First Round SOI 2018/2019

Solution Booklet

Swiss Olympiad in Informatics

October 1 – November 30, 2018

1 Mergeball

In this task we're given the sizes $b_0, \ldots, b_{N-1}$ of N balls. The action we can perform with them is to take 2 adjacent balls with the same size and merge them together, creating a new ball with double the size.

Subtask 1: Final configuration with two balls (20 Points)

In this subtask we're only given pairs of balls which we want to merge.

Because for each test case there are 2 balls and their sizes are the same, it's enough to just print their sum.

An example solution can be found here:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int T; // T is the number of test cases
  cin >> T;
  for (int t = 0; t < T; t++) { // A loop to input each test case
    int N, b0, b1;              // b0 and b1 are the sizes of the 2 balls
    cin >> N >> b0 >> b1;
    // For each test case we output the sum of the sizes
    cout << "Case #" << t << ": " << (b0 + b1) << "\n";
  }
}
```

Subtask 2: All to one (20 Points)

This subtask is building on top of the first one, but this time we have to check if we can merge N balls, all of which have the same size, into one single ball.

In order to solve it, we have to be smart – we can't just take any 2 adjacent balls at each step and merge them together. The reason for that is we might create a ball that divides 2 balls that could have been merged and we end up with multiple of them, instead of a single one, essentially getting stuck.

A way we can make sure this never happens if there is a solution is the following: We take starting from the left the *smallest* pair of balls and we merge them together. We continue doing so, until we get one single ball or we can't merge anymore.

Here's an example:
①①①①①①①① → ②①①①①①① → ②②①①①① → ②②②①① → ②②②② → ④②② → ④④ → ⑧

This is an example of a *bottom-up approach*. The argumentation of why this is indeed correct will be presented in the next subtask.

Another idea that at first sight might seem confusing, but is also correct, is the following. Let's look at the last merge. That merge has to combine 2 balls with the same values. Even more, we know that the size of these 2 balls has to be equal to half of the total sum of all the balls. If we can't split the balls in 2 sides equal to this half-sum, we can't merge everything into one. Otherwise, we can do the same check, for the 2 sides and do so, until we get to the bottom recursively. This is an example of a *top-down approach*.

There is however a very important observation we have to make. Since we start with N balls, which all have the same size, we merge every consecutive pairs of them, essentially dividing the number of balls by half. Since the new balls we get all have the same size as well, we continue this process until we either have 1 ball or we can't divide the total number of balls we currently have by 2. Therefore in order to check if in the end we'll be left with a single ball or multiple, it's enough to check if the number of balls in the beginning N is a power of 2 or not. An example solution using bittricks to check if a number is a power of 2 can be found here:

```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; // Number of balls
    for (int i = 0; i < N; i++) {
      int bi; // The size of the current ball. In this test case, all sizes are
              // the same.
      cin >> bi;
    }
    // (N & (N - 1)) is a bit trick to find out if N is a power of 2
    cout << "Case #" << t << ": " << ((N & (N - 1)) ? "Multiple" : "Single")
         << "\n";
  }
}
```

Subtask 3: All to one – Part 2 (20 Points)

This subtask is the same as the second one, except in this case we're given balls with different sizes.

The two ideas we used to solve the second subtask can be applied here as well. Again at each step we take the smallest 2 balls and we merge them. An example of this solution is:

⑧④②①① → ⑧④②② → ⑧④④ → ⑧⑧ → ⑯

This is correct because of the following argument: Look at the smallest leftmost ball. If there is *any* solution, this ball has to be merged with some other ball. Balls are either merged to the left or to the right. Because the ball to the left is larger or does not exist (by definition of smallest leftmost), it has to be merged with the one to the right. We can repeat this argument until there is only a single ball left.

Because of this, many modifications will still work - it does not need to be the two smallest balls, it can also be the first two adjacent balls with the same value. A fast solution using this idea uses a stack, which pushes the sizes from left to right and merges the two top values if they are equal. Since we access each value only twice - once when we add it to the stack and once when we take it out to merge it, for a total number of $2N$ times, the running time of this solution is $O(N)$. An example of this solution can be found here:

```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;
    stack<int> st; // A stack to store the ball sizes
    for (int i = 0; i < N; i++) {
      int bi;
      cin >> bi;
      // Check if the previous ball had the same size. We continue doing so,
      // until we can't
      while (st.size() >= 1 && st.top() == bi) {
        st.pop(); // If yes, we merge it with the current ball and remove the
                  // previous
        bi *= 2;
      }
      st.push(bi);
    }
    // If there is only 1 ball in the stack, we successfully merged everything
    cout << "Case #" << t << ": " << (st.size() == 1 ? "Single" : "Multiple")
         << "\n";
  }
}
```

Subtask 4: To infinity (20 Points)

In this subtask we have balls of the same size, from which we want to find out what sizes can be reached in the game. The limit here is that we aren't allowed to make balls bigger than size C.

Let's say the size of the balls is b_i. Since the only way to create new balls is by merging existing ones with the same size and Stofl has only balls of one size, in the beginning we can create a ball of size $2 \cdot b_i$. This ball can only be merged with another one of the same type, creating a ball of size $4 \cdot b_i$ etc. . .

So by just doubling the size at each step and stopping when that value gets bigger than C, we can find all possible sizes that can be achieved using that ball. We have to be careful and use long numbers to make sure there won't be an overflow. An example of this solution can be found here:

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
signed main() {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    int N, C, bi;
    cin >> N >> C >> bi;
    vector<int> sizes;     // A vector to store the sizes we compute
    while (bi <= C) {      // If the new value is bigger than C, we stop
      sizes.push_back(bi); // We add the size to the rest
      bi *= 2;             // We double it
    }
    cout << "Case #" << t << ": " << sizes.size();
    for (int s : sizes)
      cout << " " << s;
    cout << "\n";
  }
}
```

Subtask 5: To infinity and beyond! (20 Points)

This subtask is the same as the fourth one, except here we have many balls of different sizes.

The solution is exactly the same as the previous one, except that we have to pay attention to the fact that we might achieve the same size by using 2 different balls and we don't want to count it twice. The easiest way to do so is by using a set, where we store the values we get during the computation, which deals with the problem of having the same value twice. An example of this solution can be found here:

```
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
signed main() {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    int N, C;
    cin >> N >> C;
    set<int> sizes; // A set to store the sizes we compute
    for (int i = 0; i < N; i++) {
      int bi;
      cin >> bi;
      while (bi <= C) {   // If the new value is bigger than C, we stop
        sizes.insert(bi); // We add the size to the rest
        bi *= 2;          // We double it
      }
    }
    cout << "Case #" << t << ": " << sizes.size();
    for (int s : sizes)
      cout << " " << s;
    cout << "\n";
  }
}
```

2 Ceremony

Given are the heights $h_0, \dots, h_{N-1}$ of each building in a row of N skyscrapers. For convenience's sake, let us say that if $x < y$, then the x^{th} building is to the west or on the left of the y^{th} building, and, similarly, that if $x > y$, then the x^{th} building is to the east or on the right of the y^{th} building.

For every $0 \le i, j < N$, fireworks launched from the top of the i^{th} building are visible from the top of the j^{th} building if and only if both $i \ne j$ and $h_k < h_j$ for every k such that $\min(i, j) < k < \max(i, j)$. In that case, we say that the j^{th} building offers an ideal view on the i^{th} building, or that it is ideal for the i^{th} building.

In the three first subtasks, you're given some i and you have to determine how many buildings are ideal for the i^{th} building. In the two last subtasks, your task is to compute the maximal number of ideal buildings if one chooses i optimally.

