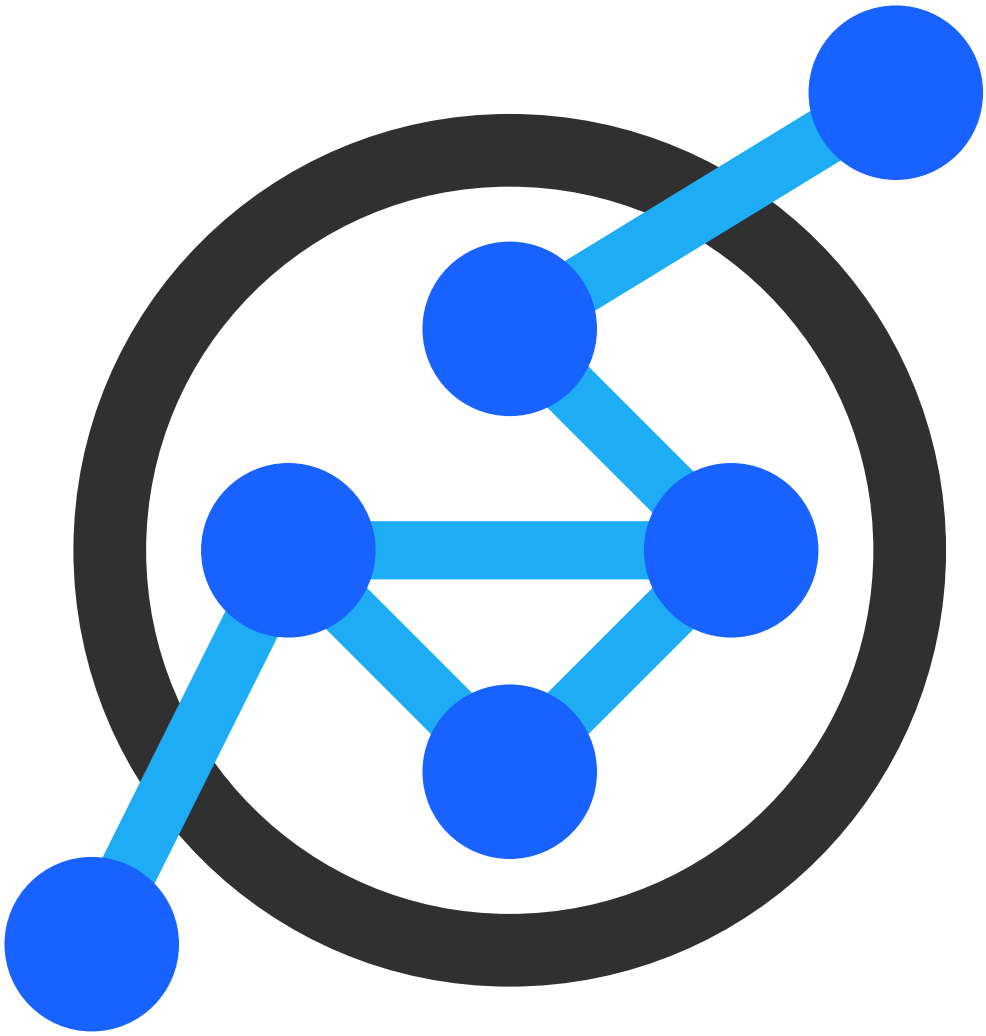
Second Round Theoretical

Solutions



Swiss Olympiad in Informatics

March 4, 2023

Goulash

Task Idea	Charlotte Knierim
Task Preparation	Charlotte Knierim
Description English	Charlotte Knierim
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Charlotte Knierim, Johannes Kapfhammer
Solution English	Charlotte Knierim

Subtask 1: Example

You can get all the ingredients from the basement in 12 minutes. For the first trip, all three mice go together, carrying up items $\{0, 1, 7\}$. This takes 3 minutes. After that Mouse 0 and Mouse 2 go to the basement again, carrying up items $\{2, 3\}$ and 6 minutes have passed since the start. Then we are left with the items 4 and 6 (there is no item 5, this was not intentional but does not harm the solution to this problem in any way). Note that Mouse 2 is the only mouse that can carry these items, so two additional trips to the basement are needed, each taking 3 minutes, giving a total of $6 + 3 + 3 = 12$ minutes.

Subtask 2: When is it possible

It is possible to cook the goulash if at least one of Stofl's friends is tall enough to reach the highest item. In this case we can just send this friend to fetch all the ingredients from the basement. So our algorithm should output "*Yes*" in this case. If no one is tall enough to reach the highest item, clearly it is not possible to cook the goulash. Our algorithm should output "*No*".

The running time of this is $O(N + M)$ as we need to compute the maximum height of an ingredient and the maximum height among Stofl's friends. We only use constant additional memory in this subtask.

```
Mice = read_Stofls_friends()
Ing = read_Ingredients()


if max(Mice) < max(Ing):
    print("No")
else
    print("Yes")
```

Subtask 3: General case

In general we can reformulate this as: "How many rounds of mice going to the basement do we need to fetch all the ingredients". Once we know the number of rounds r, the answer is just $3 \cdot r$ as every round trip to the basement takes 3 min. Note that in this Subtask, we will assume there is always a solution. We can do this by running our algorithm from the previous subtask before we start, which already tells us if it was not possible. There are two general approaches how to solve this problem:

Binary search

We can approach this problem with a binary search. Let us analyse how long we need to *validate* whether the problem can be solved in r rounds for a given number $1 \leq r \leq N$.

We do this as follows: Assume the list of ingredients and the list of Stofl's friends are sorted

by height (we do this once in the beginning). If $r \cdot M < N$, clearly, there is no way to get the ingredients from the basement in r rounds. Else we can just check $I[N-1-\ell \cdot r] \leq F[M-1-\ell]$ for all $0 \leq \ell$ with $N-1-\ell \cdot r \geq 0$. Note that there are at most M such values for ℓ as we know $r \cdot M \geq N$. Thus we can check in $O(M)$ if the M friends can get all the ingredients from the basement in r rounds.

