ICPC Europe Regionals

JET BRAINS

icpc global sponsor
programming tools

icpc international collegiate programming contest

icpc.foundation

HUAWEI

icpc diamond
multi-regional sponsor

# The 2021 ICPC Southwestern Europe Regional Contest

Official Solutions

# A Organizing SWERC

Author: Federico Glaudo
Preparation: Federico Glaudo

This is the ice breaker problem of the contest. To solve it one shall implement what is described in the statement.

One way to implement it is to keep an array `beauty`$[1 \dots 10]$ (initialized to 0), so that, for $1 \leq$ `diff` $\leq 10$, the value `beauty`[`diff`] corresponds to the maximum beauty of a proposed problem with difficulty equal to `diff`. One can update the entries of `beauty` while processing the input. In the end, if some entries of `beauty` are still 0 then the correct output is `MOREPROBLEMS`, otherwise it is the sum of the entries `beauty`$[1]$ + `beauty`$[2] + \cdots +$ `beauty`$[10]$.

Notice that the small size of the input allows for less efficient solutions. For instance, one could iterate 10 times over the problems, once for each difficulty, and find the maximum beauty associated with difficulty $i$ during the $i$-th iteration (this way, array `beauty` is not needed).

# $\boxed{\text{B}}$ Drone Photo

Author: Andrea Ciprietti
Preparation: Filippo Baroni

For each $i = 1, \ldots, n^2$, let:

- $s_i$ denote the $i$-year-old contestant;
- $u_i$ be the number of contestants on the same row as $s_i$ who are (strictly) under $i$ years old;
- $v_i$ be the number of contestants on the same column as $s_i$ who are (strictly) under $i$ years old.

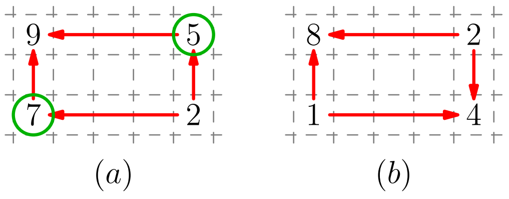
The numbers $u_i$ and $v_i$ can be easily computed in $\mathcal{O}(n^2)$ after sorting the contestants by age, by iterating on them in increasing order of age and keeping track of how many contestants in each row and column we have already considered.

Let $A$ be the answer to the problem, i.e. the number of ways to choose four contestants so that the poles do not cross. We claim that

$$2A = \sum_{i=1}^{n^2} [u_i(n-1-v_i) + (n-1-u_i)v_i].$$

This gives a solution to the problem with overall time complexity $\mathcal{O}(n^2 \log n)$, or $\mathcal{O}(n^2)$ if we use counting sort (which is feasible since the ages are in the range $[1, n^2]$). Let us now justify the claim.

Given an axis-aligned rectangle, we say that a contestant $s$ standing on one if its vertices is *intermediate* if exactly one of the two contestants on vertices adjacent to $s$ is younger than $s$. Rectangles in which the poles don't cross can be characterised by the fact that exactly two of the contestants on their vertices are intermediate (see figure (a) below for an example). On the contrary, rectangles in which the poles do cross have no intermediate contestants (figure (b)).



$(a)$ $(b)$

The right-hand side of the equation is the sum over all contestants $s$ of the number of rectangles for which $s$ is intermediate. Therefore, by the previous remark, this summation equals twice the number of rectangles with non-crossing poles, i.e. twice the answer to the problem.

# $\boxed{\text{C}}$ Il Derby della Madonnina

Author: Gerard Orriols
Preparation: Gerard Orriols

Let $x_i := vt_i - a_i$ and $y_i := vt_i + a_i$ for $i = 1, \ldots, n$. The main observation to solve this problem is that a sequence of kicks with indices $i_1, \ldots, i_k$ can be seen (in this order and starting from the first one) if and only if both sequences $x_{i_1}, \ldots, x_{i_k}$ and $y_{i_1}, \ldots, y_{i_k}$ are nondecreasing. To see this, observe that

$$x_i \le x_j \iff vt_i - a_i \le vt_j - a_j \iff a_j - a_i \le v(t_j - t_i)$$

and

$$y_i \le y_j \iff vt_i + a_i \le vt_j + a_j \iff a_i - a_j \le v(t_j - t_i).$$

Moreover

$$t_i = \frac{x_i + y_i}{2v},$$

so if both sequences are nondecreasing, then their corresponding times are in increasing order and it holds that $|a_{i_{\ell+1}} - a_{i_\ell}| \le v(t_{i_{\ell+1}} - t_{i_\ell})$, which means that each event can be reached from the previous one. The converse is clear from the previous inequalities.

In order to impose the condition that the events can be reached starting from position 0 at time 0, it is enough to remove all points which cannot be reached from the origin, that is, with $|a_i| > vt_i$. We will assume that these events have been eliminated and still denote by $n$ the total number of events.

We have thus reduced the problem to finding the longest increasing subsequence of the $y$-values when ordered by increasing $x$-value. More precisely, let $(p_1, \ldots, p_n)$ be the permutation of $(1, \ldots, n)$ such that

$$i < j \implies x_{p_i} < x_{p_j} \text{ or } (x_{p_i} = x_{p_j} \text{ and } y_{p_i} < y_{p_j}).$$

Then the solution of the problem is the length of the longest (non-strictly) increasing subsequence of the list $y_{p_1}, \ldots, y_{p_n}$.

There are classical algorithms to solve this efficiently in $\mathcal{O}(n \log n)$ time. For instance, this complexity is achieved by an approach that processes the elements from left to right and uses binary searches to update the value of the smallest possible last element of a length-$k$ increasing subsequence in every prefix for every $k$.

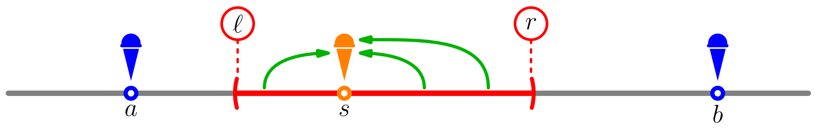
# $\boxed{\text{D}}$ Ice Cream Shop

Author: Stefanie Zbinden
Preparation: Pedro Paredes

Suppose there are ice cream shops located $a$ and $b$ meters to the right of the first hut, such that $a < b$. If we place our ice cream shop in any location in the interval $[a, b]$ it can only be strictly closer to huts located in the interval $(a, b)$, since any hut located at or before $a$ will be closer to the ice cream shop located at $a$ and any hut located at or after $b$ will be closer to the ice cream shop located at $b$. We now present two approaches to solve the problem starting from this observation.