Subtask 1: Launch the fireworks from the westmost building (10 points)

In this first subtask, the fireworks were always launched from the 0^{th} building and you had to compute from how many buildings they were visible. The limits were still quite low ($1 \le N \le 100$), so it was possible to test the visibility from every other building using the definition of visibility presented above without any further trick. For the j^{th} building, one just needs to check for every k such that $0 < k < j$ whether $h_k < h_j$. If there is some case in which this inequality does not hold, then the j^{th} building is not ideal for the i^{th} building. This approach would result into the following code:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int T; cin >> T;
  for (int t = 0; t < T; t++) {
    int N, i; cin >> N >> i; // note that i = 0
    vector<int> h(N);
    for (int j = 0; j < N; j++)
      cin >> h[j];
    int result = 0;
    for (int j = 1; j < N; j++) {
      bool visible = true;
      for (int k = 1; k < j; k++)
        if (h[k] >= h[j]) // i. e. if it is not the case that h[k] < h[j]
          visible = false;
      if (visible)
        result++;
    }
    cout << "Case #" << t << ": " << result << '\n';
  }
}
```

Overall, we perform $\sum_{k=1}^{N-2} k = \frac{(N-2)^2+N-2}{2} = \frac{N^2-3N+2}{2}$ comparisons. Therefore, the program computes the solution to any testcase in $\mathcal{O}(N^2)$.

Subtask 2: More available buildings (20 points)

This subtask was the same as the first one, except that N could be a lot larger: the limits were $1 \leq N \leq 1'000'000$. Our previous program was therefore too slow to compute the solution whithin the time limit of 5 minutes.
The key observation is the following: we were comparing a lot of values that we did not need to compare. For example, let us consider an input with heights 8, 3, 5, 12, 4, and 13. When we want to check whether the last building is ideal for the 0th building, we test four inequalities: $3 < 13$, $5 < 13$, $12 < 13$ and $4 < 13$. It is evident that if the third of these inequalities holds, all of the others do as well. For example, if $12 < 13$, then clearly $3 < 13$, because $3 < 12$ and $12 < 13$ ("being less than" is a transitive relation). More generally, one only needs to check whether $\max_{0<k<j}(h_k) < h_j$.
Moreover, if one goes through the buildings from left to right, it is easy to keep track of the heighest building met so far on the go, so that one can always compute whether the next building is ideal in constant time. This code would be an implementation of that idea:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int T; cin >> T;
  for (int t = 0; t < T; t++) {
    int N, i; cin >> N >> i; // note that i = 0
    vector<int> h(N);
    for (int j = 0; j < N; j++)
      cin >> h[j];
    int result = 0, maxSoFar = 0;
    for (int j = 1; j < N; j++) {
      if (maxSoFar < h[j])
        result++;
      maxSoFar = max(maxSoFar, h[j]);
    }
    cout << "Case #" << t << ": " << result << '\n';
  }
}
```

With that improvement, we get our runtime down to $O(N)$.

Subtask 3: Launching from another building (20 points)

In this subtask, the limits are the same as in the second one, but i, the index of the building from which the fireworks are launched, can vary. Simply reusing our previous solution but starting with the $(i+1)^{\text{th}}$ building would clearly not be enough, because one would not ever look at buildings on the left of the i^{th} building. However, we can apply the same principles to extend our solution with another loop checking for each building on the left of the i^{th} building whether it is ideal for that building. Here is an example of how one would do that:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int T; cin >> T;
  for (int t = 0; t < T; t++) {
    int N, i; cin >> N >> i; // i is not always 0 anymore
    vector<int> h(N);
    for (int j = 0; j < N; j++)
      cin >> h[j];
    int result = 0, maxSoFar = 0;
    for (int j = i + 1; j < N; j++) { // right side
      if (maxSoFar < h[j])
        result++;
      maxSoFar = max(maxSoFar, h[j]);
    }
    maxSoFar = 0;
    for (int j = i - 1; j >= 0; j--) { // left side
      if (maxSoFar < h[j])
        result++;
      maxSoFar = max(maxSoFar, h[j]);
    }
    cout << "Case #" << t << ": " << result << '\n';
  }
}
```

In total, we look at the visibility of $N - 1$ buildings, and for every one of these, we only need a constant time to compute whether they are visible. Therefore, our runtime is still $O(N)$.

Subtask 4: Find the building with optimal visibility (10 points)

Now the problem is slightly different: you are not given some i as in the previous subtasks, but you are required to find the optimal number of buildings from which the fireworks are visible if i is chosen optimally, that is, you need to find the maximal solution among the solutions for every possible value of i.
In this fourth subtask, the limits are low again: $1 \leq N \leq 100$. That means that we can still use a similar solution to that of the third subtask, but checking the solution for every i such that $0 \leq i < N$ and taking the maximum of these N values. The modification is quite simple and could be implemented in the following manner:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  int T; cin >> T;
  for (int t = 0; t < T; t++) {
    int N; cin >> N; // there is no given i anymore
    vector<int> h(N);
    for (int j = 0; j < N; j++)
      cin >> h[j];
    int bestResult = 0;
    for (int i = 0; i < N; i++) {
      int result = 0, maxSoFar = 0;
      for (int j = i + 1; j < N; j++) { // right side
        if (maxSoFar < h[j])
          result++;
        maxSoFar = max(maxSoFar, h[j]);
      }
      maxSoFar = 0;
      for (int j = i - 1; j >= 0; j--) { // left side
        if (maxSoFar < h[j])
        result++;
      maxSoFar = max(maxSoFar, h[j]);
    }
    bestResult = max(bestResult, result);
  }
  cout << "Case #" << t << ": " << bestResult << '\n';
 }
}
```

Since we need $O(N)$ time to compute each of the N partial solutions, we get a runtime in $O(N^2)$.

Subtask 5: Find the building with optimal visibility with many buildings (40 points)

This final subtask was the trickiest: with the higher limit on N ($1 \leq N \leq 1'000'000$), one needed a good idea to get under the time limit.

The idea was the following: if one knows which buildings on the left of the k^{th} building offer an ideal view of it, then it is easy to compute which buildings on the left of the $(k+1)^{\text{th}}$ building offer an ideal view of it. The heights of the buildings on the left of the k^{th} building that are ideal for it make up a strictly decreasing sequence (for it is a direct consequence of the definition of visibility that if this sequence was not strictly decreasing, then some of the buildings therein would not be ideal). The ideal buildings on the left of the $(k+1)^{\text{th}}$ building are then the same as those on the left of the k^{th} one, without the buildings that were ideal for the k^{th} building but are not for the $(k+1)^{\text{th}}$ building because their height is smaller than h_k, and with the addition of the k^{th} building itself. But the buildings that we have to remove are always a suffix of the sequence of the ideal buildings on the left of the k^{th} building, because of the fact that this sequence is strictly decreasing. In other words, we can remove the last element of that sequence, and then the next one and so on, until we meet a building whose height is greater than h_k or we reach the end of the sequence. At that point, we get a new sequence with all the buildings on the left of the $(k+1)^{\text{th}}$ building offering an ideal view of it, and we can later use that sequence to compute the sequence of all the buildings offering an ideal view from the left of the $(k+2)^{\text{th}}$ building, and so on until we reach the last building. We also know that no building on the left of the first building offers an ideal view of fireworks launched from the first building, because there is no building on the left of the first building.

By maintaining such a sequence (which is more easily represented by a stack) by starting from the leftmost building and iterating through all buildings from left to right, we can find the number of buildings on the left of every building offering an ideal view of fireworks launched from the top of that building. The procedure would go as follows:

1. We create an empty stack S.
2. We start considering the 0th building. S contains all the buildings left of the 0th which offer an ideal view of fireworks launched from the 0th building.
3. S's size is 0, so we memorize that 0 is the number of ideal buildings on the left of the 0th building.
4. The 0th building is taller than no element in S, so we don't remove anything from S, and we add the 0th building to the stack.

