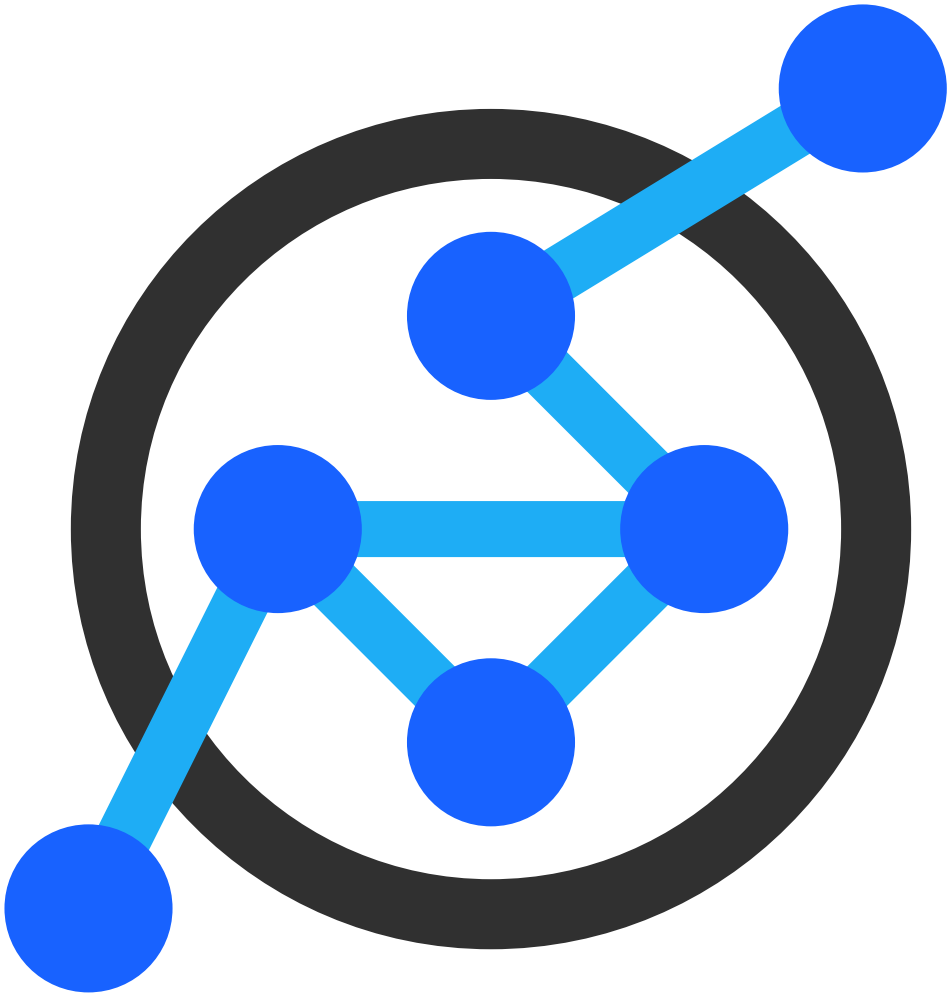
Round 2 SOI 2025/2026

Solution Booklet



Swiss Olympiad in Informatics

10 October 2025 – 30 November 2025

Registan Pattern

Task Idea	Luca Versari
Task Preparation	Ursus Wigger
Translation (en)	Ursus Wigger
Translation (de)	Benjamin Schmid
Translation (fr)	Noémie Jacquemot
Solution Author	Ursus Wigger

Subtask 1: A short wall (11 points)

In this subtask, there are only $3! = 6$ possible inputs:

1. `0 1 2` ⇒ `1 1 1`
2. `0 2 1` ⇒ `1 2`
3. `1 0 2` ⇒ `2 1`
4. `1 2 0` ⇒ `Impossible`
5. `2 0 1` ⇒ `Impossible`
6. `2 1 0` ⇒ `3`

It suffices to hardcode the answers.

- Runtime: $\mathcal{O}(1)$
- Memory: $\mathcal{O}(1)$

Subtask 2: The entrance (17 points)

Precomputing all $n!$ possible input permutations takes too many resources. However, there are a lot less segmentations than permutations.

Lemma 1. *There are 2^{n-1} possible segmentations*

Proof. Between every two consecutive elements, you have the choice of putting a segment separator there or not. There are $n-1$ such spots, so the total number of choices are 2^{n-1} □

With that, we generate all possible segmentations and check for each whether it correctly sorts the array.