It is easy to see that this property is monotone (i.e. if we can get the ingredients in r rounds we can also get them in $r+1$ rounds). Thus we can use binary search here. As the correct value for r lies within $[1, \ldots, N]$ we can find the optimal value in $O(\log N)$ rounds of the binary search.

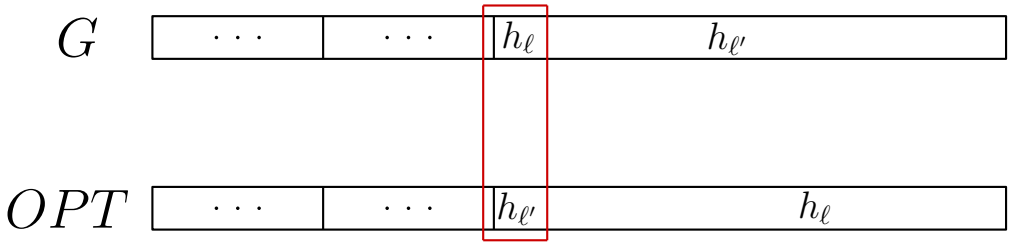
Including the sorting we do in the beginning, this gives a running time of $O(N \log N + M \log N + M \log M)$ which is dominated by $O(N \log N)$ as $N > M$.

```
def check(r):
    if N > M * r:
        return False
    else:
        for l in range(M):
            i = N - 1 - r * l
            if i >= 0 && Ing[i] > Mice[M-1-l]:
                return False
        return True

Mice = read_Stofls_friends()
Ing = read_Ingredients()

M = len(Mice)
N = len(Ing)

if not_possible():
    print("Not possible")
else:
    l_r = 0
    r_r = N

    while(l_r != r_r):
        m_r = (l_r + r_r)//2

        if check(m_r):
            r_r = m_r
        else
            l_r = m_r + 1

print(3*l_r)
```

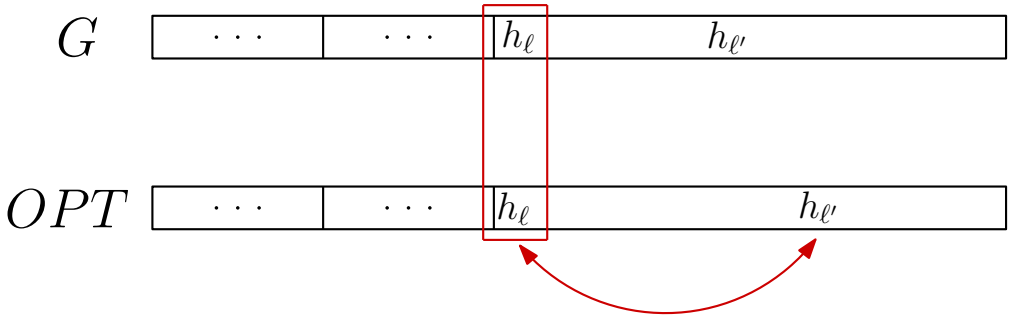
Greedy

We can solve this problem using a greedy approach. First, sort all ingredients on this list by height. Now we consider the problems in rounds. In every round every mouse j takes the item with the largest height that is less or equal than m_j. For now, let us not worry how to find these elements. Once we found them, we need can easily simulate this process to see how many rounds we need.

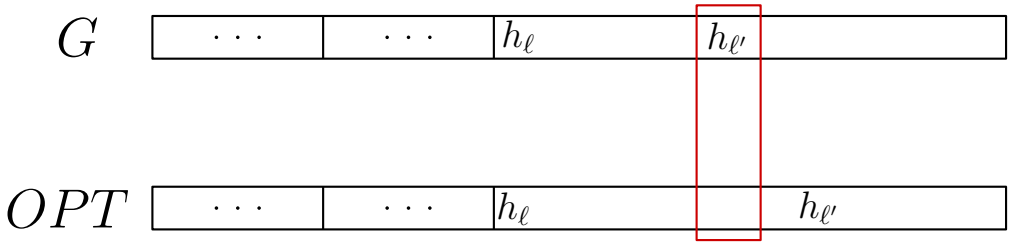
But why is this optimal? Let us assume there was a solution that is better than our greedy solution G and among all these solutions, take the one that agrees with our solution for as long as possible. We denote this solution by OPT. Let ℓ be the first time that OPT makes a choice that is different from the greedy algorithm G. Then there is a mouse m_j such that in this round the item picked by G (with height h_ℓ) is higher up than the item picked by OPT (at height $h_{\ell'}$ (as the greedy algorithm always picks the heaviest possible item).



In this case, we can surely take the heavier object instead and not make the solution worse. This is, as the mouse assigned to this ingredient is clearly tall enough to grab the higher one, and everyone who might grab the higher one later on can take the lower ingredient instead.



By this we constructed a solution that has the first difference to the greedy solution at a later time, a contradiction to the fact that we chose the one to maximize this metric.



In order to analyse the running time, we have to discuss how to find the maximum element on the list a mouse can carry. An easy approach to this is linear search. Starting at the first element we continue to go through the list until we found the first element we cannot carry. As we sorted the list in the beginning, the element before this is the heaviest element we can carry. This approach takes at most $O(N)$ time per element we carry. As we have to do this once for every element on the list we get a running time of $O(N^2)$ which dominates the time it takes to sort the list initially.

