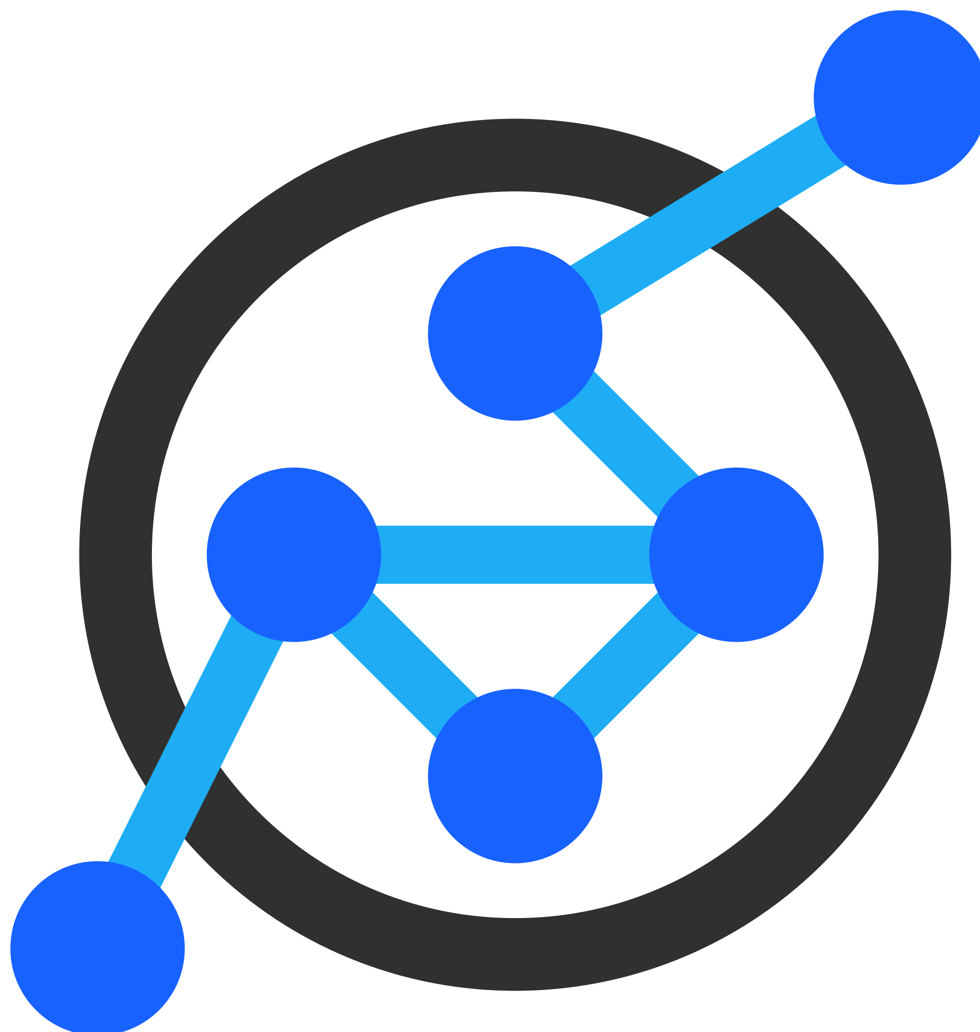
First Round SOI 2022/2023

Solution Booklet



Swiss Olympiad in Informatics

15 September – 30 November 2022

Paragliding

Task Idea	Joël Huber & Christopher Burckhardt
Task Preparation	Joël Huber
Description English	Joël Huber
Description German	Bibin Muttappillil
Description French	Florian Gatignon
Solution	Joël Huber
Correction	

Summary

All subtasks shared a common base, which will be more easily explained in this short introduction.

Given were N places on a line, where the i-th place is at x-coordinate x_i and its height above sealevel is h_i metres. A mouse can fly from one place to another if the starting place is higher than the end place and all other places inbetween. You were asked to compute for each place the longest horizontal distance Mouse Binna can fly starting from that particular place.

In subtask 5, the only thing that changed is that you were asked to output the sum of all these numbers, as otherwise the output would be big and might take long to upload.

Subtask 1: The Uetliberg (15 Points)

In the first subtask, $N = 2$ and so there are only two places. If there are only two places, one will be lower than the other and from this place, you can't fly anywhere. From the higher place, it's always optimal to fly to the lower place. This is easily done with a simple conditional, using a constant amount of time and memory.

```
1  t = int(input())
2  for case in range(t):
3      n = int(input())
4      xs = list(map(int, input().split()))
5      hs = list(map(int, input().split()))
6      dx = xs[1] - xs[0]
7      if hs[0] < hs[1]:
8          print(f"Case #{case}: 0 {dx}")
9      else:
10         print(f"Case #{case}: {dx} 0")
```

Subtask 2: Paragliding at the Gornergrat (15 Points)

In this subtask, the heights are decreasing, meaning that we have $h_i > h_j$ for all $i < j$. In this subtask, we can realize that if we want to cover as much horizontal distance as possible, it is always optimal to fly until the last place, since there is nothing stopping us. Thus the horizontal distance that can be covered starting from place i is $x_{n-1} - x_i$ (using zero-based indexing).

This will take us $O(n)$ time and $O(n)$ memory.

```
1  t = int(input())
2  for case in range(t):
3      n = int(input())
4      xs = list(map(int, input().split()))
5      hs = list(map(int, input().split()))
6      print(f"Case #{case}: {' '.join([str(xs[-1] - x) for x in xs])}")
```

Subtask 3: Mount Kékes (15 Points)

In this subtask, the coordinates are increasing at first, and then decreasing. Such a sequence is usually called a "unimodal" sequence. Also, let's call the highest place the "peak" of the mountain.

If we ignore the peak for a moment, we can make the same observation of subtask 2 for both sides of the mountain: If you are on the left side of the peak, it's optimal to fly to the rightmost point since you can't fly to the right, but nothing stops you from flying to the left as far as you want. If you are on the right side of the peak, then you can't fly to the left but it's optimal to fly as far as possible to the right as nothing stops you from flying further. It's only the peak where we actually need to make the decision in which direction to fly. But there we can just compare the distance to the leftmost point with the distance to the rightmost point and choose the longer one.

These observations just made can be expressed in an even more compact form: For each place and direction, we can either not fly in that direction at all or we can fly all the way without being stopped. Phrasing our observations that way allows us to find a neater implementation where we don't even care about finding the peak.

```
1 t = int(input())
2 for case in range(t):
3     n = int(input())
4     xs = list(map(int, input().split()))
5     hs = list(map(int, input().split()))
6     res = [
7         max(
8             xs[i] - xs[0] if i > 0 and hs[i - 1] < hs[i] else 0,
9             xs[-1] - xs[i] if i < n - 1 and hs[i] > hs[i + 1] else 0,
10        )
11        for i in range(n)
12    ]
13    print(f"Case #{case}: {' '.join(map(str, res))}")
```

Subtask 4: The Mátra Mountain Range (25 points)

This subtask is the first subtask where we try to solve the general problem. Note that in this subtask, we have $N \leq 3000$, so we can try to find a solution running in $O(N^2)$.

First, we make the observation that if we start at place i, the furthest we can fly to the left is to place $j + 1$ where j is maximal such that $j < i$ and such that $h_j > h_i$ (or to place 0 if no such j exists). Now suppose we start from place i. Then the easiest way to find out how far we can fly to the left is by using a for loop starting from place i going to the left, only stopping when we arrive at a place which is higher than the place we started from.

We then do the same but flying to the right instead of flying to the left, and take the maximum of the two answers for each place. Note that we can reuse the "flying to the left" part of our code for the "flying to the right" by just reversing the two arrays (and by taking the absolute value of the difference of the x-coordinates).

```
1 def run(n, xs, hs):
2         res = [0]*n
3         for i in range(n):
4                 for j in range(i, -1, -1):
5                         if hs[j] > hs[i]:
6                                 break
7                         res[i] = abs(xs[i] - xs[j])
8         return res
9
10 t = int(input())
11 for case in range(t):
12         n = int(input())
```

```
13        xs = list(map(int, input().split()))
14        hs = list(map(int, input().split()))
15        res = [max(a, b) for a, b in zip(
16                run(n, xs, hs),
17                reversed(run(n,
18                        list(reversed(xs)),
19                        list(reversed(hs))
20                ))
21        )]
22        print(f"Case #{case}: {' '.join(map(str, res))}")
```

Subtask 5: All of Hungary (30 points)

This subtask is almost the same as the last one, with the only differences being that N can reach 10^6 now, so our previous solution is too slow, and that we now only need to print the sum of the answers (since otherwise the output might get very big).

Recall that in our slower solution, we were trying to find for each i the biggest $j < i$ such that $h_j > h_i$. If we do this for each i separately, this is very slow. But we can try to compute this for all starting places at the same time. Let's loop from left to right - the key idea is that if we are currently at a place i and place k with $k < i$ has $h_k < h_i$, then k will never again be a place that stops us from going further to the left, since if we're able to fly over place i, we're also able to fly over place k. This leads us to what is commonly known as an ''monotone stack'' or ''increasing stack'': We can try to keep all the places that can still be stopping places in increasing order on a stack, with the lowest one being on top of the stack. From our earlier observation, we can see that these places are also ordered by decreasing x-coordinates. When we process place i, we can start popping the places on top of the stack until the lowest place still on the stack has height higher than h_i. Then the element on top of the stack is the place that stops us when we start from place i. Then finally, we push the current place on the stack. The code looks like this:

```
1  def run(n, xs, hs):
2          stack = [(-1, int(2e12))]
3          res = [0]*n
4          for i in range(n):
5                  while stack[-1][1] < hs[i]:
6                          stack.pop()
7                  res[i] = abs(xs[stack[-1][0] + 1] - xs[i])
8                  stack.append((i, hs[i]))
9          return res
10
11 t = int(input())
12 for case in range(t):
13         n = int(input())
14         xs = list(map(int, input().split()))
15         hs = list(map(int, input().split()))
16         res = [max(a, b) for a, b in zip(
17                 run(n, xs, hs),
18                 reversed(run(n,
19                         list(reversed(xs)),
20                         list(reversed(hs))
21                 ))
22         )]
23         print(f"Case #{case}: {sum(res)}")
```

Let's think about the running time of this approach. At first, it seems like this is also $O(N^2)$ as in the while loop we could pop up to N elements from our stack. The key idea here is that every item that gets popped needs to be added to the stack first. Thus if we consider the total number of times we pop something from the stack across all iterations of our outer for loop, it is $O(N)$ as we only push $O(n)$ elements on our stack. Thus the total running time is $O(n)$.

