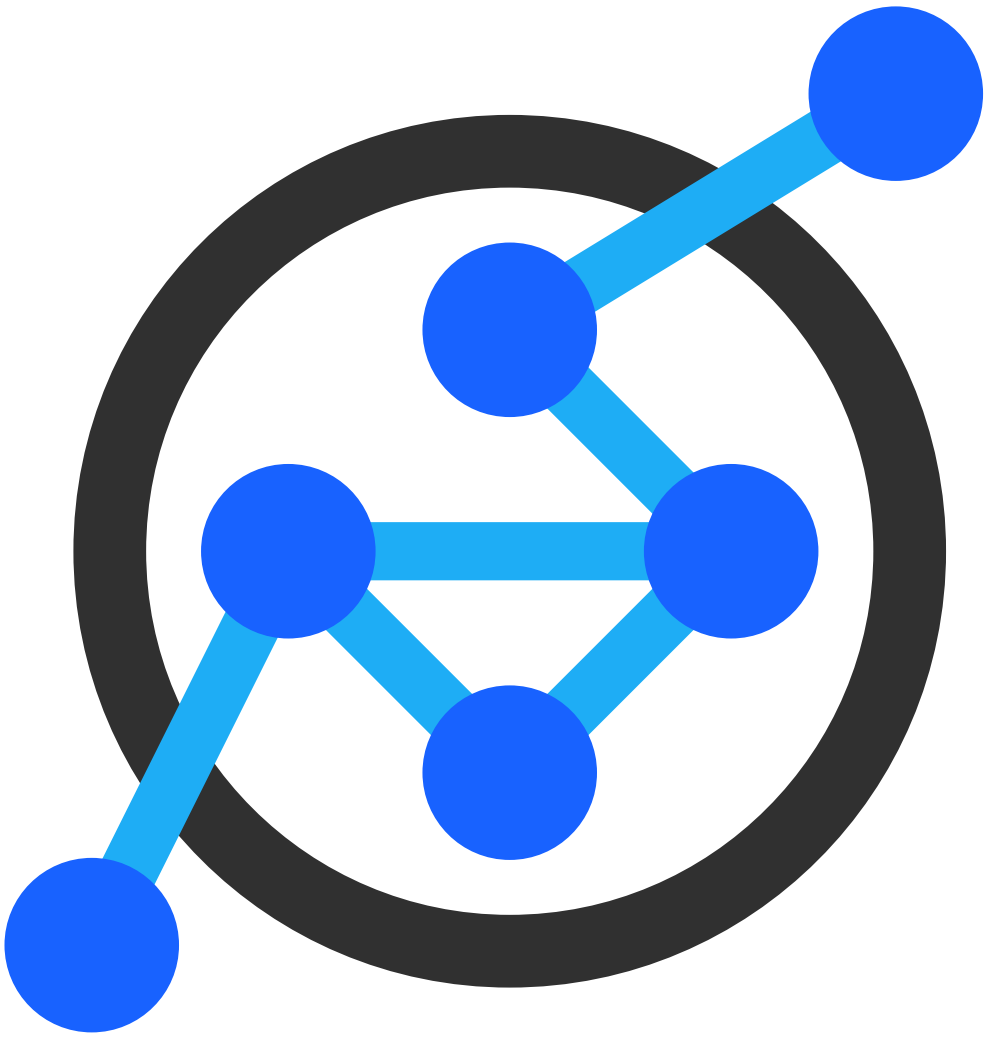
First Round SOI 2021/2022

Solution Booklet



Swiss Olympiad in Informatics

15 September – 30 November 2021

Peaks

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

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

The problem involved an array of N positive integers $h_0, \cdots, h_{N-1}$, presented in the story as the heights of the mountains on every x-coordinate of a postcard. There is a peak at a coordinate i on a postcard if and only if there are two coordinates with lower mountains directly to the left and the right of i (that is, both $i-1$ and $i+1$ are on the postcard and $h_{i-1} < h_i > h_{i+1}$).

Subtask 1: Small Picture (10 Points)

In the first subtask, $N = 3$ and you have to output "yes" if there is a peak on the postcard and "no" otherwise. Of course, a peak could only be found on coordinate 1, so one just needs to check whether $h_1 > h_0$ and $h_1 > h_2$. This is easily done with a simple conditional, using a constant amount of time and memory.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
	int T;
	cin >> T;
	for(int t = 0; t < T; t++) {
		int N;
		cin >> N;
		vector<int> h(N);
		for(int i = 0; i < N; i++) {
			cin >> h[i];
		}
		cout << "Case #" << t << ": "<<((h[0]<h[1]&&h[1]>h[2])?"yes":"no") << "\n";
	}
}
```

Subtask 2: Medium Picture (10 Points)

In this subtask, $N = 5$ and one must output the number of peaks on the postcard. There are now three candidate peaks at 1, 2 and 3. One can just check each of those individually like in the first subtask. This is still clearly feasible in constant time and memory, given the constant size of the input.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
	int T;
	cin >> T;
	for(int t = 0; t < T; t++) {
		int N;
		cin >> N;
		vector<int> h(N);
		for(int i = 0; i < N; i++) {
```

```
                        cin >> h[i];
                }
                int ans = 0;
                if(h[1]>h[0]&&h[1]>h[2]) ans++;
                if(h[2]>h[1]&&h[2]>h[3]) ans++;
                if(h[3]>h[2]&&h[3]>h[4]) ans++;
                cout << "Case #" << t << ": "<< ans << "\n";
        }
}
```

Subtask 3: Large Picture (30 Points)

In this subtask, N can reach 10^6. To check each possible coordinate, one might use a loop and apply the definition of a peak once again, keeping track of how many peaks one has found so far. Attention should be paid to the bounds of the loops, so that one doesn't check coordinates that don't exist (e.g. h_{-1}).

We loop through $N-2$ coordinates and check each possibility in constant time, so the running time is in $O(N)$. It is easiest to store the whole array of integers, using $O(N)$ additional memory. This is not optimal: it would suffice to store the last three integers while going through the loop. However, we are not very limited on space, so the solution below uses $O(N)$ memory for clarity, as will the one for subtask 5.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
        int T;
        cin >> T;
        for(int t = 0; t < T; t++) {
                int N;
                cin >> N;
                vector<int> h(N);
                for(int i = 0; i < N; i++) {
                        cin >> h[i];
                }
                int ans = 0;
                for(int i = 1; i <= N-2; i++) if(h[i]>h[i-1]&&h[i]>h[i+1]) ans++;
                cout << "Case #" << t << ": "<< ans << "\n";
        }
}
```

Subtask 4: Cut a Picture (20 Points)

N is quite small again (up to 10^4) and the input includes an integer $K \leq N$. Now, the task is to find a contiguous sequence of length K in the h_i's such that a postcard with K coordinates with heights $h_x, \ldots, h_{x+K-1}$ has as many peaks as possible. One should output the maximal number of peaks on such a postcard.

A trivial approach is to run the previous solution on every contiguous subsequence of length K in the input and to output the maximal answer. This runs in $O(N^2)$ (in the worst case, when $K = N/2$, we must check approximately $N^2/4$ possible peaks). This is fast enough given the size of the inputs.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
        int T;
        cin >> T;
        for(int t = 0; t < T; t++) {
                int N, K;
```

```
                cin >> N >> K;
                vector<int> h(N);
                for(int i = 0; i < N; i++) {
                        cin >> h[i];
                }
                int best = 0;
                for(int i = 0; i+K <= N; i++) {
                        int ans = 0;
                        for(int j = 1; j < K-1; j++) if(h[i+j]>h[i+j-1]&&h[i+j]>h[i+j+1]) ans++;
                        best = max(ans,best);
                }
                cout << "Case #" << t << ": "<< best << "\n";
        }
}
```

Subtask 5: Cut a Large Picture (30 Points)

The task is the same as the previous one, except N can reach 10^6 again. For that reason, the previous solution is too slow for the new inputs.

A key observation to make the solution faster is that we keep checking the same coordinates over and over. For example, imagine that $N = 10$ and $K = 5$. We check whether $h_4 < h_5 > h_6$ three times, for subsequences starting at 2, 3, and 4.

To solve this issue, we use a scanline technique. We compute the solution for the first subsequence $(h_0, \cdots, h_{K-1})$ as before. Then, we can compute the solution for $h_1, \cdots, h_K$ in constant time: the solution is the same as for the previous subsequence, minus 1 if there was a peak at h_1 (it cannot be a peak anymore, because it is on the edge of the postcard now), plus 1 if there is now a peak at h_{K-1} (there was not before, because it was on the edge of the postcard). We can then compute the solution for the subsequence starting at 2 in a similar fashion, and so on until we get to the end of the input. In total, this only requires a running time in $O(N)$: we check each coordinate's "peakness" only once. This is optimal and sufficient to get the full points for this subtask.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
        int T;
        cin >> T;
        for(int t = 0; t < T; t++) {
                int N, K;
                cin >> N >> K;
                vector<int> h(N);
                for(int i = 0; i < N; i++) {
                        cin >> h[i];
                }
                int cur = 0;
                for(int i = 1; i < K-1; i++) if(h[i]>h[i-1]&&h[i]>h[i+1]) cur++;
                int best = cur;
                for(int i = 1; i+K<=N; i++) {
                        if(h[i]>h[i-1]&&h[i]>h[i+1]) cur--;
                        if(h[i+K-2]>h[i+K-3]&&h[i+K-2]>h[i+K-1]) cur++;
                        best = max(best,cur);
                }
                cout << "Case #" << t << ": "<< best << "\n";
        }
}
```

Ropeways

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Bibin Muttappillil
Description French	Florian Gatignon
Solution	Jan Schär
Correction	Jan Schär

In this task, you are given the heights $h_0, h_1, \ldots, h_{N-1}$ of N islands on a line. Island 0 should be connected with island $N-1$ through a path of ropeways. Ropeways have a maximum length K (formally, a ropeway from island i to j can be built if $i < j$ and $j - i \leq K$). A ropeway from i to j is *boring* if $h_i \leq h_j$ (it needs to be motor-powered which is slow). The task was to determine the smallest achieveable number of boring ropeways. The input contains Q different values $k_0, \ldots, k_{Q-1}$ for K, the answer should be computed for each of them.

Subtask 1: Long Ropeways to Pulau Ngondo (10 Points)

In this subtask, we have $Q = 1$ and $k_0 = N - 2$. We cannot build a ropeway directly from 0 to $N-1$, but we can build all possible two-ropeway paths (any island other than the first and last can be used as the midpoint). We cannot achieve a smaller number of boring ropeways by building more than two ropeways. Thus, we can simply try out each island $1, \ldots, N-2$ as the middle island of a two-ropeway path, and take the best.

Running time $O(N)$.

Space usage $O(N)$.

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

void testcase () {
  int N, Q;
  cin >> N >> Q;

  vector<int> heights(N);
  for (int &h : heights) cin >> h;

  vector<int> maxDist(Q);
  for (int &k : maxDist) cin >> k;

  int ans = 2;
  for (int i = 1; i < N - 1; i++) {
    int boring = 0;
    if (heights[0] <= heights[i]) boring++;
    if (heights[i] <= heights[N - 1]) boring++;
    ans = min(ans, boring);
  }
  cout << ans;
}

signed main () {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    cout << "Case #" << t << ": ";
    testcase();
    cout << "\n";
  }
}
```

Subtask 2: Short Ropeways to Pulau Watubebek (20 Points)

Here, we have $Q = 2$, $k_0 = 1$ and $k_1 = 2$.