$\cdots$ (We continue by considering the next buildings)

5. We're now considering the i^{th} building. S contains all the buildings left of the i^{th} which offer an ideal view of fireworks launched from the i^{th} building.
6. S's size is now X, so we memorize that X is the number of ideal buildings on the left of the i^{th} building.
7. The i^{th} building is of at least the same height as the first Y items on top of S, so we remove these from S, and we add the i^{th} building on top of S.

. . . (We continue by considering the next buildings)

8. We've now found the number of ideal buildings on the left of the $(N-1)^{\text{th}}$ building, so we can stop.

It is possible to implement a similar process to compute the number of buildings offering an ideal view from the right of any building. Then, after we have computed both the number of ideal buildings on the left and on the right of some building, we know that the sum of these two numbers is the number of buildings offering an ideal view on fireworks launched from the top of that building. Hence, to get the overall solution, it suffices to iterate through every i between 0 and $N-1$ and remember what the maximal solution was among all of these.
Here is some code implementing this algorithm:

```cpp
1  #include <bits/stdc++.h>
2  using namespace std;
3  int main() {
4    int T; cin >> T;
5    for (int t = 0; t < T; t++) {
6      int N, result = 0; cin >> N;
7      vector<int> h(N), left(N), right(N);
8      stack<int> leftStack, rightStack;
9      for (int j = 0; j < N; j++)
10       cin >> h[j];
11     for (int i = 0; i < N; i++) { // ideal buildings on the left
12       left[i] = leftStack.size();
13       while (!leftStack.empty() && h[i] >= leftStack.top())
14         leftStack.pop();
15       leftStack.push(h[i]);
16     }
17     for (int i = N - 1; i >= 0; i--) { // ideal buildings on the right
18       right[i] = rightStack.size();
19       while (!rightStack.empty() && h[i] >= rightStack.top())
20         rightStack.pop();
21       rightStack.push(h[i]);
22     }
23     for (int i = 0; i < N; i++) { // find optimal value
24       result = max(result, left[i] + right[i]);
25     }
26     cout << "Case #" << t << ": " << result << '\n';
27   }
28 }
```

The runtime analysis of this algorithm is a little bit more complicated than for the earlier subtasks, because it is at first glance unclear how much time the inner loops on the 13th and 19th lines need. The key observation is that every building will be added once to each queue and removed at most once from each queue. Overall, our solution runs in $O(N)$.

3 Touristtrap

Given is a rectangular map and a starting spot, your task is to determine whether or not it is possible to reach any of the border tiles from the starting point.

The four different subtasks had differing representations of the map and increasing limits on map complexity and size.

Subtask 1: ASCII map (20 points)

In the first subtask the map is represented as a grid where each tile is either accessible (_) or not (#). This case can be solved by starting a depth-first-search (`https://soi.ch/wiki/dfs/`) from Mouse Stofl's starting tile on the graph where each tile is connected to it's four neighboring tiles if they are accessible. If the DFS visits any border tiles, it's possible to escape, otherwise not. Since for a graph with N nodes and M edges DFS runs in $O(N+M)$ time, and we have one node for each accessible tile and at most four edges for each tile, the solution runs in $O(WH)$.

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


signed main() {
	int T; cin >> T;
	for(int t=0; t<T; ++t) {
		int w, h, x, y; cin >> w >> h >> x >> y;
		vector<string> map;
		copy_n(istream_iterator<string>(cin), h, back_inserter(map));
		cout << "Case #" << t << ": ";
		vector<array<int,2>> stack={{x,y}};
		bool poss = false;
		while(!stack.empty()) {
			auto pos = stack.back(); stack.pop_back();
			if (pos[0]<0 || pos[0]>=w || pos[1]<0 || pos[1]>=h)
				continue;
			if (map[pos[1]][pos[0]] != '_')
				continue;
			if (pos[0]==0||pos[1]==0||pos[0]==w-1||pos[1]==h-1) {
				poss = true;
				break;
			}
			map[pos[1]][pos[0]] = '.';
			stack.push_back({pos[0]-1,pos[1]});
			stack.push_back({pos[0]+1,pos[1]});
			stack.push_back({pos[0],pos[1]-1});
			stack.push_back({pos[0],pos[1]+1});
		}
		cout << ("IMPOSSIBLE\n"+2*poss);
	}
}
```

Subtask 2: Vector map (20 points)

In this subtask, the map is now given as a list of rectangles, whose contained tiles should be considered accessible. Since the limits on the size is still there, any solution for Subtask 1 should sill work, we just need to convert to the old format. We can start with a grid filled with # characters and fill each rectangle with _ characters. This will take $O(RWH)$ time in the worst case (most rectangle covering most of the grid). The test data didn't contain many cases with large-area rectangles, but even in the worst case the code is fast enough for $R \leq 1000$ because constants are very small.

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


signed main() {
	int T; cin >> T;
	for(int t=0; t<T; ++t) {
		int w, h, x, y, r; cin >> w >> h >> x >> y >> r;
		vector<string> map(h,string(w,'#'));
		for(int i=0; i<r; ++i) {
			int w, h, x, y; cin >> x >> y >> w >> h;
			for_each(map.begin()+y, map.begin()+y+h, [x,w](auto &row) {
				fill_n(row.begin()+x, w, '_');
			});
		}
		cout << "Case #" << t << ": ";
		vector<array<int,2>> stack={{x,y}};
		bool poss = false;
		while(!stack.empty()) {
			auto pos = stack.back(); stack.pop_back();
			if (pos[0]<0 || pos[0]>=w || pos[1]<0 || pos[1]>=h)
				continue;
			if (map[pos[1]][pos[0]] != '_')
				continue;
			if (pos[0]==0||pos[1]==0||pos[0]==w-1||pos[1]==h-1) {
				poss = true;
				break;
			}
			map[pos[1]][pos[0]] = '.';
			stack.push_back({pos[0]-1,pos[1]});
			stack.push_back({pos[0]+1,pos[1]});
			stack.push_back({pos[0],pos[1]-1});
			stack.push_back({pos[0],pos[1]+1});
		}
		cout << ("IMPOSSIBLE\n"+2*poss);
	}
}
```

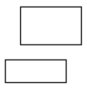
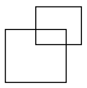
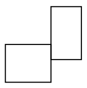
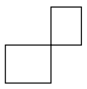
Subtask 3: Bigger map (30 points)

Now that we don't have a significant restriction on W and H (up to 10^9), our $O(WH)$ algorithm won't be fast enough.

We notice that any two tiles within the same rectangle are connected (possibly with some intermediate tiles). Because of the transitivity property of connectivity (if both tiles A and B

and tiles B and C are connected, A is also connected to C), if any two tiles of two rectangles are connected, all tiles in the rectangles are connected. Because are tiles inside a rectangle have the same connectivity, we can simplify our graph to only have one node for each rectangle. Now we just need a good way to figure out if two rectangles are connected.

We havet to consider four cases for a pair of rectangles:

Image	Description of extents	Connected?
	Disjoint in at least one dimension	×
	Intersecting in both dimensions.	✓
	Intersecting in one and touching in one dimension	✓
	Only touching in both dimensions	×

We can then create a node for each rectangle, and check every pair of rectangles to add an edge if they are connected. We then run DFS on this graph, starting from the rectangle containing Mouse Stofl's starting point (any of them if there are several). If we visit any rectangle that touches the border, we have an escape route, otherwise not.