Rubik's Knob

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Bibin Muttappillil
Description French	Mathieu Zufferey
Solution	Johannes Kapfhammer
Correction	

The setting for this task is a spiral spring with some integer value *charge*. Increasing the absolute value of the charge by 1 has a cost of 1; decreasing the absolute value by 1 has a cost of 0. If the charge is x, the indicator of the spiral spring points at position $x \bmod M$ for a fixed M.

For practical implementation, it's easiest to only consider the charge as the canonical state of the spring and deduce all secondary properties from it. The position can be computed as $x \bmod M$, and the number of turns can be computed by $\lceil x/M \rceil$ (rounded towards 0).

Subtask 1: Computing the number of wind-ups (17 Points)

The first subtask is about computing the number of operations given a sequence of positions and whether to move between them with left or right rotations.

For moving between two positions, there are six cases and we can compute the cost separately for them:

```
 a0----->a1   |
 a1<-----a0   |
     b0------|----->b1
     b1<-----|------b0
             |   c0----->c1
             |   c1<-----c0
-------------0-------------->
```

a) start and end negative:
$\max(0, |a_0| - |a_1|)$ (costs only if we increase the absolute value)

b) start positive/end negative or start negative/end positive:
$|b_1|$ (moving from b_0 to 0 is free, rest has cost 1)

c) start and end negative:
$\max(0, |c_0| - |c_1|)$ (costs only if we increase the absolute value)

All that remains now is to translate the "L" or "R" instruction to an absolute charge value – we do this by setting the target charge to the correct position with the same number of turns, and then adjust based on absolute value.

```
1  def parse():
2      n, m = map(int, input().split())
3      xs = list(map(int, input().split()))
4      dirs = input().strip()
5      validate(n, m, xs, dirs)
6      return m, xs, dirs
7
8  def segment_cost(a, b):
9      return max(0, abs(b) - (abs(a) if (a>0)==(b>0) else 0))
10
11 def solve(m, xs, dirs):
12     ans = 0
13     pos = 0
14     for to, dir in zip(xs, dirs):
15         nxt = (pos//m)*m + to
16         if nxt < pos and dir == 'R':
17             nxt += m
```

```
18          elif nxt > pos and dir == 'L':
19              nxt -= m
20          ans += segment_cost(pos, nxt)
21          pos = nxt
22          assert pos%m == to
23      return ans
24
25  if __name__ == '__main__':
26      for case in range(int(input())):
27          print(f"Case #{case}:", solve(*parse()))
```

General Observations

The following subtasks ask us to compute the minimal number of operations for a given sequence $x_0, \ldots, x_{n-1}$. Before we take a look into the subtasks, we can make some general observations.

Claim 1 *We can add* 0 *to the beginning of the sequence and* 0 *to the end of the sequence. The cost of an optimal solution will be the same.*

Proof. We already start at 0 so we get this for free. At the end, if we are somewere other than 0 we can always "fall back" to 0 using the remaining charge we have, without additional cost either. □
Let $x_0, x_1, \ldots, x_{n-1}$ be the sequence we are interested in. This claim allows us to assume without loss of generality that $x_0 = 0$ and $x_{n-1} = 0$. (If this is not the case prepend and append 0 – which makes n at most larger by 2.)

It will turn out to be useful to not look at the sequence $x_0, \ldots, x_{n-1}$ directly but instead at the deltas $d_i = (a_{i+1} - a_i) \bmod m$.

Claim 2 *The sum of all deltas is divisible by m, i.e.* $\left(\sum_{i=0}^{n-1} d_i\right) \bmod m = 0$.

Proof. Since we added 0 to the start and the end of the sequence we start at 0 and end at 0. This is only possible when we do some number of full turns which implies the sum of deltas is a multiple of m. □

Claim 3 *Without loss of generality $d_i \neq 0$ for all i. Or in other words, any deltas of value* 0 *can be ignored.*

Proof. We can fullfil them by standing still, so the cost is the same if we take them out of the sequence. □

Claim 4 *In order to move by a delta of d_i we can either increase the charge by d_i or decrease it by $n - d_i$.*

Proof. This follows directly from the definitions. Turning right increases the charge by d_i. Turning left means we have to move the opposite side, which has cost $n - d_i$. □

Subtask 2: Beginner Puzzles (16 Points)

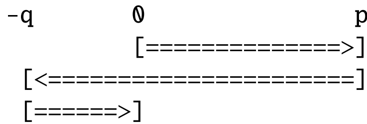
This subtask was a special case where the x_i are increasing.

The first idea that comes to mind would be to solve this with purely right turns "`RRR...RRRR`" but luckily(?) the example shows that this is not always optimal.

If we go from 0 to m with purely right turns we will have a charge of m which we are "wasting". In hindsight we want to spend this charge on one of the deltas. And in fact we can do.

Let the deltas be $d_0, d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_{n-1}$. We can do the first $d_0, d_1, \ldots, d_{i-1}$ with right turns. Then we can do d_i with a single left turn. And then we do $d_{i+1}, \ldots, d_{n-1}$ with a right turns again.

What are the costs of doing so? Now the ideas of Claim 2 come in handy! We can see that $\sum_{i=0}^{n-1} d_i = n$. The solution will look like this:



We have to pay the part from 0 to $p = d_0 + d_1 + \ldots + d_{i-1}$, then we drop down for free from p to 0. Then we have to pay the part from 0 to $-q$. And lastly we get back from $-p$ to 0 with $d_{i+1}, \ldots, d_{n-1}$. But that allows us to compute q as $q = d_{i+1} + \cdots + d_{n-1}$. So the total sum is

$$p + q = (d_0 + d_1 + \ldots + d_{i-1}) + d_{i+1} + \cdots + d_{n-1} = m - d_i$$

For what d_i is $m - d_i$ minimal? For the biggest one of course!

The following code does exactly that, it computes the deltas with `[(y-x)%m for x, y in zip(xs, xs[1:])]` and then takes the `max(...)` of them.

```python
def parse():
    n, m = map(int, input().split())
    xs = list(map(int, input().split()))
    return m, xs

def solve(m, xs):
    xs = [0] + xs + [0]
    return (m - max([(y-x)%m for x, y in zip(xs, xs[1:])]))%m

if __name__ == '__main__':
    for case in range(int(input())):
        print(f"Case #{case}:", solve(*parse()))
```

Subtask 3: Advanced: Small Puzzles (23 Points)

Subtask 4: Expert: Large Puzzles (44 Points)

This time we have to solve arbitrary sequences.

For this we need another set of observations:

Claim 5 *The optimal solution starts and ends with a charge of 0.*

Proof. We already know we start with a charge of 0 so we only need to look at whether we also end at 0.

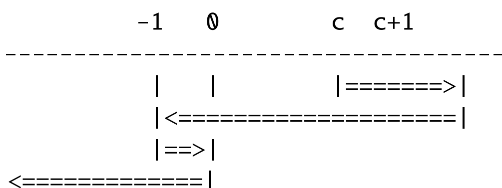
Proof by contradiction: Say we end up with a charge of $\geq m$ in the end (recall that by by Claim 1 we end up at position 0 so the charge must be at least m). Take the last move that did cost us some operations and go the opposite direction. Doing so will just unwind the spring and cost 0 charge (because we have at least m charge left we can pay this fully with the remaining charge), so the original one was not optimal. (Analogously with ≤ -1 turn.) □

Now another helper claim that helps us simplify our calculations later:

Claim 6 *If some sequence of operations starts and ends with the same charge c, the cost is the same for any other charge c'.*

Proof. Let's for a moment just compare a base charge of c and a base charge of $c + 1$.

Say we start and end at c.



```
=========>|
          |======>|
```

What happens when we shift the start from c to $c + 1$? The following operations still cost the same:

- Originally starting from ≥ 0 and going right costs 1.
- Originally starting from > 0 and going left costs 0.
- Originally starting from ≤ -1 and going left costs 1.
- Originally starting from < -1 and going right costs 0.

And the following operations now cost something diffent:

(a) Originally starting from -1 and going right did cost 0, now it will cost 1.

(b) Originally starting from 0 and going left did cost 1, now it will cost 0.

Note that since we start and end at the same position, the number of operations (a) must be the same as the number of operations (b). Which means that the total cost will still be the same.

So if c costs the same as $c + 1$, and $c + 1$ costs the same as $c + 2$ then by induction all charges $\geq c$ cost the same. By symmetry the same argument holds for c and $c - 1$, so it will be the same cost for all c. □

This gives us a pretty useful claim:

Claim 7 *Given the deltas $d_0, \ldots, d_i, d_{i+1}, \ldots, d_j, d_{j+1}, \ldots, d_{n-1}$ let R be the set of indices where we go right. Then the total cost is $\sum i \in Rd_i$.*

Proof. Because Claim 5 allowed us to assume we start and end at 0, by the previous claim we can now shift the base charge to a large value (say $\sum d_i$) without affecting the total cost. This means that all left turns are free and all right turns cost their value d_i. □

As a side remark, this leads to the following shorter solution for subtask 1 which would be pretty hard to understand without understanding Claim 7:

```python
def solve(m, xs, dirs):
    ans = 0
    pos = 0
    for to, dir in zip(xs, dirs):
        if delta == 0: continue
        delta = (to - pos)%m  # delta will be in range [0..m)
        if dir == 'R':
            pos += delta
            ans += delta
        else:
            pos += delta - m
        assert pos%m == to
    ans += max(0, -pos)
    return ans
```

With that out of the way, let's go to the key observation for this task:

Claim 8 *We can reorder the deltas without changing the total cost of the solution.*

Proof. This is a direct consequence from Claim 7: the cost of a given solution is only the cost of those deltas where we decide to go right.

We can change the order of the deltas as we want, as long as we go right for the same deltas as before the total sum does not change. □

If the order does not matter we can always assume the sequence is sorted and now we can just do that.

Claim 9 *Let the deltas be sorted ($d_0 \le d_1 \le \cdots \le d_{k-1}$). Let j be the number of times we go right in an optimal solution. The optimal solution will go right for the first j deltas ($d_0, \ldots, d_{j-1}$) and will go left for the last $n - j$ deltas ($d_j, \ldots, d_{k-1}$).*

Proof. Again proof by contradiction. Assume otherwise, meaning there are two deltas $p < q$ such that we go left at p and right at q. Since $d_p < d_q$ (we assumed the deltas are sorted) this will make the solution better, leading to a contradiction. □

Now how to find the optimal j?

Claim 10 *The optimal j is such that*

$$(d_0 + \cdots + d_{j-1}) + (d_j - m + \cdots + d_{k-1} - m) = 0.$$

Proof.

