europe

icpc

european
championship

2023
2024

icpc.foundation

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

The 2024 ICPC European Championship

Solutions

JET BRAINS

icpc global sponsor
programming tools

HUAWEI

icpc diamond
multi-regional sponsor

Jane Street

icpc europe
regional sponsor

pinely

icpc
sponsor

CITADEL | CITADEL Securities

icpc
sponsor

A Grove

AUTHOR: GIOVANNI PAOLINI
PREPARATION: GIOVANNI PAOLINI

Let $\delta = \lceil r \rceil$. Note that all trees need to be planted at least δ away from the boundary of the lawn, so their coordinates (x, y) have to satisfy $\delta \le x, y \le n - \delta$.

For integers $0 \le a_\delta, \dots, a_{n-\delta} \le n - \delta$, denote by $f(a_\delta, \dots, a_{n-\delta})$ any configuration that maximizes the number of trees while only using locations (x, y) such that $x \le a_y$. Our task is to compute $f(n - \delta, \dots, n - \delta)$. We compute values of f recursively, and store all results using memoization.

Let $\bar{x} = \max(a_\delta, \dots, a_{n-\delta})$, and let $\bar{y}$ be such that $a_{\bar{y}} = \bar{x}$. Note that, if $\bar{x} < \delta$, then $f(a_\delta, \dots, a_{n-\delta}) = \emptyset$ because any tree would be too close to the left boundary of the lawn. Assuming that $\bar{x} \ge \delta$, we try to plant or not plant a tree at the location $(\bar{x}, \bar{y})$.

If we do not plant a tree at $(\bar{x}, \bar{y})$, an optimal configuration is given by $f(a'_\delta, \dots, a'_{n-\delta})$, where $a'_y = a_y$ if $y \ne \bar{y}$ and $a'_{\bar{y}} = a_{\bar{y}} - 1$.

If we plant a tree at $(\bar{x}, \bar{y})$, an optimal configuration is given by $f(a'_\delta, \dots, a'_{n-\delta}) \cup \{(\bar{x}, \bar{y})\}$, where a'_y is the largest integer $\le a_y$ such that locations (a'_y, y) and $(\bar{x}, \bar{y})$ are at distance $\ge 2r$.

The number of recursive calls increases as r decreases. For this reason, it can be beneficial to solve by hand the cases where r is small. For instance, if $r \le 1/2$, then we can plant trees at all integer coordinates in the interior of the lawn; if $1/2 < r < \sqrt{2}/2$, we can plant trees at all integer coordinates (x, y) in the interior of the lawn such that $x + y$ is even.

If the implementation of the above algorithm is not fast enough, it is also possible to pre-compute all optimal configurations offline. Indeed, for every n, there is a finite number of intervals $(p, q] \subseteq \mathbb{R}$ such that the valid configurations of trees are the same for all $r \in (p, q]$. The boundaries of these intervals are of the form $\sqrt{a^2 + b^2}/2$ for integers $0 \le a \le b \le n$. For instance, the first two intervals are always $(0, 1/2]$ and $(1/2, \sqrt{2}/2]$, which we have already encountered above.

Finally, it is also possible to solve this problem using a generic max-clique algorithm, especially in combination with the above idea to pre-compute optimal configurations offline.

$\boxed{\text{B}}$ Charming Meals

AUTHOR: ANTON TRYGUB
PREPARATION: ANTON TRYGUB

Let's sort spicinesses, assume $a_1 \le a_2 \le \ldots \le a_n$, $b_1 \le b_2 \le \ldots \le b_n$.

Lemma: Let c, d be some nondecreasing arrays of length n. If there exists some permutation $\sigma(1), \sigma(2), \ldots, \sigma(n)$, such that $c_i \le d_{\sigma(i)}$ for all i, then $c_i \le d_i$ for all i.

Proof: for each i, there can be at most $i-1$ ds smaller than c_i: $d_{p_1}, d_{p_2}, \ldots, d_{p_{i-1}}$, as for any $j \ge i$ we have $d_{p_j} \ge c_j \ge c_i$.

Consider some optimal pairing, assume we paired a_i with $b_{\sigma(i)}$, and got a minimum charm of k. Let's call pairs with $a_i + k \le b_{\sigma(i)}$ **small**, and pairs with $a_i \ge b_{\sigma(i)} + k$ **large**. Consider small pairs, assume there are t of them. Note that, by the lemma, we can assume that a_is and $b_{\sigma(i)}$s in these pairs are nondecreasing. Similarly, they are nondecreasing in large pairs.

But then we can just pair smallest t a_is with largest t b_is, and largest $n-t$ a_is with smallest $n-t$ b_is, and we will still have a charm of at least k. More formally:

- For $1 \le i \le t$, pair a_i with $b_{i+(n-t)}$;
- For $t+1 \le i \le n$, pair the a_i with b_{i-t}.

So, it's enough to check such a pairing for each possible $0 \le 1 \le t \le n$, and to pick the best one!

Total runtime is $O(n^2)$.

