B} (x^2 \cdot \alpha^2 + y^2)}$. Thus, we reduce the problem to finding the maximum at a point among a set of linear functions, which can be solved using the convex hull trick in $O(q \cdot \log(n))$.

In total: $O((n+q) \cdot \log(n+q))$, for finding convex hull, Minkowski sum and Convex Hull Trick.

# Problem J. Tree Cutting

First, let's note that if a tree can be cut, it is always beneficial to do so, as it cannot prevent the cutting of other trees. Thus, we arrive at the following solution: while there is a tree on the segment that can currently be cut, we cut it and continue the algorithm.

Let's first try to recalculate the answer when adding one tree to the current segment. Suppose we have some segment $l..r$, and we have cut all the trees we could. Now we allow cutting the tree $r+1$. If this tree cannot be cut, the answer remains the same. If it can be cut, we fell it and then greedily fell all that we can. It is clear that we only need to check the rightmost of the remaining trees, so this can be done quickly by maintaining a stack of uncut trees; it is easy to see that the average amortized complexity of adding one tree to the right is $O(1)$, as each tree will be added and removed at most once. This idea can be used to construct even faster solutions.

To solve the problem in $O(\log n)$ per query, we will use the following idea.

Consider a certain tree. If it can be felled, it can be felled either to the left or to the right. Suppose it can be felled to the left. Then the trees to its right do not prevent this. Additionally, adding new trees to the left will also not prevent this. Thus, there is some number $a_i$ such that tree $i$ can be felled to the left if and only if the left boundary of the segment $l \le a_i$. Similarly, there is some number $b_i$ such that tree $i$ can be felled to the right if and only if the right boundary of the segment $r \ge b_i$.

Assuming we have computed the numbers $a_i$ and $b_i$. Then the problem reduces to the following: for each query $(l, r)$, we need to find the number of trees for which $l \le a_i$ and $r \ge i$ or $l \le i$ and $r \ge b_i$. This can be done using standard techniques with a sweeping line and a segment tree.

We still need to learn how to compute the numbers $a_i$ and $b_i$. Let's show how to compute the numbers $a_i$, the numbers $b_i$ are computed similarly. We will compute the numbers $a_i$ using binary search.

At any moment, we will have some half-interval $[L..R)$ and a set of trees, and we know that for all trees in this set, the number $a_i$ lies within the half-interval $[L..R)$. We will choose a value $M$, the average of $L$ and $R$. We will run the algorithm with a stack, starting from position $M$. If the tree cannot be felled to the left, then for it $a_i < M$, if it can be felled, then $a_i \ge M$. We will assign such trees to sets $A$ and $B$, respectively.

Consider the trees from set $A$. For them, we need to run a binary search on the half-interval $[L..M)$, noting that trees from set $B$ for any value from this half-interval will definitely fall, so they will not prevent the trees from set $A$. Thus, in this recursive call, we can assume that there are no trees from set $B$.

Similarly, for trees from set $B$, we need to run a binary search on the half-interval $[M..R)$, while trees from set $A$ cannot prevent them from falling, as they did not prevent them from falling for the value $M$. Thus, in this recursive call, we can assume that there are no trees from set $A$.

In total, we have obtained a recursive algorithm that divides the range of values in half each time and the trees into two non-overlapping sets. The time complexity of such an algorithm is $O(n \log n)$, as the recursion depth is $\log n$, and the total number of elements at each layer of recursion is $n$.

The final time complexity of the algorithm is $O((n+q)\log n)$.

# Problem K. Skyscrapers

Cases where $K \leq 2$ are solved by filling the grid from left to right, top to bottom, in ascending order of skyscraper heights. In this case, for any skyscraper, there will be no taller skyscrapers to the left and above it.

For $K = 4$, the answer is `YES` only if all skyscrapers have the same height.

For the case $K = 3$, we will arrange the skyscrapers in blocks in descending order of height. Let's say we want to place all skyscrapers with height $h_i$, and all taller skyscrapers have already been placed. Then, after adding the skyscrapers with height $h_i$, all the placed skyscrapers should form a solid rectangle.

Indeed, suppose we have already placed all skyscrapers with height at least $X$ and they **do not** form a solid rectangle. Since the entire grid is rectangular, the bounding box in the final answer will be completely filled. Since the subsequent filling will use only skyscrapers with strictly less height than $X$, at least one of these skyscrapers will have skyscrapers from the current figure on two or more sides in the answer.

Therefore, each new type of skyscraper should complement the already placed subrectangle. Suppose we have a formed rectangle of size $W \times H$ and we need to add $S$ skyscrapers of the next lower height. To do this, we will consider all rectangles that can be formed from $W \times H + S$ skyscrapers we have. Such rectangles with sides $A \times B$ must satisfy the conditions:

1. $A \times B = W \times H + S$;
2. $A \geq W$;
3. $B \geq H$.