## First solution

Let $a < b$ be the positions of two ice cream shops, and let us further assume that there is no other ice cream shop between $a$ and $b$. Of course, it is never convenient to put the new shop at $a$ or $b$. What happens if we place our new shop at $s \in (a, b)$? It is easy to see that the only huts whose people will buy ice creams are those in the open interval $\left(\frac{a+s}{2}, \frac{s+b}{2}\right)$. Notice that this interval has length $\frac{s+b}{2} - \frac{a+s}{2} = \frac{b-a}{2}$, i.e., half the length of $(a, b)$, which is independent of $s$.



On the other hand, consider an interval $(l, r)$ such that $a \le l < r \le b$ and $r - l = \frac{b-a}{2}$. If we choose $s = 2l - a$, the interval $\left(\frac{a+s}{2}, \frac{s+b}{2}\right)$ is exactly $(l, r)$. This means that for every interval $(l, r)$ of length $\frac{b-a}{2}$ which lies entirely in $(a, b)$, it is possible to achieve a total number of ice creams given by

$$f(l, r) := \sum_{100i \in (l, r)} p_{i+1}.$$

(Recall that the position of the $i$-th hut is $100(i - 1)$.)

Thus the answer is the maximum of $f(l, r)$, over all intervals such that there exist $a$ and $b$ as above and $r - l \le \frac{b-a}{2}$. Also notice that we can restrict our search to the intervals with $l = 100k - \frac{1}{2}$ for some $0 \le k \le n - 1$. We can therefore proceed as follows.

- Iterate over $k = 0, \dots, n - 1$.
- For each $k$, find the greatest $x_j < 100k$. This can be done in amortized constant time — or via binary search — after sorting the $x_j$'s.
- Compute $f(l, r)$, where $l = 100k - \frac{1}{2}$ and $r = \min\left(x_{j+1}, l + \frac{x_{j+1}-x_j}{2}\right)$ (assume the $x_j$'s are sorted). This is easy to do having precomputed the prefix sums of $p_1, \dots, p_n$.
- Return the maximum of all these values.

One must be careful with "boundary conditions", that is, values of $k$ such that $100k$ lies to the left or to the right of all ice cream shops. These are best handled by introducing two dummy shops at positions $-\infty$ and $+\infty$.

The solution has $\mathcal{O}(n + m \log m)$ time complexity and $\mathcal{O}(n + m)$ space complexity.

## Second solution

Pick again two ice cream shops at $a < b$, with no other ice cream shops in between, and consider a hut located at $h$ such that $a < h < b$. If we place our shop in $s$ such that $s$ is between $a$ and $h$, then we will always be closer to $h$ than the ice cream shop located at $a$. However, we are only closer than the ice cream shop at $b$ if the distance from $s$ to $h$ is smaller than the distance from $h$ to $b$. Formally this means $h - s < b - h$. If we rearrange this expression, we conclude that we will be closer to $h$ if $s > 2h - b$. Analogously, if we place our shop in $s$ such that $s$ is between $h$ and $b$, we will be closer to $h$ if $s < 2h - a$. Hence, we conclude that if we want to be closer to $h$ then we have to place our shop in the interval $(2h - b, 2h - a)$ (note that this is an open interval since we want to be strictly closer to $h$).

Using this idea, we can now compute for every hut $h$ an interval such that if we place our shop in any point in that interval then our shop will be the closest to hut $h$. Let us assume we have all these intervals, how do we determine the optimal solution from this information? Notice that the optimal solution is a point that maximizes the sum of the $p_i$ of all the intervals it is contained in. We can find this maximum sum efficiently using a *sweep-line* approach. Picture a vertical line that is going to sweep through all of these intervals from left to right. As this line moves we want to keep the sum of the $p_i$ corresponding to the intervals it currently intersects, so we store that value in a counter and whenever the line reaches the starting point of an interval we add its value to the counter, conversely whenever it reaches the endpoint of an interval we subtract its value from the counter. The solution is the maximum value the counter achieves during this process.

To implement this efficiently, the only thing we need to observe is that the only relevant "events" to this procedure are the start- and endpoints of intervals. So our sweep-line algorithm can be implemented as follows: for every interval $(l_i, r_i)$ corresponding to hut $p_i$ add a point $(l_i, p_i)$ and a point $(r_i, -p_i)$ to a list of points. Sort this list according to the first component. Iterate through the sorted points and store a counter (initially 0). Whenever we reach a point $(a, b)$ add $b$ to the counter. This requires time $\mathcal{O}(n \log n)$ since we have to sort all of the $2n$ "event" points and then iterate through them once.

Before doing that, however, we need to compute each hut's interval. To do so we have to determine the closest ice cream shops $a$ and $b$ such that $a < h$ and $b > h$ and then the interval is given by $(2h - b, 2h - a)$. If we sort the $x_i$ locations of all the ice cream shops we can find this using *binary search* or by iterating through the huts and ice cream shops simultaneously (this is also known as a *two pointers technique*). The time complexity of the binary search method is $\mathcal{O}((m + n) \log m)$, since we first sort the $x_i$ and then for each hut we perform one binary search. The time complexity of the two pointers method is $\mathcal{O}(m \log m + n)$, since we first sort the $x_i$ and then iterate through all the $n$ huts and $m$ ice cream shops once.

Combining all of this we get a solution that runs in $\mathcal{O}(m \log m + n \log n)$ time.

# $\boxed{\text{E}}$ Evolution of Weasels

Author: Stefanie Zbinden
Preparation: Stefanie Zbinden

To solve the problem we need the following observations.

- Every mutation is reversible. Hence, instead of trying to find a sequence of mutations of the string $u$ to get to the string $v$, we can try to find a sequence of mutations of the string $u$ and a sequence of mutations of the string $v$ such that both of them are the same after the mutations.
- In all possible mutations, we never change the parity of the occurrence of a character. Thus, if the parity of the occurrences of A, B and C is not the same in $u$ and $v$, there does not exist a sequence of mutations to get from $u$ to $v$. For the rest of the solution we assume that the parities are the same in both strings.