C Annual Ants' Gathering

AUTHOR: PETR MITRICHEV
PREPARATION: MICHAEL ZÜNDORF

First Solution

Since ants can not move to an empty house the region of non-empty homes always needs to stay connected. Therefore, only ants on a leaf of the tree (of non-empty homes) can ever move and they can only move in one direction. Further, notice that if the smallest group of ants on a leaf can not move the solution is impossible (all other leaf groups are larger and can therefore also never be merged into the parent group of the smallest group). We can simply simulate this process.

Second Solution

Notice that the centroids of the tree are the only homes where all ants can gather up. For a fixed root we can greedily simulate the process with a DFS for each centroid.

D Funny or Scary?

Author: Petr Mitrichev
Preparation: Petr Mitrichev

In graph terms, this problem can be formulated as follows: you are given a complete undirected graph on n vertices. Some of the edges of the graph are already colored with two colors. You need to color all remaining edges with two colors in such a way that the graph does not have long monochromatic simple paths.

First of all, what happens when no edges are already colored? It turns out this exact question is well-studied in mathematics, and the definitive answer was given 57 years ago in the following paper: L. Gerencsér, A. Gyárfás, On Ramsey-type problems, *Ann. Univ. Sci. Budapest Eötvös Sect. Math.* 10 (1967), pp. 167-170. They prove that for any such graph on n vertices there always exists a monochromatic path with at least $\lfloor \frac{2n}{3} \rfloor$ edges, and provide an example of such a graph where the longest monochromatic path has exactly $\lfloor \frac{2n}{3} \rfloor$ edges.

What does their example look like? The key idea is to use a bipartite graph with unbalanced parts. More specifically, we split all vertices into two groups, the first one of size roughly $\frac{n}{3}$, and the second one of size roughly $\frac{2n}{3}$. We color the edges between the two groups red, and the edges within each group blue. Now each blue path will have to stay within one of the groups, and therefore will not be longer than $\frac{2n}{3}$. The red edges form a bipartite graph, so each red path will have to alternate groups, which means that it will have to go to the smaller group every second step, and since the smaller group has only $\frac{n}{3}$ vertices, the path also cannot be longer than $\frac{2n}{3}$.

Sadly, not everyone will have read this seminal paper from 1967, so how can one come up with this construction during the contest? Since every edge in the complete graph will have one of the two colors, at least one of the colors will be used by a lot of edges. So we need to come up with examples of graphs with lots of edges, but no long paths. Disconnected graphs are a natural way to achieve this, and then we can consider what edges the complement of a disconnected graph must have, and arrive at the above construction. One other thing one can do when stuck is experimentation: try generating random complete graphs colored with two colors until you find some examples with no long monochromatic paths, for example where there is no monochromatic path with $n-1$ edges, and then try to generalize their structure. In fact, the answer to the second sample was generated in exactly this fashion: we repeatedly tried to color each edge randomly and independenty until we got a graph with 12 vertices and no monochromatic path with 10 edges. It actually took a few hours to find just one such graph. Studying this answer could be another way to discover the above construction, as the funny edges do form an almost-bipartite structure with nodes 5 and 11 in one part, and the rest in the other part (for example row 5 of the matrix is `FFFF.FFFFFSF`).

We still need to deal with edges that are already colored, but we also still have some reserves: our paths do not exceed $\lfloor \frac{2n}{3} \rfloor$, while this problem allows paths up to $\lceil \frac{3n}{4} \rceil$.

How bad is a wrongly colored edge to the above construction? If an edge that is supposed to be red is actually blue, it is very bad: now the two blue groups are connected, and therefore it is easy to create a blue path through all vertices. However, if an edge that is supposed to be blue is actually red, it is not so bad: even when this edge is used in a red path, all "originally" red edges still have to have one of their endpoints in the smaller group, and therefore the number of originally red edges in a red path still does not exceed two times the size of the smaller group, so changing one blue edge to red increases the boundary on the red path length by at most 1.

We have the freedom to choose which vertex belongs to which group, so we need to use this freedom to make sure that all edges that are already blue connect vertices within one group, as even one incorrectly blue edge is bad. This means that we need to find the connected components using the preexisting blue edges, and then take some of those components to the bigger group and the rest to the smaller group.

The edges that are already colored red can each increase the boundary on the red path length by 1. If all $\lfloor \frac{n}{2} \rfloor$ already colored edges are red, this would increase the path length by $\lfloor \frac{n}{2} \rfloor$, which is a lot. However, we also have the freedom to decide which one out of funny and scary corresponds to red. Let us choose one that has fewer edges, and therefore the number of already colored red edges will not exceed $\lfloor \frac{n}{4} \rfloor$.

Because the length of the red path can be increased by $\lfloor \frac{n}{4} \rfloor$ through the already colored edges, we need to make the smaller group even smaller: since the red path must not exceed $\lceil \frac{3n}{4} \rceil$, we can have at most $\lfloor \frac{\lceil \frac{3n}{4} \rceil - \lfloor \frac{n}{4} \rfloor}{2} \rfloor$ vertices in the smaller group, which is roughly $\frac{n}{4}$, and more precisely it is $\geq \lfloor \frac{n}{4} \rfloor$.