Can we do this faster? We can use **Using std::multiset** to get to a running time of $O(N \log N)$ Storing the height of the ingredients in a multiset allows us to find the highest ingredient a mouse can carry and deleting it from the list in $O(\log N)$. Using this we can simulate the greedy algorithm in $O(N \log N)$ time.

```
multiset<int> Ing = read_Ingredients()
vector<int> Mice = read_Stofls_friends()

sort(Mice.begin(),Mice.end());

int r = 0;

while(!Ing.empty)
        r++;
        for(int i = Mice.length();i>0;--i){
                auto e = Ing.lower_bound(Mice[i-1]) - 1;
```

```
                if(e == -1) break;
                Ing.erase(e)
        }

cout << 3*r <<'\n';
```

Height compression

You can realize that you do not actually need to save the height of an ingredient but instead we get the same solution by saving the height of the smallest mouse that can reach this item. This way, we only have at most M different heights and we can save a pair of (height, number of ingredients) in a set to access the next ingredient in $O(\log M)$. This gives a running time for the greedy solution of $O(N \log M)$.

The same trick brings the dominating factor of the binary search approach (sorting the ingredients by height) down to $O(N \log M + M \log M)$ for the height compression and then sorting the compressed list. We can still implement the validation step in $O(M)$, giving a total running time of $O(N \log M + M \log N)$ which is dominated by $O(N \log M)$.

All solutions without height compression need $O(N)$ additional memory as we need to sort the ingredients. If we use the height compression trick, $O(M)$ additional memory is enough as we do not need to save all the ingredients.

```python
Mice = read_Stofls_friends()
Ing = read_Ingredients()

M = len(Mice)

Mice.sort()

#using a binary search we find the smallest mouse that
#can carry the ingredient

def compress(i):
    l = 0
    r = M
    while (r != l):
        m = (r + l)//2
        if Mice[m] < i:
            l = m + 1
        else:
            r = m
    return l

#for every mouse we save for how many items
#it is the smallest mouse that can carry this item

compressed_Ing = [0 for _ in range(M)]

for i in Ing:
    compressed_ing[compress(i)] +=1


```

Jogging Map

Task Idea	Charlotte Knierim, Bibin Muttappillil
Task Preparation	Bibin
Description English	Bibin
Description German	Charlotte
Description French	Mathieu Zufferey
Solution	Johannes Kapfhammer, Ema Skottova, Charlotte, Bibin
Solution English	Bibin

Subtask 1: Solve an example

With $d(3,0) = 2$ we have a graph looking like $3 - A? - 0$. $A?$ can't be:

- 0 or 3, as they are already used
- 2 would imply $d(0,2) = 1! = 2$
- 4, would imply $2 = d(0,4) + d(4,2) \geq d(0,2) = 3$
- 5, would imply $d(3,5) = 1! = 2$

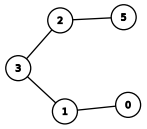
So it has to be 1.

With $d(3,5) = 2$ we have a graph looking like $3 - B? - 5$. $B?$ can't be:

- 5 or 3, as they are already used
- 0, implies cycle
- 4, has a path to 0 without 2 $\implies$ cycle

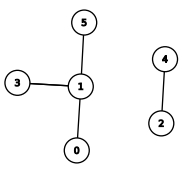
So $B?$ is either 1 or 2.

Case $B? = 2$



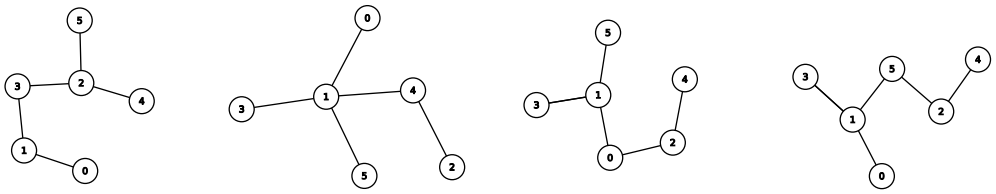
Only 4 is left, which we have to connect to 2, as $d(4,2) = 1$.

Case $B? = 1$. $d(4,2) = 1$ implies an edge between 2 and 4.



Now we need to add one more edge to connect these two components. Quickly checking all 8 possibilities we get, that we need to add either $2-3$, $2-5$ or $4-1$. All others imply an incorrect distance for $d(0,2)$.

Then we get these four graphs for this subtask:



Subtask 2: Solve the general case

The idea is to take an arbitrary root, e.g. 0 and ask all distances from it. Then we can group these vertices into or levels (or layers) by their distance. Similar to a bfs tree without edges.

Afterwards we will try to find the parent of each vertex (except the root). The parent has to lie in the level above the current vertex. So we check every one of them, by asking if their distance is 1. Additionally, we check also every vertex in our current level to look for siblings, i.e. vertex with the same parent, by asking if their distance is 2. Then we can also set the parent of our sibling.

Here is some more detailed pseudo code of the idea:

```
vector<edge> solve(int L, int Y){
    int N = L + Y;

    vector<int> non_root;  // list of all vertices expect the root = 0
    for(int v = 1; v < N; v++) non_root.push_back(v);

    // group vertices by their distance to 0, like a bfs tree, without the edges
    vector<vector<int>> level(N);
    for(int v: non_root) level[d(0, v)].push_back(v);

    vector<int> parent(N, -1);
    for(int i = N-1; i >= 1; i--){  // go through every level
        for(int l: level[i]){ // go through every vertex in this level, ...
            if(parent[l] != -1) continue;  // ...that doesn't have a parent yet

            // this line is hit number of non-leafs in level[i-1] times
            // level[i].size() <= L and level[i-1].size() <= L
            // thus the runtime of Y * L

            for(int p: level[i-1]){  // potential parents
                if(d(l, p) == 1) parent[l] = p;
            }

            for(int ll: level[i]){  // potential sibling
                if(d(l, ll) == 2) parent[ll] = parent[l];
            }
        }
    }

    // construct list of edges
    vector<edge> edge_list;
    for(int v: non_root) edge_list.push_back(edge{v, parent[v]});

    return edge_list;
}
```

Correctness: Reconstructing the map means, that we want to find all $N-1$ edges. After arbitrarily rooting the tree, the goal is equivalent of finding all parents of each vertex, except the root.