The case $k_0 = 1$ is easy, since there is only valid path: It contains all ropeways of length 1. We only need to count how many of them are boring.

The case $k_1 = 2$ is more challenging. Trying out every possible path is infeasible, because that would take time exponential in N. Instead, we make the following observation: Define d_i as the minimum number of boring ropeways for a path from 0 to i. If we already know d_{i-2} and d_{i-1}, we can easily compute d_i as follows: The ropeway before island i must have length 1 or 2. If the length is 1, then we get d_{i-1} boring ropeways, plus 1 if the ropeway $i-1$ to i is boring. If the length is 2, then we get d_{i-2} boring ropeways, plus 1 if the ropeway $i-2$ to i is boring. Then we just take the minimum of those two, and get d_i.

d_0 is equal to 0, and for d_1 the length of the last ropeway must be 1. We can compute the other d_i values in increasing order, and the final answer is then d_{N-1}. This technique is known as DP (Dynamic Programming).

Correctness I will argue that the output of this solution for the case $k_1 = 2$ is correct.

First, I will argue that the output cannot be less than the correct answer. The solution is constructive, which means that we can follow it backwards and see at each step which ropeway it has chosen (the one which had the minimum). Thus, we know that there is a path of ropeways where the number of boring ropeways is equal to the output.

Next, I will argue that the output cannot be greater than the correct answer. Assume towards a contradiction that there exists a better path with less than d_{N-1} boring ropeways. Take the first island i in this path where the number of boring ropeways up to i in the path is less than d_i. i cannot be 0, because $d_0 = 0$. This means there is an island j that is directly before i in the path, and since d_i is the first wrong value in the path, d_j must be correct. But when computing d_i in our algorithm, we consider every possible length of ropeway before i, including $i-j$, and we compute the number of boring ropeways based on d_j and whether the ropeway j to i is boring. And since we take the minimum, d_i cannot be greater than the length of the better path up to i. We arrive at a contradiction, and conclude that the output is correct.

Running time $O(N)$.

Space usage $O(N)$, we could even optimize it to $O(1)$.

```
int ans1 = 0;
for (int i = 0; i + 1 < N; i++) {
  if (heights[i] <= heights[i + 1]) ans1++;
}

int d1 = 0;
int d2 = 0;
if (heights[0] <= heights[1]) d1++;
for (int i = 2; i < N; i++) {
  int now = min(
    d1 + (heights[i - 1] <= heights[i] ? 1 : 0),
    d2 + (heights[i - 2] <= heights[i] ? 1 : 0));
  d2 = d1;
  d1 = now;
}
int ans2 = d1;

cout << ans1 << " " << ans2;
```

Note: The `c ? a : b` syntax means "if condition `c` is true, then `a`, else `b`". We could even just write `d1 + (heights[i - 1] <= heights[i])`, because in C++ a boolean can be implicitly converted to an integer, where false turns into 0 and true into 1.

Subtask 3: Fast Track to Pulau Jungok (30 Points)

In this subtask, we have $N \leq 5000$ and $Q \leq 30$, the k_j can be arbitrary. We can use the same DP approach as in the previous subtask, this time we need to consider k_j instead of just 2 lengths for the ropeway just before island i. For islands $i < k$, we only consider ropeway lengths up to i.

Correctness The argument is the same as in the previous subtask.

Running time $O(Q \cdot N^2)$.

Space usage $O(N)$.

```
for (int k : maxDist) {
  vector<int> dp(N, 0);
  for (int i = 1; i < N; i++) {
    int minBoring = N;
    for (int j = 1; j <= k && i - j >= 0; j++) {
      minBoring = min(minBoring, dp[i - j] + (heights[i - j] <= heights[i] ? 1 : 0));
    }
    dp[i] = minBoring;
  }
  cout << dp[N - 1] << " ";
}
```

Subtask 4: Galactic System of Ropeways to Pulau Glatik (40 Points)

In this subtask, the limits are much higher, we have $N \leq 1\,000\,000$ and $Q \leq 100$. The DP solution from before is too slow with these limits, but we can optimize it. The part which we want to speed up is finding the least boring predecessor for an island i.

Which predecessor is the best? Let's take two candidate predecessor islands, a and b. First, note that if $d_a < d_b$, then a is always at least as good as b. Even if $d_a = d_b - 1$, the ropeway a to i is boring, and the ropeway b to i is fun, we get the same number of boring ropeways for both choices, so it is always safe to choose a. If $d_a = d_b$ and $h_a > h_b$, then again a is always at least as good as b. It is possible that ropeway a to i is fun and b to i is boring, but the other way around is not possible. To summarize, we a is better than b if either $d_a < d_b$, or if $d_a = d_b$ and $h_a > h_b$.

The only additional constraint is that we can only choose a candidate if it is not too far left, or the ropeway would be too long. Once a candidate is too far away, we have to choose the next best candidate that is still in reach. This means: What we need is a queue which contains the best candidate in the first position, the next best candidate that is further to the right in the second position, and so on. Once the first candidate is too far away, we remove it from the queue, such that the first entry will be the best candidate within reach. Then, because we know that the first entry j of the queue is the best candidate, we can easily compute d_i as d_j, plus one if the ropeway j to i is boring. Next, we always have to add i to the end of the queue, because at some point all other entries will be too far away, such that i will be the best candidate that we know so far. But before that, we have to remove entries from the end of the queue if they are worse than i. This ensures that we actually have the best candidate in the first entry, and the next best in the second entry, and so on.

Correctness The algorithm breaks if the queue ever gets empty, so I argue that this never happens. Initially, the queue contains the element 0. When processing island i, we always have $i - 1$ at the end of the queue, this will never be too old and we will not remove it from the front of the queue. When removing candidates from the back of the queue, we will never remove the best candidate, since island i is either lower than the best candidate, or it is more boring to get there.

Next, I argue that the algorithm always constructs a correct ropeway, which then means that the algorithm is constructive and thus its output is not too low. Each island in the queue is strictly newer (more to the right) than the one before it. In each iteration, we remove the first candidate from the queue if it is too old. Hence, we always have at most one island which is too old, which will be removed at the start of the iteration. Then, the first item in the queue must be a valid

candidate.

Finally, I argue that there is no path which is less boring than the output. From the previous subtask, we know that this is the case if we always choose the best candidate.

We put all candidates into the queue, and we only remove candidates in two cases: We remove a candidate if it is too old. It will always be too old after that point. We remove a candidate if there is a newer candidate which is better. For each point where the removed candidate will be valid from that point on, the newer candidate will also be valid, so it always makes sense to choose that one instead of the removed candidate.

Now we know that the best candidate is in the queue; it remains to show that it is at the front of the queue. Whenever we add an element to the queue, we remove elements from the end, such that the added candidate is not better than the entry before it. This means that each element in the queue is not better than the one before it, and this means that the best candidate is at the front.

Running time $O(Q \cdot N)$.

Space usage $O(N)$.

```
for (int k : maxDist) {
  vector<int> dp(N, 0);
  deque<int> best;
  best.push_back(0);
  for (int i = 1; i < N; i++) {
    if (best.front() < i - k) best.pop_front();
    dp[i] = dp[best.front()] +
      (heights[best.front()] <= heights[i] ? 1 : 0);
    while (dp[best.back()] > dp[i] ||
           (dp[best.back()] == dp[i] &&
            heights[best.back()] <= heights[i])) {
      best.pop_back();
    }
    best.push_back(i);
  }
  cout << dp[N - 1] << " ";
}
```

Thanks to Elias Bauer for discovering this slightly simpler variant of the solution we found.

There are many other ways to solve this problem using various kinds of data structures, for example with a priority queue, with a stack, or with a segment tree.

T-Shirts

Task Idea	Johannes Kapfhammer
Task Preparation	Ema Skottova
Description English	Ema Skottova
Description German	Ema Skottova
Description French	Mathieu Zufferey
Solution	Timon Gehr
Correction	

There are N mice and M t-shirts. The i-th mouse can wear t-shirts of size between l_i and r_i, where the j-th t-shirt has size s_j. Our goal is to distribute as many t-shirts as possible to mice that can wear them. I.e., we want to find an assignment of t-shirts to mice such that each mouse gets at most one t-shirt and each t-shirt is given to at most one mouse.[1] If mouse i receives t-shirt j, it must hold that $l_i \leq s_j \leq r_i$, and the number of mice receiving t-shirts should be maximized.

Subtask 1: One Mouse (5 Points)

In this subtask, there is only one mouse ($N = 1$) and one t-shirt ($M = 1$), and we have to check if the t-shirt fits the mouse.

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

void solve_testcase() {
    int l, r, s;
    cin >> l >> r >> s;

    if (l <= s && s <= r) {
        cout << "YES\n"; // t-shirt fits
    } else {
        cout << "NO\n";  // t-shirt does not fit
    }
}

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int t;
    cin >> t;

    for (int i = 0; i < t; i++) {
        cout << "Case #" << i << ": ";
        solve_testcase();
    }
}
```

The running time of this solution is $O(1)$ per test case, as we execute at most a constant number of operations.

Subtask 2: Same Size (10 Points)

In this subtask, there are N t-shirts ($M = N$) that all have the same size S. As there are enough t-shirts so that every mouse could be given one, and all t-shirts are exchangeable, it suffices to compute how many of the N mice can wear a t-shirt of size S:

[1] Such an assignment is sometimes called a *matching*.

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

void solve_testcase() {
    int n;
    cin >> n;
    vector<int> l(n), r(n);
    for (int i = 0; i < n; i++) {
        cin >> l[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> r[i];
    }
    int s;
    cin >> s;

    int result = 0;

    for (int i = 0; i < n; i++) {
        if (l[i] <= s && s <= r[i]) {
            result++; // give a t-shirt to mouse i
        }
    }
    cout << result << "\n";
}
```

The running time of this solution is $O(N)$ per test case, as we execute a few for-loops with N iterations.

Subtask 3: Rats (20 Points)

In this subtask, we only have to ensure that all distributed t-shirts are large enough to fit the respective mouse.

Our solution will be a *greedy* algorithm. A greedy algorithm builds a solution in steps. In each step, the algorithm makes a decision about the final result that is never revisited later. Greedy algorithms are usually quite intuitive, but they often result in suboptimal solutions. To show that a given greedy algorithm is actually optimal, we have to show that whenever it makes a decision, the result can still be extended to an optimal solution.

Consider a smallest t-shirt j and a mouse i whose t-shirt size lower bound l_i is smallest.

If $s_j < l_i$, t-shirt j is too small to fit any mouse. We can ignore it and only consider the remaining t-shirts, as it cannot appear in any solution, including every optimal solution.

Otherwise $l_i \leq s_j$. We will give t-shirt j to mouse i, which can indeed wear it. We now have to argue that there is some optimal solution that also gives t-shirt j to mouse i. Consider an arbitrary optimal solution.[2] If this solution gives t-shirt j to mouse i, we are done. Otherwise, the optimal solution gives another t-shirt j' to mouse i or it gives t-shirt j to another mouse i' (or both). (This has to be the case, because otherwise we could just take the optimal solution and additionally give t-shirt j to mouse i, to obtain an even better solution, which is not possible.)