(Solution by Leo Chen)

```cpp
#include <bits/stdc++.h>
using namespace std;
// #define int int64_t

bool check(const vector<int>& a, const vector<int>& segments) {
    int result_pos = 0;  // Position in the final sorted array
```

```
7       int array_pos = 0;   // Current position in the original array
8
9       // Process each segment
10      for (int seg_len : segments) {
11          // Check if flipping this segment gives us the correct values
12          for (int i = 0; i < seg_len; i++) {
13              // When we flip the segment, position array_pos + i becomes
                result_pos + (seg_len - 1 - i)
14              int flipped_index = array_pos + seg_len - 1 - i;
15              if (a[flipped_index] != result_pos + i + 1) {
16                  return false;
17              }
18          }
19          result_pos += seg_len;
20          array_pos += seg_len;
21      }
22
23      return true;
24  }
25
26  void solve() {
27      int n; cin >> n;
28      vector<int> a(n);
29      for (int i = 0; i < n; i++) {
30          cin >> a[i];
31      }
32
33      // Try all possible ways to split into segments using bitmask
34      // Bit i = 1 means there's a segment boundary after position i
35      for (int mask = 0; mask < (1 << (n-1)); mask++) {
36          vector<int> segments;
37          int start = 0;
38
39          for (int i = 0; i < n-1; i++) {
40              if (mask & (1 << i)) {
41                  // There's a boundary after position i
42                  segments.push_back(i - start + 1);
43                  start = i + 1;
44              }
45          }
46          // Add the last segment
```

```
47          segments.push_back(n - start);
48
49          if (check(a, segments)) {
50              cout << "Possible\n";
51              cout << segments.size() << "\n";
52              for (int i = 0; i < segments.size(); i++) {
53                  cout << segments[i] << " ";
54              }
55              cout << "\n";
56              return;
57          }
58      }
59      cout << "Impossible\n";
60  }
61
62  signed main() {
63      ios_base::sync_with_stdio(false);
64      cin.tie(0);
65
66      int T; cin >> T;
67      for (int t = 0; t < T; t++) {
68          cout << "Case #" << t << ": ";
69          solve();
70      }
71  }
```

- Runtime: $\mathcal{O}(n \cdot 2^n)$
- Memory: $\mathcal{O}(n)$, can be optimized to $\mathcal{O}(1)$

Subtask 3: All the walls (29 points)

For this subtask, we observe that if we have a segment separator between two elements a_i and a_{i+1}, all elements from $a_0, a_1, ...a_i$ must be $0, 1, ..., i$. We can adapt this to an algorithm:

We create a DP table `vector<bool> dp(n)`, where `dp[i]` asks whether we can sort every element from $a_0, ..., a_i$ when there is a separator between a_i, a_{i+1}. However, we need to backtrack the solution if it's possible, so we need to store the segmentation up to each point i.

(Solution by Leo Chen)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
```

```
4
5   void solve() {
6       int n; cin >> n;
7       vector<int> a(n);
8       for (int i = 0; i < n; i++) {
9           cin >> a[i];
10          a[i]--; // Convert to 0-indexed
11      }
12
13      // dp[i] stores if a[0...i-1] can be partitioned, and the segments if
        so.
14      vector<pair<bool, vector<int>>> dp(n + 1, {false, {}});
15      dp[0] = {true, {}}; // Base case: empty prefix is solvable.
16
17      for (int end = 1; end <= n; ++end) {
18          // Try to form the last segment for the prefix a[0...end-1]
19          for (int start = 0; start < end; ++start) {
20              // Check if the prefix a[0...start-1] is solvable
21              if (dp[start].first) {
22                  int seg_len = end - start;
23                  bool segment_is_valid = true;
24                  // Check if flipping the segment a[start...end-1] makes
                    it sorted
25                  for (int i = 0; i < seg_len; ++i) {
26                      // The element at the original index (end - 1 - i)
27                      // should have the value (start + i) to be sorted.
28                      if (a[end - 1 - i] != start + i) {
29                          segment_is_valid = false;
30                          break;
31                      }
32                  }
33
34                  if (segment_is_valid) {
35                      // Found a valid partition for a[0...end-1].
36                      dp[end].first = true;
37                      dp[end].second = dp[start].second; // Copy segments
                        from the previous solution
38                      dp[end].second.push_back(seg_len); // Add the new
                        segment
39                      break; // Optimization: found a solution for this
                        endpoint, move to the next.
```

```
40                 }
41             }
42         }
43     }
44
45     if (dp[n].first) {
46         cout << "Possible\n";
47         cout << dp[n].second.size() << "\n";
48         for (int i = 0; i < dp[n].second.size(); ++i) {
49             cout << dp[n].second[i] << (i == dp[n].second.size() - 1 ?
               "" : " ");
50         }
51         cout << "\n";
52     } else {
53         cout << "Impossible\n";
54     }
55 }
56
57 signed main() {
58     ios_base::sync_with_stdio(false);
59     cin.tie(0);
60
61     int T; cin >> T;
62     for (int t = 0; t < T; t++) {
63         cout << "Case #" << t << ": ";
64         solve();
65     }
66 }
```

- Runtime: $\mathcal{O}(n^2)$
- Memory: $\mathcal{O}(n)$

Subtask 4: All the registans (43 points)

Lemma 2. *You only reverse decreasing segments*

Proof. If you would reverse a non-decreasing segment, the resulting segment is not strictly increasing, which is never a solution by definition of this task. □

This is the most crucial observation, as if there are ever two consecutive elements a_i, a_{i+1} such that $a_i < a_{i+1}$, then we know they can't be in the same segment. However, if $a_i > a_{i+1}$, they must be in the same segment. If they wouldn't be in the same segment, it's easy to see that you'll not sort the array using such a segmentation.

Any two consecutive elements fall into one of those relations (either $a_i < a_{i+1}$ or $a_i > a_{i+1}$). Therefore, there is exactly one way of segmenting the entire array.

We create the segmentation in this way, and check whether the resulting array is increasing.

(Solution by Ursus Wigger)

cpp

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

using namespace std;

inline bool valid(vector<int> &a, vector<int> &seg) {
  int i = 0;
  for(auto x : seg) {
    for(int j = 0 ; j < x ; ++j) {
      if(i+j != a[i+x-1-j]) {
        return false;
      }
    }
    i += x;
  }

  return true;
}

inline void solve() {
  int n;
  cin >> n;

  vector<int> a(n);
  for(int i = 0 ; i < n ; ++i) cin >> a[i];

  vector<int> res;

  int cur = 1;
  for(int i = 0 ; i+1 < n ; ++i) {
    if(a[i] < a[i+1]) {
      res.push_back(cur);
      cur = 0;
    }
    cur++;
  }
```

```
36    res.push_back(cur);
37
38    if(!valid(a, res)) {
39      cout << "Impossible\n";
40      return;
41    }
42
43    cout << "Possible\n";
44    cout << res.size() << "\n";
45    for(auto x : res) cout << x << " ";
46    cout << "\n";
47  }
48
49  signed main() {
50    ios_base::sync_with_stdio(0);
51    cin.tie(0);
52
53    int t; cin >> t;
54    for(int i = 0 ; i < t ; ++i) {
55      cout << "Case #" << i << ": ";
56      solve();
57    }
58  }
```