We can transform the string AB to the string BA via the following sequence of moves: start with AB, then insert BB at the back of the string to get ABBB, then insert the string ABAB in the second to last position to get ABBABABB. Removing the two occurences of BB we get the string AABA and then removing AA we get to BA. A similar trick can be done to transform the string BC to the string CB. While this series of steps might seem magical, viewing the problem as a group theoretic problem (see a later section) gives us this observation almost for free.

The observation above tells us that we can move the letter B to any position we want and can essentially ignore it; let $u'$ and $v'$ be the strings we obtain from $u$ and $v$ when removing all occurrences of B. When ignoring the letter B, the strings ABAB and BCBC are the same as the strings AA and CC, thus we now have strings consisting of the letters A and C and we can add and remove substrings of the form AA and CC. This is a much easier problem. We can iteratively remove occurences of AA or CC from $u'$ and $v'$ until no removal is possible anymore. If the strings are the same after the removal, a sequence of mutations is possible, otherwise it is not.

**Optimizing the runtime.** If we remove occurrences of AA and CC iteratively the runtime of our algorithm is quadratic (in the length of the strings), which is fast enough to pass. There exists a linear time algorithm: Keep a stack of the substring you visited but could not remove (at the beginning, this is empty). Then iterate through your string. If the letter you currently look at is the same as the last letter in your stack, delete the last letter in your stack. If not add the letter to your stack.

## Viewing the problem as a group theoretic problem

This is a different perspective on the problem and might require some knowledge about groups and their presentations to understand. In the language of groups, the problem statement can be reformulated as follows: Do the words $u$ and $v$ represent the same element in the group $G = \{\mathtt{A}, \mathtt{B}, \mathtt{C} \mid \mathtt{A}^2 = \mathtt{B}^2 = \mathtt{C}^2 = \mathtt{ABAB} = \mathtt{BCBC} = 1\}$? The words $u$ and $v$ represent the same element if and only if $w = uv^{-1}$ represents the identity in $G$. Furthermore A and B commute: $\mathtt{A}^2 = 1$ implies that $\mathtt{A} = \mathtt{A}^{-1}$ and the same holds for B and C. Thus, $1 = \mathtt{ABAB} = \mathtt{ABA}^{-1}\mathtt{B}^{-1}$, which directly implies $\mathtt{AB} = \mathtt{BA}$. Analogously, we get that $\mathtt{BC} = \mathtt{CB}$.

Now the only thing left to check is:

- Does the letter B occur evenly many times in $w$?

- After deleting all occurences of `B` from $w$, can we get from $w$ to the empty word by deleting `AA` or `CC`?

If the answer to one of those questions is no, then $w$ does not represent the identity. If both answers are yes, $w$ does represent the identity.

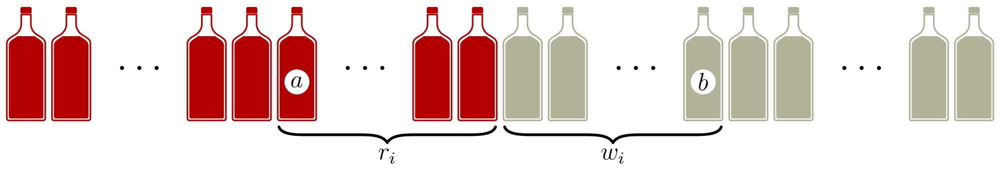
**Some background about the problem.** The problem of telling whether two words represent the same element in a group is called the *word problem*, and is a widely studied problem in Mathematics. While it is proven to be unsolvable for some groups, the group $G$ we are dealing with in this problem is a Coxeter group, where the word problem is always solvable using Tits' algorithm.

# $\boxed{\text{F}}$ Bottle Arrangements

Author: Andrea Ciprietti
Preparation: Andrea Ciprietti

Let $r_{\max}$ be the maximum number of bottles of red wines requested by one critic, i.e., $r_{\max} = \max\{r_1, r_2, \ldots, r_m\}$. Likewise, let $w_{\max}$ be the maximum number of white wines requested by one critic, i.e., $w_{\max} = \max\{w_1, w_2, \ldots, w_m\}$. Then, one can notice that, if $r_{\max} + w_{\max} > n$, there is no valid arrangement, simply because $n$ bottles are not sufficient.

On the other hand, suppose that $r_{\max} + w_{\max} \le n$. In this case, it is not difficult to produce an arrangement that satisfies all the requests: for instance, one can have the first $r_{\max}$ bottles be red wines, and the remaining $n - r_{\max}$ be white wines (note that the number $n - r_{\max}$ is nonnegative under our hypothesis). Then, if the $i$-th critic has requested $r_i \le r_{\max}$ red wines and $w_i \le w_{\max}$ white wines, they can choose the interval starting at position $a = r_{\max} - r_i + 1$ and ending at position $b = r_{\max} + w_i$ (it can be noticed that both these positions are in the range $[1, n]$).



The algorithm simply iterates over all critics, finds the values of $r_{\max}$ and $w_{\max}$, checks whether their sum does not exceed $n$ and, if that is the case, produces a valid string such as the one described.

Both the runtime and used memory are $\mathcal{O}(n + m)$.

# $\boxed{\text{G}}$ Round Table

Author: Petr Mitrichev
Preparation: Petr Mitrichev

## Main assumption

After playing a bit with the second sample test case one can notice that there are many sequences of swaps of the same length that achieve the same result, and as long as we always swap two people such that the person with a larger number is on the left before the swap, then we always achieve the goal in the minimum number of swaps.

Let us first construct a solution which relies on this assumption, and then prove that the assumption is correct. We need to find any way to arrive at the given permutation from the initial one such that in each swap the person with a larger number is on the left, and count the number of swaps needed efficiently.

## Algorithm

We will construct our permutation in steps. After the $i$-th step, the people with numbers from 1 to $i$ will be in the correct circular order, and the people with numbers from $i+1$ to $n$ will be sitting in the order of their numbers directly to the right of person $i$.

In the starting arrangement the condition above is already satisfied for $i = 2$, since there is just one circular order for two people.

To perform the $(i+1)$-th step, we need to do the following: person $i+1$ is currently sitting to the right of person $i$, but they need to be sitting to the right of some other person $j_i$ which is determined as the closest person on the left of person $i+1$ that has a number between 1 and $i$.

Since we cannot move the person $i+1$ to the left, as we cannot swap person $i$ and person $i+1$, the only way to achieve this is by moving person $i+1$ to the right. But we can't do this directly because person $i+2$ is there, so what we are going to do is move the entire block of people from $i+1$ to $n$ to the right.