If the optimal solution did not give another t-shirt to mouse i, we can just take t-shirt j from whatever mouse the optimal solution gave it to and give it to mouse i instead.

If the optimal solution gives another t-shirt j' to mouse i, we can modify the optimal solution by giving t-shirt j to mouse i instead. If there is a mouse i' that had been given t-shirt j, we can just give t-shirt j' to mouse i' instead. Mouse i' can wear t-shirt j' as it is at least as large as t-shirt j (which is the smallest t-shirt overall).

In each case, the resulting solution still has the same overall size, so it remains optimal.

Therefore, there is an optimal solution that gives t-shirt j to mouse i. Once we have made this assignment, we can optimally solve the new test instance we obtain by removing both mouse i and t-shirt j. I.e., we just continue in the same fashion until we run out of t-shirts or mice.

[2] We do not know this solution, but we can still reason about it using case distinction.

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

void solve_testcase() {
    int n, m;
    cin >> n >> m;
    vector<int> l(n), s(m);
    for (int i = 0; i < n; i++) {
        cin >> l[i];
    }
    for (int i = 0; i < n; i++) {
        int r;
        cin >> r; // ignore upper bound
    }
    for (int j = 0; j < m; j++) {
        cin >> s[j];
    }

    sort(l.begin(), r.end());
    sort(s.begin(), s.end());

    int result = 0;

    for (int i = 0, j = 0; i < n && j < m;) {
        if (l[i] <= s[j]) {
            result++; // give t-shirt j to mouse i
            i++; // remove mouse i
            j++; // remove t-shirt j
        } else {
            // t-shirt j does not fit any remaining mouse:
            j++; // remove t-shirt j
        }
    }
    cout << result << "\n";
}
```

The running time of this solution is $O(N \log N + M \log M)$ per test case. (We can sort an array of n numbers in time $O(n \log n)$.)

Subtask 4: Everything (25 Points)

For this subtask, we will now also have to consider upper bounds on t-shirt sizes. We still want to use a greedy algorithm with a similar correctness proof as for the previous subtask. We will consider the algorithm above and see what goes wrong in the correctness proof if there are upper bounds. If $s_j < l_i$, we can still ignore t-shirt j, as it is too small for any mouse to wear. However, if $l_i \leq s_j$, we may not be able to give t-shirt j to mouse i, because it may be too large. This is however easy to fix: if $r_i < s_j$, then there is no t-shirt that fits mouse i, so we can ignore mouse i.

Otherwise, we have $l_i \leq s_j \leq r_i$, and we could give t-shirt j to mouse i. Let's see what happens if we give t-shirt j to mouse i. We will again consider some optimal solution. If that solution gives t-shirt j to mouse i, or if it does not give another t-shirt j' to mouse i, or if it does not give t-shirt j to another mouse i', everything works out like before. However, if the optimal solution gives another t-shirt j' to mouse i and it gives t-shirt j to another mouse i', we run into some trouble: While it is certainly possible to give t-shirt j to mouse i instead, t-shirt j' may be too large to give to mouse i'!

This problematic case can happen when $r_{i'} < s_{j'} \leq r_i$, and this gives us a hint how to fix the problem: It would be good if there wasn't a mouse i' that can wear t-shirt j and $r_{i'} < r_i$. So instead of choosing as mouse i the mouse with smallest l_i, we will pick an i such that $l_i \leq s_j \leq r_i$ and r_i is minimized.

We now have to make sure that this modification does not interfere with the validity of any earlier steps in the correctness proof, but it is indeed the case: We did not use that l_i is the smallest possible lower bound, it was only important that t-shirt j fits mouse i. Futhermore, the problematic case now goes through: if there is a mouse i' such that the optimal solution gives

t-shirt j to mouse i' and there is a t-shirt j' such that the optimal solution gives t-shirt j' to mouse i, we can now swap them.

Mouse i can wear t-shirt j by construction. We will now show that mouse i' can wear t-shirt j'. We need to show that $l_{i'} \leq s_{j'} \leq r_{i'}$. Because the optimal solution gave t-shirt j to mouse i' and s_j is a smallest t-shirt, we know that $l_{i'} \leq s_j \leq s_{j'}$. And because the optimal solution gave t-shirt j' to mouse i and r_i is the smallest possible among all mice that can wear t-shirt j[3], we know that $s_{j'} \leq r_i \leq r_{i'}$. Therefore, mouse i' can wear t-shirt j'.

This concludes the proof that there is an optimal solution that gives t-shirt j to mouse i.

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

void solve_testcase() {
    int n, m;
    cin >> n >> m;
    vector<int> l(n), r(n), s(m);
    for (int i = 0; i < n; i++) {
        cin >> l[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> r[i];
    }
    for (int j = 0; j < m; j++) {
        cin >> s[j];
    }

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

    int result = 0;

    for (int j = 0; j < m;) {
        int i = -1; // find mouse i: t-shirt has to fit and r[i] should be minimal
        for (int i_cand = 0; i_cand < n; i_cand++) {
            if (l[i_cand] <= s[j] && s[j] <= r[i_cand]) {
                // t-shirt fits
                if (i == -1 || r[i_cand] < r[i]) {
                    i = i_cand; // found a better candidate for mouse i
                }
            }
        }
        if (i != -1) {
            result++; // give t-shirt j to mouse i
            l[i] = r[i] = -1; // remove mouse i
            j++; // remove t-shirt j
        } else {
            // t-shirt j does not fit any remaining mouse
            j++; // remove t-shirt j
        }
    }
    cout << result << "\n";
}
```

This solution runs in time $O((N + \log M) \cdot M)$ per test case, which is fast enough for this subtask.

Subtask 5: Many Participants (40 Points)

Now there can be up to 10^6 mice and t-shirts, so the solution for the previous subtask is no longer fast enough. Conceptually, the solution remains the same, but we will have to find mouse i for a given t-shirt j more efficiently.

To this end, we maintain a priority queue q that contains all mice that can wear t-shirt j. The priority queue is sorted by upper bounds, such that the top entry t has the smallest upper bound r_t.

[3](We know mouse i' can wear t-shirt j too, because the optimal solution gave t-shirt j to mouse i'.)

When we update j, it is easy to maintain the priority queue: we can first add all mice whose lower bound is now satisfied, then we can remove all mice whose upper bound is now violated.

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

void solve_testcase() {
    int n, m;
    cin >> n >> m;
    vector<int> l(n), r(n), s(m);
    for (int i = 0; i < n; i++) {
        cin >> l[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> r[i];
    }
    for (int j = 0; j < m; j++) {
        cin >> s[j];
    }

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

    // create vector with all mice:
    vector<int> mice(n);
    iota(mice.begin(), mice.end(), 0); // fill with values 0,1,...,n-1

    // sort mice according to lower bound:
    sort(mice.begin(), mice.end(), [&](int i, int j){ return l[i] < l[j]; });

    // create a priority queue of mice sorted according to upper bound:
    auto cmp = [&](int i, int j){ return r[j] < r[i]; };
    priority_queue<int, vector<int>, decltype(cmp)> q(cmp);

    int result = 0;

    for (int cur = 0, int j = 0; j < m;) {
        // add mice for which t-shirt j is now large enough:
        while (cur < n && l[mice[cur]] <= s[j]) {
            q.push(mice[cur++]);
        }
        // remove mice for which t-shirt j is now too large:
        while (!q.empty() && r[q.top()] < s[j]) {
            q.pop();
        }
        if (!q.empty()) {
            result++; // give t-shirt j to mouse i
            q.pop(); // remove mouse i (i is q.top())
            j++; // remove t-shirt j
        } else {
            // t-shirt j does not fit any remaining mouse
            j++; // remove t-shirt j
        }
    }
    return result;
    cout << result << "\n";
}
```

This solution runs in time $O(N \log N + M \log M)$ per test case.

Claw Sort

Task Idea	Johannes Kapfhammer
Task Preparation	Jan Schär
Description English	Jan Schär
Description German	Jan Schär
Description French	Mathieu Zufferey
Solution	Johannes Kapfhammer
Correction	

Clawsort was about sorting an array of integers using a sequence of the operations "<" and ">".

Subtask 1: Simulate the robot (15 Points)

In the first subtask, you had to simulate a sequence of operations.

The empty claw can be represented by some invalid value (for example -1), then it is easier to avoid special cases.

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

int main() {
  int testcases; cin >> testcases;
  for (int testcase = 0; testcase < testcases; ++testcase) {
    int n; cin >> n;
    vector<int> v(n);
    for (auto& x : v) cin >> x;
    string s; cin >> s;

    int claw = -1;
    int pos = 0;
    for (char c : s) {
      if (c == '.')
        break;
      swap(v[pos], claw);
      pos += c == '>' ? 1 : -1;
    }
    cout << "Case #" << testcase << ":";
    for (int x : v)
      cout << ' ' << x;
    cout << "\n";
  }
}
```

Subtask 2: Sort three cards (10 Points)

For all of the other subtasks, there were plenty of strategies that work. In this booklet, we show just one of those strategies.

With only 3 cards, there are only 3! = 6 (3 factorial) possible inputs. One approach that does require the least thinking is just hardcoding all of them.

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

string solve(vector<int> a) {
  map<vector<int>, string> operations = {
    {{0,1,2}, ""},
    {{0,2,1}, ">><<>"},
    {{1,0,2}, "><>"},
    {{1,2,0}, ">><<>><"},
    {{2,0,1}, "><>><>"},
```

```
    {{2,1,0}, ">><><<"},
  };
  assert(a.size() == 3);
  assert(operations.count(a));
  return operations[a];
}

int main() {
  int testcases; cin >> testcases;
  for (int testcase = 0; testcase < testcases; ++testcase) {
    int n; cin >> n;
    vector<int> a(n);
    for (auto& x : a)
      cin >> x;
    cout << "Case #" << testcase << ": " << solve(a) << ".\n";
  }
}
```

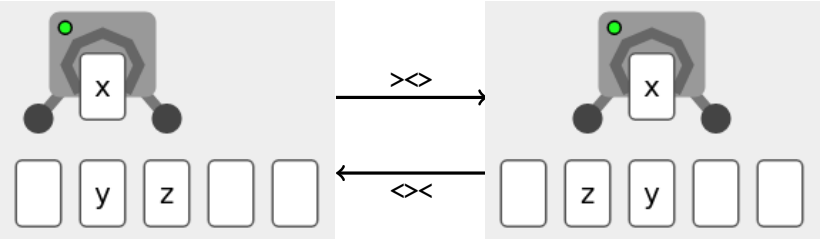
Subtask 3: Reverse the cards (20 Points)

There are many ways to reverse the cards, but one way that leads to a fast solution uses the following idea:

- $[\underbrace{n-1}_{-1}, n-2, n-3, \ldots, 2, 1, 0]$: drag $n-1$ and $n-2$ to the end.
- $[-1, n-3, \ldots, 2, \underbrace{1}_{0}, n-1, n-1]$: drag 0 and 1 to the start.
- $[0, 1, \underbrace{n-3}_{-1}, \ldots, 2, n-2, n-1]$

Now notice that elements $0, 1$ and $n-2, n-1$ are at the correct positions. So all we need to do now is reverse the range from $n-3, \ldots, 2$. This is the same problem as before, but the range is 4 smaller. So we can iterately apply this procedure until we are with only 0, 1, 2 or 3 elements left, which we have to handle specially.

How can we achieve "dragging" some elements to the end? With the "fish" pattern: ><>.



We first have to get into the state where the two elements we want to drag along are on top of each other. This can be done by a single >. Then we chain n-2 fishes to drag those elements to the right, and we do the same backwards again.

The code below achieves this. Only the `solve` function is shown, the function `main` is the same as in the previous subtask.