A vertex's parent is connected to one, and has a distance exactly one less than itself. There is no other vertex with these properties, as this would imply to different paths to the root, contradicting the acyclic property of a tree.

As we check every vertex in one layer above, which are all the vertices with one distance less, we can't miss the parent. And since we always check if their distance is 1, we check their connectedness.

For the number of questions:
Claim: Each level has at most L vertices. Every one of them lives in a different subtree. Every subtree has at least one leaf (e.g. by walking down, we need to end at some point). And those leafs can't be shared, otherwise it wouldn't be a tree.

Every time we find a parent in the first inner loop, it has to be a new parent. Otherwise, if we have seen the parent, then we also have to have seen a sibling, which would have set our parent. So the amount of time both inner loops are executed is the amount of potential parents in our layer above. Also, being a parent implies being not a leaf.
Thus, in total we need to run the two inner loops Y times, the number of non-leaf vertices, getting us $Y \cdot L$ as the number of questions. Plus the $Y + L$ initial questions for building the levels.

For the runtime: the first loop runs in $O(N)$ and the last loop in $O(N)$, as we have $N-1$ edges. Everything else are some constant operations proportional to the number of questions, so the runtime is also $O(Y \cdot L)$.

For memory: `non_root`, `parent` and `level` contain each vertex at most once. `edge_list` contains every edge at most once. So this means $O(N) = O(Y + L)$ memory.

Alternative Solutions

There are some solutions, where you ask every vertex v from the furthest away vertex l from the root r (note: l is a leaf). With this you can recreate the path from l to r, by checking if $d(l,v) + d(v,r) = d(l,r)$.

Other edges and vertices forms subtrees rooted at one of the vertices on the path. You also know the level i of a vertex w in such a subtree with $d(l,w) + d(r,w) = d(l,r) + 2 \cdot i$. Then you can find the furthest leaf again and recursively solve the subtrees.

You can also do a similar thing with finding the diameter, taking its edges away and solving the subtrees.

For brevity, we won't analyse them here, but usually we only managed to get them to ask $O(L \cdot (Y + L))$ questions.

Asking all queries

If you ask all queries possible, you will be asking $\binom{L+Y}{2}$ queries, as this is the number of possibilities you can choose 2 vertices out of $L + Y$. Also, note that $\binom{L+Y}{2}$ is in $O((L+Y)^2)$.

Now a and b are connected if and only if $d(a,b) = 1$. If you keep all the questions, that answer with a distance of 1, you get all the edges of the tree.

Path

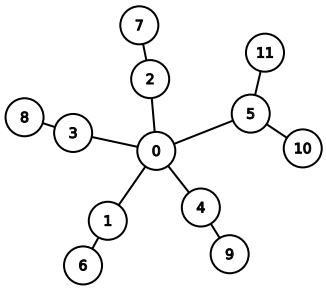
One way to solve this is to choose an arbitrary vertex a and ask all distances to a. Take a vertex l with the farthest distance, which has to be one of the leafs. Now ask the distances of all vertices from l and order them by their distance. As l is one of the leafs, every other vertex will lie on

the same side of the line. Thus, there exists an edge between two consecutive vertices in the list.

Subtask 3: Show a Lower Bound

Let $A = \lfloor \frac{N-1}{2} \rfloor$ and $B = N-1-A$. (If N is odd then $A = \frac{N-1}{2}$, $B = \frac{N-1}{2}$ else $A = \frac{N-2}{2}$, $B = \frac{N}{2}$.)

Make a graph with one (arbitrary) root, A vertices in the first level, with each of them connected to one more vertex. (Maybe two, if N is even.)



Every vertex in the second level is part of exactly one of the A subtrees, rooted at the vertices of the first level. Let s be a vertex in the second level. If we ask at most $A-2$ questions with s, then, by pigeonhole principle, there are two subtrees, where s asked no questions with any vertex on these subtrees. Note: The algorithm doesn't know yet that each vertex in the first layer has exactly one vertex other than the root adjacent.

So for the first $A-2$ questions, we answer as if the two vertices are in different subtrees. When asking the $A-1$-th question, we answer in a way, that they in the same subtree. For simplicity, we answer questions in a way such that every vertex in the second layer are in different subtrees (except for maybe 2).

Thus, every vertex s in the second layer needs to ask at least $A-1$ question to find its parent.

We then have $B \cdot (A-1)$ questions in total. Since $B \geq A \geq \frac{N-2}{2}$ we get at least $\frac{(N-2)^2}{4}$ questions which is larger than $\frac{N^2}{2023}$ for $N \geq 3$.

Subtask 4: The lost subtask

One subtask we didn't include was to find a lower bound for every L and Y, where any algorithm needs $\Omega(L \cdot Y)$ questions. (As opposed to the last subtask, not only is N fixed, but L and Y as well, so you can't choose the number of houses anymore.)

We suspect that a star-like graph, where you have one central vertex and L paths connected to it as arms, with around $\frac{Y-1+L}{L}$ length. Though it was not so clear to me, how the watch can restructure and relabel the vertices, such that you can guarantee such a worst case for any algorithm.