Moving the block of people from $i+1$ to $n$ by one position to the right takes $n-i$ swaps, and we need to do this $d_i$ times where $d_i$ is the circular distance from person $i$ to person $j_i$ when going to the right, and only considering people with numbers from 1 to $i$.

The total number of swaps can therefore be computed as $f(p) = \sum\limits_{i=2}^{n-1} d_i(n-i)$. The numbers $d_i$ can be computed using a standard data structure such as a segment tree, a Fenwick tree, or a balanced search tree in $\mathcal{O}(n \log n)$.

## Proof of the main assumption

Now we need to repay our debts and prove the assumption. It suffices to prove the following.

**Lemma.** If a permutation $q$ differs from a permutation $p$ in only one swap of adjacent elements $a$ (on the left) and $b$ (on the right) such that $a > b$, then $f(q) - f(p) = 1$, where $f(p)$ is the number of operations used by the above algorithm.

**Proof.** Indeed, almost all terms in $f(p)$ and $f(q)$ sums will be the same. Which terms are going to differ? The quantity $d_{a-1}$ will increase by 1, since the person $a$ will now need to travel one more step to the right to overtake $b$. Conversely, $d_a$ will decrease by 1, since the person $a+1$ will now need to travel one less step to the right as they will have already overtaken $b$ earlier. All other values of $d_i$ will stay unchanged. So $f(q) - f(p) = (n - (a - 1)) - (n - a) = 1$. ■

By applying the lemma in reverse, we can also see that performing a swap where the largest element is on the right will decrease $f(p)$ by 1. So every swap either increases or decreases $f(p)$ by 1, and $f(\text{id}) = 0$ (id stands for the identity permutation), therefore $f(p)$ is indeed the smallest number of swaps needed to reach permutation $p$, and it does not matter in which order we make the swaps as long as we always make swaps where the person with a larger number is on the left.

## Parting words

This problem was inspired by the following paper, where you can learn more about this setup: *Abram, Antoine, Nathan Chapelier-Laget, and Christophe Reutenauer. "An Order on Circular Permutations." The Electronic Journal of Combinatorics (2021): P3-31.*

The paper also mentions that this setup was used for USAMO 2010, Problem 2. We hope that none of the participants have seen the paper or the USAMO problem before.

We found it quite remarkable that the answer is $\mathcal{O}(n^3)$ in magnitude but the order of swaps does not matter, just like in the non-circular problem without any restrictions on swaps where one would simply count the number of inversions in a permutation. We hope you enjoyed solving this problem, too!

# $\boxed{\text{H}}$ Pandemic Restrictions

Author: Gerard Orriols
Preparation: Gerard Orriols

Let us start with some definitions that will make the presentation clearer. Given three points $X, Y$ and $Z$ in the plane, let $f(X, Y, Z)$ be the minimum of $\overline{XP} + \overline{YP} + \overline{ZP}$ over all points $P$. It is well known that this minimum actually exists and is achieved at a unique point called the Fermat point of the triangle $XYZ$.

For fixed $X$ and $Y$, let $f_{XY}(Z) := f(X, Y, Z)$ and define

$$g(W) := \max(f_{AB}(W), f_{AC}(W), f_{BC}(W)),$$

where $A, B$ and $C$ are the positions of the three friends.

For a given parameter $r$, the condition that $Z$ is a valid residence point is equivalent to $g(Z) \le r$. Thus, no valid residence point exists if and only if $\min g > r$, and therefore the minimum value of $r$ that works is equal to the minimum of $g$ over all points of the plane.

We claim that $g$ is convex, hence this minimum exists and we can find it efficiently. This relies on the following two observations:

- The pointwise maximum of a finite set of convex functions is convex.
- The function $f_{XY}$ is convex.

The first observation is a standard fact. For the second observation, let $X, Y$ be two points, $0 \le t \le 1$, and let $P, Q$ be the respective Fermat points of the triangles $ABX$ and $ABY$. Then, denoting $Z = tX + (1-t)Y$ and $R = tP + (1-t)Q$, we have that

$$\overline{AR} = |A - tP - (1-t)Q| \le t\overline{AP} + (1-t)\overline{AQ},$$
$$\overline{BR} = |B - tP - (1-t)Q| \le t\overline{BP} + (1-t)\overline{BQ},$$

and

$$\overline{ZR} = |t(X - P) + (1-t)(Y - Q)| \le t\overline{XP} + (1-t)\overline{YQ},$$

so adding up the three inequalities we get that

$$f_{AB}(Z) \le \overline{AR} + \overline{BR} + \overline{ZR} \le t f_{AB}(X) + (1-t) f_{AB}(Y).$$

In view of these facts, if we know how to compute $f(X, Y, Z)$ efficiently we can solve the problem with a double ternary search on $g$. There are ways to achieve this in $\mathcal{O}(1)$ time by distinguishing two cases:

- If one of the angles of triangle $XYZ$ is greater than $2\pi/3$, then the corresponding vertex is the Fermat point.
- Otherwise the Fermat point can be computed using the construction of the Napoleon triangle, or the sum of its distances can be computed more straightforwardly using the following formula:

  $$f(X, Y, Z) = \sqrt{\frac{a^2 + b^2 + c^2 + 4\sqrt{3}S}{2}}.$$

  (Here $a$, $b$, $c$ are the sidelengths and $S$ is the area of the triangle.)

This solution has a total complexity of $\mathcal{O}(|\log \varepsilon|^2)$, where $\varepsilon$ is the required precision. Alternatively, each of the three Fermat points can be computed using a double ternary search on its own, giving a total complexity of $\mathcal{O}(|\log \varepsilon|^4)$, which is also fast enough.

# $\boxed{\text{I}}$ Antennas

Author: Simon Mauras
Preparation: Michal Svagerka

Let's model the problem as an undirected graph, where the vertex $i$ corresponds to the antenna $i$ and vertices $i$ and $j$ are connected with an edge if and only if the corresponding antennas are able to communicate directly, that is $|i-j| \le \min(p_i, p_j)$. In this formulation, the answer to the question posed in the problem statement is simply the shortest distance between $a$ and $b$.

To compute the shortest distance, we would like to be able to construct the graph and run a breadth-first search. Unfortunately, we cannot afford to do that, as the number of edges can be in the order of $n^2$. We need a more efficient approach. We examine two options.