Checking connectivity between rectangles is a constant-time operation, so we spend $O(R^2)$ time on generating the graph. The DFS also takes at most $O(R^2)$ to run in the worst case (many connections), giving a total running time of $O(R^2)$.

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

typedef array<int, 4> int4;

namespace std {
	istream& operator>>(istream& stream, int4 &r) {
		return stream >> r[0] >> r[1] >> r[2] >> r[3];
	}
}

int connectivity(int xs, int xe, int ys, int ye) {
	if (xs > ye || ys > xe)
		return 0;
	if (xs == ye || ys == xe)
		return 1;
	return 2;
}
bool connected(int4 &a, int4 &b) {
	return connectivity(a[0], a[0]+a[2], b[0], b[0]+b[2])
	    + connectivity(a[1], a[1]+a[3], b[1], b[1]+b[3]) > 2;
}
bool has(int4 &a, int x, int y) {
	return a[0] <= x && x <= a[0]+a[2]
	   && a[1] <= y && y <= a[1]+a[3];
}
bool border(int4 &a, int w, int h) {
	return a[0] == 0 || a[1] == 0 || a[0]+a[2] == w || a[1]+a[3] == h;
}

signed main() {
	int T; cin >> T;
	for(int t=0; t<T; ++t) {
		int w, h, x, y, r, s; cin >> w >> h >> x >> y >> r;
		vector<int4> rects;
		copy_n(istream_iterator<int4>(cin), r, back_inserter(rects));

		vector<vector<int> >g(r);
		for (int i = 0; i < r; ++i) {
			if (has(rects[i], x, y))
				s = i;
			for (int j = 0; j < i; ++j) {
				if (connected(rects[i], rects[j])) {
					g[i].push_back(j);
					g[j].push_back(i);
				}
			}
		}

		cout << "Case #" << t << ": ";
		vector<int> stack={s};
		vector<bool> vis(r,false);
		bool poss = false;
		while(!stack.empty()) {
			int pos = stack.back(); stack.pop_back();
			if (vis[pos])
				continue;
			vis[pos] = true;
			if (border(rects[pos], w, h)) {
				poss = true;
				break;
			}
			copy(g[pos].begin(), g[pos].end(), back_inserter(stack));
		}
		cout << ("IMPOSSIBLE\n"+2*poss);
	}
}
```

Subtask 4: Even bigger map (30 points)

For this subtask, we need a solution that runs in $O(n \log n)$ *in the worst case*. Even if we have a smarter way of finding connecting rectangles, we can still have up to $O(R^2)$ connections (consider many long vertical and horizontal rectangles).

Dealing with rectangles touching only at corners

We first deal with rectangles that touch only a a corner. If we replace $[a,b]\times[c,d]$ by the "cross" $[3a,3b]\times[3c+1,3d-1]\cup[3a+1,3b-1]\times[3c,3d]$, then two rectangles are connected if and only if the corresponding crosses touch or intersect. We also note that two crosses never touch only at a corner, so this transformation got rid of that special case.

An elementary $O(n \log n)$ solution

We perform scanline on x and use a union-find datastructure to store the components of rectangles based on all intersection to the left of the sweepline. We mantains two sets of pairs (y-coordinates, rectangle-id):

- The set S stores the lower and upper coordinate of every rectangle that intersects the sweepline.
- The set T stores all elements of S that are in a different component than their predecessor in S or in a different component than their successor in S.
- Whenever we insert or delete something in S, we need to check whether its predecessor or successor needs to be added to or removed from T.

The intuitive idea behind the second set is the following: If many adjacent y-coordinates $b_1, \ldots, b_k$ belong to the same component, then a new rectangle $[y_1, y_2]$ that intersects some of them is either fully contained in $[b_1, b_k]$, or it contains b_1 or b_k.

At the current x-position, we first process add all rectangles that start here and then we remove all rectangles that end here. To add a rectangle R with y-range $[y_1, y_2]$, we do the following.

1. We first seek to find all components that R intersects. Let A be the set of all coordinates in T that are in $[y_1, y_2]$. We unite the R with all elements of A. Next, we remove all but the first and last element of A from T. This removal step makes the solution fast, as whenever we need a lot of time, there are a lot of elements in A, so also remove a lot of elements from T. Finally, we check whether we need to remove the first and last elements of A from T.
2. If in the first step, A was empty, the we're in the special case where all elements of S in $[y_1, y_2]$ belong to the same component. In this case, we just unite A with any element of S in $[y_1, y_2]$.
3. Insert y_1 and y_2 into S (and then as always check if we need to add / remove elements to / from T).

To remove a rectangle, simply remove its endpoints from S and T.

After the scanline, we know how the connected components of the rectangles look like, so we just iterate over all components and check if they contain both the starting point and a rectangle touching the border.

Note that this solutions uses only requires some scanline with "std::set". No fancy datastructes are used here.

Runtime analysis

This solution runs in $\Theta(r \log r)$. Sorting the events and the union-find take $\Theta(r \log r)$ time. The $2r$ insert and remove operations perfomed on S take $O(\log r)$ time each, so that's also fine. The non-obvious part is that mainting the set T takes $O(r \log r)$ time.

We first note, that T can be efficiently updated when we insert or erase some element from S, as we only check if the predecessor and successor should belong to T, so we insert or remove at most two elements from T. Hence this doesn't increase the asymptotic runtime.

What remains to show is that the first step when adding a rectangle is fast. Finding the set A and removing its interior elements from T takes $O(\log r + |A|)$ time. While $|A|$ might be large, we then also remove $|A| - 2$ elements from T. As all other steps run in $\Theta(r \log r)$ time, we add at most that many elements to T, so we can also remove at most that many elements from T. Hence the combined size of all A is at most $O(r \log r)$ and this part is also fast.

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

using ll = int64_t;
constexpr int inf = 1e9;

struct Rectangle{
    ll x1, x2, y1, y2;
    int id;
    bool contains(ll x, ll y) const {
        return x1 <= x && x < x2 && y1 <= y && y < y2;
    }
    bool touches_border(ll w, ll h) const {
        return x1 == 0 || y1 == 0 || x2 == w || y2 == h;
    }
};
struct Event{
    ll x;
    ll y1, y2;
    int id;
    bool is_deletion;
    bool operator<(Event const&o) const {
        return make_pair(x, is_deletion) < make_pair(o.x, o.is_deletion);
    }
};
struct Union_Find{
    vector<int> p;
    Union_Find(int n) : p(n) {
        iota(p.begin(), p.end(), 0);
    }
    int f(int x){
        return p[x] ==x ? x : p[x] = f(p[x]);
    }
    void u(int a, int b){
        p[f(a)] = f(b);
    }
};

signed main(){
    int T;
    cin >> T;
    for(int cas=0; cas<T; ++cas){
        cout << "Case #" << cas << ": ";
        int r;
        ll W, H, X, Y;
        cin >> W >> H >> X >> Y >> r;
        vector<Rectangle> recs;
        // read rectangles and transform them into crosses
        for(int i=0;i<r;++i){
            ll x1, y1, x2, y2, w, h;
            cin >> x1 >> y1 >> w >> h;
            x2 = x1+w;
            y2 = y1+h;
            recs.push_back(Rectangle{3*x1+1, 3*x2-1, 3*y1, 3*y2, i});
            recs.push_back(Rectangle{3*x1, 3*x2, 3*y1+1, 3*y2-1, i});
        }
        // also transform grid
        X=3*X+1; Y=3*Y+1; W*=3; H*=3;
        // compute events
        vector<Event> evs;
        for(auto const&rec:recs){
            // insertion event
            evs.push_back(Event{rec.x1, rec.y1, rec.y2, rec.id, false});
            // deletion event
            evs.push_back(Event{rec.x2, rec.y1, rec.y2, rec.id, true});
        }
        sort(evs.begin(), evs.end());
        // set up datastructures
        set<pair<ll, int> > S, T;
        Union_Find uni(r);
        auto should_be_in_T = [&](set<pair<ll, int> >::iterator it){
            // check if first or last
            if(it == S.begin()) return true;
            auto it2 = next(it);
            if(it2 == S.end()); return true;
            // check if successor is in different component
            if(uni.f(it2->second) != uni.f(it->second)) return true;
            // check if predecessor is in different component
            auto it3 = prev(it);
            if(uni.f(it3->second) != uni.f(it->second)) return true;
        };
        auto update_in_T = [&](set<pair<ll, int> >::iterator it){
            if(should_be_in_T(it)){
                T.insert(*it);
            } else {
                T.erase(*it);
            }
        };
        auto check_pred_succ = [&](pair<ll, int> const&p){
            auto it_suc = S.upper_bound(p);
            if(it_suc != S.end()) update_in_T(it_suc);
            auto it = S.lower_bound(p);
            if(it != S.begin()) update_in_T(prev(it));
        };
        auto remove_point = [&](pair<ll, int> const&p){
            S.erase(p);
            T.erase(p);
            check_pred_succ(p);
        };
        auto add_point = [&](pair<ll, int> const&p){
            auto it = S.insert(p).first;
            update_in_T(it);
            check_pred_succ(p);
        };
        // scanline
        for(auto const&ev:evs){
            if(ev.is_deletion){
                remove_point(make_pair(ev.y1, ev.id));
                remove_point(make_pair(ev.y2, ev.id));
            } else { // insertion
                auto it_l = T.lower_bound(make_pair(ev.y1, -inf)), it_r =
                ↪  T.upper_bound(make_pair(ev.y2, inf));
                // The set A of intersections is now [it_l, ..., prev(it_r)]
                if(it_l == it_r){
                    // special case: [y1, y2] is inside endpoints belonging to the same
                    ↪  component
                    // find the <=1 component [y1, y2] intersects
                    auto it = S.lower_bound(make_pair(ev.y1, -inf));
                    if(it != S.end() && it->first <= ev.y2){
                        uni.u(it->second, ev.id);
                    }
                } else {
                    // unite with elements in A
                    for(auto it = it_l;it != it_r;++it){
                        uni.u(it->second, ev.id);
                    }
                    // remove elements in the interior of A
                    auto erase_l = next(it_l), erase_r = prev(it_r);
                    if(erase_l != it_r){ // don't erase if A has size 1
                        T.erase(erase_l, erase_r);
                        // check if first and last element of A still belong into T
                        update_in_T(S.find(*it_l));
                        update_in_T(S.find(*erase_r));
                    }
                }
                add_point(make_pair(ev.y1, ev.id));
                add_point(make_pair(ev.y2, ev.id));
            }
        }
        // check if a component contains (X, Y) and touches the border.
        vector<bool> contains(r,0), touches(r, 0);
        for(auto &e:recs){
            if(e.contains(X, Y)) contains[uni.f(e.id)] = true;
            if(e.touches_border(W, H)) touches[uni.f(e.id)] = true;
        }
        bool can_escape = false;
        for(int i=0;i<r;++i){
            if(contains[i] && touches[i]) can_escape = true;
        }
        if(can_escape){
            cout << "POSSIBLE\n";
        } else {
            cout << "IMPOSSIBLE\n";
        }
    }
}
```

