# A. AND-OR closure

**Author:** Pavle Martinović
**Solved by:** 5/28
**First to solve:** *KNU_0_GB_RAM*

We can generate the AND-OR closure $B$ by setting $B$ to be $A$ initially taking any every two elements $x$, $y$ of $B$ and add $x$ `AND` $y$ and $x$ `OR` $y$ to $B$ if they aren't there already. This process will terminate since we can't ever get a number greater than $2^{40}$. However, this simulation may be slow, so we need a better description of what's going on.

First let's forget about all bits such that every value of $A$ has the same value for this bit (either 0 or 1) because it will remain the same after applying an arbitrary number of `OR` s and `AND` s. Now we can create equivalence classes between the bits, where two bits are equivalent if they're equal in every element of $A$ (but not always equal). They will remain the same after applying an arbitrary number of `OR` s and `AND` s, so we can just view them as one bit, i.e let's choose one bit from every equivalence class and forget all the other ones. Now we create a DAG on these bits that are still active, where there is an edge $i \to j$ if and only if in every element of $A$ in which $i$ is set to 1 then so is $j$. We claim that the number of elements is $B$ is equal to the number of sets $C$ in this DAG with the property $i \in C$ and $i \to j \implies j \in C$. For some bit $i$ by taking the `AND` of all the elements of $A$ where the set $i$ is set to 1, we get a number whose 1 bits are precisely those $j$ for which $i \to j$. By taking the `OR` of some of these we can get any set $C$ described above (taking the those $i$'s which are maximal in the sense that there is no $j \to i$ in $C$). On the other hand, taking `OR` s and `AND` s for these numbers corresponding to some $C$'s is equivalent to taking the union or intersection of two sets with the above property, and it's easily verified that they'll still have the above property.

Now we are in interested in finding the number of sets $C$ with the above property, which is equivalent to finding the number of anti-chains in our DAG (poset), by taking the maximal elements. This can be done by meet-in-the-middle, for the first $\frac{K}{2}$ bits we will generate all antichains among them and find which ones of the other $\frac{K}{2}$ are still "free". Then we apply sum over subsets DP to find for each subset of the other $\frac{K}{2}$ how many antichains of the first half they can be paired with. Finally, generate the antichains of the second half, and merge via the calculated SOS DP. Thus, we have solved the problem in $O(n \log^2(\max A_i) + \sqrt{\max A_i} \log(\max A_i))$.

# B. Build Permutation

**Author:** Pavle Martinović
**Solved by:** 113/121
**First to solve:** *HunTR*

First we shall sort the array $a$ (with maintaining the mapping to the original indices by, for example, sorting pairs of type (value, index)). Now if we solve it for this sorted array, we can easily reconstruct the respective permutation for the original array.

Notice that if $s = a[i] + a[\pi_i]$, then $s - a[i] = a[\pi_i]$ is sorted decreasingly, so the only valid choice of permutation $\pi$ for a sorted $a$, would have to be $\pi = (n, n-1, \ldots, 1)$. So we just check whether the sum $a[i] + a[n+1-i]$ is constant and if it is, we have found the solution, and if not, then there is no solution.

# C. Christmas Sky

**Author:** Lucian Bicsi
**Solved by:** 5/11
**First to solve:** [UoNovi Sad] Infinity

Let's imagine binary searching for the answer $d$. Let's try to achieve the answer $d$, by analyzing the constraints that the problem impose on the translation vector $t$.

By simply looking at two different points $p$ and $q$, where $p$ is from the first set and $q$ is from the second, we can notice that we need to satisfy $||p+t-q|| \leq d$, where $||\cdot||$ denotes euclidean norm. Geometrically, the feasible set looks like a disk with center $q-p$ and radius $d$. Iterating through all of the $nm$ constraints, we find that $d$ is feasible if and only if all $nm$ disks of radius $d$ have a non-empty intersection.