```
string repeat(string s, int n) {
  string t;
  for (int i=0; i<n; ++i)
    t += s;
  return t;
}

string solve(vector<int> a) {
  string ans;
  int n = a.size();
  while (n >= 4) {
```

```
    ans += ">";                 // [-1,<claw=n-2>n-1,..]
    ans += repeat("><>", n-2); // [-1,...,<claw=n-2>n-2]
    ans += "<<";                // [-1,..,<claw=0>1,n-1,n-2]
    ans += repeat("<><", n-3); // [<claw=0>1,...,n-1,n-2]
    ans += ">>";                // [0,1,<claw=-1>2,...,n-1,n-2]
    ans -= 4;                   // now solve the same subproblem but with 4 elements less
  }
  if (n == 2)
    ans += "><>";
  else if (n == 3)
    ans += ">><><<>";
  cerr << a.size() << ": " << ans.size() << "\n";
  return ans;
}
```

Subtask 4: Sort a lot of cards (55 Points)

We can generalize the above idea to if our array is not sorted: Say, we are at the start and our two relevant elements are somewhere in the middle. Let's denote with a the element that comes first, either $n-1$ or $n-2$, and with b the other one.

$$[\text{Robot}, \ldots, a, \ldots, b, \ldots]$$

We want to move right (with a sequence of >'s) and then pick up element $n-2$. Then, we need to drag that one along with a sequence of fish patterns (><>). Just before we are at b, we want to move onto it with a single > so that both are stacked on top of each other.

$$[\ldots, \underset{x}{\underline{a}}, b, \ldots] \rightsquigarrow [\ldots, x, \underset{a}{\underline{b}}, \ldots]$$

Then we continue with our sequence of fish patterns until the end, where we unpack them with <<.

$$[\ldots, y, x, \underset{a}{\underline{b}}] \rightsquigarrow [\ldots, \underset{x}{\underline{y}}, b, a]$$

Then we do the same thing in reverse.

The general idea works, but we need to take care of a few special cases, such as when b, a is not the right order, or when they are right at the start. There are no new ideas to address those, just a bit of trial and error to get it right.

Note that when we 3 or less elements left, we can just apply our solution for subtask 2.

```
struct Robot {
  int pos=0;
  int claw=-1;
  int n;
  vector<int> a;
  string cmd;

  Robot(vector<int> a) : n(a.size()), a(move(a)) {}

  void go(string_view operations) {
    for (char c : operations) {
      assert(c == '>' || c == '<');
      swap(a[pos], claw);
      pos += c == '>' ? 1 : -1;
      assert(pos >= 0 && pos < n);
    }
    cmd += operations;
  }
  int index(int i) {
    if (claw == i) return pos;
    return find(a.begin(), a.end(), i) - it;
  }
  bool is_sorted() {
    return claw == -1 && std::is_sorted(a.begin(), a.end());
```

```
  }
};

string solve(vector<int> a) {
  int n = a.size();
  Robot robot(move(a));
  if (robot.is_sorted())
    return "";

  int l = 0;
  int r = n;

  while (r - l > 3) {
    // move forward and sort r-1 and r-2
    int a = robot.index(r-2);
    int b = robot.index(r-1);

    // fix edge case where we start on top of the target
    // then we want to shuffle around so we end up in the right state
    if (min(a, b) == robot.pos) {
      robot.go(">><<><><>");
      a = robot.index(r-2);
      b = robot.index(r-1);
    }

    robot.go(repeat(">", min(a, b) - robot.pos));
    while (robot.pos < r-1)
      if (robot.a[robot.pos+1] == r-1 || robot.a[robot.pos+1] == r-2)
        robot.go(">");
      else
        robot.go("><>");

    robot.go("<");
    if (robot.a[r-1] == r-2)
      robot.go("<>><");
    robot.go("<");

    r -= 2;

    // move backward and sort l and l+1
    a = robot.index(l);
    b = robot.index(l+1);

    if (r-l <= 3) {
      robot.go(repeat("<", robot.pos - l));
      robot.go("><><");
      break;
    }

    if ((robot.claw == l || robot.claw == l+1) &&
        !(robot.a[robot.pos] == l || robot.a[robot.pos] == l+1)) {
      robot.go("<><>");
    }

    robot.go(repeat("<", max(a, b) - robot.pos));
    while (robot.pos > l)
      if (robot.a[robot.pos-1] == l || robot.a[robot.pos-1] == l+1)
        robot.go("<");
      else
        robot.go("<><");

    robot.go(">");
    if (robot.a[l] == l+1)
      robot.go("><<>");
    robot.go(">");

    l += 2;

```

```
    if (robot.claw != -1) {
      robot.go(">><<");
      if (robot.is_sorted())
        break;
      if (r-l <= 2) // might have destroyed the order
        ++r;
    }
  }
  assert(robot.claw == -1);
  if (!robot.is_sorted()) {
    assert(l == robot.pos);
    vector<int> v(robot.a.begin()+l, robot.a.begin()+r);
    for (auto& x : v) {
      assert(l <= x && x < r);
      x -= l;
    }
    if (r-l == 3) {
      map<vector<int>, string> operations = {
        {{0,1,2}, ""},
        {{0,2,1}, "><><>"},
        {{1,0,2}, "><>"},
        {{1,2,0}, "><<><><"},
        {{2,0,1}, "><><><>"},
        {{2,1,0}, "><<><<>"},
      };
      robot.go(operations[v]);
    } else if (r-l == 2) {
      robot.go("><>");
    }
  }

  assert(is_sorted(robot.a.begin(), robot.a.end()));
  assert(robot.claw == -1);
  return robot.cmd;
}
```

The above code has optimal worst-case, but is a bit wasteful for some inputs that are partially sorted (it would give 52/55 points for this subtask).

It can be improved by hardcoding a few more special cases, most notably the case when b comes before a.

The reference solution that was used for scoring can be found here: `https://soi.ch/media/files/clawsort-sub4-reference-solution.py`

Dancing Mice

Task Idea	Joël Mathys, Johannes Kapfhammer
Task Preparation	Petr Mitrichev
Description English	Petr Mitrichev
Description German	Johannes Kapfhammer
Description French	Mathieu Zufferey
Solution	Petr Mitrichev
Correction	Petr Mitrichev

N mice are dancing in a circle, at each moment in time each mouse may dance together with at most one of its neighbors. You need to design a dance with the shortest overall duration that allows mice i and $i+1$ (or $N-1$ and 0 for $i = N-1$) to dance as a pair for at least t_i seconds for each i.

The statement of this problem is very similar to the problem Storytelling from 2019 Round 1. The key difference is that in that older problem, the dance for each pair must have been happening in one uninterrupted segment, while here we can split it into multiple parts. It turns out that while some ideas remain the same, most of the solution has to change because of this.

Subtask 1: Three Mice (5 Points)

In this subtask, $N = 3$. Therefore only one pair of mice can dance together at the same time, so we just need to stack together the three segments, one for each pair of mice. The duration of the resulting dance is simply the sum of the input durations. Here is the code:

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

vector<pair<string, double>> solve(int n, const vector<int>& t) {
  return {
    {"<>.", t[0]},
    {".<>", t[1]},
    {">.<", t[2]},
  };
}

// This part is the same for all tasks.
int main() {
  int num_tests;
  cin >> num_tests;
  cout.precision(20);
  for (int test_id = 0; test_id < num_tests; ++test_id) {
    int n;
    cin >> n;
    vector<int> t(n);
    for (auto& x : t) cin >> x;
    auto res = solve(n, t);
    cout << "Case #" << test_id << ": " << res.size() << "\n";
    for (auto p : res) {
      cout << p.second << " " << p.first << "\n";
    }
  }
  return 0;
}
```

Subtask 2: All the Same (10 Points)

In this subtask, $t_i = 1$ for all i. The two example cases demonstrate the two patterns that we need to apply.

In case N is even, we can have just two segments, in the first segment each even-numbered

mouse will dance with the next mouse, and in the second segment each even-numbered mouse will dance with the previous mouse. The total duration of this dance is always 2.

In case N is odd, it is not as simple as in each segment one of the mice will not be dancing at all. Once the mouse that is not dancing is chosen, the pairings of the other mice follows uniquely. Therefore we will have N segments, each with the same duration. To determine this duration, we notice that each pair will be dancing together in $\frac{N-1}{2}$ segments, and therefore the duration of each segment must be $\frac{2}{N-1}$.

Both of those approaches are optimal, because the maximum number of pairs of mice are dancing at every moment, and each pair is dancing together for exactly one second. Therefore a shorter dance can not exist, as there would simply be not enough time to accumulate the total amount of dancing-together over all pairs (which is N in this subtask). Here is the code:

```
vector<pair<string, double>> solve(int n, const vector<int>& t) {
  vector<pair<string, double>> res;
  string pattern;
  for (int i = 0; i < n / 2; ++i) {
    pattern += "<>";
  }
  if (n % 2 == 0) {
    res.emplace_back(pattern, 1);
    res.emplace_back(pattern.substr(1) + pattern[0], 1);
  } else {
    pattern += ".";
    for (int i = 0; i < n; ++i) {
      res.emplace_back(pattern.substr(i) + pattern.substr(0, i), 2.0 / (n - 1));
    }
  }
  return res;
}
```

Subtask 3: With a Zero (12 Points)

In this subtask, N and t_i can be arbitrary, with one exception: $t_{N-1} = 0$.

In order to solve such problems that ask one to come up with a strategy that minimizes a certain value, it often helps to think: what lower boundaries do we have on the answer? Since one mouse can participate in at most one pair at the same time, the total duration will definitely be at least $t_i + t_{i+1}$ for any i (here and below we treat t_N as an alias for t_0, and t_{-1} as an alias for t_{N-1}). In other words the total duration will be at least $\max_i(t_i + t_{i+1})$ (note that this idea is shared with the 2019 Storytelling problem). Let us denote $s_i = t_i + t_{i+1}$ for the rest of the solution.

Having come up with such boundary, the next natural question to ask is: is the boundary tight? Does there exist an example where we need more time than that? We have seen such examples in subtasks 1 and 2, but here we have the additional constraint that $t_{N-1} = 0$.

When after some time we do not see such counter-example, we start to think that the boundary might be tight for this subtask. And even though we do not have a proof for that yet, we can actually use the hypothesis to drive our problem solving: let us try to come up with a way to find a dance segment such that its length is some value x, and such that subtracting x from the t_i values participating in that dance segment reduces the number $\max_i s_i$ by x.

If we are able to find such dance segment for every configuration with $t_{N-1} = 0$, then this will prove our hypothesis and give us a solution for this subtask! The reason for this is that we can repeat this process, and on every step the value of $\max_i s_i$ will decrease by the duration of the dance segment we added. Therefore when this value reaches 0, which means that all t_i will be equal to 0, we will have constructed a dance with the total duration of $\max_i s_i$. And since we know this is a lower boundary for the total duration, we know that such dance is optimal.