- *there is always such a j*: We know by Claim 5 that $\sum d_i = 0$, therefore the sum $(d_0 + \cdots + d_{j-1}) + (d_j - m + \cdots + d_{k-1} - m)$ is divisible by n for any j. Increasing (resp. decreasing) j by 1 changes the solution by m (resp $-m$). If $j = 0$ the sum is ≤ 0 and if $j = n$ it is $\ge m$. Therefore for some j in the middle the sum is 0.
- *any other value $\neq j$ is worse*: By Claim 5 we know that any solution that does not start and end at 0 is non-optimal.
- *there is an optimal solution for this j*: Then, Claim 9 shows us that given this j we can find an optimal solution by taking the smallest j deltas as right turns.

□

With all those observations out of the way, here the full solution:

```
1  def solve(m, xs):
2      xs = [0] + xs + [0]
3      deltas = sorted([(x-y)%m for x, y in zip(xs, xs[1:])])
4      rsum = 0
5      lsum = sum(m - d for d in deltas)
6      for d in deltas:
7          rsum += d
8          lsum -= m - d
9          if lsum == rsum:
10             return rsum
```

Running time is $O(n \log n)$ since we need to sort, space usage is $O(n)$.

Boulders

Task Idea	Daniel Rutschmann
Task Preparation	Timon Gehr
Description English	Timon Gehr
Description German	Charlotte Knierim
Description French	Florian Gatignon
Solution	Timon Gehr
Correction	

In this task, there is a grid with N rows and M columns. You can execute a number of operations: For each operation, you can either roll a boulder down a row (left to right) or a column (top to bottom). The boulder will keep rolling into the original direction until it hits the other border of the grid or another boulder. It will then always stop on some grid square.

You are given a grid where some grid squares are occupied by a boulder. You have to determine whether it is possible to obtain the given pattern of boulders using a sequence of operations as described above.

Subtask 1: Subtask 1 (10 Points)

In this subtask, we have $N = 1$. I.e., there is only a single row. We can simply roll a boulder down each column that should have one. In this way, we can obtain an arbitrary pattern of boulders. (This is not always the only way to generate a given pattern, but it always works.)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    int N, M;
    cin >> N >> M;
    assert(N == 1); // N is 1 in this subtask
    string grid;
    cin >> grid;                        // read a single line
    cout << "Case #" << t << ": Yes\n"; // always possible
    for (int i = 0; i < M; i++) {
      if (grid[i] == '#') {         // boulder in column i
        cout << "C " << i << "\n"; // put it
      }
    }
  }
}
```

The running time of this solution is $\Theta(M)$, as we perform a constant number of operations per column.

Subtask 2: Subtask 2 (25 points)

In this subtask, we have $N = 2$. I.e., there are exactly two rows, a top row and a bottom row. We can process the columns from right to left. There are now a few simple observations:

- If a column is empty, we don't have to do anything.
- If a column i has a boulder at the bottom, we can simply roll a boulder down column i.
- If a column i has two boulders, we can additional roll a second boulder down column i.

- The only tricky case is if a column i has a boulder in the top row but not in the bottom row. In this case, we can't roll any boulder down column i, because otherwise we would end up with a boulder in the bottom row in that column. Therefore, the only possible way to place that boulder is to roll a boulder down the top row. This boulder will always be able to reach column i, but it can only stop there in case column i is the last column ($i = M - 1$), or there already is a boulder in the top row of column $i + 1$. In case there should indeed be a boulder at that position, we have already placed it (because we are processing columns from right to left). Otherwise, it will be impossible to place the boulder and to obtain the given pattern, so we can abort.

In our implementation, we first check whether it is possible to place the boulders and we only place them if it is actually possible.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5    int T;
6    cin >> T;
7    for (int t = 0; t < T; t++) {
8      int N, M;
9      cin >> N >> M;
10     assert(N == 2); // N is 2 in this subtask
11     string top, bot;
12     cin >> top; // read top row
13     cin >> bot; // read bottom row
14     // we first check if it is possible
15     bool possible = true;
16     for (int i = 0; i < M; i++) {
17       if (top[i] == '#' && bot[i] == '.') {
18         possible &= i == M - 1 || top[i + 1] == '#';
19       }
20     }
21     cout << "Case #" << t << ": " << (possible ? "Yes" : "No") << "\n";
22     if (possible) {
23       for (int i = M - 1; i >= 0; i--) { // from last to first column
24         if (bot[i] == '#') {              // boulder at bottom in column i
25           cout << "C " << i << "\n";      // put it
26           if (top[i] == '#') {            // boulder at top in column i
27             cout << "C " << i << "\n";    // put it
28           }
29         } else {                   // no boulder at bottom in column i
30           if (top[i] == '#') { // but there is one at the top
31             assert(i == M - 1 || top[i + 1] == '#');
32             cout << "R 0\n"; // put it
33           }
34         }
35       }
36     }
37   }
38 }
```

The running time of this solution is still $\Theta(M)$, as we perform a constant number of operations per column.

Subtask 3: Subtask 3 (30 points)

(Note that the solution for subtask 4 is possibly easier to understand than the solution for subtask 3.)

In this subtask, there are at most 20 boulders. (But the number of rows and columns can be big.)

For a solution that exploits the low number of boulders in this subtask, we will characterize the conditions under which we can place a boulder. To be able to place a single boulder:

- It has to be *supported*:
  - Either it has to be in the last row or in the last column.
  - Or there needs to be a boulder already placed at the spot immediately to the right or at the spot immediately to the bottom.
- It can not be *blocked*: We cannot place a boulder in case there is already another boulder in the same row at a lower column index as well as another one in the same column at a lower row index. Otherwise we can place it.

Those conditions taken together are necessary and sufficient. If we are given a permutation of the boulders, we can check whether we can place the boulders in that order using these conditions, one-by-one for each boulder. Unfortunately, there are up to 20! = 2432902008176640000 permutations of 20 boulders. We cannot check all of those in just 5 minutes (it would likely take multiple decades), so we need to be a little bit more clever.

We can optimize this idea as follows using dynamic programming: For each set boulders, we will compute whether it is possible to place that specific set. In particular, this will allow us to determine whether it is possible to place all boulders. To compute the result for a given set, we will be able to use results for its strict subsets if we compute the results in a suitable order.

The recurrance for our dynamic programming algorithm is as follows:

- Base case: It is always possible to place the empty set of boulders (by running zero operations).
- Step: For a non-empty set of boulders, if it is possible to place them, there in particular has to be one boulder we can place last. Therefore, we can check for each boulder individually whether it can be placed last if all the other boulders have already been placed. For this, we can make use of the conditions laid out above. If a boulder can be placed last, and it is possible to place all the other boulders in the set before it, it is possible to place our set, otherwise it is not.

The implementation uses a bit-mask encoding of sets of boulders: A set is encoded as an integer m such that the i-th boulder is in the set if and only if the i-th least significant bit of m is one.

If we can determine that it is possible to place all boulders, we can then use the intermediate results in the DP table to reconstruct one of the solutions.

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct Boulder {
5   int row;
6   int col;
7 };
8
9 int main() {
10   int T;
11   cin >> T;
12   for (int t = 0; t < T; t++) {
13     int N, M;
14     cin >> N >> M;
15     // create a list of all boulders:
16     vector<Boulder> boulders;
17     for (int i = 0; i < N; i++) {
18       string row;
19       cin >> row;
20       for (int j = 0; j < M; j++) {
21         if (row[j] == '#') {
22           boulders.push_back(Boulder{i, j});
23         }
24       }
25     }
```

```
    int K = boulders.size();
    assert(K <= 20);
    // check if we can place a new boulder if
    // some set of boulders encoded in mask
    // has already been placed
    auto check = [&](Boulder new_boulder, int mask) -> tuple<bool, bool> {
      bool supported_down = new_boulder.row == N - 1;
      bool supported_right = new_boulder.col == M - 1;
      bool blocked_top = false;
      bool blocked_left = false;
      for (int i = 0; i < boulders.size(); i++) {
        if (!(mask & (1 << i))) {
          // ignore boulders that have not been placed already
          continue;
        }
        // can we stack the new boulder on top of boulder i?
        // does boulder i prevent us from placing the new boulder?
        if (boulders[i].row == new_boulder.row) {
          supported_right |= new_boulder.col + 1 == boulders[i].col;
          blocked_left |= boulders[i].col <= new_boulder.col;
        }
        if (boulders[i].col == new_boulder.col) {
          supported_down |= new_boulder.row + 1 == boulders[i].row;
          blocked_top |= boulders[i].row <= new_boulder.row;
        }
      }
      auto row_placeable = supported_right && !blocked_left;
      auto col_placeable = supported_down && !blocked_top;
      return make_tuple(row_placeable, col_placeable);
    };
    vector<bool> table(1 << K, false);
    // it's always possible to place
    // the empty set of boulders:
    table[0] = true;
    for (int mask = 1; mask < (1 << K); mask++) {
      // try to find a boulder we can place last
      for (int last = 0; last < K; last++) {
        if (!(mask & (1 << last))) {
          // ignore boulders that have not been placed
          continue;
        }
        int others = mask & ~(1 << last);
        if (table[others]) {
          // it's possible to place all other boulders
          // check if we can place boulder last:
          auto [row_placeable, col_placeable] = check(boulders[last], others);
          bool can_place = row_placeable || col_placeable;
          if (can_place)
            table[mask] = true;
        }
      }
    }
    int mask = (1 << K) - 1; // mask with all boulders
    bool possible = table[mask];
    cout << "Case #" << t << ": " << (possible ? "Yes" : "No") << "\n";
    if (possible) {
      // we have to reconstruct a solution
      vector<pair<char, int>> solution_reversed;
      while (mask) {
        // try to find a boulder we can place last
        bool found = false;
        for (int last = 0; last < K; last++) {
          if (!(mask & (1 << last))) {
            // ignore boulders that have been processed
            continue;
          }
          int others = mask & ~(1 << last);
          if (table[others]) {
```