However, an equivalent condition is that the point $t$ is at distance at most $d$ from all $nm$ points of form $q-p$. Otherwise speaking, there is a circle of radius $d$ that encloses all $nm$ points.

Of course, at this point it should become apparent that the minimum such $d$ is the radius of the **minimum enclosing circle** of the set of points $q-p$, which can be computed in expected linear time by Welzl's randomized algorithm.

The total complexity is expected $O(nm)$. It can be improved to $O(n \log n + m \log m)$ by first reducing the point sets to their convex hull and then computing their Minkowski sum in linear time, but this was not required for solving the problem.

# D. Distinct Game

**Author:** Roman Bilyi
Denis Banu
**Solved by:** 0/9
**First to solve:** *N/A*

Taking another look at the problem, the first player wins when they obtain two equal values.

First, assume that there are no consecutive equal values.

If the last values of both arrays are equal, the first player will take one of them, and the second player should take the other. As long as the last values are equal, we can continue removing them.

Now, if the last values are not equal, let's denote them as $x$ and $y$. Let's consider another occurrence of $x$. If it's at an odd position (0-based indexing) in either of the arrays, the first player can take it now and ensure they take another occurrence of $x$. The same applies to $y$. So, let's examine the scenario where both $x$ and $y$ appear at even positions. Suppose the first player takes $x$. If the second player takes $y$, the first player can then force the second player to take another occurrence of $y$. Thus, the second player should take from the same array as the first player.

Now, the last values are not equal, denote them $x$ and $y$. Let's look at another occurrence of $x$. If it's on an odd position(0-based indexing) in any of the arrays, the first player can take it now and then ensure taking another occurrence of $x$. Same for $y$. So let's look at the case when both $x$ and $y$ appear on even positions. Suppose the first player took $x$, if the second player take $y$, the first player can now force the second player to take another occurrence of $y$. So the second player should take from the same array as the first player.

Let's examine any other game state in the middle of the game before the first player's turn. In cases other than the two described previously, it's possible that another occurrence of $x$ or $y$ has already been taken by a player. If the first player took it, they could win. If the second player took it, the second player cannot take another one and should also make a move to the same array as the first player.

Let's look at any other game state in the middle of the game before the turn of the first player. Rather than the two cases described before, it's possible that another occurrence of $x$ or $y$ was already taken by a player. If it was taken by the first player, they could win. If it was taken by the second player, the second player can't take another one and should also make a move to the same array as the first player.

So, we have now determined the strategy for the second player:

- Take from the different array as the first player if the last elements are equal.
- Take from the same array if the last values are different.

With the second player's strategy now fixed, we can consider the game as a one-player game, always in a state before the first player's turn.

Let's assume both arrays have even lengths after removing equal pairs of the last elements. Since another occurrence of the value $x$ (the last element of the first array) is at an even position, the first player can take from the first array while it's not empty, and the last elements will not be equal at any moment of the game. Therefore, the first player can force the second player to take all elements at even positions. On the other hand, the second player can always take all elements at even positions, so the second player would win only if all elements at even positions are distinct. Let's refer to the two arrays as even-distinct if all elements at even positions are distinct.

What if both arrays have odd lengths? It could be shown that the second player wins only if the following conditions are met:

- The arrays have reversed odd-length suffixes:

  $a_1, a_2, \ldots, a_{2k}, 1, 2, \ldots, 2x+1$

  $b_1, b_2, \ldots, b_{2m}, 2x+1, 2x, \ldots, 1$
- The prefixes $a$ and $b$ are even-distinct.

The proof is omitted, as it's much easier from the contestant's point of view to find by brute force.

So, the solution can be divided into three phases:

1. Remove all equal last elements.
2. If both arrays have odd lengths, find odd-length reversed suffixes.
3. Check if the remaining arrays are even-distinct.

For a better understanding of the following solution, let's say that if phase 2 is skipped and phase 1 is non-empty, the last pair of equal elements in phase 1 is phase 2.