Now, how can we find a dance segment of some length x that reduces $\max_i s_i$ by x? For each pair i that is dancing in this segment, the values of $s_i = t_i + t_{i+1}$ and $s_{i-1} = t_i + t_{i-1}$ will reduce by x, assuming $x \le t_i$ (since t_i can not go negative). So we need to choose the set of dancing pairs i satisfying the following two constraints:

- for all chosen i, all $t_i > 0$, so we can choose a non-zero value of x.

- the set of s_{i-1} and s_i for all chosen i must cover all sums which are equal to $\max_i s_i$. This ensures that the maximum will decrease if we decrease all those sums.

Let us consider an example: suppose the array t is

1	2	3	2	0

and the two cells highlighted in gray are the pairs that will be dancing in our segment. Then the array s looks like

3	5	5	2	1

and the four cells highlighted in gray are the values that will be decreasing if the two pairs highlighted above are dancing.

This dance segment satisfies the two requirements mentioned above. The only s_i that is not decreasing is the value $s_3 = 2 + 0$, but it is not one of the maximum values.

This idea can be generalized to an arbitrary array t_i which has a 0 inside it. If we take any segment of consecutive non-0 values that has 0s on the both ends of it, and choose the first, third, fifth, ..., pairs in this segment, then we will cover almost all s_i values in this segment, except maybe the last one. But that last s_i has the form $a + 0$ ($2 + 0$ in the above example), while the previous one has the form $b + a$ ($3 + 2$ in the above example), therefore $a + 0$ will not be the maximum while $b > 0$ since $b + a > a + 0$. So even though this one s_i will not decrease during this dance segment, $\max_i s_i$ will still decrease.

We can do this for all segments of consecutive non-0 values to obtain an entire dance segment. It turns out that the implementation of this is really simple: we can just go from left to right starting from an element that comes after a 0 (since we know that $t_{N-1} = 0$, it means we can just start from the first element), and whenever we encounter a non-0 t_i, we take it to our dance segment unless we have already taken the previous one. The code follows:

```
vector<pair<string, double>> solve(int n, vector<int> t) {
  vector<pair<string, double>> res;
  while (true) {
    string pattern;
    bool any = false;
    for (int i = 0; i < n; ) {
      if (t[i] > 0) {
        pattern += "<>";
        any = true;
        i += 2;
      } else {
        pattern += ".";
        ++i;
      }
    }
    if (!any) break;
    int duration = (int) 1e9;
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      duration = min(duration, t[i]);
    }
    assert(duration > 0);
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      t[i] -= duration;
    }
    res.emplace_back(pattern, duration);
  }
  return res;
}
```

To recap, we started by finding a lower boundary for the overall dance duration. Then we hypothesized that this lower boundary is in fact the answer, and to prove that we found a way to always add a new dance segment in such a way that this lower boundary will decrease by the length of this segment. This has given us both the proof that our duration is optimal, and a way to actually construct the dance sequence.

One other consideration is the number of segments in our dance: the problem statement requires this amount to not exceed $3 \cdot N$. However, we can notice that every iteration of our loop makes one more value of t_i equal to 0, therefore we have at most $N - 1$ segments in this solution.

Subtask 4: Even Mice (16 Points)

In this subtask, the number N of mice is even.

In case at least one of the durations t_i is equal to 0, we can just use a slightly adapted solution from subtask 3 (instead of starting from the beginning, we need to start each iteration from the element after the 0). So we only need to consider the case where all $t_i > 0$.

It turns out that the idea from subtask 3 works just as well for this case: the value of $\max_i s_i$ is a lower boundary for the answer, and in order to decrease that value by x using a dance segment of length x we simply need to make a dance segment out of all odd-numbered (or all even-numbered) pairs. Then all s_i will decrease, and $\max_i s_i$ together with them.

After making such dance segment with maximum possible duration, we will have at least one $t_i = 0$, after which we can switch to the solution from subtask 3. This proves that the duration of the optimal dance in this subtask is also $\max_i s_i$. Here is the code:

```
vector<pair<string, double>> solve_zero(int n, vector<int> t) {
  vector<pair<string, double>> res;
  int start = 0;
  while (t[start] != 0) ++start;
  while (true) {
    string pattern(n, '.');
    bool any = false;
    for (int i = 0; i < n; ) {
      int pos = (start + i) % n;
      if (t[pos] > 0) {
        pattern[pos] = '<';
        pattern[(pos + 1) % n] = '>';
        any = true;
        i += 2;
      } else {
        ++i;
      }
    }
    if (!any) break;
    int duration = (int) 1e9;
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      duration = min(duration, t[i]);
    }
    assert(duration > 0);
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      t[i] -= duration;
    }
    res.emplace_back(pattern, duration);
  }
  return res;
}

vector<pair<string, double>> solve(int n, vector<int> t) {
  vector<pair<string, double>> res;
  string pattern;
  int duration = (int) 1e9;
  for (int i = 0; i < n; i += 2) {
    pattern += "<>";
    duration = min(duration, t[i]);
  }
  for (int i = 0; i < n; i += 2) {
    t[i] -= duration;
  }
  res.emplace_back(pattern, duration);
  auto zero_res = solve_zero(n, t);
  res.insert(res.end(), zero_res.begin(), zero_res.end());
  return res;
}
```

The function `solve_zero` solves subtask 3, but where the 0 can be in any position, not necessarily as t_{N-1}.

Subtask 5: Odd Mice (57 Points)

This subtask completes our solution of the problem: now the number N of mice is odd. This subtask was worth so many points because here we need to combine all ideas from the previous subtasks and then add some more.

First of all, while $\max_i s_i$ is still a lower boundary on the total dance duration, it is no longer equal to it. For example, as we learned in subtask 2, for the case $N = 5$, $t_i = 1$ for all i the optimal duration is equal to 2.5, while $\max_i s_i = 2$.

This raises a natural question: is there a better lower boundary that we can use? Again, from subtask 2 we learned that since at most $\frac{N-1}{2}$ pairs of mice can dance at the same time, this gives us a boundary using the sum of all t_i: the total dance duration must be at least $\frac{2 \cdot \sum_i t_i}{N-1}$.

Now we have two lower boundaries on the dance duration: $\max_i s_i$ and $\frac{2 \cdot \sum_i t_i}{N-1}$. In subtasks 3 and 4 the first one was equal to the answer, while in subtasks 1 and 2 the second one was equal to the answer. A logical hypothesis follows: maybe the answer is always equal to the bigger of those two lower boundaries, in other words to $\max(\max_i s_i, \frac{2 \cdot \sum_i t_i}{N-1})$?

Just as we did in subtask 3, let us try to build a dance segment of length x that would reduce this value by x, and the fact that we can always do it will be our proof that this value is in fact always equal to the answer.

The easiest way to make sure that this value decreases by x is to make sure that both $\max_i s_i$ and $\frac{2 \cdot \sum_i t_i}{N-1}$ decrease by x. $\frac{2 \cdot \sum_i t_i}{N-1}$ will decrease by x after a dance segment of length x if and only if this dance segment has exactly $\frac{N-1}{2}$ dancing pairs, in other words all mice except one are dancing. If j is the number of the only mouse that is not dancing, then such a segment will decrease all values of s_i by x, except the value s_{j-1}. So we just need to choose such j that s_{j-1} is less than $\max_i s_i$, and choose x in such a way that it does not exceed $(\max_i s_i) - s_{j-1}$, and then such segment will decrease both $\max_i s_i$ and $\frac{2 \cdot \sum_i t_i}{N-1}$ by x as we wanted!

But what if there is no such j, in other words, what if all s_i are equal? It turns out that when N is odd, this means that all t_i are also equal: $s_i = s_{i+1}$ means $t_i + t_{i+1} = t_{i+1} + t_{i+2}$, which leads to $t_i = t_{i+2}$, which leads to $t_i = t_{i+2k}$ for all k, which leads to $t_i = t_{i+N+1}$ (since $N + 1$ is an even number), which means $t_i = t_{i+1}$, which means that all t_i are equal (in this paragraph we liberally assume that all indices are taken modulo N).

Therefore repeating the step we described above will eventually lead us to one of the two states:

- All t_i are equal.
- At least one $t_i = 0$.

And now we can use the solution from either subtask 2 or subtask 3 to complete our solution and our proof! The code follows:

```
vector<pair<string, double>> solve_all_equal(int n, const vector<int>& t) {
  vector<pair<string, double>> res;
  string pattern;
  for (int i = 0; i < n / 2; ++i) {
    pattern += "<>";
  }
  pattern += ".";
  for (int i = 0; i < n; ++i) {
    res.emplace_back(pattern.substr(i) + pattern.substr(0, i), t[0] * 2.0 / (n - 1));
  }
  return res;
}

vector<pair<string, double>> solve(int n, vector<int> t) {
  vector<pair<string, double>> res;
  bool all_equal = false;
  while (true) {
    int max_s = 0;
    int min_s = (int) 1e9;
    int min_s_at = -1;
    for (int i = 0; i < n; ++i) {
```

```
      int s = t[i] + t[(i + 1) % n];
      if (s > max_s) {
        max_s = s;
      }
      if (s < min_s) {
        min_s = s;
        min_s_at = i;
      }
    }
    if (min_s == max_s) {
      all_equal = true;
      break;
    }
    string pattern(n, '.');
    for (int i = 0; i < n - 1; i += 2) {
      int pos = (min_s_at + 2 + i) % n;
      pattern[pos] = '<';
      pattern[(pos + 1) % n] = '>';
    }
    int duration = max_s - min_s;
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      duration = min(duration, t[i]);
    }
    if (duration == 0) {
      break;
    }
    for (int i = 0; i < n; ++i) if (pattern[i] == '<') {
      t[i] -= duration;
    }
    res.emplace_back(pattern, duration);
  }

  vector<pair<string, double>> extra_res;
  if (all_equal) {
    extra_res = solve_all_equal(n, t);
  } else {
    extra_res = solve_zero(n, t);
  }
  res.insert(res.end(), extra_res.begin(), extra_res.end());
  return res;
}
```

At each iteration, the number of s_i which are equal to $\max_i s_i$ increases by 1. Therefore we will reach either subtask 2 or subtask 3 after at most $N - 1$ dance segments, and then we can solve each of those subtasks using at most N dance segments, so the total number of segments in our solution is at most $2 \cdot N - 1$.

Note that we learned the solution presented here from two of the contestants, Vivienne Burckhardt and Ursus Wigger. The solution we used to prepare this problem was slightly more complicated, so thanks a lot to Vivienne and Ursus for coming up with such a logical and clean approach.

The basic concept of replacing the search for the smallest value of a function subject to some constraints with the search for the maximum lower boundary on that value that we can deduce from those constraints is called *duality*, and it appears in many areas of algorithms and mathematics, for example some of you might have seen it in previous SOI lectures where we have demonstrated that the size of the maximum matching is equal to the size of the minimum cover in a bipartite graph. Duality has many forms, and you can't know them all, so just keep this general idea in mind for the future problems!

Ferry Rerouting

Task Idea	Johannes Kapfhammer
Task Preparation	Joël Huber
Description English	Joël Huber
Description German	Joël Huber
Description French	Mathieu Zufferey
Solution	Joël Huber
Correction	Joël Huber

Summary

In the problem "Ferry Rerouting", you were given two trees in the input, let's call them A and B. The task was to find a sequence of moves of minimal length which transforms A to B. Each move consisted of removing an edge currently in the tree and replacing it by any other edge. However, after every move, the graph has to stay a tree, meaning it has to stay connected.