4 Pipeline

Given are N spots to place pumps ($x_0 < x_1 < \cdots < x_{N-1}$). For any subset of points including the start and the end, that is all $\{x_{i_0}, x_{i_1}, \ldots, x_{i_P}\}$ with $0 = i_0 < i_1 < \ldots i_P = N-1$), we define the LTC as the maximum distance between any two adjacent points distance, i.e. LTC $= \max\limits_{0 \le j \le P-2} \{x_{i_{j+1}} - x_{i_j}\}$.

The three subtasks were:

1. Find the minimal P for a fixed LTC.
2. Find the minimal LTC for a fixed P.
3. Find the minimal P for all LTC $\in \{1, 2, \ldots, M\}$ with an algorithm that has minimal space usage.

Subtask 1: Fixed LTC (15 points)

To find the minimal number of pumps for a fixed LTC, we do the following algorithm: Go from left to right and only take those pumps that are absolutely necessary. A pump is necessary if taking the next pump has a distance larger than LTC from the previous pump that has been taken.

An algorithm that solves a problem by making locally optimal decision is called *greedy*. In this case, we take the optimal pump location at each step, ignoring the possible spots after that point or before the last station that was taken. While this gives us the best solution locally, it does not necessarily give us a globally optimal solution. A proof of why this is indeed correct for this task is given in the last subtask.

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

vector<int> greedy(int ltc, vector<int> const& xs) {
  vector<int> take = { xs[0] }; // always take the first
  int x_prev = xs[0];
  for (auto x : xs) {
    if (x - take.back() > ltc)
      take.push_back(x_prev); // gap too large, need to take the last one
    if (x - take.back() > ltc)
      return {}; // impossible to cover, return empty vector
    x_prev = x;
  }
  if (x_prev != take.back()) // take the last, if not already taken
    take.push_back(x_prev);
  return take;
}

int main() {
  int t; cin >> t;
  for (int tc=0; tc<t; ++tc) {
    int n, ltc; cin >> n >> ltc;
    vector<int> xs;
    copy_n(istream_iterator<int>(cin), n, back_inserter(xs));
    cout << "Case #" << tc << ": ";
    auto answer = greedy(ltc, xs);
    if (answer.empty())
      cout << "Impossible";
    else if (answer.size() > 9000)
      cout << answer.size();
    else
      copy(answer.begin(), answer.end(), ostream_iterator<int>(cout<<answer.size()<<' ', " "));
    cout << '\n';
  }
}
```

Subtask 2: Fixed Pumps Count (15 points)

This time we want to minimize the LTC given a fixed number of pumps. The key idea was to not construct the solution directly, but to use the algorithm of the previous subtask.

Say we want to use 4 pumps. We can run the previous algorithm for multiple values of LTC, for example:

- Solve with LTC = 1 $\rightsquigarrow$ impossible
- Solve with LTC = 2 $\rightsquigarrow$ impossible
- Solve with LTC = 3 $\rightsquigarrow P = 9$
- Solve with LTC = 4 $\rightsquigarrow P = 6$
- Solve with LTC = 5 $\rightsquigarrow P = 5$
- Solve with LTC = 6 $\rightsquigarrow P = 4$
- Solve with LTC = 7 $\rightsquigarrow P = 3$
- Solve with LTC = 8 $\rightsquigarrow P = 3$
- Solve with LTC = 9 $\rightsquigarrow P = 3$
- Solve with LTC $\geq 10 \rightsquigarrow P = 2$

And then we see that we can get an LTC of 6 using exactly 4 pumps. A smaller LTC, say LTC=5 requires at least 5 pumps.

What if we have $P = 3$ pumps? Because we want to minimize the LTC, we take the *smallest*, in this case LTC=7. Also what if we have $P = 7$ pumps? We can't ever select spots with an LTC of 3, so we have to do an LTC=4 with 6 pumps. As we're required by the task description to actually use 7 pumps, we just add another pump that has been previously unused.

In general, the algorithm would be to take the smallest LTC that uses at most P pumps and then fill up with remaining pumps until it uses exactly P pumps.

If we were to code just that, trying all possible LTC's until we're good is too slow. However, we can do it smarter. We notice that the values of the P are decreasing when the LTC is increasing. This allows us to make use of binary search: We start with some value, say LTC = 6. If we use at least P pumps, we decrease it, say to LTC = 3. If we use less than P pumps, we increase it, say to LTC = 12.

More formally, we have two variables: l (for left) and r (for right). Those two variables define our search range $\{l + 1, l + 2, \ldots, r\}$ (meaning we know that the optimal LTC lies in this range) and satisfy the following conditions:

- if we have LTC = l, it's either impossible or we need $\geq P$ pumps

- if we have LTC = r, its possible and we need $< P$ pumps

Initially, we set $l = 0$ (then it's never possible) and $r = x_{N-1} - x_0$ (then we only need 2 pumps). We then take a value in the middle, $m = \lfloor \frac{l+r}{2} \rfloor$. We check what happens with $LTC = m$. If it's possible and we need less than P pumps, we set the variable r to m. If not, it's either impossible or we need at least P pumps, so we can set l to m. Note that our search range decreased from $r - l$ to $l - m$ or $r - m$ and in both cases it has been halved. So if originally we had $l = 0$ and $r = 2^{30}$, we need exactly 30 steps to have $r = l + 1$. In this case, we have found our optimal LTC of r. As the distance between the last and the first pump station are at most $10^9 < 2^{30}$, we will use at most $\lceil \log_2 10^9 \rceil = 30$ steps. So this algorithm runs in $O(n \log C)$ (where $C = x_{N-1} - x_0$).

Note that the initial values as given above are incorrect for $N = 1$. In this case, solving it with $l = 0$ is always possible. We can fix that by starting with $l = -1$.

There are many ways to "fill up" the remaining pumps. One is to put all pump locations selected by the greedy algorithm in a `std::set<int>` and insert values until its size is P. A set keeps each value only once, so this works and the running time would be $O(n \log(\max n, C))$ is fast because inserting into a set is logarithmic time and we do it at most n times.

Another way to do it is to make use of the fact that the values are already sorted, so we can compute the unused values in linear time and then take as many of them as we need. The function `set_difference` computes the difference between two sorted lists ($A \setminus B = \{x \in A | x \notin B\}$) and the function `inplace_merge` sorts a list that consists of two sorted halves. Doing it this way gets rid of a log-factor in the last step. The total running time thus is $O(n \log C)$.