If the smaller group has $\lfloor \frac{n}{4} \rfloor$ vertices, the bigger group has $n - \lfloor \frac{n}{4} \rfloor = \lceil \frac{3n}{4} \rceil$ vertices, which means that blue paths have at most $\lceil \frac{3n}{4} \rceil - 1$ edges. Notice how all pieces of the puzzle come together now: this is good enough for this problem, but only barely, by 1 edge. So we can allow the smaller group to have either $\lfloor \frac{n}{4} \rfloor$ or $\lfloor \frac{n}{4} \rfloor - 1$ vertices.

The only remaining question is: since we add vertices to groups not one by one, but an entire preexisting blue edge component at a time, can we actually achieve the required size of the smaller group? It turns out that we can, using the following simple approach: sort the components in increasing order of size, and take them in this order to the smaller group until the size of the smaller group is $\lfloor \frac{n}{4} \rfloor - 1$ or $\lfloor \frac{n}{4} \rfloor$.

The only way this could fail is if we jump from at most $\lfloor \frac{n}{4} \rfloor - 2$ to at least $\lfloor \frac{n}{4} \rfloor + 1$, meaning that the component we have added has at least 3 vertices. But since we add components in increasing order of size, it means that the remaining components also all have at least 3 vertices, which means that components with at least 3 vertices cover at least $n - (\lfloor \frac{n}{4} \rfloor - 2) = \lceil \frac{3n}{4} \rceil + 2$ vertices of the graph. Since we need 2 edges for a component with 3 vertices, and with bigger components we still need at least $\frac{2k}{3}$ edges for a component with k vertices, we need at least $\frac{2(\lceil \frac{3n}{4} \rceil + 2)}{3}$ edges to get into this situation, and $\frac{2(\lceil \frac{3n}{4} \rceil + 2)}{3} \geq \frac{\frac{3n}{2} + 4}{3} > \frac{\frac{3n}{2}}{3} = \frac{n}{2}$. So we need more than $\frac{n}{2}$ edges to get into this situation, which is impossible under the constraints of the problem, therefore the greedy approach of taking already blue components in increasing order of size always works.

The solution is complete now, and it runs in time linear in the input size, in other words $O(n^2)$. Why does the problem have the very low limit of $n \leq 24$ in this case? The reason is that we also need to implement the checker that will read your output and check if there are no long monochromatic paths. For that we use a relatively standard dynamic programming approach that runs in $O(n^2 \cdot 2^n)$, sped up using bitmasks to do only $O(n \cdot 2^n)$ operations with integers. It takes a few seconds for $n = 24$ so we could not go much higher. We have explored using various heuristic approaches to find the longest monochromatic paths for larger values of n, but it turns out that even though those heuristic approaches work quite well on random graphs, they actually cannot find the longest path reliably in the type of graph output by the solution above, namely an unbalanced bipartite graph with a few additional edges.

The low value of n allows to simplify the implementation of the above solution in the following manner: we can skip finding connected components of preexisting blue edges, instead just iterating over all 2^n subsets of vertices as candidates for the smaller group, and then checking if the size of the smaller group is appropriate and that there are no preexisting blue edges going from the smaller group to its complement.

E Damage per Second

Author: Michael Zündorf, Federico Glaudo
Preparation: Michael Zündorf

Once the sugar-coating is removed, the problem boils down to:

Let $f(x) = \sum_{i=1}^{n} \left\lceil \frac{h_i}{x} \right\rceil$*, and* $g(x) = \frac{f(x)}{k-x}$*. Find the minimum of* $g(x)$*.*

Sort the values so that $h_1 \geq h_2 \geq \cdots \geq h_n$. Define $H := h_1 + \cdots + h_n$. From now on, we will assume that k is even, so that $x = \frac{k}{2}$ is a valid choice (the case k odd is absolutely equivalent, just more annoying to write down).

Let us begin with two simple observations.

Observation 1. What are the "most interesting" values of x? If we ignore the ceiling in the definition of g, we get $g(x) \geq \frac{H}{x(k-x)}$ whose minimum is achieved by $x = \frac{k}{2}$. Unless the values $h_1, \dots, h_n$ are well-crafted, we expect the minimum of g to be achieved for an x very close to $\frac{k}{2}$.

Observation 2. Speeding up the computation of g. Let us describe a strategy to compute $g(x)$. Iterate from $i = 1$ to n until $i < \frac{h_i}{x} \log n$. Say that such condition is satisfied for $1 \leq i \leq q(x)$ and fails for $q + 1$ (for notational clarity we will omit the dependence of q on x). We compute the first part of $g(x)$, that is $\sum_{i=1}^{q} \lceil \frac{h_i}{x} \rceil$, naively in $O(q)$. For $q + 1 \leq i \leq n$ the quantity $\lceil \frac{h_i}{x} \rceil$ takes at most $\frac{h_{q+1}}{x}$ different values. Therefore the second part of $g(x)$, that is $\sum_{i=q+1}^{n} \lceil \frac{h_i}{x} \rceil$, can be computed in $O(\frac{h_{q+1}}{x} \log n) = O(q)$ by using binary search.

So, we have described a way to compute $g(x)$ in $O(q(x))$ where q satisfies $h_q \geq \frac{qx}{\log n}$.

Description of the algorithm. These two observations suggest the following algorithm.