It turned out that there were various solutions to this problem. In this booklet, we describe the master solution, but only a minority coded this solution during the first round. The other participants came up with other creative solutions to this problem.

General Observations

There are two very important observations which one needs to even get started. One can figure them out by just experimenting with various trees manually. The first one might seem a bit trivial, but it is what allows us to get going. Note that since this is the official solution, we will be proving these observations, but you don't need to do that when you're solving the problem. In fact, if you make the second observation, you can get a confirmation from the grader by submitting it to subtask 1 (this is the so-called "proof by AC").

Observation 1: Look at any edge e in tree A, which is not in tree B. Then there is an edge e' in tree B which can replace the edge e without violating the condition.

Proof: After removing e from the tree, we get two components. Let's call them c_0 and c_1. There is no edge that is in both A and B which connects c_0 and c_1, as otherwise there would be a cycle consisting of that edge and e in A, contradicting that A is a tree. Now suppose there is no edge e' satisfying the condition above. Then every edge in B stays in the same component, thus B consists of two components, contradicting that B is a tree. So this edge e' will always exist.

Observation 2: The number of moves we need to do is exactly the number of edges in A, which are not in B (and vice versa, the number of edges in B, which are not in A).

Proof: Let k be the number of edges in A but not in B. This observation can be proven by induction on k. The base case with $k = 0$ is trivial. For the induction step, we can just choose any edge e in A, but not in B. By observation 1, there is an edge e' to replace it with. So we can replace our edge by e'. What we get is the same problem once again, but now with $k - 1$, thus we can finish by the induction hypothesis.

Subtask 1: Number of Days (10 Points)

This subtask can be solved by applying observation 2. We need to find the number of edges which are in A, but not in B.

This can be done in $O(n \log n)$ by first putting all edges from both trees as pairs into a list, such that the first vertex is the one with the smaller index. Then one can just sort the edges and remove the duplicates.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

void solve_testcase()
{
    int n;
    cin >> n;

    vector<pair<int,int>> p1(n - 1);
    vector<pair<int,int>> p2(n - 1);

    for (int i = 0; i < n - 1; i++) cin >> p1[i].first;
    for (int i = 0; i < n - 1; i++) cin >> p1[i].second;
    for (int i = 0; i < n - 1; i++) cin >> p2[i].first;
    for (int i = 0; i < n - 1; i++) cin >> p2[i].second;

    for (int i = 0; i < n - 1; i++)
        if (p1[i].first > p1[i].second) swap(p1[i].first, p1[i].second);
    for (int i = 0; i < n - 1; i++)
        if (p2[i].first > p2[i].second) swap(p2[i].first, p2[i].second);

    sort(p1.begin(), p1.end());
    sort(p2.begin(), p2.end());

    vector<pair<int,int>> res;

    // This merges two sorted lists p1 and p2, removing duplicate elements while
    // inserting them into the list res.
    set_union(p1.begin(), p1.end(), p2.begin(), p2.end(), back_inserter(res));

    // res will now have size 2(n - 1) - [no. duplicates]
    // Thus it has size (n - 1) + number of edges appearing in A, but not in B
    cout << res.size() - (n - 1) << "\n";
}

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int t;
    cin >> t;

    for (int i = 0; i < t; i++)
    {
        cerr << i << "\n";
        cout << "Case #" << i << ": ";
        solve_testcase();
    }
}
```

However, since our graph is a tree, there is actually a way to do it more efficiently. Root the first tree and for each vertex v, find its parent p_v. Now look at the edges in the second tree. An edge a, b from the second tree appears in the first tree if and only if either $p_a = b$ or $p_b = a$. Thus we can check each edge of the second tree in $O(1)$, giving a total running time of $O(n)$. (Note that expected $O(n)$ can also be achieved by using unordered data structures, but that would be boring.)

Subtask 2: Regional Transports (20 Points)

This is the first time we try to solve the real problem. By observations 1 and 2, we realize that the edges are more or less independent, meaning that we can first find a replacement edge e' for the

first edge e appearing in A but not in B, and just repeat the procedure for every edge appearing in A but not in B.

Thus let's try to find a procedure which finds a replacement edge e' given some edge e. In this subtask, we can try each edge appearing in B but not in A as possible replacement edge e'. Then to check whether e' really replaces e, we just construct the new tree and use dfs to check whether the new graph is connected.

In total, this takes $O(n)$ for processing each edge e time $O(n)$ for processing each replacement edge times $O(n)$ for building the new tree and running the dfs, resulting in a running time of $O(n^3)$.

```
struct graph
{
    int n;
    vector<vector<int>> adj;
    vector<bool> vis;

    graph(int _n) : n(_n), adj(n), vis(n, false) {}

    void dfs(int curr)
    {
        if (vis[curr]) return;
        vis[curr] = true;

        for (auto nx : adj[curr]) dfs(nx);
    }

    bool check_connected()
    {
        dfs(0);
        bool ok = find(vis.begin(), vis.end(), false) == vis.end();
        return ok;
    }

    void add_edge(pair<int,int> p)
    {
        adj[p.first].push_back(p.second);
        adj[p.second].push_back(p.first);
    }
};

void solve_testcase()
{
    // I/O and printing the number of moves has been left out in order
    // to keep the solution shorter. See sub1 for this

    // Stores whether an edge is currently active (still in the graph)
    // or inactive (removed / not yet inserted)
    vector<bool> active_a(n - 1, true);
    vector<bool> active_b(n - 1, false);

    for (int i = 0; i < n - 1; i++)
    {
        if (binary_search(edges_b.begin(), edges_b.end(), edges_a[i])) active_a[i] = false;
        if (binary_search(edges_a.begin(), edges_a.end(), edges_b[i])) active_b[i] = true;
    }

    // Checks whether edge j is a possible replacement for the currently inactive edge
    auto check = [&](int j)
    {
        if (active_b[j]) return false;
        assert(j < n - 1);

        // Construct the new graph
        graph g(n);
        for (int i = 0; i < n - 1; i++) if (active_a[i]) g.add_edge(edges_a[i]);
```

```
56          for (int i = 0; i < n - 1; i++) if (active_b[i]) g.add_edge(edges_b[i]);
57          g.add_edge(edges_b[j]);
58
59          // Check the new graph
60          return g.check_connected();
61      };
62
63      for (int i = 0; i < n - 1; i++)
64      {
65          if (!active_a[i]) continue;
66          active_a[i] = false;
67
68          // Check each replacement edge
69          int tk = 0;
70          for (; !check(tk); tk++);
71
72          active_b[tk] = true;
73          cout << edges_a[i].first << " " << edges_a[i].second << " ";
74          cout << edges_b[tk].first << " " << edges_b[tk].second << "\n";
75      }
76  }
```

Subtask 3: Inhabited Islands of Indonesia (30 Points)

This subtask can be solved by improving the running time of the previous subtask. Note that in the previous subtask, we repeated the same thing a lot of times. Namely, we constructed the graph again and again even though not much changed. Additionally, running dfs every time is also doing a lot of redundant work.

However, the previous algorithm can be optimized. Let's again look at what we can do to find a replacement edge for an edge e.

Remove e from the tree. Now we get two components c_0 and c_1. Use dfs to find for every vertex the component it belongs to. Now we can just check for each possible replacement edge e', whether its two endpoints lie in different components. If yes, then we can take it.

Thus for each edge e of the $O(n)$ edges, we build a graph in $O(n)$. Then we check every of the $O(n)$ possible replacement edges in $O(1)$. So in total, this takes $O(n^2)$ time.

```
1   struct graph
2   {
3       vector<vector<int>> adj;
4       vector<int> comp;
5
6       void dfs(const int& curr, const int& par, const int& cmp)
7       {
8           comp[curr] = cmp;
9           for (auto nx : adj[curr]) if (nx != par) dfs(nx, curr, cmp);
10      }
11
12      void add_edge(const int& a, const int& b)
13      {
14          adj[a].push_back(b);
15          adj[b].push_back(a);
16      }
17
18      graph(const int& _n) : adj(_n), comp(_n) {}
19  };
20
21  void solve_testcase()
22  {
23      // I/O and printing the number of moves has been left out in order
24      // to keep the solution shorter. See sub1 for this
25
26      for (int i = 0; i < n - 1; i++)
27      {
```

```
28          if (!active_a[i]) continue;
29          active_a[i] = false;
30
31          graph g(n);
32          for (int j = 0; j < n - 1; j++)
33          {
34              if (active_a[j]) g.add_edge(edges_a[j].first, edges_a[j].second);
35              if (active_b[j]) g.add_edge(edges_b[j].first, edges_b[j].second);
36          }
37
38          g.dfs(edges_a[i].first, edges_a[i].second, 0);
39          g.dfs(edges_a[i].second, edges_a[i].first, 1);
40
41          for (int j = 0; j < n - 1; j++)
42          {
43              if (active_b[j]) continue;
44              if (g.comp[edges_b[j].first] == g.comp[edges_b[j].second]) continue;
45              cout << edges_a[i].first << " " << edges_a[i].second << " ";
46              cout << edges_b[j].first << " " << edges_b[j].second << "\n";
47              active_b[j] = true;
48              break;
49          }
50      }
51  }
```

Subtask 4: The World (40 Points)

In order to solve this subtask, we need to change the approach. We still have not yet used that we could look at the edges in a different order to speed some things up. Additionally, to get below $O(n^2)$, we need to process the edges "simultaneously".

The general Idea

Let's first compress all the edges that are already correct in the first tree (meaning the edges we don't need to replace). Now we get a new tree where each edge needs to be replaced. Consider a leaf in this tree. Note that we can cut it off the tree. By observation 1, there is an edge from this leaf to a different vertex in the tree. We can now connect the leaf with this vertex and then compress the leaf and the node it's connected to, as the edge connecting them is now correct as well.

Doing it in $O(n \log n)$

For the compression part, the union find structure comes in handy. With it, we can keep track on which vertices have been compressed to which other vertices. Thus the procedure is somewhat clear now: Merge connected nodes using the unionfind, then run a dfs. As soon as we return from the dfs, the vertex we're at is a leaf (as its children have been merged to other vertices). Thus find an edge not yet in our tree which leaves our current compressed vertex to some other vertex. Thus the only thing left is how to keep track of the eadges leaving a vertex.

Keep a vector of edges leaving each compressed vertex. When we now merge two vertices, we need to merge these two lists. This can be easily done by copying the contents of one list to the other list. However, if we just do this naively, we end up with a running time of $O(n^2)$ again. In order to speed this up, we need to copy the contents of the smaller list to the larger one. This is a well-known trick that on the first glance only seems to speed up the constant. However, applying this will reduce the running time of our algorithm to $O(n \log n)$.

Optimizing to $O(n\ \alpha(n))$

However, we can optimize this even further. Let's look again at what we really need from the data structure containing the edges. We have several operations:

- We want to look at the first element (to get the replacement edge)

- We want to remove the first element (to remove an edge that we used)
- We want to merge two lists efficiently (when we merge two vertices)

Note that these requirements can be satisfied by using a linked list (which is implemented as std::list in the standard library), which can do all three operations in $O(1)$. Using a linked list instead of using vectors and copying smaller to larger will give us an even better running time of $O(n\ \alpha(n))$.