## Improving the BFS

The inefficiency of a breadth-first search on a dense graph lies in the fact that the majority of the traversed edges lead to a vertex that has already been visited. Ideally, for every vertex, we would like to process only one of the incoming edges. Should we achieve this property, the running time would be linear in the number of vertices, not in the number of edges.

Let's decompose each edge into two directed edges, one in each direction. As soon as we traverse the first edge incoming to a particular vertex, we remove all other edges incoming to this vertex. This idea leads to the desired property of only processing each vertex once, nevertheless on its own it doesn't affect the time complexity as we still generate the full graph.

Let's consider for a brief moment how can we generate the graph in the first place. A naïve approach is to consider, for a fixed $i$, all $1 \le j \le n$ and check whether $i$ and $j$ can communicate with each other. However, we can observe that we do not need to test all $j$'s up to $n$ — it is sufficient to only iterate up to $\min(n, i+p_i)$, because $|i-j| = j-i \le p_i$ must hold. Similarly, a tighter lower bound for the iteration is $\max(1, i-p_i)$ instead of $i$.

We can do even better than that. Define $l_j := j - p_j$. For $j \in (i, i+p_i]$ to be able to communicate with $i$, we just need $l_j \le i$. Similarly, for $j \in [i-p_i, i)$ we need $r_j := j + p_j \ge i$. To summarise, we need to find all $j \in [i-p_i, i)$ such that $r_j \ge i$ and all $j \in (i, i+p_i]$ such that $l_j \le i$. We can do this by storing the values $r_j$ and $l_j$ in two segment trees that return the minimum and maximum on a queried interval, respectively, together with the index where this minimum or maximum is achieved, and querying the appropriate interval as long as the inequality $r_j \ge i$ or $l_j \le i$ holds. Now, recall the previous idea — removing all edges incoming to a vertex once we find its distance from $a$. In this implementation, we can achieve this in $\mathcal{O}(\log n)$ time by simply setting the corresponding $l_j$ to $\infty$ and $r_j$ to $-\infty$.

Using these techniques, we can process a vertex in $\mathcal{O}(d \log n)$, where $d$ is the out-degree of the vertex when we are processing it. Since, for each vertex, we only process at most a single incoming edge, the total running time is $\mathcal{O}(n \log n)$.

## Iterating on a restricted problem

Let us consider a restricted version of the problem. In this version, antenna $i$ is allowed to send a message to $j$ if $|i-j| \le \min(p_i, p_j)$ and $i < j$. In other words, a message can only be sent to an antenna with a strictly larger index, and never in the opposite direction. This problem can be solved by a single sweep, as follows.

Maintain a segment tree $S$ of size $n$, that returns the minimum on an interval, and supports single element update. We process the antennas in order of increasing index. When processing an antenna

$i$, we do the following operations:

- Set $S[i] = \text{dist}[i]$ where $\text{dist}[i]$ is the length of the shortest known path from $a$ to $i$. Initially $\text{dist}[a] = 0$, and all other values are initialised to $\infty$.
- For all antennas $j$ such that $j + p_j - 1 = i$ (i.e. $j$ is able to reach $i - 1$ but not $i$), set $S[j] = \infty$.
- Set $\text{dist}[i] = 1 + S.\text{min}(i - p_i, i)$.

The purpose of removing an antenna by setting $S[j] = \infty$ is evident — as the antennas are processed in order of increasing indices, we know that none of the antennas to be processed can communicate with $j$. The complexity of this algorithm is $\mathcal{O}(n \log n)$. Note that it is trivial to modify this algorithm for a version of the problem where all communication goes to antennas with a *lower* index instead.

How does this help to solve our original problem? We can alternate between the two versions — the one that only allows communication towards antennas with larger indices (i.e. to the right), and the one with smaller ones (i.e. to the left). This process is repeated while the array dist is changing. Once a fixed state is reached, we have a correct solution.

We argue that only $\mathcal{O}(\log n)$ iterations of this outer loop will be performed, in other words there exists a shortest path that "changes direction" at most $\mathcal{O}(\log n)$ times. The basic idea of the proof is that when two successive direction changes are performed, the distances between antennas at least doubled, otherwise there would exist a path of a shorter or equal length with fewer direction changes. The details of the proof are left as an exercise to the reader.

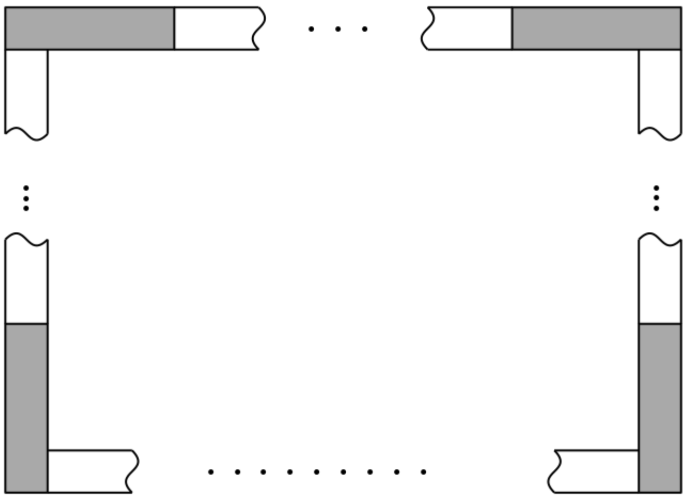
All things considered, this yields an $\mathcal{O}(n \log^2 n)$ algorithm, which is sufficient given a careful implementation.

# J Boundary

Author: Michal Svagerka
Preparation: Michal Svagerka

First, observe that we can tile any rectangle with tiles of size $1 \times 1$. From now on, we will consider a fixed $a > 1$.

Let's investigate what happens in the corners of the rectangle. Each corner is covered either by a tile that is placed horizontally, or by a tile that is placed vertically. Below is an example of one of $2^4 = 16$ possibilities, with the corner tiles in gray.



Once we place those tiles (up to 4, fewer if the tile length is greater than $\frac{w}{2}$ or $\frac{l}{2}$) the positions of the rest are uniquely determined.

We can now put four constraints on the value of $a$, one for each side of the rectangle. These constraints are of form $a \mid w - x$ or $a \mid l - x$, where $\mid$ is the "divides without remainde" relation, and $x$ is the number of positions on that side that are covered by a tile placed in a perpendicular direction, e.g. a vertical tile on a horizontal side. Clearly, this can only happen in the corners of the rectangle, hence $x$ is between 0 and 2 for all sides.