```cpp
vector<int> greedy(int ltc, vector<int> const& xs) {
  ... // same as in previous subtask
}

pair<int, vector<int>> binary_search(size_t max_pumps, vector<int> const& xs) {
  // invariant:
  // - impossible with <=max_pumps and LTC=left
  // - possible   with <=max_pumps and LTC=right
  int left=-1, right=xs.back() - xs.front();
  while (right - left > 1) {
    int mid = (left + right)/2;
    auto ans = greedy(mid, xs);
    if (ans.empty() || ans.size() > max_pumps)
      left = mid; // impossible
    else
      right = mid; // possible
  }
  auto take = greedy(right, xs);
  vector<int> unused;
  set_difference(xs.begin(), xs.end(), take.begin(), take.end(), back_inserter(unused));
  int necessary = take.size();
  copy_n(unused.begin(), max_pumps - necessary, back_inserter(take));
  inplace_merge(take.begin(), take.begin() + necessary, take.end());
  return {right, take};
}

int main() {
  int t; cin >> t;
  for (int tc=0; tc<t; ++tc) {
    int n, p; cin >> n >> p;
    vector<int> xs;
    copy_n(istream_iterator<int>(cin), n, back_inserter(xs));
    auto [ltc, answer] = binary_search(p, xs);
    cout << "Case #" << tc << ": " << ltc << '\n';
    if (answer.size() <= 9000)
      copy(answer.begin(), answer.end(), ostream_iterator<int>(cout << '\n', " "));
    else
      cout << "Over 9000";
    cout << '\n';
  }
}
```

Subtask 3: Walking down the BTC (10 points)

To find the minimal P for all LTC $\in \{1, 2, \ldots, M\}$, it was enough to run the solution of subtask 1 all different values of LTC. The goal of this task was to prepare for the last subtask.

Subtask 4: Walking down the ETH in $\mathcal{O}(N \cdot M)$ (20/60 points)

This section shows the intended solution for 20 out of 60 points. We will look at the solution that score more than that in the following section. We structure the proof as follows. First, we show that greedy works and is indeed correct. Then, we start with the trivial solution that runs greedy on all values and transform he solution into another one that computes the same results but uses less memory.

So, why does greedy work? Looking again at the source code for subtask 1:

```
vector<int> greedy(int ltc, vector<int> const& xs) {
  vector<int> take = { xs[0] }; // always take the first
  int x_prev = xs[0];
  for (auto x : xs) {
    if (x - take.back() > ltc)
      take.push_back(x_prev); // gap too large, need to take the last one
    if (x - take.back() > ltc)
      return {}; // impossible to cover, return empty vector
    x_prev = x;
  }
  if (x_prev != take.back()) // take the last, if not already taken
    take.push_back(x_prev);
  return take;
}
```

The idea was to always select the last pump station we can take without violating the LTC. By construction, we always take the first and the last element and if we find a solution, we always find one that is viable (not violating any of the constraints). What remains to be shown is that what we select is optimal, meaning it has the minimal number of pump stations.

As with many greedy algorithm, we can do a proof by contradiction followed by an *exchange argument*. Consider an arbitrary input. Our algorithm will produce the solution $A = \{a_0, a_1, \ldots, a_{P-1}\}$. Look at an optimal solution $O = \{o_0, o_1, \ldots, o_{Q-1}\}$. Assume that this other solution is *better* than ours, that is $Q < P$. Without loss of generality, let's further assume that in case there are multiple optimal solutions, we take the one that agrees with ours on the longest possible prefix. In other words take that O such that there is an k where both mismatch ($a_i = o_i$ for $0 \leq i \leq c$ and $a_k \neq o_k$)

and this k is maximal. Such an k always exists because the last element of both solutions must be equal (as it was required by the task statement that start and end are in there) and as they have different length, they must differ at some element. Furthermore, we have $2 \leq k < Q$ because the start is the same in both elements.

Now, by construction of A, we must have $a_k \geq o_k$. Why? Firstly, a_k is not the last element ($a_k \neq x_{N-1}$) because only a_{Q-1} can be equal to x_{N-1} and we know that $Q < P$. So we can only have chosen a_k on line 6. The condition on line 5 tells us that the spot following a_k is larger than $a_{k-1} + \text{LTC}$, so it would be illegal to be chosen. Therefore, $o_k > a_k$ is impossible (illegal), $o_k = a_k$ is impossible (by definition of k), so we must have $o_k < a_k$.

As $o_k \neq x_{N-1}$, we must have $k < Q - 1$ and thus o_{k+1} exists. Now comes the clue: We construct a new solution $O' = \{o_0, o_1, \dots, o_{k-1}, a_k, o_{k+1}, \dots, o_{Q-1}\}$. This solution is valid, because (a) $a_k - o_{k-1} \leq \text{LTC}$ (because $o_{k-1} = a_{k-1}$), (b) $o_{k+1} - a_k \leq \text{LTC}$ (because $a_k > o_k$), (c) $o_{k+1} - a_k > 0$ (otherwise, $O \setminus \{o_k\}$ would be a valid solution and thus O would not be optimal). However, this solution agrees with A on the first k elements, and a mismatch only happens at $k' \geq k + 1$, contradicting the assumption that k was maximal. We have a contradiction, thus we can only conclude that such an O can not exist. $\square$

The running time of this solution is clearly $O(n)$ as we iterate through each value exactly once.

Doing this naively for every LTC we end up with the following algorithm:

```cpp
vector<int> serial_greedy(int max_ltc, vector<int> const& xs) {
  const int INF = 1e9; // sentinel
  vector<int> last(max_ltc+1, xs[0]);
  vector<int> need(max_ltc+1, 1);
  for (int ltc=1; ltc <= max_ltc; ++ltc) { // ltc-loop
    for (int i=0; i<xs.size(); ++i) { // xs-loop
      if (xs[i] - last[ltc] > ltc) {
        last[ltc] = xs[i-1]; // gap too large, need to take the last one
        need[ltc] += 1;
      }
      if (xs[i] - last[ltc] > ltc)
        need[ltc] = INF; // impossible
    }
    if (xs.back() != last[ltc]) // take the last, if not already taken
      need[ltc] += 1;
  }
  need.erase(need.begin()); // ltc=0 is uninteresting
  replace_if(need.begin(), need.end(), [](int n) { return n >= INF; }, -1);
  return need;
}
```

This code is slightly different to the one shown before in order to prepare for what's about to come. We don't return early if we see something is impossible, we set the number of required pumps to $+\infty$ instead. We also store our temporary variables inside the arrays `last` and `need`. Functionally the code remains equivalent after these changes.

The memory required for this solution is $O(N + M)$. We need an array of size N for all values $\{x_0, \dots, x_{N-1}\}$. We also need two arrays of size $M + 1$ for our temporary variables.

We can't get rid of this $O(N)$ easily because we need all the values at different points in time. What we need to do first is to swap the two for loops. Doing so will not change *what's* computed, it will only change *when* it's computed.

```cpp
vector<int> parallel_greedy(int max_ltc, vector<int> const& xs) {
  const int INF = 1e9; // sentinel
  vector<int> last(max_ltc+1, xs[0]);
  vector<int> need(max_ltc+1, 1);
  for (int i=0; i<xs.size(); ++i) { // xs-loop
    for (int ltc=1; ltc <= max_ltc; ++ltc) { // ltc-loop
      if (xs[i] - last[ltc] > ltc) {
        last[ltc] = xs[i-1]; // gap too large, need to take the last one
        need[ltc] += 1;
      }
      if (xs[i] - last[ltc] > ltc)
        need[ltc] = INF; // impossible
    }
  }
  for (int ltc=1; ltc <= max_ltc; ++ltc)
    if (xs.back() != last[ltc]) // take the last, if not already taken
      need[ltc] += 1;
  need.erase(need.begin()); // ltc=0 is uninteresting
  replace_if(need.begin(), need.end(), [](int n) { return n >= INF; }, -1);
  return need;
}
```

Now we don't need to store all N values of x's anymore at the same time! We only ever look at `xs[i]` and `xs[i-1]`. We can thus read the values on the fly, only storing `x` (replacing `xs[i]`) and `x_prev` (replacing `xs[i-1]`). We've reached a streaming algorithm that only keeps $O(M)$ values at the same time, but still computes the same as the correct greedy algorithm.

```cpp
vector<int> streaming_greedy(int max_ltc, int n) {
  const int INF = 1e9; // sentinel
  int x, x_prev;
  cin >> x;
  vector<int> last(max_ltc+1, x);
  vector<int> need(max_ltc+1, 1);
  for (int i=1; i<n; ++i) {
    x_prev = x;
    cin >> x;
    for (int ltc=1; ltc <= max_ltc; ++ltc) {
      if (x - last[ltc] > ltc) {
        last[ltc] = x_prev; // gap too large, need to take the last one
        need[ltc] += 1;
      }
      if (x - last[ltc] > ltc)
        need[ltc] = INF; // impossible
    }
  }
  for (int ltc=1; ltc <= max_ltc; ++ltc)
    if (x != last[ltc]) // take the last, if not already taken
      need[ltc] += 1;
  need.erase(need.begin()); // ltc=0 is uninteresting
  replace_if(need.begin(), need.end(), [](int n) { return n >= INF; }, -1);
  return need;
}
```

The solution that runs in $(N \cdot M^{0.5}(\log M)^2)$ will be released in an update to this booklet in the online version.

5 Storytelling

In this task you were given n mice sitting around a circular table about to tell their stories. You were given rules how they were allowed to talk and your goal was to calculate the minimum time in which all mice could finish storytelling.

Subtasks 1 to 4

The first four subtasks had increasing input size which distinguished among different solutions. The Subtask 1 was solvable through a bruteforce (e.g., try all $n!$ permutations to find the answer) and Subtask 2 by a binary search.

To earn a full score one had to come up with a linear (both in time and space) solution which uses just a few minimums and maximums. One such implementation is listed below. The proof why this works is discussed in the following section and would earn you full score for the theoretical part.

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

using namespace std;

void schedule(int n, vector<long long>& V, vector<long long>&ret) {
  ret.resize(n);

  if (n % 2 == 0) {   // even case
    for (int i=0; i<n; ++i)
      if (i % 2 == 0)
        ret[i] = V[i];  // even mice speak from the beginning
      else              // odd mices speak when they can
        ret[i] = V[i] + max(V[(i-1+n)%n], V[(i+1)%n]);
  } else {              // odd case
    long long min_3 = V[n-2] + V[n-1] + V[0];
    int min_i = 0;  // find the mice min_i which will wait for min_i-2 and min_i-1
    for (int i=0; i<n; ++i) {
      long long s = V[(n+i-2)%n] + V[(n+i-1)%n] + V[i];
      if (s < min_3) {
        min_3 = s;
        min_i = i;
      }
    }

    for (int i = min_i + 1; i < min_i + n - 1; ++i ) {
      if (i % 2 != min_i % 2)
        ret[i%n] = V[i%n];  // odd mices relative to min_i speak first
      else
        ret[i%n] = V[i%n] + max(V[(n+i-1)%n], V[(n+i+1)%n]);
    }
    // min_i-1 speaks once min_i-2 is finished
    ret[(min_i-1+n)%n] = V[(min_i-2+n)%n] + V[(min_i-1+n)%n];
    // in the end once min_i-1 and min_i+1 are finished min_i speaks
    ret[min_i] = max(min_3, V[min_i] + V[(min_i+1)%n]);
  }
}

int main() {
  int T, N;

  cin >> T;
  for (int t=0; t<T; ++t) {
    cin >> N;
    vector<long long> V(N);
    for (int i=0; i<N; ++i) cin >> V[i];

    vector<long long> res; // endtimes of speaking
    schedule(N, V, res);
    cout << "Case #" << t << ": " << *max_element(res.begin(), res.end()) << '\n';
    for (int i=0; i<N; ++i)
      cout << res[i]-V[i] << (i == N-1 ? "\n" : " ");
  }
}
```

Subtask 5

In order to have simpler formulas, all mouse indices in this solution are implicitly considered modulo n. For example, mouse $n+3$ is the same as mouse 3, and $y_{3-n} = y_{3+n} = y_3$.

First, let's assume that n is even, i.e., $n = 2k$ for some integer k.

We'll start by showing how to find a lower bound: for any instance $y_1, \ldots, y_n$ we will find a time Y such that no solution can finish earlier than at Y. For each i we can make the following argument: Mice i and $i+1$ cannot speak at the same time, so we know that mouse i cannot be done speaking and listening sooner than at the time $y_i + y_{i+1}$. Hence, the entire group cannot be done sooner than at the time $Y = \max_{1 \le i \le n}(y_i + y_{i+1})$.

Next, we will show that this Y is also an upper bound. In other words, we will find an algorithm that will always construct a schedule such that everybody will finish talking by the time Y. One such algorithm looks as follows:

- All mice with odd indices start telling their stories at time 0.
- Each mouse with an even index starts telling its story as soon as both of its neighbours finish telling theirs.