```cpp
// The implementation of Union Find has been left out to make the solution shorter
// Please see https://soi.ch/wiki/union-find/#implementierung for an implementation

struct cmptree
{
    vector<vector<int>> adj;
    vector<int> par;

    void dfs(int curr, int pr)
    {
        par[curr] = pr;
        for (auto nx : adj[curr])
        {
            if (nx == pr) continue;
            dfs(nx, curr);
        }
    }

    cmptree(int n) : adj(n), par(n) {}
};

struct tree
{
    vector<vector<pair<int,int>>> adj;
    vector<list<int>> lists;
    vector<pair<int,int>> eb;

    ufind uf;

    vector<pair<int,int>> sol;

    tree(int n) : adj(n), lists(n), uf(n) {}

    void dfs(int curr, int par, int pnum)
    {
        for (auto np : adj[curr])
        {
            int nx = np.first;
            if (par != -1 && nx == par) continue;
            dfs(nx, curr, np.second);
        }
        if (pnum == -1) return;

        auto check = [&](const int& ed)
        {
            return uf.find(eb[ed].first) == uf.find(eb[ed].second);
        };

        int repr = uf.find(curr);

        // Popping the already used edges
        while (assert(lists[repr].size() > 0), check(lists[repr].front())) lists[repr].pop_front();
        int ed = lists[repr].front();

        sol.emplace_back(pnum, ed);

        // Merging the vertices
        pair<int,int> ep = {uf.find(eb[ed].first), uf.find(eb[ed].second)};
        if (ep.first != uf.find(repr)) swap(ep.first, ep.second);
```

```
60          lists[ep.first].splice(lists[ep.first].end(), lists[ep.second]);
61
62          uf.unite(ep.first, ep.second);
63          if (uf.find(ep.first) != ep.first) swap(lists[ep.first], lists[ep.second]);
64      }
65  };
66
67  void solve_case()
68  {
69      // Reading n and the edges has been left out to make the solution
70      // shorter.
71      int n;
72      vector<pair<int,int>> ea(n - 1);
73      vector<pair<int,int>> eb(n - 1);
74
75
76      cmptree ct(n);
77
78      for (int i = 0; i < n - 1; i++)
79      {
80          ct.adj[ea[i].first].push_back(ea[i].second);
81          ct.adj[ea[i].second].push_back(ea[i].first);
82      }
83
84      ct.dfs(0, -1);
85
86      tree t(n);
87      for (int i = 0; i < n - 1; i++)
88      {
89          if (ct.par[eb[i].first] == eb[i].second || ct.par[eb[i].second] == eb[i].first)
90          {
91              t.uf.unite(eb[i].first, eb[i].second);
92          }
93      }
94
95      t.eb = eb;
96      for (int i = 0; i < n - 1; i++)
97      {
98          int a = t.uf.find(ea[i].first);
99          int b = t.uf.find(ea[i].second);
100         if (a == b) continue;
101         t.adj[a].emplace_back(b, i);
102         t.adj[b].emplace_back(a, i);
103     }
104
105     for (int i = 0; i < n - 1; i++)
106     {
107         int a = t.uf.find(eb[i].first);
108         int b = t.uf.find(eb[i].second);
109         t.lists[a].push_back(i);
110         t.lists[b].push_back(i);
111     }
112
113     t.dfs(t.uf.find(0), -1, -1);
114
115     cout << t.sol.size() << "\n";
116     for (auto p : t.sol)
117     {
118         cout << ea[p.first].first << " " << ea[p.first].second << " ";
119         cout << eb[p.second].first << " " << eb[p.second].second << "\n";
120     }
121 }
```

Tinmining

Task Idea	Jonas Meier, Johannes Kapfhammer, Daniel Rutschmann
Task Preparation	Jonas Meier
Description English	Jonas Meier
Description German	Jan Schär
Description French	Mathieu Zufferey
Solution	Jonas Meier, Johannes Kapfhammer, Daniel Rutschmann, Timon Gehr
Correction	Jonas Meier

The problem can be abstracted in the following way.

Given a circular graph as an N-gon, on which there is a permutation of the first N numbers on the nodes, find a path that is increasing in the labels and not self-intersecting. (Subtasks 1 and 3 just impose further conditions on this path).

Subtask 1: First Mine (15 Points)

In this subtask, we have the restriction on the path that it has to end in the node with highest stability (node N, if we think about permutations).
We can find a solution that runs in $O(N^3)$ by using DP.
Our DP table consists of three dimensions: DP$[N][N][b]$.
The entry DP$[i][j][b]$ has the following meaning: Let $N_{i,j}$ be the set of nodes in the range "clockwise from i to j" defined as follows:

$$N_{i,j} = \begin{cases} \text{if } (i < j): & \{v \mid i \leq v \leq j\} \\ \text{if } (i \geq j): & \{v \mid v \leq j \text{ or } v \geq i\} \end{cases}$$

The meaning of an entry of the DP table is defined as follows:

DP$[i][j][0]$ = "Length of maximal path ending at i, while only using nodes in $N_{i,j}$"
DP$[i][j][1]$ = "Length of maximal path ending at j, while only using nodes in $N_{i,j}$"

An entry of the DP table can be computed as follows: The base cases are DP$[i][i] = 1$ for all i.

$$DP[i][j][0] = \max\{ \max_{\substack{k \in (N_{i,j} - \{j\}) \\ \text{stability}[k] < \text{stability}[i]}} \{DP[i][k][0] + 1\}, DP[i][j+1][0]\}$$

$$DP[i][j][1] = \max\{ \max_{\substack{k \in (N_{i,j} - \{i\}) \\ \text{stability}[k] < \text{stability}[i]}} \{DP[k][j][1] + 1\}, DP[i+1][j][1]\}$$

The order of computation has to be so that the lengths of the intervals are increasing. The length of the interval (i, j) is in this case defined as $|N_{i,j}|$ (so the number of elements in the clockwise sequence from i to j).

It takes $O(N)$ to compute one entry and in total, we have N^2 entries. So the running time is $O(N^3)$.
The solution can be found in the entry $\max\{$DP$[0][N][0]$, DP$[0][N][1]\}$.

This can be implemented in C++ in the following way:

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int T;
  cin >> T;

  for (int t = 0; t < T; t++) {

    int n;
    cin >> n;

    vector<int> stability(n);

    int max_element = 0;
    for (int i = 0; i < n; i++) {
      cin >> stability[i];
      if (stability[i] > stability[max_element])
        max_element = i;
    }

    //base cases are implicitly given if array is set to 1
    vector<vector<vector<int> > > dp(n, vector<vector<int> >(n, vector<int>(2, 1)));

    for (int size = 2; size <= n; size++) {
      for (int pos = 0; pos < n; pos++) {
        for (int k = pos; k < pos + size; k++) {

          int left = pos;
          int right = (left + size - 1) % n;
          int middle = k % n;

          dp[left][right][0] = max(dp[left][(right - 1 + n) % n][0], dp[left][right][0]);
          dp[left][right][1] = max(dp[(left + 1) % n][right][1], dp[left][right][1]);

          if (k > pos && stability[left] > stability[middle]) {
            dp[left][right][0] = max(dp[middle][right][0] + 1, dp[left][right][0]);
          }

          if (k > pos && stability[left] > stability[right]) {
            dp[left][right][0] = max(dp[middle][right][1] + 1, dp[left][right][0]);
          }

          if (k < pos + size - 1 && stability[right] > stability[left]) {
            dp[left][right][1] = max(dp[left][middle][0] + 1, dp[left][right][1]);
          }

          if (k < pos + size - 1 && stability[right] > stability[middle]) {
            dp[left][right][1] = max(dp[left][middle][1] + 1, dp[left][right][1]);
          }
        }
      }
    }

    cout << "Case #" << t << ": "
         << max(dp[max_element][(max_element - 1 + n) % n][0],
                dp[(max_element + 1) % n][max_element][1])
         << "\n";
  }
}
```

Subtask 2: Scaling up (15 Points)

Let $a_1, a_2, \ldots, a_k$ be an allowed path. Then there are two paths going from a_1 to a_k (one clockwise and one counterclockwise).
Let P_{ccw} be all element that belong to the original (allowed) path and also to the clockwise path

from a_1 to a_k in the order they appear in the original path.
In a similar way, let P_{cw} be all the elements belonging to the original allowed path and also to the counterclockwise path from a_1 to a_k in the order they appear in the original path.
We can see that the elements in P_{ccw} do not only appear in the order they appeared originally in the original allowed path but also in the order of the clockwise path from a_1 to a_2.
If we rotate the array, such that a_1 is at the first position, then the original allowed path corresponds to a sequence that is first increasing (the clockwise segment path to a_k) and then decreasing (the reverse of the counterclockwise path from a_1 to a_k).
If a sequence has this property (that is, it is first increasing and then decreasing (or vice versa), then we call it "unimodal").
We see that every allowed path corresponds to a unimodal sequence in one of the possible rotations of the original array.

In fact, we can show that every unimodal sequence also corresponds to a path. We can show this by construction.
We find the minimal and the maximal element in the unimodal sequence. Our path definitely has to start at the minimal node and end at the maximal one.
Now, we can simply grow our path from the minimal element by greedily choosing the smaller element on each side of it (from the lists that we called P_{ccw} and P_{cw}).

Therefore, our task is now reduced to the problem of finding the longest unimodal sequence that ends at the biggest node.
This can be achieved by rotating the array in such a way that the biggest node is the last element.
We can use the Longest Increasing Subsequence DP in order to find the lengths of the longest increasing and also longest decreasing subsequences ending at any element in this array.
This allowes us to quickly compute the length of the longest unimodal sequence in which the maximal element is at any given node (by adding the values and subtracting 1 as we count the highest element in both sequences).

Rotating the array takes $O(N)$ time, figuring out the lis and the lds takes $O(N \log(N))$. Then finding the maximum takes $O(N)$ time.
The total running time is therefore $O(N \log(N))$.
This can be implemented in the following way:

```
#include <bits/stdc++.h>

using namespace std;

struct lis{
  vector<int> data;
  vector<int> res;

  lis(vector<int> const& dat){
    this-> data = dat;
    computeLIS();
  }

  int smallestIndex (vector<int> const& in, int lower, int upper, int element) {
    while(upper - lower > 1) {
      int mid = (lower + upper) / 2;
      if (in[mid] >= element) {
        upper = mid;
      } else {
        lower = mid;
      }
    }
    return upper;
  }

```

```
  void computeLIS(){
    res = vector<int>(data.size(),1);
    vector<int> endElement(data.size(),0);

    int length = 1;

    endElement[0] = data[0];

    for (int i = 1; i<data.size(); i++) {
      if (data[i] < endElement[0]) {
        endElement[0] = data[i];
      } else if (data[i] > endElement[length - 1]) {
        endElement[length++] = data[i];
        res[i] = length;
      } else {
        int idx = smallestIndex(endElement, 0, length - 1, data[i]);
        endElement[idx] = data[i];
        res[i] = idx+1;
      }
    }
  }
  void printlislist() {
    for (int i = 0; i<res.size(); i++) {
      cout<<res[i]<<" ";
    }
    cout<<endl;
  }
  int get(int idx) {
    return res[idx];
  }

};

int solve(vector<int>& in) {

  int res = 0;
  int shamt = 0;

  for (int i = 0; i<in.size(); i++) {
    if (in[i] > in[shamt]) {
      shamt = i;
    }
  }

  vector<int> shift(in.size());
  for (int j = 0; j<in.size(); j++) {
    shift[j] = -in[(shamt + j) % in.size()];
  }

  vector<int> reversed(shift.rbegin(), shift.rend());
  lis dec(reversed);
  lis inc(shift);

  for (int j = 0; j<in.size(); j++){
    res = max(res, inc.get(j) + dec.get(in.size()-j-1) - 1);
  }

  return res;
}

int main() {
  cin.tie(nullptr);
  ios::sync_with_stdio(false);
  int T;
  cin>>T;
  for (int t = 0; t<T; t++){
    cout << "Case #" << t<<": ";

```

```
94      int N;
95      cin>>N;
96
97      vector<int> in(N);
98
99      for (int i = 0; i<N; i++){
100         cin>>in[i];
101     }
102
103     cout<<solve(in)<<"\n";
104   }
105 }
```

Subtask 3: Efficient exploitation (15 Points)

This subtask allows for the same bounds as subtask 1. The only difference is that the restriction that the path has to end at the maximal node is dropped.
The solution can be found by taking the maximal element in the DP table of the solution for subtask 1.