In the above figure, the top side yields the constraint $a \mid w - 0$, the bottom one $a \mid w - 2$, and both the left and right sides $a \mid l - 1$. Put together, we have a necessary condition $a \mid \gcd(w - 0, w - 2, l - 1, l - 1)$, where gcd is the greatest common divisor. Clearly, the condition is also sufficient, i.e. for an $a$ fulfilling this divisibility there is a rectangle tiling.

To summarise, we consider all 16 possibilities for the tile orientations in the corners, find the gcd of the constraints using Euclidean algorithm, and then compute its divisors by a trial division. Finally, we return the sorted set of all the divisors. The total time complexity is $\mathcal{O}(\sqrt{\min(w, l)})$ per test case, dominated by the trial division.

Note that some of the 16 cases are the same up to mirroring or rotation, and some are clearly impossible. Hence we only need to factor the following values:

- $\gcd(w - 1, l - 1)$
- $\gcd(w, l - 2)$

- $\gcd(w-2, l)$
- $\gcd(w-1, l-2, l)$
- $\gcd(w-2, w, l-1)$

The last two can only be 1 or 2, and if they are both 1, this implies that one of the first three gcds is divisible by 2. Therefore $a = 2$ is always a solution — we can simply insert it to the solution set, and only compute and factor the first three gcds. This does not change the asymptotic time complexity, nevertheless it is a useful optimisation.

# $\boxed{\text{K}}$ Gastronomic Event

Author: Cesc Folch
Preparation: Cesc Folch

This is a classical problem in which one must find a "simple" characterization of what the optimal solution looks like, and then restrict the search for the maximum to the (much smaller, much more regular) family of instances that satisfy the characterization.

In our specific case, the first part of this paradigm is by far the most difficult and the one that requires all the important insights, while the second part can be solved through the application of a standard technique. We are going to split the core of the solution in two parts, the first of which deals with the characterization of optimal solutions. Then, in the second, we will explain how to compute the answer efficiently in the restricted search space and we will discuss some improvements and further considerations.

## Preliminary observations and notation

First, however, let us set up the bases. It goes without saying that we are dealing with a graph — actually, a tree — problem. Also, the statement is deceptively straightforward: assign a permutation of $\{1, \ldots, n\}$ to the vertices so as to maximize the number of increasing paths.

And here comes the first non-trivial (however simple) idea: we are actually going to consider ways to *orient* the edges of the tree, and find an orientation that maximizes the number of *directed paths*; we forget about the vertex numeration altogether. After all, an assignment of the numbers $1, \ldots, n$ naturally induces an orientation of the edges, where edge $(u, v)$ points towards $v$ if $u < v$ and towards $u$ otherwise. This begs the question of whether the converse is also true, i.e., does an orientation of the edges "induce" a numbering of the vertices? The answer is, in general, no. Nevertheless, we would still be happy if it were true that **every orientation is induced by at least one numbering** (for this would mean that, in switching to the orientation perspective, we wouldn't be considering forbidden configurations). Luckily, it is, and it follows immediately from the existence of a topological sorting of the vertices of a directed tree (which is, in fact, a DAG): just number the vertices in their topological order! (Incidentally, note that *any* topological order works, and since it is generally not unique, neither is the numbering.)

From now on, we shall call a *path* any path in the undirected tree, as opposed to *directed path* which refers to the underlying orientation of the edges (supposing we have fixed one in the context of the sentence). A single vertex is both a path and a directed path. Given an orientation of the tree, we call its *value* the number of directed paths it produces; therefore, the problem asks to maximize the value of an orientation. We say that a (directed) path has *length* $\ell$ if it is made up of $\ell$ vertices. Finally, we denote $d(u, v)$ the distance between $u$ and $v$ in the undirected tree.

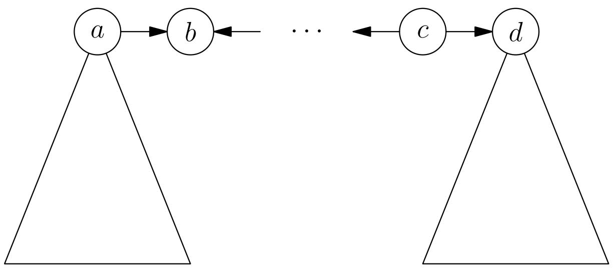
## Characterizing the optimal solutions

The next two results are crucial.

**Lemma 1.** Fix an orientation of the edges. Suppose there exist four distinct vertices $a$, $b$, $c$ and $d$ such that (refer to the figure below for clarity):

- there are edges $a \to b$ and $c \to d$;
- there is a directed path from $c$ to $b$.

Then, we can produce an orientation with a higher value by either reversing the edge $a \to b$ and all the edges in the subtree rooted at $a$ which does not contain $b$, or by reversing the edge $c \to d$ and all the edges in the subtree rooted at $d$ which does not contain $c$.



**Proof.** Let's see what happens, in terms of value, when we reverse edge $a \to b$ and the subtree rooted at $a$. Let:

- $A$ be the number of directed paths (in the initial orientation) ending at $a$, including the path of length 1;
- $B_{\text{out}}$ be the number of directed paths of length $\geq 2$ starting at $b$;
- $B_{\text{in}}$ be the number of directed paths of length $\geq 2$ ending at $b$ that do *not* go through edge $a \to b$.

When inverting the edges, most paths stay the same (or are inverted as a whole). It isn't hard to see that only two kinds of paths change in nature (that is, either become or cease to be directed). The directed paths starting at a vertex in the subtree rooted at $a$, going through edge $a \to b$, and ending at some vertex $u \neq b$, are no longer directed after the inversion; there are $A \cdot B_{\text{out}}$ of these. However, the (non-directed) paths of the form

$$v \to \cdots \to b \leftarrow a \leftarrow \cdots \leftarrow w,$$

where the first part has only right-edges, the second part has only left-edges, and $v \neq b$, become directed. There are $A \cdot B_{\text{in}}$ of these. Adding up the contributions, we get that the value difference is $\Delta_1 = A(B_{\text{in}} - B_{\text{out}})$.