- Runtime: $\mathcal{O}(n)$
- Memory: $\mathcal{O}(n)$, can be optimized to $\mathcal{O}(1)$

Timurid Riders

Task Idea	Johannes Kapfhammer
Task Preparation	Leo Chen
Translation (de)	Hannah Oss
Translation (fr)	Noémie Jacquemot
Solution Author	???

The solution author could not be reached for comment, sorry :).

Subtask 1: Orders of Samarkand (8 points)

Subtask 2: Caravan Trail to Samarkand (9 points)

Subtask 3: Equal Roads of Bukhara (14 points)

Subtask 4: Balanced Roads of Shahrisabz (18 points)

Subtask 5: A Province of the Realm (24 points)

Subtask 6: The Empire of Timur (27 points)

Mouse Dances

Task Idea	Johannes Kapfhammer
Task Preparation	Karolina Alexiou
Translation (en)	Karolina Alexiou
Translation (de)	Linus Lüchinger
Translation (fr)	Noémie Jacquemot
Solution Author	Karolina Alexiou
Further Credits	Johannes Kapfhammer, Ursus Wigger, Cheng Zhong

In this problem, we need to arrange mice with different heights to minimize the maximum height difference between neighbors. The problem has three variants: arranging in a line, in a circle, and in two circles.

Subtask 1: Tiny line (6 points)

For a line arrangement, we want to minimize the maximum difference between consecutive mice.

Since $n \leq 3$, we can solve this subtask by trying all $n!$ permutations of the mice (at most $3! = 6$ arrangements). For each permutation, we compute the maximum consecutive difference, keep track of the best arrangement and output it at the end.

```python
from itertools import permutations

n = int(input())
a = list(map(int, input().split()))

best_max = float('inf')
best_arrangement = []

for perm in permutations(a):
    max_diff = 0
    for i in range(1, len(perm)):
        max_diff = max(max_diff, abs(perm[i] - perm[i-1]))
    if max_diff < best_max:
        best_max = max_diff
        best_arrangement = list(perm)

print(best_max)
print(" ".join(map(str, best_arrangement)))
```

Runtime: $\mathcal{O}(n!)$ for small n. Memory: $\mathcal{O}(n)$ (iff we don't store all the permutations)

Subtask 2: Medium line (12 points)

We can also observe a general principle that works for all line arrangements:

Lemma 1. *The optimal arrangement for a line is to sort the mice by height.*

Proof. Consider any arrangement where mice are not sorted. There must exist adjacent mice a_i and a_j where $a_i > a_j$ but there's a mouse $a_k < a_i$ positioned after a_i, or a mouse $a_l > a_j$ positioned before a_j.

If we rearrange the mice in sorted order, the maximum consecutive difference can only decrease or stay the same, because:

- The sorted order minimizes "jumps" between consecutive elements
- The largest difference in the sorted array is between some a_i and a_{i+1}
- Any non-sorted arrangement would either have this same difference or a larger one

□

For the sorted array $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$, the answer is:

$$\max_{i=1}^{n-1}(a_i - a_{i-1})$$

```python
n = int(input())
a = list(map(int, input().split()))
a.sort()
maxdiff = 0
for i in range(1, len(a)):
    maxdiff = max(maxdiff, a[i] - a[i-1])
print(maxdiff)
print(" ".join(map(str, a)))
```

Runtime: $\mathcal{O}(n \log n)$ for sorting, and then $\mathcal{O}(n)$ for a single pass through the array. Memory: $\mathcal{O}(n)$

Subtask 3: Huge line (15 points)

Same approach as the previous subtask, but we only need to output the maximum difference value, not the arrangement itself.

Runtime: $\mathcal{O}(n \log n)$ for sorting, and then $\mathcal{O}(n)$ for a single pass through the array. Memory: $\mathcal{O}(n)$

Subtask 4: Medium circle (13 points)

For a circle, the last mouse is a neighbor of the first mouse, so we have an additional constraint. Simply using the sorted order would create a large difference between the smallest and largest mouse.

The key insight is to use an "even-odd sort" arrangement:

Lemma 2. *After sorting the mice, place mice at even indices at the beginning of the circle, and mice at odd indices at the end in reverse order.*

More precisely, if we have sorted array $[a_0, a_1, a_2, a_3, a_4, a_5, ...]$, we rearrange it as:

$$[a_0, a_2, a_4, ..., ..., a_5, a_3, a_1]$$

This creates a pattern where heights increase, then decrease, minimizing both the wraparound difference between last and first mouse and all consecutive differences.