So I leave this challenge to the reader and if you find a solution, feel free to share it with us :)

Coding Accident

Task Idea	Timon Gehr
Task Preparation	Timon Gehr
Description English	Timon Gehr
Description German	Priska Steinebrunner
Description French	Mathieu Zufferey
Solution	Timon Gehr
Solution English	Timon Gehr

Subtask 1: Small example

Recall the example:

- $p_0 = 1$
- $p_1 = 0$
- $p_2 = 3$
- $p_3 = 4$
- $p_4 = 5$
- $p_5 = 2$
- $p_6 = 7$
- $p_7 = 8$
- $p_8 = 9$
- $p_9 = 10$
- $p_{10} = 6$

The following sequence of 5 rotations solves the example:

1. `rot3`$(0, 2, 1)$ results in
   - $\mathbf{p_0 = 0}$
   - $\mathbf{p_1 = 3}$
   - $\mathbf{p_2 = 1}$
   - $p_3 = 4$
   - $p_4 = 5$
   - $p_5 = 2$
   - $p_6 = 7$
   - $p_7 = 8$
   - $p_8 = 9$
   - $p_9 = 10$
   - $p_{10} = 6$
2. `rot3`$(2, 1, 5)$ results in
   - $p_0 = 0$
   - $\mathbf{p_1 = 1}$
   - $\mathbf{p_2 = 2}$
   - $p_3 = 4$
   - $p_4 = 5$
   - $\mathbf{p_5 = 3}$
   - $p_6 = 7$
   - $p_7 = 8$
   - $p_8 = 9$
   - $p_9 = 10$
   - $p_{10} = 6$
3. `rot3`$(3, 4, 5)$ results in
   - $p_0 = 0$
   - $p_1 = 1$
   - $p_2 = 2$
   - $\mathbf{p_3 = 3}$
   - $\mathbf{p_4 = 4}$
   - $\mathbf{p_5 = 5}$
   - $p_6 = 7$
   - $p_7 = 8$
   - $p_8 = 9$
   - $p_9 = 10$
   - $p_{10} = 6$
4. `rot3`$(6, 7, 10)$ results in
   - $p_0 = 0$
   - $p_1 = 1$
   - $p_2 = 2$
   - $p_3 = 3$
   - $p_4 = 4$
   - $p_5 = 5$
   - $\mathbf{p_6 = 6}$
   - $\mathbf{p_7 = 7}$
   - $p_8 = 9$
   - $p_9 = 10$
   - $\mathbf{p_{10} = 8}$
5. `rot3`$(8, 9, 10)$ results in

- $p_0 = 0$
- $p_1 = 1$
- $p_2 = 2$
- $p_3 = 3$
- $p_4 = 4$
- $p_5 = 5$
- $p_6 = 6$
- $p_7 = 7$
- $\mathbf{p_8 = 8}$
- $\mathbf{p_9 = 9}$
- $\mathbf{p_{10} = 10}$

This permutation is now sorted.

It is not possible to solve the example with at most 4 rotations.

Subtask 2: Cyclic Shift of 2023 Lines

General Observations

Consider the decomposition of p into cycles. (I.e., consider the directed graph $G = (V, E)$ with $V = \{0, 1, \ldots, N-1\}$ and $E = \{(i, p_i) \mid i \in \{0, 1, \ldots, N-1\}\}$. Each vertex then has in- and out-degree both equal to 1. Therefore, each strongly connected component of the graph is a cycle. We consider those components.)

Let $c(p)$ denote the number of cycles that p decomposes into in this way.

Note that p is sorted if and only if $c(p) = N$.

An operation `swap`(i, j) operates on cycles as follows:

- If i and j are part of the same cycle, the cycle decays into two cycles.
- If i and j are part of different cycles, those two cycles are merged into a single larger cycle.

In particular, each operation `swap`(i, j) changes $c(p)$ by either -1 or 1.

Note that the rotation `rot3`(i, j, k) can be decomposed into two swaps, for example `swap`(j, k) followed by `swap`(i, j).

It immediately follows that each operation `rot3`(i, j, k) changes $c(p)$ by either -2, 0 or 2.

Solution

We now know enough to be able to solve our example. Let p' be the sorted version of p that we are aiming for.

No solution can use fewer than $(c(p') - c(p))/2 = (2023 - 1)/2 = 1011$ operations (as each operation can add at most 2 to $c(p)$). We will now describe one such optimal sequence of 1011 operations:

- `rot3`$(2020, 2021, 2022)$
- `rot3`$(2018, 2019, 2020)$
- `rot3`$(2016, 2017, 2028)$
- `rot3`$(2014, 2015, 2016)$
- $\vdots$
- `rot3`$(2, 3, 4)$
- `rot3`$(0, 1, 2)$

I.e., the i-th operation (0-based) is given by `rot3`$(2020 - 2 \cdot i, 2021 - 2 \cdot i, 2022 - 2 \cdot i)$.

Claim: After k evaluated operations, p is given by

- $p_0 = 1$

- $p_1 = 2$
- $\vdots$
- $p_{2022-2\cdot k} = 0$
- $p_{2022-2\cdot k+1} = 2022 - 2 \cdot k + 1$
- $p_{2022-2\cdot k+2} = 2022 - 2 \cdot k + 2$
- $p_{2021} = 2021$
- $p_{2022} = 2022$

In effect, each of the `rot3` operations reduces the index of the value 0 by two while preserving the relative order of all other values.

More precisely:

- For all i with $0 \leq i < 2022 - 2 \cdot k$, we have $p_i = i + 1$.
- We have $p_{2022-2\cdot k} = 0$.
- For all i with $2022 - 2 \cdot k + 1 \leq i < 2023$, we have $p_i = i$.

Proof: We show the claim by induction on the number k of evaluated operations. For the base case $k = 0$, we have to show that in the beginning, p satisfies:

- For all i with $0 \leq i < 2022 - 2 \cdot 0$, we have $p_i = i + 1$.
- We have $p_{2022-2\cdot 0} = 0$.
- For all i with $2022 - 2 \cdot 0 + 1 \leq i < 2023$, we have $p_i = i$.