- Maintain the running minimum m of $g(x)$, initializing it as $m = g(\frac{k}{2})$.
- Iterate over all x, starting from those closest to $\frac{k}{2}$ and moving further away.
- For each x, if $\frac{H}{x(k-x)} \geq m$ then skip this choice of x (as it cannot be a minimizer in view of the first observation).
- Otherwise compute $g(x)$ in $O(q(x))$ as described in observation 2 and update m accordingly.

This algorithm produces the correct answer, but is it provably fast enough? Unexpectedly yes! We will see that this algorithm has the remarkable running time $O(\sqrt{n \log n}k)$.

Running time analysis. Let us begin by making one further remark about the second observation. By definition of $q(x)$, we have

$$H \geq h_1 + \cdots + h_q \geq qh_q \geq \frac{q^2 x}{\log n} \implies q \leq \sqrt{\frac{H \log n}{x}}.$$

So, we are able to compute $g(x)$ in $O(\sqrt{\frac{H \log n}{x}})$ time. This is awful if H is huge, but can be potentially very good if H is controlled and x is comparable to k. We will see that the opposite happens for observation 1: it is very powerful if H is huge and very weak if H is small. So there is a tradeoff: for H large the first observation is powerful, for H small the second one is powerful. This will be enough to provide a good running time for the whole algorithm.

Let us now discuss the first observation. For any $1 \leq x \leq k-1$,

$$\frac{H}{x(k-x)} \leq g(x) \leq \frac{H}{x(k-x)} + \frac{n}{k-x}.$$

We want to use this inequality to bound the values of x that we need to check.

We have $g(\frac{k}{2}) \leq \frac{4H}{k^2} + \frac{2n}{k}$ and $\frac{H}{x(k-x)} \leq g(x)$. Therefore, if

$$\frac{4H}{k^2}+\frac{2n}{k} < \frac{H}{x(k-x)} \iff \frac{k^2}{4}-\left(x-\frac{k}{2}\right)^2 = x(k-x) < \frac{H}{\frac{4H}{k^2}+\frac{2n}{k}} \iff \left(x-\frac{k}{2}\right)^2 > \frac{k^2}{4}-\frac{H}{\frac{4H}{k^2}+\frac{2n}{k}} = \frac{nk^3}{8H+4nk}$$

we don't have to check x as it cannot yield the minimum value for g. In particular, we care only about the values of x that satisfy

$$\left|x-\frac{k}{2}\right| \leq \sqrt{\frac{nk^3}{8H}}.$$

As we anticipated, when H is huge this bound is great, while when H is small it is not particularly useful.

Conclusion. Now we have all the tools and we can conclude by considering two cases (which, morally, correspond to H large and H small).

If $\sqrt{\frac{nk^3}{8H}} < \frac{k}{4}$, then the interesting values of x are comparable to k. Thus $g(x)$ can be computed in $O(\sqrt{\frac{H\log n}{k}})$ and we shall do such computation $2\sqrt{\frac{nk^3}{8H}}$ times. So the overall complexity is $O(\sqrt{n\log n}k)$.

On the other hand, if $\sqrt{\frac{nk^3}{8H}} \geq \frac{k}{4}$ (which implies $H \leq 2nk$) then we use only the second observation to bound the complexity. Let us assume that we care only about values of x of magnitude comparable to k, the other cases are easy to handle. Then, the computation of $g(x)$ requires $O(q\log n)$. To compute $g(x)$ for all $1 \leq x \leq k$, we need

$$O\Big(\sum_{x=1}^{k}\sqrt{\frac{H\log n}{x}}\Big) = \sqrt{nk\log n}O\Big(\sum_{x=1}^{k}x^{-1/2}\Big) = O(\sqrt{n\log n}k).$$

$\boxed{\text{F}}$ Dating

AUTHOR: FEDERICO GLAUDO
PREPARATION: LUCIAN BICSI

After formalizing the statement, we get the following:

Given n sets $S_1, S_2, \ldots, S_n$ of activities, find a pair (a, b) such that all three sets $S_a \setminus S_b$, $S_b \setminus S_a$, $S_a \cap S_b$ are non-empty.

In essence, this means that for a pair (a, b) to **not** be good, it must suffice that S_a and S_b are either disjoint or one included in the other. Consequently, it means that, if there are no such good pairs, then the sets must induce a tree structure! In this structure, some vertex u is an ancestor of vertex v if and only if $S_v \subseteq S_u$.

The solution now can be summarized by the following idea: try to construct the tree (assuming there are no good pairs), and check for errors along the process (this will, in fact, expose a good pair). There are multiple ways of achieving that, both top-down and bottom-up (and they are more or less equivalent). We will describe a solution based on a top-down approach.

Top down approach. Let's start by sort the activity sets in decreasing order of lengths. For all activity sets S_i in this order, we will create a parent-child relationship $j = p(i)$ with the last processed index j that contains some activity (or -1, if there is no such index). The activity based on which we choose j is not relevant, for reasons that will be apparent as follows. After finding such j (or deciding that $j = -1$), we have some cases to consider:

1. If $j \neq -1$ and $S_i \not\subseteq S_j$, then (i, j) is a good pair, as S_j cannot possibly include S_i due to the length of S_j being at least as much as S_i.
2. If some activity $x \in S_i$ is also in some other set S_k where $p(k) = p(i)$, then (i, k) is a good pair.
3. Otherwise, the activity sets still form a tree, therefore no good pairs exist (yet).