Since not every rectangle $A \times B$ can be formed from the set of skyscrapers, we will maintain a list of achievable rectangles and update it when adding new skyscrapers.

We will iterate through the conjugate divisors $A \times B$ of the total number of skyscrapers and check that at least one of the old rectangles $C \times D$ is a subrectangle of $A \times B$.

To do this, we will store the previous layer of rectangles sorted by $C$, and maintain prefix minimums among the sides $D$ on the corresponding prefix.

The overall complexity of the solution is $O(M \cdot N)$ — for each of the $M$ types of skyscrapers, we iterate through $O(\sqrt{N^2})$ divisors. For each pair of divisors, we find the corresponding prefix on the previous layer using two pointers.

# Problem L. Table Game

Let us iterate over the mask of remaining rows of the matrix; there are $2^n$ of them. Now, for every column $c$, we know the sum of it, which is $f_c$. Computing this takes $O(2^n \cdot nm)$ time.

Now, we want to check whether some columns can be selected to achieve the desired sum. In other words, we want to find some subset of array $f$, which sums to a given number $s$. This is a classical knapsack

problem, which can be solved in $O(2^{\frac{m}{2}} \cdot m)$ using the meet-in-the-middle approach. To do so, we split columns into two equal parts; in each part, we calculate the sum of each column-mask. Then we iterate over the chosen mask in the left part and search for the remaining sum in the right part.

The total complexity is $O(2^{n+\frac{m}{2}} \cdot m)$.

# Problem M. Game on a Tree

Note that the first player must place their piece exactly at the center of the diameter on their first move. Otherwise, the second player can place their piece in an adjacent vertex towards the center, and then the first player will lose. Thus, we know the first move of the first player, and we can assume that the diameter has an even length.

We will iterate over the vertex where the second player will place their piece. Now the game process can be simulated quite simply. On their turn, the first player can choose not to move towards the second player, in which case we can independently calculate the maximum distance each player can travel. The first player benefits from doing this if they win, meaning their maximum distance is strictly greater than the maximum distance of the second player. Otherwise, the first player has no choice but to move towards the second player. Similarly for the second player: they can either move away from the first player if it results in a win, or they will move towards the first player.

Using this idea, a solution can be written in $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ (depending on the efficiency of calculating the distances described above) — iterate over the first move of the second player and simulate the game process as described above. Let's solve it faster.

Let the distance from the starting vertex of the first player to the starting vertex of the second player be denoted as $d$. We will number the vertices on the path from the starting vertex of the first player to the starting vertex of the second player from 0 to $d$. Let $a_i$ denote the maximum distance that can be traveled if the path is deviated at vertex $i$.

Then the first player will deviate from the path on the $i$-th move (the first move of the piece corresponds to turn 0) if:

$$a_i > \max_{j=i+1}^{d-i}((d-i)-j) + a_j$$

The second player will deviate from the path on the $i$-th move if:

$$a_{d-i} > \max_{j=i+1}^{d-i-1}(j-(i+1)) + a_j$$

Let $b_i = a_i + i$ and $c_i = a_i - i$. Then the first condition can be transformed as:

$$b_i > \max_{j=i+1}^{d-i} c_j + d$$

And the second as:

$$c_{d-i} + d \geq \max_{j=i+1}^{d-i-1} b_j$$

We will traverse the tree from the center using DFS, assuming that the second player initially places their piece at the vertex where the DFS is currently located. Thus, the sequence $a$ can be maintained quite easily. When transitioning to a child in the DFS, the last element in the sequence $a$ changes, and one more element is added to the end.

From this, we can achieve a solution in $O(n^2 \cdot \log(n))$. We will store $b$ and $c$ in a data structure that allows us to change the value of an element and find the maximum over a segment (for example, a segment tree). Then for each starting vertex of the second player, we can simulate the game in $O(n \cdot \log(n))$.

Instead of simulation, we can find the minimum $i$ ($i \leq \frac{d}{2}$) for which the condition for the first player's win holds. And the minimum $j$ ($j < \frac{d}{2}$) such that the condition for the second player's win holds at $d-j$. After that, the player with the smaller found number wins. Note that one or both numbers may not exist, but these cases are also handled.

Note that $\sum_{i=0}^{d} a_i \leq n$. Also, note that if $a_i < (\frac{d}{2} - i) \cdot 2$, then the first player will definitely not win if they deviate at vertex $i$, because the second player can simply continue along the path to win. The same applies to the second player. We will use root decomposition. We will remember the indices of positions where $a_i > Q$. Then for the first player, we need to check these positions, as well as the positions $\frac{d}{2} - \frac{Q}{2} \leq i \leq \frac{d}{2}$. Similarly for the second player. Thus, the check will work in $O((\frac{n}{Q} + Q) \cdot \log(n))$. If we choose $Q = \sqrt{n}$, the total running time will be $O(n \cdot \sqrt{n} \cdot \log(n))$, but the solution has a very good constant factor.