Proof. The even-odd arrangement ensures:

- The circle goes: smallest → medium-small → medium-large → ... → largest → ... → medium-large → medium-small → smallest
- The maximum difference occurs between consecutive elements in the sorted array that are 2 positions apart
- The wraparound connects similar-sized mice (first and last in the rearranged order are close in the sorted order)
- This is optimal because any other arrangement would either:
  - Create a larger wraparound difference, or
  - Create larger consecutive differences by placing non-adjacent sorted elements next to each other

□

```python
def even_odd_sort(a):
    b = [-1] * len(a)
    for i in range(len(a)):
        if i % 2 == 0:
            new_i = i >> 1
        else:
            new_i = len(a) - 1 - (i >> 1)
        b[new_i] = a[i]
    return b

def get_maxdiff(a):
    maxdiff = abs(a[0] - a[-1])
    for i in range(1, len(a)):
        maxdiff = max(maxdiff, abs(a[i] - a[i-1]))
    return maxdiff

n = int(input())
a = list(map(int, input().split()))
a.sort()
```

```
a = even_odd_sort(a)
maxdiff = get_maxdiff(a)
print(maxdiff)
print(" ".join(map(str, a)))
```

Runtime: $\mathcal{O}(n \log n)$ for sorting, and then $\mathcal{O}(n)$ for the additional passes through the array. Memory: $\mathcal{O}(n)$

Subtask 5: Huge circle (18 points)

Same approach as Subtask 4, but only output the value.

Runtime: $\mathcal{O}(n \log n)$ for sorting, and then $\mathcal{O}(n)$ for the additional passes through the array. Memory: $\mathcal{O}(n)$

Subtask 6: Two medium circles (14 points)

Now we need to split the mice into two circles. The key observation:

Lemma 3. *The optimal split divides the sorted array into two contiguous parts.*

Proof. Suppose we have an optimal solution where the two circles contain non-contiguous elements from the sorted array. That means there exist mice $a_i < a_j < a_k$ where a_i and a_k are in one circle, but a_j is in the other circle.

Consider swapping a_j with a_k. The circle containing a_i and a_k now contains a_i and a_j, which are closer in value, so its maximum difference cannot increase. Similarly for the other circle. Thus, we can always rearrange in this manner to get a contiguous split without making the solution worse. □

Given an already sorted array, for each possible split point, we apply even-odd sort to both parts and compute their maximum differences:

```python
best = max(a) - min(a)
for i in range(1, n):
    a1 = a[:i]
    a2 = a[i:]
    a1 = even_odd_sort(a1)
    a2 = even_odd_sort(a2)
    cand = max(get_maxdiff(a1), get_maxdiff(a2))
    if cand <= best:
        best = cand
```

Runtime: $\mathcal{O}(n^2)$ since we try $\mathcal{O}(n)$ splits and each split takes $\mathcal{O}(n)$ time. Memory: $\mathcal{O}(n)$

Subtask 7: Two huge circles (22 points)

For the huge version, we need to optimize the $O(n^2)$ solution to $O(n)$.

The key insight is that we can precompute the maximum differences for all possible left and right parts using dynamic programming:

Lemma 4. *For a circle formed from elements $[a_i, a_{i+1}, ..., a_j]$ of the sorted array, after applying even-odd sort, the maximum consecutive difference is:*

$$\max_{k \in \{i+1, i+3, i+5, ...\}} (a_k - a_{k-2})$$

(considering wraparound for the circle)

We can compute two arrays:

- `bestl[i]` = maximum difference for the first circle using elements $[a_0, ..., a_i]$
- `bestr[i]` = maximum difference for the second circle using elements $[a_i, ..., a_{n-1}]$

For `bestl`, we process left to right:

- At position i, the new difference introduced is between a_i and a_{i-2} (in the even-odd arrangement)
- `bestl[i] = max(bestl[i-1], a[i] - a[i-2])` (with appropriate base cases)

For `bestr`, we process right to left similarly.

Then for each split point k, the answer is $\max(\text{bestl}[k], \text{bestr}[k+1])$.

```python
def find_best(a, right_to_left=False):
    res = [0] * len(a)
    rng = range(1, len(a)) if not right_to_left else range(len(a)-2, 0, -1)
    curr_max = 0
    for i in rng:
        if right_to_left:
            if i == len(a) - 2:
                new_diff = a[i+1] - a[i]
            else:
                new_diff = a[i+2] - a[i]
        else:
            if i == 1:
                new_diff = a[i] - a[i-1]
            else:
                new_diff = a[i] - a[i-2]

        curr_max = max(curr_max, new_diff)
        res[i] = curr_max
    return res

```

```
21 a.sort()
22 bestl = find_best(a, right_to_left=False)
23 bestr = find_best(a, right_to_left=True)
24
25 best = max(a) - min(a)
26 for i in range(n - 1):
27     left = bestl[i]
28     right = bestr[i+1]
29     cand = max(left, right)
30     best = min(best, cand)
31
32 print(best)
```

Runtime: $\mathcal{O}(n \log n)$ for sorting, then $\mathcal{O}(n)$ for the two passes and finding the minimum. Memory: $\mathcal{O}(n)$

Fire

Task Idea	Italian Olympiad in Informatics
Task Preparation	Cheng Zhong
Translation (de)	Benjamin Schmid
Translation (fr)	Cheng Zhong
Solution Author	Cheng Zhong
Further Credits	Priska Steinebrunner

We have an $N \times N$ grid with several fire sources that spread outward in Manhattan distance as time passes. Mouse Binna wants to move from $(0, 0)$ to $(N-1, N-1)$. Any cell that is on fire at or before time t cannot be used at time t. We must determine the latest time at which Binna can start such that a completely safe path to the goal still exists.