```
94              auto [row_placeable, col_placeable] = check(boulders[last], others);
95              if (row_placeable) {
96                solution_reversed.push_back({'R', boulders[last].row});
97                mask = others;
98                found = true;
99                break;
100             } else if (col_placeable) {
101               solution_reversed.push_back({'C', boulders[last].col});
102               mask = others;
103               found = true;
104               break;
105             }
106           }
107         }
108         assert(found);
109       }
110       assert(solution_reversed.size() == K);
111       for (int i = K - 1; i >= 0; i--) {
112         cout << solution_reversed[i].first << " " << solution_reversed[i].second
113              << '\n';
114       }
115     }
116   }
117 }
```

The running time of this solution is $O(N \cdot M + 2^K \cdot K^2)$. First, we read the input in time $O(N \cdot M)$. Then, we are iterating over 2^K sets of boulders. For each of those sets, we check up to K elements, and for each element, we compute in $O(K)$ time whether we can place the boulder given by that element as the last boulder in the set by iterating over the set once.

(Note that we could easily optimize this solution to time $O(N \cdot M + 2^K \cdot K)$, but this was not necessary to score full points for this subtask.)

Subtask 4: Subtask 4 (30 points)

Recall the observations from subtask 3:

To be able to place a single boulder:

- It has to be *supported*:
  - Either it has to be in the last row or in the last column.
  - Or there needs to be a boulder already placed at the spot immediately to the right or at the spot immediately to the bottom.
- It can not be *blocked*: We cannot place a boulder in case there is already another boulder in the same row at a lower column index as well as another one in the same column at a lower row index. Otherwise we can place it.

If we are given a permutation of the boulders, we can check whether we can place the boulders in that order using these conditions, one-by-one for each boulder. In our solution to subtask 3, we constructed permutations using dynamic programming. However, it turns out that this is not necessary. There are in fact simple ways to generate permutations of boulders that always work in case it is possible to place boulders in the given pattern at all.

The constraints suggest an ordering of boulders according to position, such that boulders at smaller positions are placed first:

- $(\text{row} + 1, \text{col}) < (\text{row}, \text{col})$. I.e., the boulder at position (row, col) should be placed after the boulder in position $(\text{row} + 1, \text{col})$.
- Similarly, $(\text{row}, \text{col} + 1) < (\text{row}, \text{col})$.

This makes sense because a boulder at (row, col) may rely on a boulder at (row + 1, col) or (row, col + 1) for support. Therefore, we should place the supporting boulders first if we can.

Conveniently, those constraints also ensure that boulders are placed in an order where no boulder is ever blocked from reaching its position by another boulder.

There are multiple ways to satisfy those constraints. A simple way is to iterate over both rows and columns backwards using nested for loops. As the ordering ensures no boulder is ever blocked by another boulder (neither in rows now in columns), we will only need to check that each boulder is supported. (Either by the right or bottom border of the grid, or by another boulder.)

In our implementation, we first check whether it is possible to place the boulders and we only place them if it is actually possible.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5    int T;
6    cin >> T;
7    for (int t = 0; t < T; t++) {
8      int N, M;
9      cin >> N >> M;
10     vector<string> grid(N);
11     for (int i = 0; i < N; i++) {
12       cin >> grid[i];
13     }
14     // we first check if it is possible
15     bool possible = true;
16     // it suffices to check that each boulder is supported
17     // by the grid border or another boulder.
18     // if so, we can then place them in a suitable order,
19     // such that no boulder blocks another boulder from
20     // reaching its position.
21     for (int i = 0; i < N; i++) {
22       for (int j = 0; j < M; j++) {
23         if (grid[i][j] == '#') {
24           // found a boulder, check if it is supported
25           bool supported_down = i == N - 1 || grid[i + 1][j] == '#';
26           bool supported_right = j == M - 1 || grid[i][j + 1] == '#';
27           possible &= supported_down | supported_right;
28         }
29       }
30     }
31     cout << "Case #" << t << ": " << (possible ? "Yes" : "No") << "\n";
32     if (possible) {
33       // possible, place boulders
34       for (int i = N - 1; i >= 0; i--) {
35         for (int j = M - 1; j >= 0; j--) {
36           if (grid[i][j] == '#') {
37             // found a boulder
38             bool supported_down = i == N - 1 || grid[i + 1][j] == '#';
39             bool supported_right = j == M - 1 || grid[i][j + 1] == '#';
40             if (supported_down) {
41               cout << "C " << j << "\n";
42             } else if (supported_right) {
43               cout << "R " << i << "\n";
44             } else {
45               assert(false);
46             }
47           }
48         }
49       }
50     }
51   }
52 }
```

The running time of this solution is still $\Theta(N \cdot M)$, as we perform a constant number of operations for each grid cell.

Cardgame

Task Idea	Joël Huber
Task Preparation	Luc Haller
Description English	Luc Haller
Description German	Charlotte Knierim
Description French	Mathieu Zufferey
Solution	Luc Haller
Correction	

Summary

You are given two lists of N cards, one yours and one Stofl's. The union of the lists contains each number from 1 to $2N$ exactly once. In some of the subtasks, the cards additionally have different scores. You're playing a game against Stofl, where a higher card beats a lower card. Stofl already told you in which order he's going to play all his cards. You now have to decide on an order for your cards.

We can notice that what matters is only which pairs of cards we match up, and the actual order of the turns doesn't matter. This allows us to reorder Stofl's list in a way that makes it simpler for us to determine a strategy.

Subtask 1: Subtask 1 (20 Points)

In the first subtask, you have to find an order in which you can play your cards such that you win each turn, or determine that there is no such order. Consider the strategy where we sort Stofl's and our list of cards each in ascending order and play our lowest card against Stofl's lowest card, our second lowest against his second lowest, and so on. We can see that if we don't win every turn with this strategy, there's no way to win each turn: If the i-th position is the first one where our card is lower than Stofl's, we can't do anything about it: Playing a lower card in its place instead will also lose the turn, and playing a higher card in its place we would have to play the i-th card against a higher one of Stofl's cards, where it would also lose.

Thus, sorting the two lists and comparing the numbers at each index is enough, and will take $O(N \log N)$ time and $O(N)$ memory.

```
1  t = int(input())
2  for case in range(t):
3      n = int(input())
4      as = list(map(int, input().split()))
5      bs = list(map(int, input().split()))
6      as.sort()
7      bs.sort()
8      winall = True
9      for i in range(n):
10         if as[i] > bs[i]:
11             winall = False
12             break
13     print(f"Case #{case}:", f"Yes {' '.join(map(str, bs))}" if winall else "No")
```

Subtask 2: Subtask 2 (20 Points)

In this subtask, you have to find the worst possible way to play. Since the lists are quite long, we only ask you to print the number of turns you lose.

We're trying to lose as many turns as possible, so we want to get rid of our high cards. This leads

to the idea that whenever we can't avoid winning a turn, we want to play our highest remaining card: We certainly won't be able to win fewer turns by keeping a higher card for later turns than necessary. With this modification, the greedy strategy from subtask 1 of matching up the sorted lists still works.

```
t = int(input())
for case in range(t):
    n = int(input())
    as = list(map(int, input().split()))
    bs = list(map(int, input().split()))
    as.sort()
    bs.sort()
    won = 0
    for i in range(n):
        if as[i] < bs[i-won]:
            # match up as[i] and bs[n-1-won]
            won += 1
    print(f"Case #{case}: {won}")
```

Subtask 3: Subtask 3 (20 Points)

In this subtask, you're trying again to win. Now each of your cards has a score attached (but Stofl's cards don't). Your total score is the sum of the scores attached to the cards with which you win a turn.

We still consider Stofl's cards in ascending sorted order. Since we now want to win, it's best to lose the turns against Stofl's highest cards, i.e. if one of our cards can't win its turn, it's best to play it against Stofl's highest remaining card (the opposite of what we did in subtask 2). It's also again good to win turns with the lowest remaining cards we have, and keep the higher ones to beat Stofl's higher cards. (That's the strategy from subtask 1.) But because there's now a score attached, not all cards are equivalent, and thus it might have been better to win the turn with a higher card if it has a higher score and we can't win later turns with it. To adjust for this, we'll "retroactively" exchange higher cards which would lose their turn with lower cards with a lower score which won their turn. That is, as we go over the sorted lists, we keep track of the scores of the cards we already assigned to a winning turn, and consider replacing the one with the lowest score with the current card.

We again only scanned over the lists once. In addition to that we have the cost of sorting, and the cost of keeping track of the smallest element. We can use a priority queue or multiset for that. In total that still takes $O(N \log N)$ time and $O(N)$ memory (or $O(N \log N)$ memory if we use a multiset).

```
#include <bits/stdc++.h>
using namespace std;

using vi=vector<int>;
using pii=pair<int,int>;
using vpii=vector<pii>;

void solve() {
    int n; cin >> n;
    vi as(n);
    vpii bs(n);
    for (int& ai : as) cin >> ai;
    for (auto& [bi, _] : bs) cin >> bi;
    for (auto& [_, si] : bs) cin >> si;
    sort(as.begin(), as.end());
    sort(bs.begin(), bs.end());
    multiset<int> won;
    int ai = 0;
    for (int i=0; i<n; ++i) {
        if (bs[i].first > as[ai]) {
            won.insert(bs[i].second);
```

```
22              ++ai;
23              continue;
24          }
25          if (!won.empty() && bs[i].second > *won.begin()) {
26              won.erase(won.begin());
27              won.insert(bs[i].second);
28          }
29      }
30      int sum = 0;
31      for (int x : won) sum += x;
32      cout << sum << "\n";
33  }
34
35  signed main() {
36      int t; cin >> t;
37      for (int i=0; i<t; ++i) {
38          cout << "Case #" << i << ": ";
39          solve();
40      }
41  }
```

Subtask 4: Subtask 4 (20 Points)

In this subtask, Stofl's cards have scores while yours don't.