Subtask 4: Scaling up (15 Points)

This subtask uses the same idea as subtask 2. The only difference is that we try all possible rotations to allow the path to end at another location. This accounts for an additional factor of $O(N)$ in runtime. The total runtime is therefore $O(N^2 \log(N))$.

Subtask 5: Theoretical (10 + 30 Points)

Subtask 5 a

We can solve this problem with $C = 1$. The strategy is to use a LIS or a LDS (longest decreasing subsequence) of length at least $\sqrt{N}$.
With a similar argument as in subtask 2 (as an increasing/decreasing sequence is a unimodal sequence), we can show that such a sequence gives us a valid path.
It remains to show that the length of either the LIS or the LDS is at least $\sqrt{N}$.
Proof (by contradiction):
WLOG, we can assume that the LIS is longer than the LDS (otherwise, we could reverse the sequence).
We assume for the sake of contradiction that the length of the LIS is smaller than $\sqrt{N}$.
Let a_i be the length of the longest increasing subsequence ending at i. We know (by our assumption) that a_i is smaller than $\sqrt{N}$ for all i.
Since a_i has to be at least 1, we know that there are at most $\sqrt{N} - 1$ different numbers of the form a_i. By the pigeon-hole principle (sometimes referred to as Dirichlet's drawers) we know that one element has to appear at least $\sqrt{N}$ times.
Let b_i be a subsequence of a, such that all b_i are the same value. Since they are not increasing (otherwise the LIS would be bigger) and every element in the array is unique (as it is a permutation), we can conclude that the sequence of the b_i are a LDS that is longer than $\sqrt{N}$.
This is a contradiction.

Subtask 5 b

The algorithms described to solve subtasks 3 and 4 have running times $O(N^3)$ and $O(N^2 \log N)$, respectively. We will now present a couple of solutions whose running time is below $O(N^2)$ for small k.

First, we will slightly change the problem so we do not have to deal with circular indices. We will take the original permutation of stabilities and concatenate it twice. We will call the resulting

sequence of stabilities s. Now, a stability-increasing path starting at mine i corresponds to a unimodal sequence starting at s_i and ending at s_{N+i}.

Opimizing an Enumerative Approach There is a simple enumerative approach that generates all such unimodal sequences: We will maintain a set of unimodal sequences, initially it is empty. We then consider the N mines in order of increasing stability. For mine i, we add all unimodal sequences to the set whose largest element is s_i. If s_i is also the smallest element, the unimodal sequence is simply (s_i, s_{i+N}). Otherwise, note that if we remove s_i from such a unimodal sequence, the result is again a unimodal sequence with a smaller maximum element, therefore it is already in our set. To generate all unimodal sequences with maximum element s_i, we therefore just have to iterate over our set and check for each unimodal sequence if it can be extended by element s_i. Each unimodal sequence can grow within two segments. For example, the unimodal sequence $(1, 3, 4, 8, 6, 2)$ (formed as a subsequence of the underlying sequence s) can be extended by adding an element that is larger than 8 either directly to the left or directly to the right of 8. We will call the segments $[4, 8]$ and $[8, 6]$ (in the underlying sequence s) the *grow segments* of the unimodal sequence. Therefore, whenever s_i falls within one of the grow segments of one of our unimodal sequences, we obtain a new unimodal sequence.

This enumerative approach will generate all possible unimodal sequences corresponding to all increasing paths, in particular, it will find the longest one. However, this solution is slow as there can be exponentially many different such unimodal sequences. Note that there is some redundancy: Consider pairs of unimodal sequences and their grow segment (there are two such pairs for each unimodal sequence). Whenever we grow a unimodal sequence within a grow segment, we create two new such pairs, by extending the unimodal sequence with s_i and by splitting the grow segment at s_i.

We say that pair a dominates pair b if the unimodal sequence of a is at least as long as the unimodal sequence of b and the grow segment of a is contained in the grow segment of b. Note that when we grow the unimodal sequence of a using the grow segment of a, the two new pairs of unimodal sequence and grow segment we obtain will be dominated by the two pairs we obtain by extending b in the same fashion. Therefore, if a pair is dominated, we can ignore it, as dominating pairs provide grow sequences that are at least as long.

As we are only interested in the length of the optimal solution, it suffices to store for each length of unimodal sequence l a set of grow segments S_l for a unimodal sequence of that length, where we include at least all grow segments that occur in non-dominated pairs. Let t be the largest length of a unimodal sequence we have found so far. Whenever we consider the next-largest element s_i, we loop over $S_t, \dots S_2$. For S_i, we check if it contains grow segments around s_i. If so, let $[s_a, s_b]$ be the s_i-enclosing grow segment with smallest possible a and let $[s_c, s_d]$ be the s_i-enclosing grow segment with largest possible d. We then add the new grow segments $[s_a, s_i]$ and $[s_i, s_d]$ to S_{i+1}. Any other grow segments that can be obtained by splitting grow segments from S_i at s_i would be contained in one of those two segments, so this is sufficient. Finally, we add grow segment $[s_i, s_{i+N}]$ to S_2.

In the end, let t be the largest possible value such that S_t is not empty. Then, the final result is $k = t - 1$.

Therefore, what we need is a data structure to represent the sets S_i that supports the following queries efficiently:

- add(l, r): add a new segment $[l, r]$
- query_l(x): find minimal l such that there is a segment $[l, r]$ with $l \le x \le r$, or ∞ if there is none.
- query_r(x): find maximal r such that there is a segment $[l, r]$ with $l \le x \le r$, or $-\infty$ if there is none.

We will only consider query_l, as query_r is analogous. Whenever we add a new segment $[l, r]$, we have to update the answer query_l(i) to min(query_l$(i), l)$ for each i where $l \le i \le r$. One way to do this is to have a segment tree with point queries and range-minimum updates, supporting all operations in time $\log N$. Another way would be to have a segment tree with point updates and range-minimum queries, storing at each position r the smallest l such that

there is a segment (l, r) (or ∞ otherwise). To compute the answer to query_l(i), we just query the minimum of the range $[i, \infty)$. If the result is larger than i, there is no segment, otherwise the returned result is our answer.

If we store only non-empty S_i, the total running time of this approach is $O(N \cdot k \cdot \log N)$.

Rolling LIS We will now discuss an alternative approach. First, note that we can reduce the problem of finding a longest unimodal subsequence to a plain LIS instance. Take the original sequence $s_0, s_1, \ldots, s_{N-1}$, and split each element s_i into two pairs $s'_{2 \cdot i}$ and $s'_{2 \cdot i+1}$, where $s'_{2 \cdot i} = (\searrow, s_i)$ and $s'_{2 \cdot i+1} = (\nearrow, s_i)$.

Intuitively, a pair $(\nearrow, x)$ means "element x and still increasing", while a pair $(\searrow, x)$ means "element x and already decreasing". We can define the following total order on pairs:

- $(\nearrow, x) < (\searrow, x')$,
- $(\nearrow, x) < (\nearrow, x)$ iff $x < x'$, and
- $(\searrow, x) < (\searrow, x)$ iff $x' < x$.

Each increasing subsequence in s' corresponds to a unimodal subsequence in s of the same length and vice-versa.

Therefore, we have reduced our problem to the following:

Given a sequence s' of length $4 \cdot n$, compute the length of the longest increasing subsequence for contiguous subsequences $s'_{2 \cdot i}, \ldots, s'_{2 \cdot (i+N)-1}$ for $i = 0, \ldots, N-1$.

To this end, it suffices to implement a queue data structure that supports the following operations:

- push(x): append a new element to the right of the current sequence.
- pop(): remove the leftmost element of the current sequence.
- query(): compute the longest length of an increasing subsequence of the current sequence.

We can fill the queue with the first $2n$ elements of s', then repeatedly feed two new elements, pop two old elements and query, until we have computed all required lengths. Our goal is to implement all operations in amortized time $O(k)$, such that the resulting running time is $O(n \cdot k)$.

Our data structure is a list of buckets, where each bucket is a double-ended queue (e.g., a `vector<deque<pair<int,int>>>`). The i-th bucket contains all elements (both value and index) for which the longest increasing subsequence ending at that element has length i. We maintain all buckets in increasing sorted order by value and in decreasing sorted order by index. (I.e., each bucket is sorted in both ways independently.)

Lemma 1: There can be no elements in bucket $t-1$ that have both a larger index and a larger value than some element in bucket t. *Proof*: This is true, as those elements from bucket $t-1$ would be able to extend a LIS from bucket t, so it is impossible for them to end up in bucket $t-1$.

To push a new element x, we find the first bucket that has only elements that are larger than x (if needed, we allocate a new bucket), and push x to the front of that bucket. This correctly updates our data structure because if we put a new element in bucket t, this means that there was at least one smaller element in bucket $t-1$, whose LIS we can extend by element x, and all previous LIS of length t ended at elements that are larger than x. Furthermore, the new element has the largest index of any element in the data structure, so it is in particular the element with the largest index in the bucket. We can do this update efficently in time $O(k)$, as we only have to compare x with at most the first element of each bucket. (We can even find the bucket in time $O(\log k)$ using binary search, but this is not required here.)

When we pop an element x, note that x is the first element of the sequence, so it is the last element of bucket 1. When we remove x, there may now be some elements in later buckets such that all LIS ending there start with x. We have to move those elements down one bucket.

Claim: The elements that have to be moved form a suffix in each bucket, and they will again be a suffix of the new bucket after they were moved. Namely, in the first bucket, we have to move element x, and in any other bucket t, we have to move all elements whose index is smaller than the smallest index that remains in bucket $t-1$.

Proof: Elements from bucket t with an index larger than one that remains in bucket $t-1$ do not have to be moved, as they can still extend some of the same LIS from bucket $t-1$ that they did before due to Lemma 1. On the other hand, elements from bucket t whose index is smaller than any of the indices from bucket $t-1$ can not extend any LIS of length $t-1$, and therefore, they need to be moved.

We also have to make sure that moving such a suffix maintains our invariant, namely, bucket $t-1$ also has to remain sorted by value. This is the case, however, due to Lemma 1.

Note that each element has to be moved from a bucket t to a bucket $t-1$ at most k times. Therefore, the amortized running time of our $\Theta(N)$ push and pop operations is indeed $O(N \cdot k)$.