Let's return to the problem when consecutive equal elements are possible. Let's call a game simplified if all such pairs are recursively removed. Such removed pairs can create blocks like $[1, 2, 3, 3, 2, 4, 4, 1]$.

If the first player takes the first element of the block, the second player should also take from that block. Otherwise, the first player could get more than half of the elements in the block, which means some elements would be equal.

On the other hand, if the second player takes the first element of the block, the first player is not forced to take elements from that block. Therefore, the first player can decide to remove any number of consecutive equal values. If the first player wins the simplified game, they will also win the normal one.

Now, we can find the strategy for the second player as well:

- If the first player takes any element of the block, the second player should take from the same array.
- Otherwise, follow the strategy of a simplified game, even if they should take from a block instead of an element from a simplified game.

Let's examine the impact of the pairs on the result. Any pair in phase 3 of the game doesn't change anything, as the arrays will still be even-distinct.

The second player can still win if and only if:

- All pairs are in phase 2 of the game.
- The pairs are only after an odd number of elements in the reversed part.
- All blocks contain only a single pair.

Otherwise, the first player can force the second player to take both elements of the same pair.

Now, let's look at some examples. Phase 3 is empty, and the second player wins the simplified game in all these cases.

The second player wins in these scenarios:

- $[1, 2, 3, 4, 4]$ and $[3, 5, 5, 2, 1]$.
- $[1, 3, 3, 2]$ and $[1, 2]$.

The first player wins:

- $[1, 2]$ and $[1, 2, 3, 3]$.
- $[1, 2, 3]$ and $[3, 2, 4, 4, 1]$.
- $[1, 2, 3]$ and $[3, 4, 5, 5, 4, 2, 1]$.

# E. Eliminate Tree

**Author:** Pavle Martinović
**Solved by:** 55/84
**First to solve:** *HunTR*

First let's assume that we don't add any vertices, and are interested whether it is possible to eliminate the tree using only operation of type 2. This is possible if and only if the tree has a perfect matching. One direction of this claim is obvious, while the other follows from the fact that if a rooted tree has a perfect matching, then the edge with the lowest depth will always connect a vertex with degree 1 and a vertex with degree $\leq 2$ (if the "lower" vertex isn't a leaf, then there has to be another edge in its subtree, and if the "upper" vertex has degree at least 3, then it has at least two children, and in some that isn't matched to has to have a deeper matched edge).

Let $m$ be the size of the largest matching in the tree. We claim that the answer is $2n - 3m$. First, it is always possible with this many operations, by first taking $n - 2m$ operations of type 1 to add a leaf attached to every unmatched vertex, resulting in a tree with $2n - 2m$ vertices and a perfect matching, which we can destroy with an additional $n - m$ operations of type 2 by what we have already proved. On the other hand, if we ignore all operations of type 2 we will get a tree with $n + t$ vertices, which we eliminate in $\frac{n+t}{2}$ operations. The operations of type 2 applied to this expanded tree have to be a matching so the expanded tree has to have a perfect matching. However, one can notice that with every operation of type 1 the maximal matching size increases by at most 1, so $t \geq n - 2m$, and then the number of required operations is at least $t + \frac{n+t}{2} \geq 2n - 3m$.

It is possible to calculate $m$ with a standard tree dp, allowing us to solve the problem in $O(n)$.

# F. Fast XORting

**Author:** Pavle Martinović
**Solved by:** 35/54
**First to solve:** *3PeasInAPot*

First we notice that operations of type 1 (swaps) and type 2 (XORs) commute, meaning that if we apply two operations: swap $i$ and $i+1$ and then XOR by $x$, it will have the same result as first XORing by $x$ and then swapping $i$ and $i+1$. Then we can also notice that applying XOR $x$ and then XOR $y$ is the same as applying XOR $x$ `XOR` $y$, with one less operation. Using these two insights, we may assume that all of the XORing operations happen at the beginning, and then we may assume that there is only one XORing operation. After XORing, we have to swap using adjacent swaps, and it's well known that the number of operations needed for this is the number of inversions in the permutations.