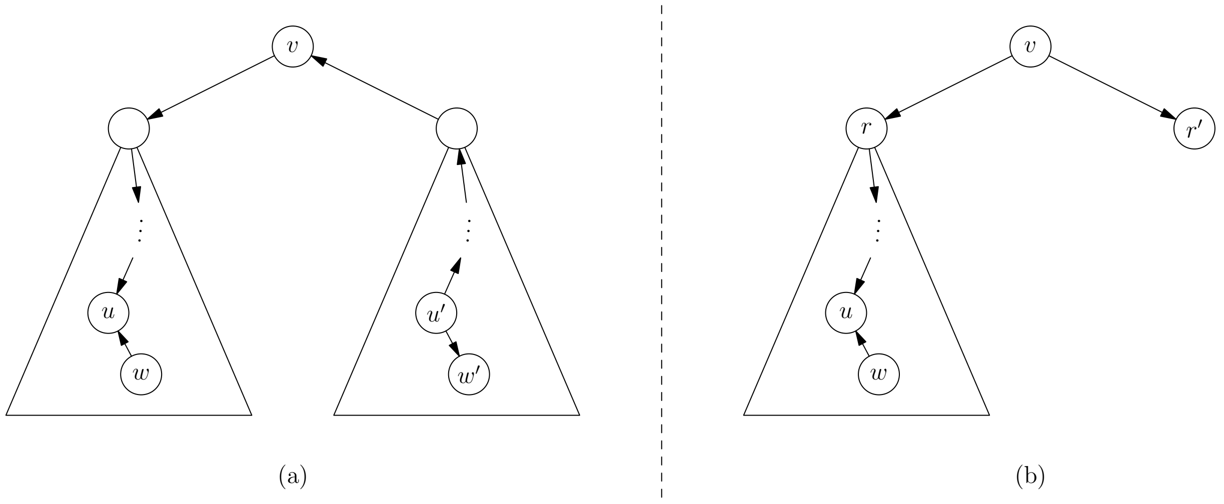
With similar reasoning and notation, if we invert the edge $c \to d$ and the subtree rooted at $d$, the vaue increases (or decreased) by $\Delta_2 = D(C_{\text{out}} - C_{\text{in}})$. Now, since every directed path beginning at $b$ can be extended to the left until we get to $c$ (i.e. we can pre-append the path $c \to \cdots \to b$), and since $c \to \cdots \to b$ is itself a path beginning at $c$, it holds $C_{\text{out}} \geq B_{\text{out}} + 1$. Likewise, we can prove $B_{\text{in}} \geq C_{\text{in}} + 1$. Thus, $(B_{\text{in}} - B_{\text{out}}) + (C_{\text{out}} - C_{\text{in}}) \geq 2$ and without loss of generality $B_{\text{in}} - B_{\text{out}} > 0$, which implies, since $A > 0$, that $\Delta_1 > 0$ as desired. ■

From Lemma 1 it follows that, in an orientation that yields the optimal solution, every path "changes direction" at most once. Formally: given a path $(v_1, v_2, \ldots, v_k)$, there exists at most one index $2 \leq i \leq k-1$ such that the path $(v_{i-1}, v_i, v_{i+1})$ is not directed. This condition is equivalent to a very simple property which opens up the way to an efficient algorithm to find the solution.

**Lemma 2.** Suppose a directed tree satisfies the aforementioned condition. Then, there exists a (not necessarily unique) vertex $v^*$ such that all paths having $v^*$ as an endpoint are also directed paths.

**Proof.** We shall start by fixing an arbitrary vertex $v$ and rooting the tree at $v$. Call a vertex $u \neq v$ *evil* if it has a child $w$ such that the path $(v, \ldots, u, w)$ changes direction at $u$. If there are no evil vertices, $v$ is a valid candidate for $v^*$ and we are done.

Otherwise, all evil vertices must lie in the same $v$-subtree. Indeed, if $u$ and $u'$ were two evil vertices belonging to different subtrees, and $w$, $w'$ the respective children, the path $(w, u, \ldots, v, \ldots, u', w')$ would change direction at least twice (see figure below, (a)). Moreover, if $u$ is an evil vertex and $r$ is the root of its subtree, the edge $(v, r)$ must point in a different direction than all other edges incident on $v$. To see why, suppose without loss of generality $v \to r$ and $v \to r' \neq r$; then the path $(w, u, \ldots, r, v, r')$ changes direction twice (see figure below, (b)).



Thanks to the previous observations, if we replace $v$ with $r$ the set of evil vertices decreases or stays the same. If we repeat this argument while there are evil vertices, we will produce a downward path in the tree rooted at $v$, which must stop eventually, at which point the number of evil vertices must vanish. ■

Thanks to Lemmas 1 and 2 combined, we know that any optimal orientation admits the existence of a vertex $v^*$ satisfying the property of Lemma 2. This makes us finally able to tackle the problem algorithmically.

## Computing the maximum

Call a vertex $v^*$ that satisfies the property of Lemma 2 a *crossroad.* We are going to iterate over all $n$ vertices and find the maximum value of an orientation where said vertex is a crossroad.

Thus, fix a vertex $v^*$ and consider the $\deg(v^*)$ subtrees the tree gets split into if we erase $v^*$. In an optimal orientation, in each subtree all edges point either "inwards" or "outwards": let $S$ be the sum of the sizes of subtrees of the first kind, and $T$ be the sum of the sizes of the subtrees of the second kind. Of course, $S + T = n - 1$. Let's calculate the value of such an orientation.

- There are $n$ directed paths having $v^*$ as an endpoint.
- The directed paths lying entirely in one subtree are $\sum_{v \neq v^*} d(v^*, v)$.
- Finally, the paths starting in a subtree of the first kind and ending in a subtree of the second kind contribute with an additional $S \cdot T$ term.

Observe that the first two terms are independent of $S$ and $T$. It is trivial to compute the sizes of the subtrees in $\mathcal{O}(\deg(v^*))$ after preprocessing. Furthermore, the sum of distances can be updated in $\mathcal{O}(1)$ when moving from $v^*$ to one of its neighbors. As for the third term, it is clear that it is maximized when $S$ and $T$ are as close as possible to $\frac{n-1}{2}$. If one of the subtrees has size $k \geq \frac{n}{2}$

(there can only be one of those), the way of doing so is trivial: just orient the edges of the big subtree one way, and all other edges the other way, so that $ST = k(n-1-k)$.