So this time it doesn't matter with which cards we win turns, just which subset of Stofl's cards we beat. So we can use highest cards we have to win the best turns we can with them (if we skipped a higher card and used a lower one, we could equivalently win that turn with the higher card instead). In an optimal solution we always win the maximum number of turns we can. We can find a maximal set of turns to win by doing a greedy scan over both card lists in reverse order, and whenever we can't beat Stofl's highest remaining card we throw away our lowest remaining card, otherwise we use the highest remaining card. This is basically the opposite of the solution to subtask 2. This maximal set of turns isn't necessarily optimal though: It could be better to replace one of Stofl's cards we win against with a lower card with a higher score. This just requires the same approach as in Subtask 3, of keeping track of the card scores we beat and potentially replacing the lowest one as we go.

```
1       // the rest of the solution is analogous to the one in Subtask 3
2       sort(as.begin(), as.end());
3       sort(bs.begin(), bs.end());
4       multiset<int> won;
5       int bind = n-1;
6       for (int i=n-1; i>=0; --i) {
7           if (bs[bind] > as[i].first) {
8               won.insert(as[i].second);
9               --bind;
10              continue;
11          }
12          if (!won.empty() && as[i].second > *won.begin()) {
13              won.erase(won.begin());
14              won.insert(as[i].second);
15          }
16      }
```

Subtask 5: Subtask 5 (20 Points)

In the final subtask, all cards have scores. The basic ideas from the other solutions are still applicable, but since both our and Stofl's cards have scores, the logic for improving earlier turns becomes a bit more complex.

```
1       sort(as.begin(), as.end());
2       sort(bs.begin(), bs.end());
3       priority_queue<int> stofl_skipped;  // The scores of Stofl's cards in turns we haven't won.
```

```
    priority_queue<int, vi, std::greater<int>> our_won;  // The scores of our cards in turns we've won.
    vi stofl_won;  // The scores of Stofl's cards in turns we've won.
    int aind = 0;
    for (int i=0; i<n; ++i) {
        if (aind < n && bs[i].first > as[aind].first) {
            // Our current (i-th) card beats Stofl's current (aind-th) card.
            our_won.push(bs[i].second);
            for (; aind < n && bs[i].first > as[aind].first; ++aind) {
                // There may be a whole range of Stofl's cards we could beat with our i-th card.
                stofl_skipped.push(as[aind].second);
            }
            // We use our i-th card to win against Stofl's highest-scoring card (with a lower value) which we haven't wo
            stofl_won.push_back(stofl_skipped.top());
            stofl_skipped.pop();
        } else if (!stofl_skipped.empty()) {
            // Our i-th card doesn't beat Stofl's aind-th card, but it can still beat
            // all of Stofl's cards which we put in `stofl_skipped` for earlier i.
            stofl_won.push_back(stofl_skipped.top());
            stofl_skipped.pop();
            our_won.push(bs[i].second);
        } else if (!our_won.empty() && bs[i].second > our_won.top()) {
            // We can't win any additional turn with our i-th card, but it has a
            // higher score than the worst lower-valued card we won a turn with,
            // so we can improve that turn by swapping them.
            our_won.pop();
            our_won.push(bs[i].second);
        }
    }
    long long sum = 0;
    while (!our_won.empty()) {
        sum += our_won.top();
        our_won.pop();
    }
    for (int x : stofl_won) sum += x;
    cout << sum << "\n";
```

Palatinal Crypt

Task Idea	Tobias Feigenwinter and Daniel Rutschmann
Task Preparation	Tobias Feigenwinter
Description English	Tobias Feigenwinter and Johannes Kapfhammer
Description German	Priska Steinebrunner
Description French	Florian Gatignon
Solution	Tobias Feigenwinter
Correction	Charlotte Knierim and Jan Schär

In this task, you were asked to prepare a crypt to open it for the public. The crypt is a $N \times M$ grid of rooms and you should connect some of the rooms (by tearing down some of the walls). Afterwards, in every subtask except for the first, all of the following conditions should hold:

1. The rooms should be connected, i.e. it should be possible to reach every room from every other room.
2. You may only tear down strictly less walls than there are rooms, i.e. you may tear down at most $N \times M$ walls.
3. No room may have all four walls removed, and no room may have exactly two walls removed.

You were asked to output your crypt layout in an ASCII layout like the following:

```
o.o.o
|.|.|
o-o-o
|...|
o-o.o
```

In this representation, the letter o represents a room of the crypt and the pipe characters and minuses represent a connection between two of the rooms. The dots do not have any particular meaning, but are used as spacing such that the other characters are at the correct position.

Subtask 1: Klotild Wing

In this subtask, the third condition in the list above is relaxed: It is allowed to remove exactly two walls of a room. It still is not allowed to remove exactly four walls, and the other two conditions also hold.

To solve this subtask, we can find some easy pattern such as the following:

```
o-o
|.|
o.o
```

You can extend this pattern to the right by repeating the red and purple parts, and to the bottom by repeating the blue and purple parts. If we repeat the red and purple parts four times and the blue and purple parts two times, we get the following 3×5 crypt:

```
o-o-o-o-o
|.|.|.|.|
o.o.o.o.o
|.|.|.|.|
o.o.o.o.o
```

This also solves $1 \times M$ crypts: for this, we just repeat the blue and purple part zero times. In the same way, we get the solutions for $N \times 1$ crypts.

The following python code outputs crypts using this pattern:

```
t = int(input())
for T in range(t):
    print("Case #{}:".format(T))
    n, m = map(int, input().split())
    print("o-"*(m-1)+"o")
    for i in range(n-1):
        print("|."*(m-1)+"|")
        print("o."*(m-1)+"o")

```

Subtask 2: Subtask 2: Ladislaus wing

From this subtask onward, the crypts have to fulfill all conditions. In particular, it is now forbidden for any room to have exactly two doors ripped out.

This subtask has a somewhat special format, in that your program is allowed to choose the crypt dimensions (as long as no size is used twice). So you just have to find any twenty sizes for which you can find a valid crypt layout. One possible solution is given by the following pattern:

```
o.o.o-o-o.o
|.|...|...|
o-o-o-o-o-o
|...|...|.|
o.o-o-o.o.o
```

Similar to subtask 1, you'll get different sizes of crypts by replicating the red part any number of times. The following python code outputs crypts based on this pattern.

```
T = int(input())
for t in range(T):
    print("Case #{}: {} {}".format(t, 3, 2+4*t))
    print("o." + "o.o-o-o."*t + "o")
    print("|." + "|...|..."*t + "|")
    print("o-" + "o-o-o-o-"*t + "o")
    print("|." + "..|...|."*t + "|")
    print("o." + "o-o-o.o."*t + "o")
```

Subtask 3: Esztergom Wing

Subtask 4: World Crypt Association

Since the fourth subtask is to theoretically analyze the solution for the third subtask, they are discussed here together.

The format of the third subtask is similar to the first subtask, with one major difference: As not every crypt size admits a valid crypt layout, you will first have to output weather or not there is any solution.

We will start by showing that all valid crypts should have a number of rooms that is even, but not divisible by four. Next we will discuss cases where either the width or the height of the crypt is at most two. Finally, we will show a pattern for constructing the larger cases.

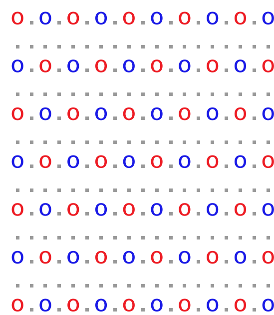
The number of rooms has to be even

For every room, we count the number of removed walls, and we add all those numbers together. If we do this, every wall has been counted twice: once for each of the rooms it connects. In particular, this number is even. Let's call this number D.

Remember that we are not allowed to remove exactly two or exactly four walls of a room. This means we have to remove one or three walls, i.e. an odd number[1]. So every room contributes an odd summand to D, which means that the number of rooms for a valid crypt layout must be even.

The number of rooms must not be divisible by four

We start by coloring the rooms of the crypt in a checkerboard pattern:



We split the red rooms into two groups based on the number of walls removed. Let r_1 be the number of red rooms with one wall removed and r_3 the number of red rooms with three walls removed.

We have previously shown that any valid crypt layout has an even number of rooms. Since a checkerboard with an even number of spaces evenly splits its spaces into the two colors, exactly half of the rooms are red:

$$r_1 + r_3 = \frac{n}{2},$$

where n is the number of rooms in the crypt.

Next, we note that for the entire crypt, we will remove exactly $n - 1$ walls: By the task statement, we are not allowed to remove more than that, and we can't connect all the rooms when removing less than that. Each wall is the wall of exactly one red (and one blue) room. This means that the number of walls removed is equal to the number of walls removed from red rooms, i.e. equal to the number of red rooms where one wall is removed plus three times the number of red rooms where three walls are removed:

$$r_1 + 3r_3 = n - 1$$

Combining those two equations gives us

$$r_3 = \frac{n-2}{4}$$

r_3, the number of red rooms with three walls removed, must be a whole number. But $\frac{n-2}{4}$ is not a whole number when n is divisible by four. So, if the number of rooms is divisible by four, no valid crypt layout exists.

[1] Actually, a room may also have zero walls removed. But since such a room would not be connected to the rest of the crypt, this can only happen in a one by one crypt. We don't have to consider this special case because it does not satisfy the limit $2 \leq N \cdot M$ given in the task statement.

Mirroring crypts

If we found a valid crypt with dimensions $N \times M$, we can mirror it to get a valid crypt with dimensions $M \times N$. For example, we can mirror the following 3×6 crypt

```
o.o-o-o.o.o
|...|...|.|
o-o-o-o-o-o
|.|...|...|
o.o.o-o-o.o
```

to obtain the following 6×3 crypt

```
o-o-o
..|..
o.o-o
|.|..
o-o.o
|.|.|
o.o-o
..|.|
o-o.o
..|..
o-o-o
```

Cases with width or height one

WloG assume that the height is one (otherwise, mirror the crypt).

It is possible to construct a 1×2 crypt:

```
o-o
```

It is not possible to construct a $1 \times M$ crypt for any $M > 2$. Since the height of the crypt is one, the only way to connect all the rooms is in a line. Therefore, we would have to tear down two walls of every room except for the outermost ones. But it is not allowed to tear down exactly two walls of a room, showing that we cannot construct a valid crypt. In the following example of a 1×3 crypt, the offending room is highlighted:

```
o-o-o
```

Cases with width or height two

WloG assume that the height is two (otherwise, mirror the crypt).

It is possible to construct a 2×1 as well as a 2×3 crypt:

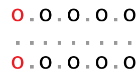
```
o
|
o
```

```
o-o-o
..|..
o-o-o
```

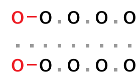
A valid 2×2 crypt is impossible since the number of rooms is divisible by four, which we showed is not allowed.

A valid $2 \times M$ crypt with $M > 3$ is also not possible. To see why, first consider the two leftmost

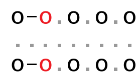
rooms. This is demonstrated below with a 2×5 room, but the argument works for any width larger than three.



Each of these rooms needs to have one wall removed (it can't have three walls removed as there are only two adjacent rooms). If we did this by connecting the two leftmost rooms to each other, they would not be connected to the rest of the crypt, so we will have the two leftmost rooms to the rooms on their right:



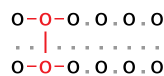
We now turn our attention to the second rooms from the left.



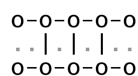
There are three possibilities of how many connections each of these rooms may have:

- Both rooms have one connection
- One of the rooms has one connection, the other has three
- Both rooms have three connections

In the fist case, the crypt would be disconnected. The second case can't happen: We can't give one of the rooms three connections without giving the other room a second connection. This only leaves the third option where both rooms have three connections:



Repeatedly applying this argument, we can show that the crypt will have to look like this:



In particular, all walls except the one on the very left and the one on the very right have to be torn down. This requires tearing down $3M - 4$ walls. For all M greater than 3, this is more than the allowed number of $2M - 1$. So, as we wanted to show, there are no valid $2 \times M$ crypts with $M > 3$.