Now, we notice that whether $a[i]$ `XOR` $x$ is greater or less than $a[j]$ `XOR` $x$ depends on only one bit. Indeed, look at the first bit where $a[i]$ and $a[j]$ differ, if that bit is 0 in $x$ then they remain in the same order as $a[i]$ and $a[j]$, and if it is 1 their order swaps. This means that for any $i$ and $j$ whether $a[i]$ and $a[j]$ are in an inversion or not depends only on a single bit i.e. all the bits are independent. This means that for each value of $2^k$ we may apply XOR $2^k$ to the array and look whether the number of inversions has decreased and if it has, add $2^k$ to $x$ which we will xor by in the beginning. In the end we just XOR everything by $x$, find the number of inversions and add 1 if x isn't 0 to get our answer. Since there are multiple ways to count inversions in $O(n \log n)$, and we need to do so for each bit once, we get a solution in $O(n \log^2 n)$.

Challenge: Solve the problem in $O(n \log n)$.

# G. Graph Race

**Author:** Roman Bilyi
Aleksa Milisavljević
**Solved by:** 13/36
**First to solve:** *[UoNovi Sad] Infinity*

Since there is an edge between vertices 1 and $v$ it's possible to go from vertex $v$ to vertex 1 and then from vertex 1 to vertex $u$. That means $dist(v,u) \le dist(1,u)+1$. Similarly, $dist(1,u)-1 \le dist(v,u)$. So there are 3 possible cases:

- $dist(v,u) = dist(1,u) - 1$;
- $dist(v,u) = dist(1,u)$;
- $dist(v,u) = dist(1,u) + 1$.

Let $f(u,x) = a_u - b_u \cdot (dist(1,u)+x)$, we are interested only in values of $f$ with $-1 \le x \le 1$. Since $b_u \ge 0$, $f(u,-1) \ge f(u,0) \ge (f,1)$. So if it's possible to reach $u$ using $d(1,u)-1$ edges, we can still update the answer using all values $f(u,-1)$, $f(u,0)$ and $f(u,1)$.

We already know that it's possible to reach any vertex using $dist(1,u)+1$ edges. That means we can initialize $ans_v = \max_{u \ne v} f(u,1)$. After that, we need to update the answers with values $f(u,-1)$ and $f(u,0)$ when it's possible to reach $u$ using fewer edges.

To reach vertex $u$ using $dist(1,u)-1$ edges we can only use directed edges from $x$ to $y$ such that $dist(1,y) = dist(1,x)+1$. Let's call all such edges forward. To reach vertex $u$ using $dist(1,u)$ edges we can use exactly 1 edge $(x,y)$ such that $dist(1,x) = dist(1,y)$ and all other forward edges.

So we can create another directed graph with $2n$ vertices such that if we have forward edge $x$ to $y$, we will create 2 edges: from $x$ to $y$ and from $x+n$ to $y+n$. If we have edge $(x,y)$ such that $dist(1,x) = dist(1,y)$, we will create edges from $x$ to $y+n$ and from $y$ to $x+n$. It's easy to see that if it's possible to reach vertex $u (u \le n)$ from vertex $v$ in a new graph, then $dist(v,u) = dist(1,u) - 1$ and we can update $ans_v$ with $f(u,-1)$. And if it's possible to reach vertex $u (u > n)$ from vertex $v$, we can update $ans_v$ with $f(u-n,0)$.

Additionally, the new graph is a directed acyclic graph, so we can found all answers using dynamic programming.

Complexity of such solution is $O(n+m)$.

# H. High Towers

**Author:** Roman Bilyi
Denis Banu
**Solved by:** 0/13
**First to solve:** *N/A*

At first, let's say that each tower can communicate with itself as well.