The matter is more complicated if all subtrees have sizes smaller than $\frac{n}{2}$. In this case, $v^*$ is a centroid of the tree. Recall that any tree has exactly one or two centroids — and the latter can only occur if $n$ is even. What we are dealing with is an instance of the knapsack problem: we are given positive integers $k_1, \ldots, k_{\deg(v^*)}$ — the sizes of the subtrees — whose sum is $n-1$, and we shall find a subset whose sum is as close as possible to $\frac{n-1}{2}$. The naïve dynamic programming solution runs in $\mathcal{O}(n^2)$ at worst (with $\mathcal{O}(n)$ space), way too slow for our purposes. We can optimize the runtime, reducing it by a factor 32, via the use of bitsets. This is still not enough. There is a standard, yet somewhat advanced, trick to speed the algorithm up to $\mathcal{O}\left(\frac{n\sqrt{n}}{\text{sizeof(int)}}\right)$. We won't go over it here, but you can find a complete tutorial in this Codeforces blog entry (under *Subset Sum Speedup 1*), together with a number of other interesting related techniques. Since we only need to run this algorithm at most twice, this final optimization, carefully implemented, does the trick.

Finally, let's comment further on the nature of the characterization. So far, we know very little about the crossroad $v^*$, but intuitively it makes sense that it should be "in the middle" of the tree in order to yield an optimal solution.

**Lemma 3.** There is an optimal solution where at least one centroid of the tree is a crossroad.

**Proof.** Consider an optimal orientation where a vertex $v^*$, which is not a centroid, is a crossroad. Let $\tau$ be the subtree of size $\geq \frac{n}{2}$. We have already observed that all edges of $\tau$ must point (WLOG) toward $v^*$, and all other edges away from $v^*$. Now let $v$ be the neighbor of $v^*$ belonging to $\tau$. If we make $v$ a crossroad, by inverting edge $v \to v^*$ and keeping the orientations of all other edges, a simple computation shows that the value of the new configuration has not decreased. We can therefore iterate until we reach a centroid. ■

Although this result is not necessary to solve the problem, it is relatively easy to guess and prove, and it simplifies the implementation. One can even prove this slightly stronger conclusion: for each centroid, there is an optimal orientation in which it is a crossroad.

[It was briefly considered whether to give this problem with $n \leq 10^7$. It turns out that the "slow" part of the algorithm is *not* the optimized knapsack, but rather the tree traversal needed to find the centroid. To fit the new constraint, one must come up with a way to do this without any traversal. Taking a look at the input format might be a good starting point...]

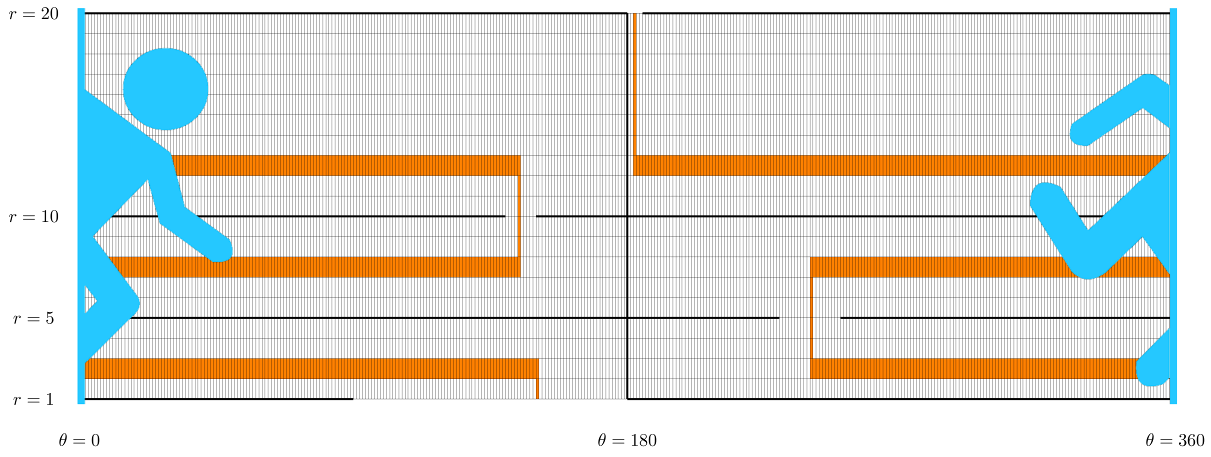
# $\boxed{\text{L}}$ Circular Maze

Author: Simon Mauras
Preparation: Simon Mauras

In this task, you are given circular mazes described by collections of walls, which can be either circular (constant radius) or straight (constant angle). The goal is to determine if a maze can be solved, that is, if there is a path from the center to the outside which does not cross any wall.

## Reducing the problem to a grid maze

The first remark to be made is that this task can be reformulated into the much simpler problem of finding a path in a grid maze (with the left and right sides of the grid being connected by some portal). The image bellow illustrates the grid obtained from the first input sample, where each row corresponds to a radius (between 1 and 20) and each column corresponds to an angle (between 0 and 360). A possible path solving the maze is displayed in orange.



The first part of the solution consists of building the grid maze, and fill 2D arrays with the walls. Because walls do not overlap, it is sufficient to iterate over each wall (number of operations is in the order of $t \cdot 20 \cdot 360 \leq 10^6$). Observe that with overlapping walls, the naïve implementation might be too slow (worst case is $t \cdot n \cdot 360 \geq 10^8$), but one could compute cumulative sums of arrays containing the endpoints of walls.

Finally, we build the undirected graph where each cell is connected to the neighbouring cells such that there is no wall in between.

## Solving the grid maze

Now that we have a grid maze, where each cell is connected to up to 4 neighbours, we need to determine if there is a path between the inside and the outside of the maze. The only remaining detail is to describe how we deal with the inside (below radius $r = 1$) and the outside of the maze (above radius $r = 20$). Multiple approaches exist:

- Adding two "sentinel" rows of cells to the maze, one at the bottom (inside) and one at the top (outside). Because there are no walls in sentinel rows, it is enough to pick (arbitrarily) one cell from each row and check if there is a path between them.
- Adding two special cells, one for the inside and one for the outside, connected (when there is no wall) to cells from the bottom and the top row. It is now enough to check if there is a path

between these two special cells.

Finally, checking if there is a path between two cells can be done with various algorithms computing the **connected components** of a graph.

- Depth-first search: the procedure builds the component of a starting cell by exploring (recursively) adjacent cells, making sure it does not explore the same cell twice.
- Breadth-first search: the procedure builds the component of a starting cell by exploring (iteratively) cells at distance 1, then 2, etc.
- Disjoint-Set Union data structure: starting from a collection of singleton sets (each containing one cell), the data structure can merge sets (containing two neighbouring cells). The resulting collection of sets corresponds to connected components.