The first two constraints are exactly the definition of the initial p given in the task statement, while the last constraint is vacuously true because there is no i that satisfies the condition.

For the step case, consider the state of p after $0 \leq k < 1011$ operations. According to the induction hypothesis, it is given by

- For all i with $0 \leq i < 2022 - 2 \cdot k$, we have $p_i = i + 1$.
- We have $p_{2022-2\cdot k} = 0$.
- For all i with $2022 - 2 \cdot k + 1 \leq i < 2023$, we have $p_i = i$.

To get the state after $k + 1$ operations, we have to evaluate the k-th operation, which is given by `rot3`$(2020 - 2 \cdot k, 2021 - 2 \cdot k, 2022 - 2 \cdot k)$. Only the values at those three indices will change. We will refer to the state of p after evaluating this operation as p'.

Therefore,

- for all i with $0 \leq i < 2020 - 2 \cdot k$, we have $p'_i = p_i = i + 1$.
- we have $p'_{2020-2\cdot k} = p_{2022-2\cdot k} = 0$.
- we have $p'_{2021-2\cdot k} = p_{2020-2\cdot k} = (2020 - 2 \cdot k) + 1 = 2021 - 2 \cdot k$.
- we have $p'_{2022-2\cdot k} = p_{2021-2\cdot k} = (2021 - 2 \cdot k) + 1 = 2022 - 2 \cdot k$.
- For all i with $2022 - 2 \cdot k + 1 \leq i < 2023$, we have $p'_i = p_i = i$.

This is equivalent to

- For all i with $0 \leq i < 2022 - 2 \cdot (k + 1)$, we have $p_i = i + 1$.
- We have $p_{2022-2\cdot(k+1)} = 0$.
- For all i with $2022 - 2 \cdot (k + 1) + 1 \leq i < 2023$, we have $p_i = i$.

Which concludes the step case. It now just remains to show that after 1011 operations, the permutation p is sorted. We have just proven that for $k = 1011$, the state is given by

- For all i with $0 \leq i < 2022 - 2 \cdot 1011$, we have $p_i = i + 1$.
- We have $p_{2022-2\cdot1011} = 0$.
- For all i with $2022 - 2 \cdot 1011 + 1 \leq i < 2023$, we have $p_i = i$.

The first constraint is vacuously true and the other two combine to state that for all i, we have $p_i = i$. Therefore, our sequence of 1011 operations indeed sorts p. As we have already shown that a lower number of operations cannot work, this concludes the proof of correctness.

Subtask 3: Cyclic Shift of 2024 Lines

Recall that each operation `rot3`(i, j, k) changes $c(p)$ by either -2, 0 or 2. In particular, the parity of the number of cycles remains unchanged. However, in the end, we need 2024 cycles and in the beginning, we have a 1 large cycle. The parities of 2024 and 1 are not the same, therefore there is no solution.

Subtask 4: Reversal

General Observations

We can generalize our insights from the previous two subtasks: An odd cycle of length $2 \cdot k + 1$ can be sorted using k operations. An even cycle of length $2 \cdot k$ cannot be sorted by itself.

However, we may be able to combine multiple cycles. To this end, we investigate how `rot3`(i, j, k) affects the cycle decomposition of p:

- If i, j and k are in the same cycle, this cycle either decays into 3 cycles (of arbitrary positive lengths adding up to the correct total) or it stays as one cycle.
- If i, j and k are in three distinct cycles, they are merged into a single cycle.
- If i is in one cycle, and j and k are in another cycle, the effect is to split the second cycle arbitrarily and merge one of the parts into the first cycle.
- The remaining two cases are analogous. (While there are non-trivial constraints regarding which precise cycles can be formed in each case, what is important for us is that one rotation involving two cycles will preserve the number of cycles, but can otherwise arbitrarily rebalance the number of elements between the two cycles.)

Conceptually, our problem therefore reduces to the following task: Given is a list of positive numbers (those are our cycle lengths). In one step, we can either

- replace three of them by their sum,
- replace one of them by three positive integers that sum up to it, or
- replace two of them summing up to at least 3 by two other positive integers with the same total sum.

In the end, we want to end up with a list whose elements are all 1, and we want to minimize the number of operations executed.

Any solution to this new problem can be turned into a solution to our original problem with the same number of operations, as there is always a way to choose a `rot3` operation transforming the cycle lengths accordingly, and if all cycles have length 1, the permutation is sorted. Similarly, each solution to the original problem immediately yields a solution with the same number of operations for this new problem.

Note that the second kind of operation is the only one that increases the number of cycles. However, any even cycle must participate in at least one operation of one of the other two types.

We can formalize this intuition as follows: Let $c'(p) = c(p) - e(p)$, where $e(p)$ is the number of even cycles in the cycle decomposition of p. Note that we still have $c'(p) = N$ if and only if p is sorted.

We now investigate how our three operations may affect $c'(p)$.