Let's look at the first tower. It can communicate with some prefix of $c$ towers and $s$ more towers, such that each of them if higher than all towers to the left of it. Let's call those $s$ towers an increasing stack of towers. Also assume that $c$ is the largest such value, so the first tower can't communicate with tower $c+1$ if it exists.

Each of the towers from second to $c-1$-th cannot communicate with any other tower that the first can't communicate. So $a_i \le a_1$ for each $2 \le i \le c-1$. On the other hand, $c$-th tower can communicate with all the towers that the first tower can and with tower $c+1$ as well, so $a_c > a_1$. That means we can find $c$ as the first value $a_c$ larger than $a_1$. If such value doesn't exist, $a_1$ should be equal to $n$, and we can make the first tower as the highest and solve the remaining problem.

Now tower $c$ can communicate with the first $c$ towers, $s$ towers from increasing stack and also some range of towers from $c+1$ to $c+r$. We can find value of $r$ as $a_c - c - s$. There are 2 possible cases:

1. Tower $c+r+1$ is the one from increasing stack.
2. Tower $c+r+1$ is not the one from increasing stack, while $h_{c+r} = h_c$.

Let's look at value $a_{c+r}$. In the first case, tower $c+r$ cannot communicate with any other towers either than towers in range $[c, c+r-1]$ and towers in increasing stack. So $a_{c+r} \le r+s$. In the second case, tower $c+r$ can communicate with all towers in range $[c, c+r-1]$, all $s$ towers from increasing stack and with tower $c+r+1$. So $a_{c+r} \ge r+s+1$. That means we can distinguish those 2 cases. Then we can continue such process to find a non-decreasing stack of towers, split the problem into smaller subproblems and solve each of them separately.

Complexity of the solution is $O(n)$.

# I. Impossible Numbers

**Author:** Lucian Bicsi
**Solved by:** 0/5
**First to solve:** *N/A*

Obviously, a number $x$ can be written iff there is a matching in the bipartite graph having vertices corresponding to digits $0-9$ on the left side and one vertex for each cube on the right side, and adding the edge $(i, j)$ iff cube $j$ has digit $i$. Also, vertices on the left side must have a capacity equal to the number of occurrences of each digit in $x$.

Considering that, let's invoke Hall Theorem. A number $x$ can be written with the cubes iff for each subset $S$ of digits, the number of cubes having at least one of the digits from $S$ is at least equal to the total occurrences of digits in $x$ belonging to $S$. Conversely, a number cannot be written iff there is some "witness"set $S$ for which the above does not hold.

Let's take each set $S$ (max $2^{10}$) and first calculate the number of cubes, as described above. Then, we will generate numbers that are a counter-example for $S$ in increasing order using a simple backtracking "branch and bound" algorithm (fix digit by digit, and keep the number of digits fixed from $S$). The answer is the smallest $k$ elements of the merge of these $2^{10}$ infinite lists. In order to merge efficiently, we will keep an answer list $ans$ and we will simulate the merging with the list for each set $S$ while backtracking (e.g. using two pointers), and break when the merge result has $k$ elements.

If coded carefully, the solution should have time complexity $O((n+kd)\cdot 2^d)$ and memory complexity $O(k)$, where $d = 10$.

# J. Jackpot

**Author:** Anton Trygub
**Solved by:** 95/112
**First to solve:** *CodeBusters*

Let $b_1 \le b_2 \le \ldots \le b_{2n}$ be these elements in the sorted order. It's easy to see that the answer can't be larger than $(b_{n+1} + b_{n+2} + \ldots + b_{2n}) - (b_1 + \ldots + b_n)$. From the other side, we can achieve it.

Color elements $b_1, b_2, \ldots, b_n$ yellow, and elements $b_{n+1}, \ldots, b_{2n}$ blue, then at any point in time there will be two adjacent elements of different colors. Perform the operation with any such pair. For any such pair, the difference between the blue and the yellow numbers will be added to your score.