Complexity is $\mathcal{O}(k \log k)$ or $\mathcal{O}(k)$, where k is the total number of activities, depending on whether one uses ordered sets or hash maps. *It is possible to implement this solution in $\mathcal{O}(k)$ using just arrays, but this is not required.*

Alternative approach. One may solve this problem without leveraging the tree structure of the activity sets. It stems from the observation that any good pair (a, b) must have a common element x. Let's fix all such x, and sort the sets having x as an element in the order of length. Then, similarly to the first solution, one may prove that it is sufficient that a set with a smaller length has an element not present in some set with a larger length. We can solve this by simply keeping an array which keeps track of existing elements present in the sets, and iterating from right to left.

This approach does not quite work yet, as it has quadratic complexity; however, by considering the sets with large size (i.e., larger than $\sqrt{k}$) separately than the elements with a small size (i.e. smaller than $\sqrt{k}$), then one may achieve $\mathcal{O}(k\sqrt{k})$ complexity using this approach.

$\boxed{\text{G}}$ Scooter

AUTHOR: GIOVANNI PAOLINI
PREPARATION: LUCIAN BICSI

First and foremost, let's notice that we can ignore professors that are already in a proper building, as well as empty buildings (in essence, all positions i such that $c_i = p_i$).

More so, if there are no positions i such that $c_i \neq p_i$ then we can simply answer `YES` without doing any operations.

From this point on, we will assume that for all i, we have $c_i \neq p_i$.

General idea. Intuitively, we would want to iteratively pick up the professor from some building and drive them to a building where a corresponding class is held. However, we must make sure to not visit the same place twice. Multiple greedy solutions might solve this, but we also have to take into account implementation effort and the number of edge cases to consider. In this sense, we advise the reader to not rush into writing an implementation just yet, and instead try to simplify the solution as much as possible on paper.

An easier subproblem. The issue with the problem currently is that there is much room for making choices and, consequently, much room for making an erroneous decision. Let's consider the easier subproblem where there are exactly as many professors of a given class types as there are classes. Even though one might say that this makes the problem tighter, it actually paradoxically makes it easier! This is mainly because, as tight as it is, there is not much room for bad decisions.

A key observation is that this subproblem (and, as it turns out, the original problem as well) is **impossible** if there are no buildings where no class is being held, as there is simply no way of returning to the starting position to drop off the corresponding professor (note, again, that we are ignoring positions i with $c_i = p_i$).

Therefore, a greedy strategy that works for this subproblem is simply repeatedly choosing a professor from a building where no class is being held and driving them to one of the buildings where their class is being held. If one prioritizes buildings where there are professors before those where there are no professors, the key condition never becomes unsatisfied, thus the problem is solved.

Solving the original problem. Going back, let's solve the original problem. We can do that by simply reducing it to the easier version, by removing "excess" professors from some of the buildings. At the same time, note that we have to make sure the key observation above remains valid throughout the removal process. One easy and correct way of doing that is, yet again, the greedy approach of simply prioritizing removing professors from buildings where there are classes to buldings with no classes.

Final complexity of the solution is $\mathcal{O}(n)$.

H Division Avoidance

Author: Petr Mitrichev
Preparation: Petr Mitrichev

Looking at the solution for the sample, the most difficulty seems to come from the condition in bold: **A division of a cell (a, b) can only happen if the cells $(a+1, b)$ and $(a, b+1)$ are not yet part of the organism.** It forces us to clean up space, potentially recursively, before a division that we want to happen can actually happen.

What happens if we remove this condition, instead allowing to have multiple copies of a cell with the same coordinates in the organism? Then of course it is possible to avoid any finite set of forbidden cells, so this version of the problem is not very useful. However, consider the following modification instead: we allow to have multiple copies of a cell with the same coordinates temporarily, but the final state of the organism must have at most one copy of any cell, and of course zero copies of the forbidden cells. It turns out that this modification is equivalent to the original problem!

To see why, consider a sequence of operations that starts from $(0, 0)$ and ends in a valid state as described above, but has multiple copies of some cells in its intermediate states. We claim that it is possible to reoder its sequence of operations in such a way that all operations stay valid, but we never have multiple copies of the same cell. Notice that reodering the operations does not change the end state, because each operation can be seen as subtracting 1 from the number of occurrences of the cell (a, b) and adding 1 to the numbers of occurrences of the cells $(a+1, b)$ and $(a, b+1)$, and such subtractions and additions clearly commute.

To avoid getting multiple copies of the same cell, we can repeatedly do the following to produce a reordering: from all operations that are yet to be done in the reference sequence of operations, we choose the operation (a, b) such that (a, b) is part of the organism and $a + b$ is maximized. Maximizing $a + b$ guarantees that we are not going to have two copies of the same cell: when we decide to divide (a, b), we cannot have $(a + 1, b)$ in the organism, as otherwise $(a + 1, b)$ would also be one of the operations yet to be done (since the final state has at most one cell at $(a + 1, b)$), so we should have chosen $(a + 1, b)$ as the opeation to do over (a, b) since $a + 1 + b > a + b$, which is a contradiction.