Clearly, the last mouse to finish talking will be an even-numbered mouse. Mouse $2j$ will finish talking precisely at the time $\max(y_{2j-1} + y_{2j}, y_{2j} + y_{2j+1})$. It is obvious that the maximum of all these times is precisely Y.

Now let's assume that n is odd, i.e., $n = 2k+1$ for some integer k.

The lower bound shown in the even case still holds, as the argument is valid for any number of mice. However, in some situations this lower bound cannot be achieved. For example, if we have three mice with $y_1 = y_2 = y_3 = 1$, we get the lower bound $Y = 2$, but the actual optimal solution needs 3 seconds. Below, we will show a second lower bound that will cover such situations.

Suppose that we have any valid schedule telling each mouse when it should start speaking. As the number of mice is odd, there has to be a mouse who starts speaking **after** one of its neighbors but **before** the other one. (The opposite is impossible, because if you only have mice who start speaking before both neighbors and mice who start speaking after both neighbors, these two type of mice have to alternate, which is impossible if the total number of mice is odd.)

Without loss of generality, suppose that mice q, $q+1$, and $q+2$ have the property that mouse $q+1$ speaks after mouse q but before mouse $q+2$. Then, mouse $q+2$ cannot finish its own story sooner than at the time $y_q + y_{q+1} + y_{q+2}$.

Note that for the lower bound we now need to take the **minimum** (and not the **maximum**) of all such triples. This is because all we know is that each valid schedule has to contain at least one such triple, and it is possible that the optimal solution will contain the cheapest of all such triples. Thus, the second lower bound we get is $Y' = \min_{1\le i\le n}(y_i + y_{i+1} + y_{i+2})$.

To finish this task, we will now show an algorithm for $n = 2k+1$ that always produces an optimal solution – i.e., a solution that ends at the time $\max(Y, Y')$. This new algorithm looks as follows:

- Rotate the mice around the table until you get the situation where $y_{n-2} + y_{n-1} + y_n = Y'$.
- All mice with odd indices, except for mouse n, can start telling their stories at time 0.
- Each mouse with an even index, except for mouse $n-1$, starts telling its story once all its neighbours from the first group finish telling theirs.
- Mouse $n-1$ starts telling its story after mouse $n-2$.
- Finally, mouse n can tell its story once both its neighbors are done.

Why is this algorithm optimal? The last mouse to finish talking is either an even-numbered mouse, or mouse n. An even-numbered mouse $2j$ will finish telling its story at the time $\max(y_{2j-1} + y_{2j}, y_{2j} + y_{2j+1}) \le Y$. Mouse n will finish its story at the time $\max(y_n + y_1, y_{n-2} + y_{n-1} + y_n)$. The first of these two is at most Y, while the second one is exactly Y'. As a result, all mice will finish their stories by the time $\max(Y, Y')$.

Note that the entire algorithm can be implemented in linear time and space – each of the first three steps can be implemented as a single pass through the array of talking times.

6 Tank Golf

Scoring

Bugs

10 points were awarded for submitting running code. This means that the submission should compile and be able to handle all matches without crashing.

Elo

After fixing all crashes, the bots were tested against each other. Each bot was pitted against every other bot multiple times. An Elo rating was calculated for each bot using the Bayesian Elo algorithm[1]. This means that Elo scores were chosen such that they maximize the likelihood of correctly predicting every match. The Elo ratings of the bots are shown in figure 6.1.

Elo scores r can be converted to playerstrength γ using

$$\gamma = 10^{\frac{r}{400}}$$

The win chance of a player E_1 can then be calculated using

$$E_1 = \frac{\gamma_1}{\gamma_1 + \gamma_2}$$

90 points were awarded proportionally to the playerstrength of the submission, i.e. score $= 90\frac{\gamma}{\gamma_{\text{ref}}}$. The maximum score γ_{ref} was set at the third place submission by Nicolas Camenisch.

Strategies

Condition checking

By simulating multiple possible shooting directions or spawn locations, each possible outcome can be tested for a few conditions. This way the bot can recognize which shots will throw the enemy off the map. Advanced conditions such as trying to return ones own tank back into an upright position if it was flipped over can be evaluated as alternative options. The main weakness with checking for conditions directly is that there are too many cases to efficiently cover all of them.

State scoring

Instead of checking for conditions strictly, each condition can be converted into a score. These scores are added up to give a numeric value to every possible game state. Then, from all possible actions the highest scoring one is chosen. In addition to the previous conditions, things like distance to the nearest ledge and distance to the enemy tank can be included in the score.

[1]`https://www.remi-coulom.fr/Bayesian-Elo/#theory`

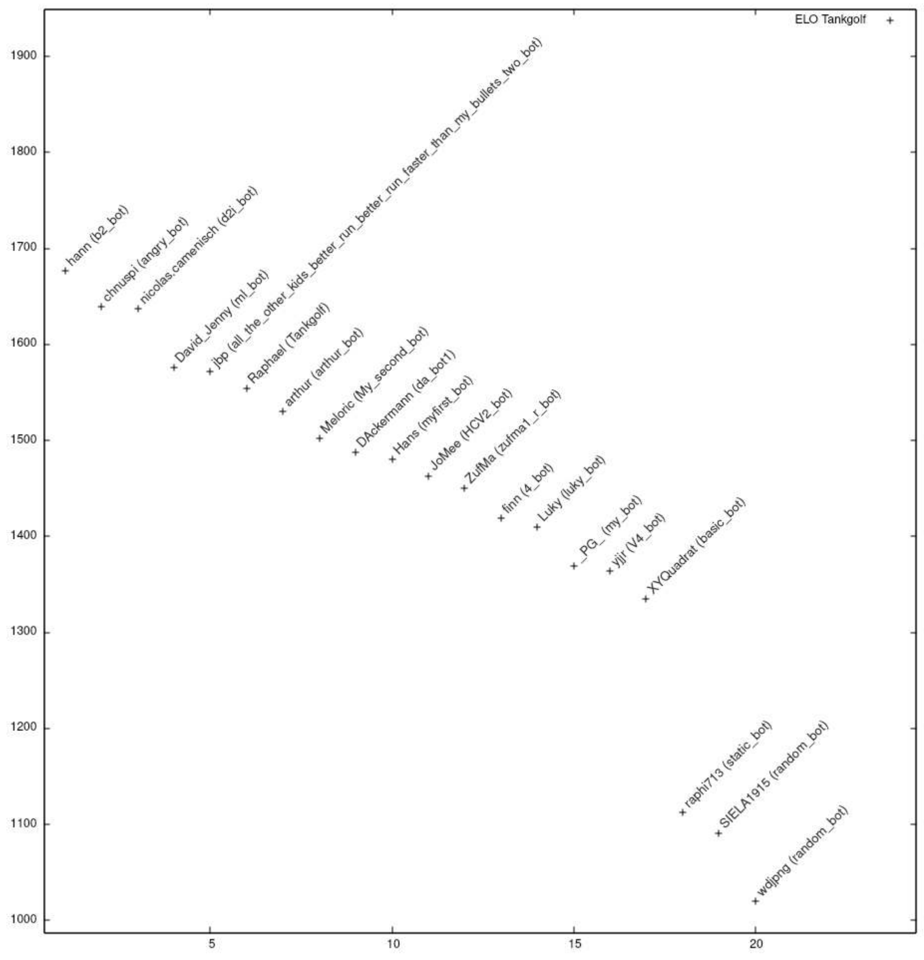


Figure 6.1: Distribution of Elo ratings depending on the rank.

Targeted search

Once a scoring function has been established, possible shooting positions can be refined. As only a limited number of requests are available, it makes sense to focus the search near high scoring actions. Variations of gradient ascent can be used to find local maxima.