- When merging three cycles, $c(p)$ decreases by 2. There are four cases:
  - If all three cycles are odd, $e(p)$ remains unchanged. Therefore, $c'(p)$ decreases by 2.
  - If one of three cycles is even, $e(p)$ remains unchanged. Therefore, $c'(p)$ decreases by 2.
  - If two of three cycles are even, $e(p)$ decreases by 2. Therefore, $c'(p)$ remains unchanged.
  - If all three cycles are even, $e(p)$ decreases by 2. Therefore $c'(p)$ remains unchanged.
- When splitting a cycle into three cycles, $c(p)$ increases by 2. There are (the same) four cases:
  - If all three cycles are odd, $e(p)$ remains unchanged. Therefore, $c'(p)$ increases by 2.
  - If one of three cycles is even, $e(p)$ remains unchanged. Therefore, $c'(p)$ increases by 2.
  - If two of three cycles are even, $e(p)$ increases by 2. Therefore, $c'(p)$ remains unchanged.
  - If all three cycles are even, $e(p)$ increases by 2. Therefore $c'(p)$ remains unchanged.
- When replacing two cycles by two other cycles, $c(p)$ remains unchanged. There are are two cases:
  - If the sum of cycle lengths is even, there are two cases:
    - If both original cycles are even:
      - Both resulting cycles could be even, in this case $e(p)$ remains unchanged. $c'(p)$ remains unchanged in this case.
      - Both resulting cycles could be odd, in this case $e(p)$ decreases by 2. Therefore, $c'(p)$ increases by 2.
    - If both original cycles are odd:
      - Both resulting cycles could be odd, in this case $e(p)$ remains unchanged. $c'(p)$ remains unchanged in this case.
      - Both resulting cycles could be even, in this case $e(p)$ increases by 2. Therefore, $c'(p)$ decreases by 2.
    - If the sum of cycle lengths is odd, that means that exactly one original cycle is even and exactly one resulting cycle is even. Therefore $e(p)$ remains unchanged. $c'(p)$ also remains unchanged in this case.

We have shown that each possible operation changes $e(p)$ by either -2, 0 or 2. It immediately follows that there is no solution if $e(p)$ is odd. (Because we cannot change the parity of the number of even cycles, and in the end, there should be exactly 0 even cycles.)

Finally, we have shown that each possible operation changes $c'(p)$ by either -2, 0 or 2. In particular, no operation can increase $c'(p)$ by more than 2. Therefore, if we find solutions that increase $c'(p)$ by 2 in each step, those solutions are optimal. (Because our goal is to change p such that $c'(p)$ takes on the maximum possible value N.)

Solution

Note that p decomposes into $\lfloor \frac{N}{2} \rfloor$ disjoint swaps. (It swaps lines p_i and p_{N-1-i} for each i with $0 \leq i < N/2$.) Swaps are cycles of length 2. If N is odd, we also have one additional cycle of length 1 at $p_{(N-1)/2} = (N-1)/2$.

$e(p)$ counts the number of such disjoint swaps. Therefore, there must be an even number of swaps and $\lfloor \frac{N}{2} \rfloor$ must be even, otherwise there is no solution.

Consider the two disjoint swaps given by $p_0 = N-1$, $p_{N-1} = 0$, $p_1 = N-2$, $p_{N-2} = 1$. We can order those correctly using the following two rotations:

- `rot3`$(N-1, 0, 1)$ results in $p_0 = 0$ and the rotation $p_1 = N-1$, $p_{N-1} = N-2$, $p_{N-2} = 1$. Note that this operation turns two even cycles into two odd cycles. Therefore $c'(p)$ increases by 2.
- `rot3`$(1, N-1, N-2)$ reverses the leftover rotation from above and we obtain $p_0 = 0$, $p_1 = 1$, $p_{N-2} = N-2$, $p_{N-1} = N-1$. This operation turns one odd cycle into three odd cycles. Therefore $c'(p)$ increases by 2 as well.

Therefore, we can sort our permutation two such swaps at a time. This works because there is an even number of swaps. Furthermore, we have used the smallest possible number of operations, because each operation increases $c'(p)$ by 2, which is the largest possible amount for a single operation.

We can implement this to run in time $O(N)$ with $O(1)$ additional memory.

Subtask 5: General Case

The general case combines the insights from the previous subtasks. We have already shown that if p has an odd number of even cycles, there is no solution.

Otherwise, we pair up the even cycles and rotate once across each such pair to turn them into two odd cycles each. (It suffices to set up the rotation such that it moves one of the elements of one of the cycles to its correct location.) Each of those operations preserves $c(p)$ and reduces $e(p)$ by 2, hence it increases $c'(p)$ by 2.

After this step, we are left with some odd cycles. Similarly to subtask 2, we can sort each odd cycle independently. One way to do so is to rotate three subsequent elements in the opposite direction of the cycle. This will put two of those elements to their final location. The remaining elements will again form an odd cycle. Each of those operations preserves $e(p)$ and increases $c(p)$ by 2, hence it increase s $c'(p)$ by 2.

As those operations correctly sort p and each operation increases $c'(p)$ by 2, which is the largest amount possible, we are using the smallest possible number of operations.

To execute this solution, we need to find the cycles of p, which we can do in $O(N)$ time with $O(N)$ memory. The subsequent steps can also be implemented in this total running time. (One simple way would be to simulate the rotations on even cycles on p, recomputing the new cycle decomposition and then use an algorithm that proceeds like in subtask 2, independently on each odd cycle.)

Therefore, the total running time and memory usage are both $O(N)$.

Odd Canals

Task Idea	Michał Stawarz, Bibin Muttappillil
Task Preparation	Bibin Muttappillil
Description English	Bibin Muttappillil
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Michał Stawarz, Bibin Muttappillil, Johannes Kapfhammer
Solution English	Michał Stawarz

Clearly our problem can be modeled as a graph. Let $G = (V, E)$ be an undirected graph, where V is the set of the sites and E is the set of canals connecting them. We have $|V| = 2n$ and $|E| = m$. We want to direct edges so that there are exactly n vertices with odd out-degree.

Subtask 1: Solve an example (10 points)

Sample orientation satisfying the conditions

0 : 1, 2, 3, 4, 7
1 : 2, 3, 4, 7
2 : 3, 4, 7
3 : 4, 5
4 : 6
5 :
6 :
7 :
8 : 7
9 : 7
10 : 13
11 : 12
12 :
13 :

Subtask 2: Odd Number of Vertices? (5 points)

Let $G' = (V', E')$ be a graph with $|V'| = n'$, $|E'| = m'$. We know that for all $v \in V'$ holds $\deg(v) = 1 \mod 4$. In particular every degree is odd. Assume in sake of contradiction that n' is odd. It follows then that $\sum_{v \in V'} \deg(v)$ is odd as a sum of odd many odd numbers. On the other hand by handshaking lemma we have $\sum_{v \in V'} \deg(v) = 2m'$. So this sum is even. Contradiction.
Thus n' must be even i.e the number of vertices in the graph is always even.