Larger crypts

It remains to find valid crypts with width and height at least 3. As we've previously shown, we only have to consider crypts whose number of rooms is even, but not divisible by four.

Either the width or the height of the crypt has to be even, but not divisible by four. Assume WloG that it is the width (otherwise, mirror the crypt). Then, we know that the height is odd.

We now describe one possible way of constructing the solutions. It distinguishes between crypts with $N \bmod 4 = 1$ and crypts with $N \bmod 4 = 3$ (where N is the height).

N mod 4 = 1

In this case, we can use the following pattern:

```
o.o.o.o.o.o.o.o.o.o
|.|.|.|.|.|.|.|.|.|
o-o-o-o-o.o-o-o-o-o
|.......|.|.......|
o-o-o.o-o.o-o.o-o-o
|.|.....|.|.....|.|
o.o-o.o-o.o-o.o-o.o
..|.....|.|.....|..
o.o-o.o-o.o-o.o-o.o
|.|.....|.|.....|.|
o-o.o.o-o.o-o.o.o-o
|.|.|...|.|...|.|.|
o.o-o.o-o.o-o.o-o.o
..|.|...|.|...|.|..
o-o.o.o-o.o-o.o.o-o
..|.....|.|.....|..
o-o-o.o-o-o-o.o-o-o
```

Again, we can extend it to the right by repeating the red and purple part and to the bottom by repeating the blue and purple part. Crypts with width six can be obtained by completely omitting the red and purple parts. Crypts of height five can be obtained by omitting the blue and purple part.

N mod 4 = 3

In this case, we can use the following pattern:

```
o-o-o-o-o.o-o-o-o-o
..|.|.|.....|.|.|..
o-o.o.o-o.o-o.o.o-o
..|...|.....|...|..
o.o-o.o-o.o-o.o-o.o
|.|...|.....|...|.|
o-o.o.o-o.o-o.o.o-o
|.|.|.|.....|.|.|.|
o.o-o.o-o.o-o.o-o.o
..|.|.|.....|.|.|..
o-o.o.o-o.o-o.o.o-o
..|...|.....|...|..
o.o-o.o-o.o-o.o-o.o
|.|...|.....|...|.|
o-o.o.o-o.o-o.o.o-o
|.|.|.|.....|.|.|.|
o.o-o.o-o.o-o.o-o.o
..|.|.|.....|.|.|..
o-o.o.o-o-o-o.o.o-o
..|...|.|.|.|...|..
o-o-o.o.o.o.o.o-o-o
```

Runtime and memory usage

Let us consider the memory usage first. Note that both of the pattern consist of a constant number of building blocks, each of constant size. These patterns are the only thing we need to save. When outputting the grid, we can calculate the index and output the corresponding symbol from the pattern. Thus we only use $O(1)$ memory.

Coming to the running time of our algorithm we can check whether the grid exists in $O(1)$ by checking the conditions we established. If it exists we need $O(N \cdot M)$ time to iterate over all rooms and outputting the corresponding wall configurations from the patterns.

Implementation

The following python code constructs crypts using the patterns discussed above.

```python
def parse():
    n, m = list(map(int, input().split()))
    return n, m

def flip(map):
    def flip_char(c):
        if c == '-':
            return '|'
        if c == '|':
            return '-'
        return c
    return [
        ''.join(flip_char(map[j][i]) for j in range(len(map))) for i in range(len(map[0]))
    ]

def solve(n, m):
    if n * m == 1:
        return ['o']
    if n == 1 and m == 2:
        return ['o-o']
    if n == 2 and m == 1:
        return [
            'o',
            '|',
            'o',
        ]
    if n == 2 and m == 3:
        return [
            'o-o-o',
            '..|..',
            'o-o-o',
        ]
    if n == 3 and m == 2:
        return [
            'o.o',
            '|.|',
            'o-o',
            '|.|',
            'o.o',
        ]
    if min(n, m) <= 2:
        return None
    if (n * m) % 4 != 2:
        return None

    if n % 4 != 2:
        return flip(solve(m, n))

    assert n % 4 == 2
    if m % 4 == 1:
        start = [
            'o-o-o-o.' + 'o-o-o.o.' * (m // 4 - 1) + 'o',
            '..|.|...' + '..|...|.' * (m // 4 - 1) + '|',
            'o-o.o-o-' + 'o-o-o-o-' * (m // 4 - 1) + 'o',
            '..|.|.|.' + '|...|...' * (m // 4 - 1) + '|',
            'o-o.o.o.' + 'o.o-o-o.' * (m // 4 - 1) + 'o',
        ]
        join = '..|.....' + '........' * (m // 4 - 1) + '.'
        extend = [
            'o-' + 'o.' * (m - 2) + 'o',
            '..' + '|.' * (m - 2) + '|',
            'o-' + 'o-' * (m - 2) + 'o',
            '..' + '..' * (m - 2) + '|',
            'o-' + 'o-' * (m - 2) + 'o',
```

```
65              '..' + '|.' * (m - 2) + '|',
66              'o-' + 'o.' * (m - 2) + 'o',
67          ]
68      else:
69          assert m % 4 == 3
70
71          start = [
72              'o.o.' + 'o-o-o.o.' * (m // 4) + 'o',
73              '|.|.' + '..|...|.' * (m // 4) + '|',
74              'o-o-' + 'o-o-o-o-' * (m // 4) + 'o',
75              '|...' + '|...|...' * (m // 4) + '|',
76              'o-o.' + 'o.o-o-o.' * (m // 4) + 'o',
77          ]
78          join = '|...' + '........' * (m // 4) + '.'
79          extend = [
80              'o-' * (m - 2) + 'o-o',
81              '|.' * (m - 2) + '|..',
82              'o.' * (m - 2) + 'o-o',
83              '..' * (m - 2) + '|..',
84              'o.' * (m - 2) + 'o-o',
85              '|.' * (m - 2) + '|..',
86              'o-' * (m - 2) + 'o-o',
87          ]
88
89      lines = []
90      lines.extend(start)
91      lines.append(join)
92      for i in range(n // 4 - 1):
93          lines.extend(extend)
94          lines.append(join)
95      lines.extend(reversed(start))
96      return lines
97
98  def outformat(lines):
99      if lines is None:
100         return 'Impossible'
101     return 'Possible\n' + '\n'.join(lines)
102
103 if __name__ == '__main__':
104     for case in range(int(input())):
105         print(f"Case #{case}:", outformat(solve(*parse())))
```

Dada

Task Idea	Joël Huber
Task Preparation	Petr Mitrichev
Description English	Petr Mitrichev
Description German	Timon Gehr
Description French	Mathieu Zufferey
Solution	Petr Mitrichev
Correction	Timon Gehr

In this task, you are given a connected labyrinth that has N rooms and $N-1$ passages, in other words a tree. Each room has a bucket of paint of a unique color, and for each color we are given a list of rooms where we need to carry the corresponding bucket, and in the end we need to bring it back to where it was originally. We can only carry one bucket at a time, and our goal is to minimize the total number of times we carry a bucket along a passage.

Note that in the input file we were given the list of colors that need to be carried to a particular room instead of a list of rooms where a particular color needs to be carried, so we needed to convert the data from one representation to another as the first step.

Subtask 1: A single color (10 Points)

In this subtask there is just one bucket of paint in room 0 and $N \leq 1000$.

We will denote as *important* the rooms that either have the bucket in the beginning and at the end (room 0), or that need this bucket to complete the painting. Let us first make a few observations:

- For each passage such that there are important rooms in the parts of the tree on both sides of this passage, we need to carry the bucket along this passage at least twice (there and back).
- For each passage such that all important rooms are in one of the two parts that this passage separates the tree into, we do not need to carry the bucket along this passage.

It turns out that we can always find a way to carry the bucket in such a way that for each passage of the first kind we need to carry the bucket along it exactly twice. In order to see that, consider the case where we carry the bucket along it at least four times (twice there and twice back), and denote our overall route as A-there-B-back-C-there-D-back-E, where the letters from A to E denote whole segments of the route. But in that case we can make our route shorter by going A-there-B-D-back-C-E.

Therefore we just need to count the number of passages such that there are important rooms in the subtrees on both sides of it, and multiply this amount by 2. The most straightforward way to do it would run in $O(N^2)$, but we can easily improve to $O(N)$ by running a depth-first search that checks if a subtree has interesting rooms in it, starting it from some important room (for example, from room 0 which has the bucket initially). The interesting passages are those that lead to a subtree which has an important room in it.

Here is the main part of the code:

```
1 // Returns whether the subtree had any important rooms, and the total walking time inside it.
2 pair<bool, int> dfs(const vector<vector<int>>& adj, const vector<bool>& need, int at, int skip) {
3   bool any = need[at];
4   int res = 0;
5   for (int x : adj[at]) if (x != skip) {
6     auto [child_any, child_res] = dfs(adj, need, x, at);
```

```
7     if (child_any) {
8       any = true;
9       res += 1 + child_res;
10    }
11  }
12  return {any, res};
13 }
14
15 int solve(int start, const vector<vector<int>>& adj, const vector<bool>& need) {
16   return 2 * dfs(adj, need, start, -1).second;
17 }
```

The code uses a recursive lambda for the `dfs` function, which is a useful trick, but it was by no means necessary — one could have just used a normal function instead.

Subtask 2: Many colors (15 Points)

In this subtask we still have $N \leq 1000$, but now there is a bucket of paint in each room.

Since we do not care about the number of times we walk along the passage without a bucket in hand, the work related to each color is completely independent, and therefore we can just use the solution from subtask 1 for each color and add up the results.

Subtask 3: Apprentices (20 Points)

In this subtask we still have $N \leq 1000$, but there are also 'M' apprentices, and for each color you can send each of the apprentices along one passage from the vertex that has the bucket with that color initially, and they will take care of the rooms in that subtree, so that you only need to deal with the subtrees that no apprentice has gone to.

Thanks to the fact that we start the depth-first search from the vertex initially containing the bucket, the solution above already computes the number of steps we need for each subtree, and we just need to remove M largest ones before adding them up.

Here is the updated code:

```
1 // Returns whether the subtree had any important rooms, and the total walking time inside it.
2 pair<bool, int> dfs(const vector<vector<int>>& adj, const vector<bool>& need, int at, int skip,
3                     int num_free_children) {
4   bool any = need[at];
5   vector<int> children;
6   for (int x : adj[at]) if (x != skip) {
7     auto [child_any, child_res] = dfs(adj, need, x, at, 0);
8     if (child_any) {
9       any = true;
10      children.push_back(1 + child_res);
11    }
12  }
13  sort(children.begin(), children.end());
14  int res = 0;
15  for (int i = 0; i + num_free_children < children.size(); ++i) res += children[i];
16  return {any, res};
17 }
18
19 int solve(int start, const vector<vector<int>>& adj, const vector<bool>& need, int m) {
20   return 2 * dfs(adj, need, start, -1, m).second;
21 }
```

Subtask 4: A Huge Artwork (15 Points)

In this subtask we have a bigger tree: $N \leq 3 \cdot 10^5$, but we know additionally that the number of (a room, a color needed in this room) pairs also does not exceed $3 \cdot 10^5$.