Total runtime: $O(n \log n)$ for sorting (but can also be done in $O(n)$, since we only need to find the median).

# K. $K$ Subsequences

**Author:** Anton Trygub
**Solved by:** 61/87
**First to solve:** *KhNURE_KIVI*

Let $f(a) = t$. Then $f$ of at least one subsequence has to be at least $\lceil \frac{t}{k} \rceil$. We can show that this is achievable.

Let's add elements to the sequences one by one; let $p_i$ be the current largest sum of the suffix of subsequence $i$, and $p$ be the current largest sum of $a$. When we add 1 to the $i$-th subsequence, $p_i \to p_i + 1$, when we add $-1$, we get $p_i \to max(0, p_i - 1)$.

Now, we can do a simple greedy algorithm: go over elements of $a$ from left to right, if it's 1, add it to the subsequence with the smallest $p_i$, if it's $-1$, add it to the subsequence with the largest $p_i$. Then all $p_i$s always differ by at most 1, and we always have $p_1 + p_2 + \ldots + p_k \leq p$, so we never get $p_i > \lceil \frac{t}{k} \rceil$.

Total runtime: $O(n \log k)$ (can also be done in $O(n)$).

# L. LIS on Grid

**Author:** Anton Trygub
**Solved by:** 0/8
**First to solve:** *N/A*

Let's state the criteria, when we can make LIS at most $k$: when

$$\sum_{i=1}^{m} \max(0, a_i - k) \leq k \cdot (n - k)$$

First, let's show that it's necessary. Let's use the Dilworth's theorem:

**Dilworth's Theorem.** In any finite partially ordered set, the largest antichain has the same size as the smallest chain decomposition.

For this problem, chains are increasing sequences of cells, and antichains are sequences of cells, such that no pair of them is decreasing; they form nondecreasing sequences: sequences in which we can go only up or to the right. Then, if the LIS is $k$, then we must be able to decompose all cells into $k$ such nondecreasing sequences.

For each nondecreasing sequence, count the number of times we go one cell up (and stay in the same column). This can happen at most $n - k$ times. From the other side, if $a_i > k$, then this has to happen at least $a_i - k$ times just for that column alone. Hence the inequality.

Now, let's show that it's achievable. First, we can assume that all $a_i \geq k$ (we can replace $a_i$ smaller than $k$ with $k$, construct a solution, and then delete some cells). We will now build $k$ nondecreasing sequences of cells; the $i$-th sequence will start at cell $(n - k + i, 1)$. We will also keep $b_i = a_i - k$: we will make each subsequence will have at least one cell in each column, so $b_i$ represents the "extra"amount of black cells that we need.

We will build such sequences one by one. When we process sequence $i$, we will do the following:

1. Assume we are in column $c$. While $b_c > 0$, and the cell above is not black, we go down, decrease $b_c$ by 1, and add the current cell to the sequence.
2. Then, we go to the right and add the current cell to the sequence.

It's easy to see that this construction works (the proof is left to the reader as an exercise).

Total runtime: $O(nm)$.

# M. Max Minus Min

**Author:** Anton Trygub
**Solved by:** 59/83
**First to solve:** *Infinity*

Let's fix the segment on which we add $x$. Denote the maximum and minimum values on this segment as $max_1, min_1$, and outside of this segment as $max_2, min_2$. Then, we can get total $max - min$ to be $\min(max_1 - min_1, max_2 - min_2)$. So, we just need to minimize this value.

Note that if we can improve $max - min$, the chosen segment has to contain $max$ or $min$. Now, let's binary search over $max - min$. Let's check if we can make it $\leq x$.

Assume the chosen segment contains $max$. Then, it shouldn't contain any elements smaller than $max - x$. Choose the maximal such segment. Then, it's enough to check if $max - min$ for remaining elements is $\leq x$ too. Afterwards, do the same check, assuming that the chosen segment contains $min$.

Total runtime: $O(n \log maxA)$.