Subtask 3: Just Odd is Impossible (10 points)

We will construct a counterexample by combining two types of vertices. Those with degree congruent to 1 modulo 4 and respectively congruent to 3 mod 4. Note that it is required to use both types of vertices to obtain any counterexample.
The most straightforward example of a graph, involving both types of vertices, is a tree with 4 vertices and 3 leaves. As one can observe (e.g. by listing 8 possible directions of edges) in such a graph it is impossible to have exactly 2 vertices with odd out-degree.

Subtask 4: Find an algorithm (75 points)

In this section we will call vertices with odd out-degree black and the remaining ones white. Our goal is to find an orientation with exactly n black vertices.

Playing with a tree

Let us focus on tree case. We direct edges arbitrarily. Let x be the number of black vertices in the graph.

Lemma: $x = n \mod 2$

Proof: The total number of edges in the graph is equal to the sum of out-degrees i.e. $m = \sum_{v \in V} \text{out-deg}(v)$. By taking the remainders of both sides modulo 2, we can omit in the summation all even out-degrees and hence $m = x \mod 2$. By handshaking lemma: $2m = \sum_{v \in V} \deg(v) = \sum_{v \in V} 4k_v + 1 = 2n + 4\sum_{v \in V} k_v$. Thus $m = n + 2\sum_{v \in V} k_v$ implies $m = n \mod 2$. Combining two equations we get $n = x \mod 2$.

Assume $x < n$. We will repeat the following process to increase the number of black vertices at each step: Find any two white vertices s.t. on the path between them there are only black vertices and reverse all edges on the path. Why does it work? Reversing an edge corresponds to flipping colors of its both endpoints. Thus by reversing the edges on the path we at first make two white vertices adjacent and then make them black. Furthermore x increases exactly by 2. Because x and n have the same parity we will reach n by exactly $(n - x)/2$ steps.

The case when $x > n$ is symmetric (instead of pair of white vertices we find pair of black vertices).

Straightforward implementation leads to $O(n^2)$ time complexity.

This can be however speeded up as follows. We root our tree at vertex 1 and run a dfs from it. We assign to every vertex a pre-order traversal number. Then we mark first $n - x$ white vertices in respect to pre-order ordering.

Lemma: We claim that we need to reverse an edge if and only if there is an odd number of marked vertices within its subtree.

Proof: We can pair up together consecutive marked vertices so that they doesn't violate the rule. Instead of directly reversing a path from u to v, we can reverse 2 paths: from 1 to u and from 1 to v. So an edge will be reversed exactly if there is an odd number of paths passing through it. Because all paths go to the root, so it is equivalent to having odd number of marked vertices in the subtree.

Number of marked vertices within each subtree can be computed via dynamic programming.

Overall it yields to $O(n + m)$ solution to tree case.

Euler tours

Now we will solve general case.

Let's add one super vertex s to the graph and connect it to every other vertex in V. Then $\deg(s) = |V| = 2n$ and for all $v \in V$ $\deg(v) = 2 \mod 4$. So we can conclude that every vertex in the new graph has an even degree. Moreover the new graph is connected (as all vertices are connected to s), thus there exists an Euler cycle in the new graph.
We direct all edges according to the Euler cycle.

Lemma: For every vertex v $\text{out-deg}(v) = \text{in-deg}(v)$.

Proof: Euler cycle visits every edge exactly once, so for each edge we can define the immediate successor in the cycle. Clearly no two distinct edges will have the same successor, hence this function is bijective. Then when we look at edge which is directed to v we can match it with its successor which is directed from v. Therefore the in-degree is the same as out-degree for every vertex v.

Let $v \in V$. We have $2 \cdot \text{out-deg}(v) = \text{out-deg}(v) + \text{in-deg}(v) = \deg(v) = 4k + 2$ for some $k \in \mathbb{Z}$. So $\text{out-deg}(v) = 2k + 1$ is an odd number.

So all out-degrees of vertices in V are odd. On the other hand $2 \cdot \text{out-deg}(s) = \text{out-deg}(s) + \text{in-deg}(s) = \deg(s) = 2n$. So vertex s has exactly n out-going edges.

Now, when we remove the vertex s from the graph and all edges incident to it, exactly n vertices will flip their out-deg parity. Thus after removing s exactly n vertices have an odd out-degree. We found desired orientation. Time complexity $O(n + m)$.

"Reverse engineering"

Let us describe another, surprisingly easy, solution. We will use the facts proved in tree subtask.

Statement: Direct edges arbitrarily. Then start reversing them one by one (e.g. in order $1, 2, .., m$). We claim that at some point we will have exactly n black vertices in the graph.

Proof: Let x denotes the number of black vertices at the beginning i.e. after initial direction of edges. We have already shown that $x = n \mod 2$. Now consider the number of black vertices at the end (when we reversed all the edges). As every degree is odd, so every black vertex becomes white and vice-versa. Therefore we will have $2n - x$ black vertices at the end. Now observe that $n = \frac{x+(2n-x)}{2}$, thus $n \in [\min\{x, (2n-x)\}; \max\{x, (2n-x)\}]$. Every single reverse operation changes x to $x - 2$, x or $x + 2$ (see tree subtask). As x and n have the same parity, it follows that we must hit n at some point.

Running time for this algorithm is $O(n + m)$ and we need $O(n)$ additional memory (keep track of the color of each vertex; then iterate over the edges; if we hit n at index i then we print all edges up to i in reverse order and after i in forward order).