Subtask 1: Single fire source (12 points)

Here we only have one fire source at coordinates (X, Y). At time t, all cells whose Manhattan distance from (X, Y) is at most t are on fire. Geometrically, that is a diamond-shaped region. As t grows, this diamond gets larger and longer, and eventually it will stretch across the grid in such a way that it completely cuts off all paths from the start to the end.

What we want is the first moment T^* when this single diamond really blocks every possible path from $(0, 0)$ to $(N-1, N-1)$. Then the answer is $L = T^* - 1$: before T^*, Binna can still barely find some way around; at exactly T^*, every possible route is broken.

If you draw a picture and think about how to walk from top-left to bottom-right without being stopped by that diamond, you basically have two "ways around" it. They correspond to two directions in which you can try to squeeze around the fire:

- One way is to go around the "top or right" side of the diamond. The width of that channel is controlled by the minimum of the distances from the fire to the top edge and to the right edge, that is: $\min(X, N-1-Y)$.
- The other way is to go around the "left or bottom" side. That channel's width is controlled by: $\min(Y, N-1-X)$.

As long as at least one of these two channels is still open, there is still a safe path. Only when both of them have been burned through do we really lose all routes. So the critical time T^* is the later of these two "death times":

$$T^* = \max(\min(X, N-1-Y), \min(Y, N-1-X))$$

The final answer is

- $L = T^* - 1$ if this is at least 0,
- otherwise, if even time 0 does not work, $L = -1$.

In code, we may just compute these two mins, take their max as T^*, and then output $T^* - 1$ with the usual check; there is no need for any BFS here.

cpp

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

const int M_MAX = 20000;

int solve_subtask1(int N, int M, int X[], int Y[]) {
    (void) M; // unused in this subtask: there is exactly one fire

    int x = X[0];
    int y = Y[0];

    int option1 = min(x,       N - 1 - y);
    int option2 = min(y,       N - 1 - x);
    int best    = max(option1, option2);

    return best - 1;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int Tcase;
    if (!(cin >> Tcase)) return 0;

    static int X[M_MAX + 2], Y[M_MAX + 2];

    for (int test_case = 0; test_case < Tcase; ++test_case) {
        int N, M;
        if (!(cin >> N >> M)) break;

        for (int i = 0; i < M; ++i) {
            cin >> X[i] >> Y[i];
        }

        int ans = solve_subtask1(N, M, X, Y);
        cout << "Case #" << test_case << ": " << ans << "\n";
    }
```

```
39      return 0;
40  }
```

Subtask 2: Aligned fire sources (19 points)

Now there are many fire sources, but they are all on the same row $x = x_0$. You can imagine that whole row as a line of fire. As time goes on, this line burns upward and downward, forming a thicker and thicker horizontal band of fire. The mouse starts at the top-left, wants to reach the bottom-right, and can try to bypass this band either above it, below it, or through some gap where there is temporarily no fire between two fire sources.

If we sort all the fire sources by their column coordinate Y_i, and remove duplicates, we can think about three kinds of possible "channels":

1. A channel below all fire. Its vertical room depends on how far the fire row x_0 is from the bottom edge; its effective vertical height is some value we can call bottom_space. Horizontally, this channel is also limited by how close the nearest fire source is to the boundary on the left or right. That gives a bottleneck time we can call K_1.
2. A channel above all fire. Symmetrically, the vertical room depends on how far x_0 is from the top edge, and the horizontal room depends on the closest fire to the top side of the grid. This gives another bottleneck time K_2.
3. The best gap between two neighboring fire sources. Between two fire sources whose column coordinates differ by some gap, there is a column interval that is not on fire initially. The larger this gap, the wider this "window" in the middle. The largest such gap, say max_gap, can be used as a middle channel. Fire from the two sides moves toward each other, and after roughly half of this gap (rounded appropriately) the gap is fully closed. So the horizontal part of the bottleneck is about half_gap = max_gap / 2. But to use that gap, you must also have enough vertical space above or below the fire band to reach it; that vertical space is limited by the distance from x_0 to the nearest horizontal boundary. So this channel's bottleneck time is $K_3 = \min(\text{vertical\_space}, \text{half\_gap})$.

Among these three candidates, the time T^* that we can hold out until is the maximum: $T^* = \max(K_1, K_2, K_3)$.

Intuitively, we have three different escape routes. Each one will eventually be burned shut, and K_1, K_2, K_3 are the times when that happens for each route. We choose the route that survives the longest; the moment that route dies is the first moment when the grid is completely cut. The answer is again $L = T^* - 1$. If this is negative, we output -1.

cpp
```
#include <bits/stdc++.h>
using namespace std;
using int64 = long long;

int main() {
```

```
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    if (!(cin >> T)) return 0;

    for (int tc = 0; tc < T; ++tc) {
        int64 N; int M;
        cin >> N >> M;

        vector<int64> X(M), Y(M);
        for (int i = 0; i < M; ++i) cin >> X[i] >> Y[i];