Now we know that we can focus on what operations to do, and no longer care about the order in which we do them. It means that we can choose the order that suits us best, and as long as we do only necessary operations, we will always arrive at exactly the same result. It is natural to go by diagonals: first consider all operations with $a + b = 0$, then all operations with $a + b = 1$, then all operations with $a + b = 2$, and so on. The operations with $a + b = z$ only create new cells with $a + b = z + 1$, so those operations are independent within each value of z, and we never have to return to smaller values of z.

This leads to the following solution: we go in increasing order of z, and before processing a given value of z we know how many copies of each cell with $a + b = z$ we have, as an array $d_{z,a}$. Now for each forbidden cell (a, b) with $a + b = z$ we have to apply division to it $d_{z,a}$ times, which means that we have to add $d_{z,a}$ to $d_{z+1,a}$ and to $d_{z+1,a+1}$. Similarly, for each non-forbidden cell we have to apply division to it $\max(0, d_{z,a} - 1)$ times. We continue while the array d_z has at least one non-zero value. If this process terminates, the answer is `YES`. If it runs infinitely, the answer is `NO`.

For example, in the first sample case we get the following arrays:

- $d_0 = (1)$
- $d_1 = (1, 1)$
- $d_2 = (1, 2, 1)$
- $d_3 = (0, 2, 2, 0)$
- $d_4 = (0, 1, 2, 1, 0)$
- $d_5 = (0, 0, 1, 1, 0, 0)$
- $d_6 = (0, 0, 0, 0, 0, 0, 0)$

However, this solution is still not practical: how do we check if the process runs infinitely? Looking at a few examples can help spot a pattern: if we ever get a 3, then the process never stops. Having identified this pattern, it is relatively easy to prove. Since we always add the same number to two adjacent elements of d_{z+1}, getting a 3 in it means that one of its adjacent numbers will be at least 2. And then, even if there are no forbidden cells at all anymore, $3, 2$ becomes $2, 3, 1$ on the next diagonal, so we have a 3 adjacent to a 2 again, and this will repeat forever.

What if we never get a 3? Then it turns out that as soon as we pass all forbidden cells, we will start going down towards all zeros and eventually get there. With no forbidden cells, a 1 does not propagate to the next diagonal at all, and a 2 propagates two adjacent $+1$s to the next diagonal. So if we have a sequence that looks like $(2, 2, \ldots, 2)$ with k 2s, on the next diagonal we will get $(1, 2, 2, \ldots, 2, 1)$ with $k - 1$ 2s, so after k diagonals the 2s will disappear.

This argument also demonstrates that the process will converge after $O(n)$ steps, since every forbidden cell will create a constant amount of additional 2s, and without forbidden cells the number of 2s decreases by at least 1 per step. Therefore we finally have a complete solution, albeit one that runs in $O(n^2)$ which is too slow for $n = 10^6$.

In order to make it faster, we have to take a closer look at the changes of the array d. Suppose the array looks like $(0, 0, \ldots, 0, 1, 2, 2, \ldots, 2, 1, 0, 0, \ldots, 0)$ for a certain diagonal. How is it going to look for the next diagonal depending on the locations of the forbidden cells on this diagonal?

- If a cell with a 0 is forbidden, this changes nothing.
- If a cell with a 2 is forbidden, then we are going to get a 3 on the next diagonal, and the answer will be `NO`.
- So the only interesting question is whether the two cells with 1 are forbidden:
- If none of the 1s are forbidden, then we get the same structure on the next diagonal, but with the left 1 moving one position to the right (and therefore the number of 2s decreases by one).
- If only the left 1 is forbidden, then we still get the same pattern on the next diagonal, but the left 1 stays in the same position (and therefore the number of 2s is unchanged).
- If only the right 1 is forbidden, then it moves one position to the right on the next diagonal (and therefore the number of 2s is also unchanged).
- Finally, if both 1s are forbidden, then we still get the same pattern on the next diagonal, but with the number of 2s actually increasing by one.

This suggests the following idea: when there are no 3s or higher on a diagonal, and when not all numbers are 0s, the array d for it always has the form $(0, 0, \ldots, 0, 1, 2, 2, \ldots, 2, 1, 0, 0, \ldots, 0)$: exactly two 1s, zero or more 2s in between the 1s, and 0s on the outside. This is not entirely correct: in the sample discussed above there is also the case of $(0, 0, \ldots, 0, 2, 2, 0, 0, \ldots, 0)$: exactly two 2s surrounded by all 0s. This pattern always appears after $(0, 0, \ldots, 0, 1, 2, 1, 0, 0, \ldots, 0)$ when the 2 is forbidden, and it is actually the only situation where a 2 can be forbidden without getting a 3. It turns out that this is the only special case, and we can easily prove by induction that we always have one of the patterns mentioned above.

Therefore as we go in increasing order of z, for the diagonal $a + b = z$ we can simply store two integers and a boolean: the position of the left 1, the position of the right 1, and whether we have the special $0, 2, 2, 0$ case. We need to find all forbidden cells on the given diagonal, which we can achieve by sorting the forbidden cells by the sum of the coordinates and keeping a pointer into the sorted array as we increase z. After that we can determine the state of the next diagonal in $O(1)$, so the total running time of this solution is $O(n \log n)$ if we use normal sorting, or $O(n)$ if we use counting/radix sorting (we can use counting sorting since we only care about the first $O(n)$ diagonals).