We can no longer afford to run a separate depth-first search for each color. There are two directions that we can pursue to overcome this: either we process all colors together instead of one-by-one, or we make the processing of each color faster. It turns out that both options can lead us to a fast solution.

First solution

First, let us try to process all colors together in a single depth-first search. The depth-first search for subtasks 1 through 3 returns a boolean flag denoting if an important vertex for the current color appears in the subtree, and an integer denoting the number of passages that we have to traverse to visit all such vertices. Instead, we will now return a map from a color to a pair containing the number of this color's important vertices in the subtree, and the number of passages that we have to traverse to visit all such vertices.

Since we no longer can afford to root the tree at the vertex with the bucket for each color, we will encounter the vertex with the bucket at some point during the unified depth-first search. At this point the values returned from the children allow us to find out the work for an apprentice sent to one of the children, but we do not know the work that an apprentice sent towards the root would do. This last value we can obtain later by subtraction: when we reach the lowest vertex such that all important vertices for some color are in its subtree, we will know the total amount of work needed to reach all important vertices for this color, and we can subtract the work for all apprentices going towards the children from the bucket vertex to get the work for the apprentice going towards the root from the bucket vertex.

This solution still runs in $O(N^2)$ or maybe even in $O(N^2 \log N)$ because of the map. However, we can use the small-to-large technique to make it faster: when combining the maps coming from several children, we need to pass the maps by pointer or by reference, and merge the smaller maps into the biggest one, to avoid making a copy of the biggest one.

In addition to merging the maps, we also need to increase the second value (the total number of passages needed to traverse all important vertices in the subtree) for all elements in the map by 2. Doing this by traversing all elements of the map would once again lead to $O(N^2)$ running time, but we can avoid this by storing an additional integer *bonus* together with the map, increasing that integer by 2 instead, and adding that integer to the map value on every map read.

To prove that this works fast, first consider a slightly different approach: let us take the map with the largest sum of values as the main one when merging. Then every time we need to traverse an element the sum of values in the map containing it increases at least twice, because we merge its map into one with a bigger sum of values. Therefore we traverse each element at most $\log N$ times, and the total running time is either $O(N \log N)$ or $O(N \log^2 N)$, depending on whether we use a hash map or balanced tree. Now we can notice that our way of merging, that is choosing the map with the most elements as the main one, leads to the same or fewer operations than choosing the map with the largest sum of values, because we keep the same or more elements untouched at every step.

Note that running a depth-first search on a tree with $3 \cdot 10^5$ vertices can quite easily lead to a stack overflow, so one needs to know how to increase the default stack size for their programming environment, or to rewrite the depth-first search without using recursion.

Second solution

Alternatively, we can learn how to process each color faster. Even though we potentially have to walk the entire tree for each color, the number of important vertices can not be big for all colors, as their total number is bounded. So we need to learn to handle each color in time proportional to the number of its important vertices, potentially after some color-independent preprocessing.

If we know the order of visiting the important vertices of each color, and the grouping of

important vertices into parts accessible by one apprentice, then we just need to be able to quickly determine the distance between two vertices in a tree. This is a textbook problem that can be solved using the lowest common ancestor (LCA) algorithms that allow to answer such queries in $O(\log N)$ or even $O(1)$ after $O(N)$ or $O(N \log N)$ preprocessing, which is fast enough for our purposes.

It turns out that establishing the optimal order of visiting the important vertices is also not hard. Suppose that we first enumerate all vertices in the order the depth-first search visits them, also called the *preorder* traversal of the tree. Then it turns out that it is always optimal to visit the important vertices in this order, because when we do it this way each passage that has important vertices in its subtree is traversed two times (when we cross the boundaries of the subtree in the preorder traversal), which is exactly what we counted in Subtask 1.

Finally, the preorder traversal also gives us an easy way to identify the important vertices belonging to one apprentice, since each subtree forms a contiguous range of vertices in that order. It is also worth noting that the preorder traversal is a building block of many LCA algorithms, therefore we get it automatically while implementing that algorithm and the additional implementation on top of LCA remains manageable.

Subtask 5: Theoretical (40 Points)

First, let us state the answer, and then we will explain how to find it.

- When $N - 1 \le M$, $f(N, M) = 0$.
- When $\frac{N}{2} \le M < N - 1$ and $2 \le M$, $f(N, M) = \Theta(N - M)$.
- When $2 \le M < \frac{N}{2}$, $f(N, M) = \Theta(N \frac{\log N}{\log M})$.
- When $0 <= M <= 1$ and $M < N - 1$, $f(N, M) = \Theta(N^2)$.

The big-Theta notation $f(N, M) = \Theta(\dots)$ denotes both the lower bound up to a constant factor and the upper bound up to a constant factor at the same time. Since we are able to prove that the same formula is both the lower bound and the upper bound up to a constant factor, we know that this is the best possible solution and those bounds cannot be improved.

Note that because we do not care about a constant factor, there are many other equivalent ways to state this answer, for example $\frac{N}{2}$ can be replaced with $\frac{N}{10}$, or $\log M$ can be replaced by $\log 2M + 1$ when $M \ge 2$.

To arrive at this answer, first let us remove some insignificant parts of the problem statement. Since we are interested in the highest possible number of times a bucket is carried, we can assume that every color needs to be used in every vertex. Also, if we find the *centroid* of the tree (such vertex that after removing it all parts have at most $\frac{N}{2}$ vertices) and root the tree from the centroid, then for every vertex except the root we will send one of the apprentices towards the root, as it would cover the most vertices of all options for this apprentice (at least half of all vertices). Finally, for each edge along which Stofl carries the bucket he needs to carry it back, so the answer is always even, and we can compute half the answer since we do not care about the constant factor.

Now let us consider various cases. When $N - 1 \le M$, we always have enough apprentices to cover the entire tree, so Stofl can just rest, therefore $f(N, M) = 0$.

When $0 \le M \le 1$ and $M < N - 1$, consider a tree that is a chain with N vertices. For a vertex that is i edges away from the closest end of the chain Stofl will need to carry the bucket at least i times towards the closest end of the chain, while the apprentice (if there is one) can carry it towards the farthest end of the chain. Therefore Stofl's total effort would be at least $0 + 0 + 1 + 1 + ... + \lfloor \frac{N-1}{2} \rfloor = \Omega(N^2)$. On the other hand, even when there are no apprentices, there

are at most $N - 1$ edges to be traversed for each color (as a reminder, we are computing half the answer), so the total effort is at most $N \cdot (N-1) = O(N^2)$, therefore $f(N, M) = \Theta(N^2)$.

Now let us turn to the case $M \geq 2$. A chain would be completely covered by two apprentices, so we need a tree with more branching. Consider an almost-balanced tree that is built recursively from the root: we take the root, then split the remaining $N-1$ vertices into $2M$ groups of roughly equal size (some larger by 1 than others), and recursively construct a subtree for each group that is connected to the root. In case at some level of recursion there are less than $2M + 1$ vertices remaining, we just take the root and make all other vertices its children.

Since the apprentices can cover at most M children of every vertex (actually, they can cover at most $M - 1$ children for all vertices except the root because one of the apprentices has to go towards the root, but since we do not care about the constant factor such precision is not required), there will be at least M uncovered children for all vertices that have $2M$ children. And since the subtrees for each child are roughly the same in size, for each level of the tree Stofl will have to cover at least roughly half of all edges below that level himself.

The total number of vertices at each level is $2M$ times the number of vertices on the previous level, so the number of levels is around $\log_{2M} N = \frac{\log N}{\log 2M} = \Theta(\frac{\log N}{\log M})$. And for almost all levels more than half of all edges are below that level, so Stofl has to do $\Theta(N\frac{\log N}{\log M})$ work on this tree.

But is such tree really the worst case? It turns out it is, for the following reason: consider any tree rooted at its centroid as described above, and let us count the number of times Stofl has to visit a particular vertex. As we go from this vertex towards the root of the tree, Stofl will only go to the subtree containing this vertex if there are at least $M - 1$ other subtrees that are bigger, therefore the total size of the subtree multiplies by at least M when this happens, so this cannot happen more than $\log_M N = \frac{\log N}{\log M}$ times, therefore the total work is $O(N\frac{\log N}{\log M})$, and since our lower bound and upper bound are the same, we find that $f(N, M) = \Theta(N\frac{\log N}{\log M})$.

Why does the answer stated above have one more case? In the above proof we used terms like *roughly half* and *almost all*, and it turns out that we need to pay more attention to what that means. For example, when $\frac{N}{2} <= M$, the tree stops growing as we expect right from the root. To see what happens in that case, notice there is at most one vertex with degree more than M, so Stofl will only need to deal with that vertex, and the apprentices will still cover at least M of $N - 1$ other vertices, so Stofl's part of the work is at most $N - 1 - M$. On the other hand, on a star (a root connected to $N - 1$ other vertices), Stofl's part of the work is exactly $N - 1 - M$, so for this case $f(N, M) = \Theta(N - M)$.

When $M < \frac{N}{2}$ the first level of the tree works as expected, so Stofl needs to do at least $\Omega(N)$ work for the root of the tree, and we can check that the error introduced by *roughly half* and *almost all* is not significant enough to affect the overall $f(N, M) = \Theta(N\frac{\log N}{\log M})$ bound.

To see why this is the case, consider two possibilities. If $M + 1 < N \leq M^{20}$, then $1 < \frac{\log N}{\log M} \leq 20$, in other words this fraction is bound by a constant, and therefore $\Omega(N)$ (the amount of work we have to do for the root of the tree) is the same as $\Omega(N\frac{\log N}{\log M})$ for this case and we have a lower bound and an upper bound that match.

And when $N > M^{20}$, because $M \geq 2$ we can see that $N > M^{20} \geq (2M)^{10} > 1 + 2M + (2M)^2 + \cdots + (2M)^9$, therefore the tree has at least 11 levels (where the root is also counted as a level), and on all levels except the last two each vertex has exactly 2M children. On the i-th level Stofl's part of work will be at least $\frac{N-1-2M-(2M)^2 \cdots -(2M)^i}{2} - M$ (half of all vertices below that level minus M to account for the fact that the sizes of the parts can differ by 1), and $\frac{N-1-2M-(2M)^2 \cdots -(2M)^i}{2} - M > \frac{N}{4} - M > \frac{N}{8}$ for all levels except the last two. Therefore the total amount of Stofl's work is at least $\frac{N}{8}(\log_{2M} N - 4) \geq \frac{N}{8}\frac{\log_{2M} N}{2} = \Omega(N\frac{\log N}{\log M})$.