        // Subtask: all x_i are equal
        int64 x0 = (M ? X[0] : 0);

        int64 ans = 0;
        if (M == 0) {
            // No fire: can wait arbitrarily long.
            // Here we output 0, but it could be replaced by a large number if required.
            ans = 0;
        } else {
            sort(Y.begin(), Y.end());
            // Remove duplicates (not strictly necessary)
            Y.erase(unique(Y.begin(), Y.end()), Y.end());
            M = (int)Y.size();

            // Distance from bottom edge (y=0) to the nearest fire
            int64 B0   = llabs(0      - Y.front());       // = Y.front()
            // Distance from top edge (y=N-1) to the nearest fire
            int64 BN1  = llabs((N-1)  - Y.back());        // = (N-1) - Y.back()

            // Maximum gap between two consecutive fires (in sorted Y)
            int64 max_gap = 0;
            for (int i = 0; i + 1 < M; ++i) {
                max_gap = max<int64>(max_gap, Y[i+1] - Y[i]);
            }
            int64 half_gap = max_gap / 2; // usable clearance inside the largest gap
```

```
44
45              // Horizontal clearances from x0 to both sides
46              int64 left_space  = N - 1 - x0;
47              int64 right_space = x0;
48              int64 mid_space   = min(left_space, right_space);
49
50              // Three candidate bottlenecks:
51              int64 K1 = min(left_space,  B0);
52              int64 K2 = min(right_space, BN1);
53              int64 K3 = min(mid_space,   half_gap);
54
55              int64 K = max({K1, K2, K3});
56              ans = K - 1;
57          }
58
59          cout << "Case #" << tc << ": " << ans << "\n";
60      }
61      return 0;
62  }
```

Subtask 3: Small grid (21 points)

In this subtask the grid is small enough that we can simulate everything directly. The basic algorithm is: given a time t, treat all currently burning cells as blocked, and check whether Binna can still go from $(0, 0)$ to $(N - 1, N - 1)$ using only non-burning cells. Then increase t by 1 by expanding the fire one step, and repeat until the check fails.

Concretely, we maintain the set of burning cells at the current time. At time 0, exactly the fire sources are burning. Each time step, fire spreads to the four neighbors of every burning cell, so the burning region grows by one "layer" in Manhattan distance.

After updating the burning cells for the current time, we run a standard BFS on the grid to test reachability. Burning cells are treated as walls; all other cells are traversable. If $(0, 0)$ or $(N - 1, N - 1)$ is burning, the answer is immediately impossible for that time. Otherwise, BFS determines whether a safe path exists. This reachability test corresponds to the function reach() in the code.

As long as the BFS succeeds, time t is still feasible, so we advance the fire by one step and try again. The first time the BFS fails, it means the current time is already too late. Therefore, if the first failing time is t, the largest feasible start time is $t - 1$ (equivalently, if we count how many successful expansions we performed, the answer is one less than that count).

This approach may do multiple BFS runs, but for $N \leq 500$ it is still acceptable: the fire cannot expand more than $O(N)$ steps before the whole grid burns, and each BFS is $O(N^2)$.

cpp

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

const int M_MAX = 20000;

int N;
int sol = 0;
vector<vector<bool>> G;      // burning cells
queue<pair<int,int>> F;      // current fire frontier

bool reach() {
    if (G[0][0] || G[N - 1][N - 1]) return false;

    vector<vector<bool>> seen(N, vector<bool>(N, false));
    queue<array<int, 2>> q;
    q.push({0, 0});

    while (!q.empty() && !seen[N - 1][N - 1]) {
        auto f = q.front();
        int x = f[0];
        int y = f[1];
        q.pop();

        if (seen[x][y]) continue;
        seen[x][y] = true;

        if (x < N - 1 && !G[x + 1][y] && !seen[x + 1][y])
            q.push({x + 1, y});
        if (y < N - 1 && !G[x][y + 1] && !seen[x][y + 1])
            q.push({x, y + 1});
        if (x > 0     && !G[x - 1][y] && !seen[x - 1][y])
            q.push({x - 1, y});
        if (y > 0     && !G[x][y - 1] && !seen[x][y - 1])
            q.push({x, y - 1});
    }

    return seen[N - 1][N - 1];
}

void step() {
```

```
41      auto remaining = F.size();
42      ++sol;
43
44      while (remaining > 0) {
45          auto f = F.front();
46          int x = f.first;
47          int y = f.second;
48
49          F.pop();
50          --remaining;
51
52          if (x > 0 && !G[x - 1][y]) {
53              F.emplace(x - 1, y);
54              G[x - 1][y] = true;
55          }
56          if (y > 0 && !G[x][y - 1]) {
57              F.emplace(x, y - 1);
58              G[x][y - 1] = true;
59          }
60          if (x < N - 1 && !G[x + 1][y]) {
61              F.emplace(x + 1, y);
62              G[x + 1][y] = true;
63          }
64          if (y < N - 1 && !G[x][y + 1]) {
65              F.emplace(x, y + 1);
66              G[x][y + 1] = true;
67          }
68      }
69  }
70
71  int solve_subtask3(int n, int m, int X[], int Y[]) {
72      N = n;
73      G.assign(N, vector<bool>(N, false));
74
75      for (int i = 0; i < m; ++i) G[X[i]][Y[i]] = true;
76
77      while (!F.empty()) F.pop();
78      for (int i = 0; i < m; ++i) F.emplace(X[i], Y[i]);
79
80      while (reach()) step();
```

```
81
82      return sol - 1;
83  }
84
85  int main() {
86      ios::sync_with_stdio(false);
87      cin.tie(nullptr);
88
89      int Tcase;
90      if (!(cin >> Tcase)) return 0;
91
92      static int X[M_MAX + 2], Y[M_MAX + 2];
93
94      for (int test_case = 0; test_case < Tcase; ++test_case) {
95          int n, m;
96          cin >> n >> m;
97
98          for (int i = 0; i < m; ++i) {
99              cin >> X[i] >> Y[i];
100         }
101
102         sol = 0;
103         G.clear();
104         while (!F.empty()) F.pop();
105
106         int ans = solve_subtask3(n, m, X, Y);
107         cout << "Case #" << test_case << ": " << ans << "\n";
108     }
109
110     return 0;
111 }
```