The time limit in this problem was pretty tight, so even the $O(n)$ solution had to be implemented with care. The reason for this was that the $O(n^2)$ solution mentioned above yields itself perfectly to optimizations. The only quadratic part is a couple of small inner loops looking roughly like this:

```
for (int i = left; i <= right; ++i) {
  nam[i] = am[i - 1] + am[i];
}
```

The C++ compilers are very good at vectorizing such code using SIMD instructions (which you can enable using some pragmas). But we can also help the compiler because we know that the values in our arrays are always only 0, 1 or 2, so we can actually represent our array as two bitsets. We could not get the compiler to efficiently vectorize `std::bitset`, but it turns out that `gcc` supports a relatively new syntax

```
typedef uint64 fast_vec __attribute__ (( vector_size(32) ));
```

This defines a new type that works like an array of 4 `uint64`'s, but supports element-wise operations including bitwise operations that automatically use vectorization. Using this trick made the $O(n^2)$ solution mentioned above run in under 7 seconds for $n = 10^6$, therefore to make sure this solution does not pass we had to set the tight time limit.

One other potentially relevant remark for implementing the solution: we have mentioned above that we only care about the first $O(n)$ diagonals, but what is the constant hidden in the O-notation? Initially one might think that a case with the following forbidden cells is the worst: $(0, 0), (1, 0), (2, 0), \ldots, (\lceil \frac{n-1}{2} \rceil, 0), (0, 1), (0, 2), \ldots, (0, \lfloor \frac{n-1}{2} \rfloor)$. The final state of the organism in this case is a big rectangle (so it also catches solutions that assume we only do a linear number of divisions), and we have $n + 1$ nonzero diagonals in this case.

However, the $0, 2, 2, 0$ exception from the solution above allows to construct a testcase that is twice nastier: $(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), \ldots, (n-3, n-3)$. In this case the array d constantly changes from $1, 2, 1$ to $0, 2, 2, 0$ and back, and we get $2n-2$ nonzero diagonals. So if your solution hardcodes the number of diagonals to process you need to go as high as $2n-2$.

For more context about the origins of this problem and related topics you can check out the following paper: F. Chung, R. Graham, J. Morrison, A. Odlyzko, Pebbling a Chessboard, *Amer. Math. Monthly* 102 (1995), pp. 113–123.

I Disks

Author: Giovanni Paolini
Preparation: Giovanni Paolini

To start, we build the graph that describes the tangency relation between the disks: each disk is represented by a node in the graph, and an edge connects two nodes if and only if the corresponding disks are tangent. Constructing this graph can be done in time $O(n^2)$ by checking all pairs of disks. In fact, it can be done in faster than quadratic time, but this is not necessary to solve the problem with the given constraints.

Suppose that you change the radii of the disks by real numbers $\delta_1, \dots, \delta_n$. In order to maintain tangency, we must have $\delta_i + \delta_j = 0$ whenever the i-th and j-th disks are initially tangent. In fact, this is condition is also sufficient, provided that $\delta_1, \dots, \delta_n$ are small enough in absolute value (to avoid overlaps and to keep all the radii positive).

Fix a connected component of the tangency graph, and fix any node i in this component. The value δ_i determines the values of δ_j for all j in the same connected component, and each δ_j has to be equal to either δ_i or $-\delta_i$. If the connected component is not bipartite, it contains a cycle of odd length, and such a cycle shows that $\delta_i = -\delta_i$ and so $\delta_i = 0$. On the other hand, if the connected component is bipartite, then it is possible to assign any real value to δ_i. Color the nodes of such a bipartite connected component black and white, so that every edge connects a white node and a black node; use white to color the node i. The sum of the radii in this connected component changes by $\delta_i \cdot k$, where k is the difference between the number of white nodes and the number of black nodes. In order to strictly decrease the sum of the radii, we need a different number of black and white nodes ($k \neq 0$); if this happens, we can pick δ_i to be any sufficiently small real number such that $\delta_i \cdot k < 0$.

To summarize, the answer is `YES` if and only if there is at least one connected component which is bipartite and has a different number of white and black nodes. This condition can be checked in time $O(n)$.

J Amanda the Amoeba

Author: Lucian Bicsi
Preparation: Martin Kacer

To solve this problem, we first find a path P_{AB} connecting two pixels A and B such that

- Pixel A is a part of the initial position.
- Pixel B is a part of the final position.
- All inner pixels of P_{AB} (let us denote their sequence as Q_{AB}) are free pixels, neither inside the initial position nor the final one.

Note that it is generally allowed to select two neighbouring pixels or even having $A = B$, in which case Q_{AB} will be empty. For the solution to work, it is not important how such a path is found, one possible way is to find the shortest path between the initial and final position.

If no such path exists, the problem has no solution, as it means the initial and final position are completely separated by blocked cells. Otherwise, the final position is reachable and the problem has a solution, which will be proven by describing its construction.