We have provided the solution in an informal fashion first, with terms like *roughly half* and *almost all* present, and then made it formal at the end, because we wanted to show how does one come up with such a solution. When writing down your solution during the round you did not need to write down the informal part, and could just focus on the formal proof.

Thermal Springs

Task Idea	Johannes Kapfhammer
Task Preparation	Daniel Rutschmann
Description English	Daniel Rutschmann
Description German	Timon Gehr
Description French	Mathieu Zufferey
Solution	Daniel Rutschmann
Correction	

In this task, you were given n thermal springs with characteristics t_i, m_i, s_i and p_i. Binna could go from spring a to spring b if $t_a < t_b$, $m_a < m_b$, $s_a > s_b$ and $p_a < p_b$.

Subtask 1: All the Springs (20 Points)

If Binna wants to bathe at all the springs, she has to visit the springs in order of increasing temperature t_i. Thus, we can sort the springs and check whether this gives a valid route.

```
def solve(n, t, m, s, p):
    order = list(sorted(range(n), key = lambda i : t[i]))
    for i in range(n-1):
        a = order[i]
        b = order[i+1]
        if not (t[a] < t[b] and m[a] < m[b] and s[a] > s[b] and p[a] < p[b]):
            print("NO")
            return
    print("YES")
    print(*order)

def parse():
    yield int(input())
    yield from (list(map(int, input().split())) for _ in range(4))

if __name__ == '__main__':
    for case in range(int(input())):
        print(f"Case #{case}:", end=' ')
        solve(*parse())
```

Subtask 2: As Many Springs as Possible (20 Points)

Similar to the first subtask, we can sort the springs by t_i. Then, use dynamic programming: Let DP$[i]$ be the length of the longest valid route ending at spring i. Such a route either starts at spring i, or consist of some route to spring j followed by going from j to i. This yields the formula

$$\mathrm{DP}[i] = \max\left(1, \max_{\substack{t_j<t_i \\ m_j<m_i \\ s_j>s_i \\ p_j<p_i}} (1 + \mathrm{DP}[j])\right)$$

Filling this DP table takes $O(N^2)$ time. To extract the longest route, it can be useful to store the optimal j for every i (in addition to the DP value).

```
def solve(n, t, m, s, p):
    order = list(sorted(range(n), key = lambda i : t[i]))
    dp = [1 for _ in range(n)]
    best = [-1 for _ in range(n)]

    for i,e in enumerate(order):
```

```
        for j,f in enumerate(order[:i]):
            if t[f] < t[e] and m[f] < m[e] and s[f] > s[e] and p[f] < p[e]:
                if dp[j] >= dp[i]:
                    dp[i] = dp[j] + 1
                    best[i] = j

    cur = max(range(n), key = lambda i : dp[i])
    print(dp[cur])
    out = []
    while cur >= 0:
        out.append(order[cur])
        cur = best[cur]
    out.reverse()
    print(*out)

def parse():
    yield int(input())
    yield from (list(map(int, input().split())) for _ in range(4))

if __name__ == '__main__':
    for case in range(int(input())):
        print(f"Case #{case}:", end=' ')
        solve(*parse())
```

Subtask 3: Two Mice (20 Points)

Once again, sort the springs by t_i. Then, use dynamic programming: Let DP[(i, j)] be 1 if $i < j$ and there is a pair of routes with one route ending at spring i and one route ending at spring j such that the springs $1, \ldots, j$ are each visited by at exactly one mouse, and 0 otherwise. From (i, j), either Stofl or Binna has to go to spring $j+1$ and we try both options, namely $(j, j+1)$ and $(i, j+1)$. (One can also interpret this as a BFS / DFS on a state graph.) In total, this uses $O(N^2)$ time.

```
#include <bits/stdc++.h>
using namespace std;

struct Spring{
    int t, m, s, p;
    int index;

    friend bool operator<(Spring const&a, Spring const&b){
        return a.t < b.t;
    }
};
bool valid(Spring const&a, Spring const&b){
    return a.t < b.t && a.m < b.m && a.s > b.s && a.p < b.p;
}
const int inf = 1.01e9;

void solve(){
    int n;
    cin >> n;
    vector<Spring> v(n);
    for(auto &e : v) cin >> e.t;
    for(auto &e : v) cin >> e.m;
    for(auto &e : v) cin >> e.s;
    for(auto &e : v) cin >> e.p;
    for(int i=0; i<n; ++i) v[i].index = i;
    // add two dummy springs to simplify the implementation
    v.push_back(Spring{-inf, -inf, inf, -inf, -99});
    v.push_back(Spring{inf, inf, -inf, inf, -99});
    n += 2;
    sort(v.begin(), v.end());

    vector<vector<int> > dp(n, vector<int>(n, 0));
    // dummy springs ensure we start at 0, 1
```

```
34     dp[0][1] = -1;
35     for(int j=0; j+1<n; ++j){
36         for(int i=0; i<j; ++i){
37             const int k = j+1;
38             if(dp[i][j]){
39                 if(valid(v[i], v[k])) dp[j][k] = 1+i;
40                 if(valid(v[j], v[k])) dp[i][k] = -1-j;
41             }
42         }
43     }
44     // dummy springs ensure we end at n-2, n-1
45     int i = n-2, j = n-1;
46     if(!dp[i][j]){
47         cout << "NO\n";
48         return;
49     }
50     cout << "YES\n";
51     vector<int> a, b;
52     while(i != 0 || j != 1){
53         assert(i < j);
54         b.push_back(v[j].index);
55         const int k = dp[i][j];
56         if(k > 0){
57             a.swap(b);
58             j = i;
59             i = k-1;
60         } else {
61             j = -1-k;
62         }
63     }
64     b.push_back(v[1].index);
65     reverse(a.begin(),a.end());
66     reverse(b.begin(),b.end());
67     // remove dummy springs
68     if(a.back() == -99) a.pop_back();
69     if(b.back() == -99) b.pop_back();
70     for(auto &e : a) cout << e << " "; cout << "\n";
71     for(auto &e : b) cout << e << " "; cout << "\n";
72 }
73
74 signed main(){
75     int T;
76     cin >> T;
77     for(int t=0; t<T; ++t){
78         cout << "Case #" << t << ":";
79         solve();
80     }
81 }
```

Subtask 4: Big Plans (40 Points)

In the last subtask, we need to reduce the number of states (i, j) we consider to $O(N)$. The key observation is the following: For any r, consider the minimal l such that all the springs $l, l+1, \ldots, r-1, r$ can be visited by the same mouse. (Then in particular, a mouse cannot go from spring $l-1$ to l.) Then

- there are no states (i, r) with $i < l-1$, as that would imply the mouse at r visited both the springs $l-1$ and l,
- we have to consider the state $(l-1, r)$, and
- out of all states (j, r) with $l \leq j < r$, we only care about the state with smallest j: Suppose there are states (j_1, r) and (j_2, r) with $l \leq j_1 < j_2 < r$, then by assumption a mouse could go from spring j_1 to j_2, hence any state reachable from (j_2, r) is also reachable from (j_1, r).

In particular, for each r, it suffices to consider the two states (k, r) with smallest or second-smallest

k. This reduces the running time to $O(N)$.

```
#include <bits/stdc++.h>
using namespace std;

struct Spring{
    int t, m, s, p;
    int index;

    friend bool operator<(Spring const&a, Spring const&b){
        return a.t < b.t;
    }
};
bool valid(Spring const&a, Spring const&b){
    return a.t < b.t && a.m < b.m && a.s > b.s && a.p < b.p;
}
const int inf = 1.01e9;

void solve(){
    int n;
    cin >> n;
    vector<Spring> v(n);
    for(auto &e : v) cin >> e.t;
    for(auto &e : v) cin >> e.m;
    for(auto &e : v) cin >> e.s;
    for(auto &e : v) cin >> e.p;
    for(int i=0; i<n; ++i) v[i].index = i;
    // add two dummy springs
    v.push_back(Spring{-inf, -inf, inf, -inf, -99});
    v.push_back(Spring{inf, inf, -inf, inf, -99});
    n += 2;
    sort(v.begin(), v.end());

    vector<vector<pair<int, int> > > dp(n);
    auto add = [&](int r, pair<int, int> s){
        // avoid duplicates
        if(count(dp[r].begin(), dp[r].end(), s)) return;
        // only keep the two minimal states
        dp[r].push_back(s);
        sort(dp[r].begin(), dp[r].end());
        if(dp[r].size() > 2) dp[r].pop_back();
    };
    dp[1].push_back(make_pair(0, -1));
    for(int j=0; j+1<n; ++j){
        for(auto const&[i, _] : dp[j]){
            const int k = j+1;
            if(valid(v[i], v[k])) add(k, make_pair(j, 1+i));
            if(valid(v[j], v[k])) add(k, make_pair(i, -1-j));
        }
    }
    if(dp[n-1].empty()){
        cout << "NO\n";
        return;
    }
    int i = dp[n-1][0].first, j = n-1;
    cout << "YES\n";
    vector<int> a, b;
    while(i != 0 || j != 1){
        b.push_back(v[j].index);
        int k = 0;
        for(auto &e : dp[j]) if(e.first == i){
            k = e.second;
        }
        if(k > 0){
            a.swap(b);
            j = i;
            i = k-1;
        } else {
```

```
67              j = -1-k;
68          }
69      }
70      b.push_back(v[1].index);
71      reverse(a.begin(),a.end());
72      reverse(b.begin(),b.end());
73      if(!a.empty() && a.back() == -99) a.pop_back();
74      if(!b.empty() && b.back() == -99) b.pop_back();
75      for(auto &e : a) cout << e << " "; cout << "\n";
76      for(auto &e : b) cout << e << " "; cout << "\n";
77  }
78
79  signed main(){
80      int T;
81      cin >> T;
82      for(int t=0; t<T; ++t){
83          cout << "Case #" << t << ":";
84          solve();
85      }
86  }
```

Trees

Task Idea	Benjamin Schmid
Task Preparation	Benjamin Schmid
Description English	Benjamin Schmid
Description German	Priska Steinebrunner
Description French	Mathieu Zufferey
Solution	
Correction	Benjamin Schmid

Final tournament results: `https://creativity.soi.ch/tournaments/2760`.