Subtask 4: Few fire sources (22 points)

In this subtask, the grid side length N can be very large (up to something like 10^9), but the number of fire sources M is at most 100. This makes it impossible to do any BFS over the whole grid, but very attractive to ignore the grid and focus only on the fire sources plus the boundaries.

The basic idea will be very close to the full solution: we consider each fire source as a node in a graph, and add two special nodes:

- LB: the union of the left & bottom boundary
- RT: the union of the right & top boundary

In total we have $M + 2$ nodes. After introducing LB and RT, our goal becomes to find the earliest time when there exists a continuous "firewall chain" connecting LB to RT; once such a chain exists, every safe path from $(0, 0)$ to $(N - 1, N - 1)$ is cut off.

Between any two nodes i and j we imagine there is an edge with weight $w(i, j)$. This weight is defined as the earliest time when fire coming from the two sides represented by i and j is enough to build a complete firewall between them, so that the "gap" between those two regions is fully blocked. For two fire sources, this can be computed from their Manhattan distance. For a fire source and LB or RT, it is some minimum distance from the fire source to the relevant boundaries. In all cases the formulas can be derived from geometry; conceptually, $w(i, j)$ is just "how long until the passage between i and j is completely burned shut".

Directly, these $w(i, j)$ values give us the "blocking time" if we only use that one edge. But of course we can have more complex blocking routes that go through intermediate fire sources. For example, a route from LB to RT via some fire source k and then another fire source j has a blocking time equal to $\max(w(LB, k), w(k, j))$ because both segments must be burned through to make the whole route a wall. There can be many such routes, so we want, for any pair (i, j), the minimal possible value of "maximum edge time" over all paths between i and j.

This is exactly a classic "minimize the maximum edge on the path" problem, and we can solve it with a Floyd–Warshall style triple loop. We maintain a matrix $\text{bott}[i][j]$, initially set to $w(i, j)$ when there is a direct relationship, or to a very large number otherwise. Then we repeatedly try to improve $\text{bott}[i][j]$ by going through an intermediate node k:

$$\text{bott}[i][j] = \min(\text{bott}[i][j], \max(\text{bott}[i][k], \text{bott}[k][j])).$$

In words: if going from i to j via k means the worst edge we see is no more than the worst edge among "i to k" and "k to j", then that gives a candidate bottleneck for i to j, and we keep the smallest possible one.

After the triple loop finishes, $\text{bott}[LB][RT]$ represents T^*, the earliest time when some route from LB to RT becomes completely burned, i.e., the earliest time when a full firewall can separate start from end. If this T^* is 0, it means the grid is already blocked at the beginning and the answer is -1. Otherwise, the answer is $L = T^* - 1$.

Here $M \leq 100$, so the number of nodes $M + 2$ is at most 102, and the Floyd–Warshall style algorithm runs in time proportional to $(M + 2)^3$, which is perfectly fine. In this way, we solve the connectivity of a huge grid using only a small graph built from the fire sources and two special boundary nodes.

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

```

```cpp
static inline ll touch_pair_xy(ll x1, ll y1, ll x2, ll y2) {
    ll dx = llabs(x1 - x2);
    ll dy = llabs(y1 - y2);
    ll need = max(0LL, dx - 1) + max(0LL, dy - 1);
    return (need + 1) / 2; // ceil(need/2)
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    if (!(cin >> T)) return 0;
    for (int tc = 0; tc < T; ++tc) {
        ll N; int M;
        cin >> N >> M;
        vector<pair<ll,ll>> pts(M);
        for (int i = 0; i < M; ++i) cin >> pts[i].first >> pts[i].second;

        const int LB = M, RT = M+1, V = M + 2;
        const ll INF = (1LL<<60);
        vector<vector<ll>> bott(V, vector<ll>(V, INF));
        for (int i = 0; i < V; ++i) bott[i][i] = 0;

        // fire-fire
        for (int i = 0; i < M; ++i)
            for (int j = i+1; j < M; ++j) {
                ll w = touch_pair_xy(pts[i].first, pts[i].second,
                                     pts[j].first, pts[j].second);
                bott[i][j] = bott[j][i] = min(bott[i][j], w);
            }

        // fire-arc (LB = LeftuBottom, RT = RightuTop)
        for (int i = 0; i < M; ++i) {
            ll x = pts[i].first, y = pts[i].second;
            ll wLB = min(y, (N-1) - x);      // reach y=0 or x=N-1
            ll wRT = min((N-1) - y, x);      // reach y=N-1 or x=0
            bott[i][LB] = bott[LB][i] = min(bott[i][LB], wLB);
            bott[i][RT] = bott[RT][i] = min(bott[i][RT], wRT);
        }
```