As the next step, we will construct a tree T_A rooted in pixel A and containing all pixels of the initial position (and no other pixels), and also a tree T_B rooted in pixel B and containing all pixels of the final position (and only them). Again, it is not important how the trees look like, as far as they contain all appropriate pixels. It is easy to construct them using either DFS or BFS.

The solution then consecutively removes pixels of T_A bottom-up, starting from leaves and proceeding to parents when they become leaves. Such removed pixels are first added to fill the sequence Q_{AB}, and then to tree T_B top-down, starting with B and going to lower nodes after their parents were added. In the end, pixels of Q_{AB} need to be removed (and added to the bottom of T_B). After that, the amoeba will occupy the exact final position.

There is one more important issue to cope with. The trees T_A and T_B are not necessarily disjoint, which means that some (or even many) pixels may appear in both. While removing pixels from T_A and adding them to T_B, the following situations may occur:

- Some pixel should be removed from T_A but we know it is also part of T_B and has not been added yet. In this case, the algorithm proceeds just normally, removing the pixel. It will later be added again to the body, which adds to the total number of moves, but there is no requirement to minimize that number.
- Some pixel should be added to T_B but it is still part of the body, as it was not yet removed from T_A. In such a case, the pixel is skipped (and stays a part of the body to keep it connected) and it has to be marked as such. When the algorithm later proceeds to that pixel as part of T_A, it must not be removed anymore, and the algorithm proceeds to other pixels in T_A.

These steps keep the amoeba connected at all times, and in the end, all pixels of T_B are occupied, which means reaching the final position. Although the algorithm may remove some pixels and then add the same pixels again, this happens at most once for each of the pixels. In fact, each pixel is removed at most once and also added at most once, which means the maximal number of steps produced by the algorithm is limited by the total number of pixels in the pixture, $r \cdot c$, which is way below the required limit.

$\boxed{\text{K}}$ Make Triangle

AUTHOR: ANTON TRYGUB
PREPARATION: ANTON TRYGUB

We will denote these groups as A, B, C correspondingly.

Wlog $n_a \le n_b \le n_c$, and $x_1 \le x_2 \le \ldots \le x_n$. Let $x_1 + x_2 + \ldots + x_n = S$. We just want the sum in each group to be smaller than $\frac{S}{2}$. Let's note some obvious conditions for construction to be possible:

- The largest **group** is not too large: $x_1 + x_2 + \ldots + x_{n_c} < \frac{S}{2}$;
- The largest **item** is not too large: $x_n + (x_1 + x_2 + \ldots + x_{n_a-1}) < \frac{S}{2}$.

It turns out that these conditions are sufficient! We will prove even stronger statement.

Lemma. Assume we already placed some numbers, which are $\ge$ than any remaining numbers. Assume current sum in group g is S_g, the number of empty spots in group g is n'_g, and there are $n'_a + n'_b + n'_c = n'$ numbers remaining, $x_1 \le x_2 \le \ldots \le x_{n'}$. Then, it's possible to distribute the remaining numbers if and only if the following conditions hold:

- No group is too large: for any group g,

$$S_g + x_1 + x_2 + \ldots + x_{n'_g} < \frac{S}{2}$$

- The largest item is not too large: there exists a group g with $n'_g > 0$, such that

$$S_g + x_{n'} + (x_1 + x_2 + \ldots + x_{n'_g-1}) < \frac{S}{2}$$

Proof. These conditions are obviously necessary; let's show that they are sufficient. Forget about $n_a \le n_b \le n_c$ for now Wlog we can put the largest element $x_{n'}$ in group a, so $S_a + x_{n'} + (x_1 + x_2 + \ldots + x_{n'_a-1}) < \frac{S}{2}$. Put the remaining numbers in the other two groups arbitrarily. If both have sum less than $\frac{S}{2}$, we are good! Otherwise, wlog, sum in group B is at least $\frac{S}{2}$.

Now, start swapping free elements in B and C one by one. If at some point sums in both B and C were smaller than $\frac{S}{2}$, we are good. Note that it's not possible that at one moment, the sum in B is $\ge \frac{S}{2}$, and at next the sum in C is $\ge \frac{S}{2}$: the sum in C currently is at most $S - \frac{S}{2} - x_{n'}$, and it will decrease by at most $x_{n'}$ and increase by at least 1 after the swap, so it will still be at most $S - \frac{S}{2} - 1 < \frac{S}{2}$. So, the only possibility is that the sum in B is still $\ge \frac{S}{2}$.

Now, note that if we swap free elements in A and C, the sum in B is still $\ge \frac{S}{2}$. Also, note that if $n'_C = 0$, that is, there are no free elements in C remaining, then we swap free elements in A and B, and we will either get a valid partition, or the sum in B will still be $\ge \frac{S}{2}$, as it's not possible for sum in A to get $\ge \frac{S}{2}$ after one swap, for the same reason.

So, here is the idea: we will start swapping elements, so that the smallest n'_B end up in B. We will get a contradiction, since we know that $S_B + x_1 + x_2 + \ldots + x_{n'_B}) < \frac{S}{2}$.

This lemma also gives us a way to find a construction: add elements from largest to smallest. When deciding where to put an element, put it in any group, such that the conditions will hold after we put it there. We can check these conditions in $O(1)$ after we sorted x_is and precomputed their prefix sums.

Runtime $O(n \log n)$.