```
45
46          // Floyd-Warshall for minimum bottleneck path
47          for (int k = 0; k < V; ++k)
48              for (int i = 0; i < V; ++i)
49                  for (int j = 0; j < V; ++j)
50                      bott[i][j] = min(bott[i][j], max(bott[i][k], bott[k]
                        [j]));
51
52          ll Tstar = bott[LB][RT];
53          ll ans = (Tstar == 0 ? -1 : Tstar - 1);
54
55          cout << "Case #" << tc << ": " << ans << "\n";
56      }
57      return 0;
58  }
```

Subtask 5: More fire sources (26 points)

In this final subtask, both the grid size N and the number of fire sources M can be large, so neither a grid-based BFS nor the $O(M^3)$ Floyd–Warshall approach from the previous subtask is fast enough. However, the model we use is exactly the same as before:

- Each fire source is a node in a graph.
- We add two special nodes LB and RT representing the "start side" and the "end side" of the grid.
- In total there are $M + 2$ nodes.
- For any two nodes i and j, we define an edge weight $w(i, j)$ as the earliest time when fire from the regions represented by i and j can cooperate to completely block the passage between them.

The geometric meaning of $w(i, j)$ and the way we compute it (for fire–fire pairs and for fire–boundary pairs) are the same as in the previous subtask, just written with slightly different closed-form expressions in code.

As before, any path from LB to RT becomes a solid firewall at the time given by the maximum $w(i, j)$ along that path. Among all such paths, the earliest one to be fully burned is the path whose largest edge weight is as small as possible. If that optimal value is T, then the mouse must start no later than $T - 1$.

The only real difference in this subtask is how we compute that optimal bottleneck value between LB and RT. Instead of running Floyd–Warshall on the $(M + 2) \times (M + 2)$ matrix, we run a Dijkstra-like minimax algorithm on the same graph:

- We keep an array $D[i]$ where $D[i]$ means: over all known paths from LB to node i, the path whose maximum edge weight is smallest has value $D[i]$.

- Initially, $D[LB] = 0$ and all other $D[i]$ are set to a large value (for example N).
- Repeatedly, we pick the unvisited node u with the smallest $D[u]$ and "finalize" it.
- For every node v, we consider using the edge (u, v) to improve $D[v]$. If we go from LB to v via u, the maximum edge weight on that path is $\max(D[u], w(u, v))$. So we update $D[v] = \min(D[v], \max(D[u], w(u, v)))$.

This is the standard "minimize the maximum edge on the path" variant of Dijkstra's algorithm. It runs in $O(M^2)$ with a simple array-based implementation, which is fine for M up to the limits of the full task.

At the end of this process, $D[RT]$ gives the critical time T at which some path from LB to R T first becomes completely burned. As in previous subtasks, the answer we output is $T - 1$: if T is the earliest time at which all routes are blocked, then $T - 1$ is the last time when the mouse can still start and find a safe path.

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

const int M_MAX = 20000;

bool used[M_MAX + 2];     // node already finalized
int  dist_lb[M_MAX + 2];  // minimax distance from LB

int solve_subtask5(int N, int M, int X[], int Y[]) {
    const int LB = M;
    const int RT = M + 1;

    auto compute_dist = [&](int i, int j) -> int {
        // Edge between LB and RT
        if ((i == LB || j == LB) && (i == RT || j == RT)) return N;

        // Edge involving LB and a fire
        if (i == LB || j == LB) {
            int idx = i + j - LB;
            return min(X[idx], N - 1 - Y[idx]);
        }

        // Edge involving RT and a fire
        if (i == RT || j == RT) {
            int idx = i + j - RT;
            return min(Y[idx], N - 1 - X[idx]);
        }

```

```
29          // Edge between two fires
30          int dx = abs(X[i] - X[j]);
31          int dy = abs(Y[i] - Y[j]);
32          if (dx == 0 || dy == 0) {
33              return (dx + dy) / 2;
34          } else {
35              return (dx + dy - 1) / 2;
36          }
37      };
38
39      for (int i = 0; i < M + 2; ++i) {
40          dist_lb[i] = N;
41      }
42      dist_lb[LB] = 0;
43
44      // Minimax Dijkstra
45      while (true) {
46          int best = -1;
47          for (int j = 0; j < M + 2; ++j) {
48              if (!used[j] && (best == -1 || dist_lb[j] < dist_lb[best]))
                    best = j;
49          }
50          if (best == -1) break;
51          used[best] = true;
52
53          for (int j = 0; j < M + 2; ++j) {
54              int cand = max(dist_lb[best], compute_dist(best, j));
55              dist_lb[j] = min(dist_lb[j], cand);
56          }
57      }
58
59      return dist_lb[RT] - 1;
60  }
61
62  int main() {
63      ios::sync_with_stdio(false);
64      cin.tie(nullptr);
65
66      int Tcase;
67      if (!(cin >> Tcase)) return 0;
68
```

```
69      static int X[M_MAX + 2], Y[M_MAX + 2];
70
71      for (int test_case = 0; test_case < Tcase; ++test_case) {
72          int N, M;
73          cin >> N >> M;
74
75          for (int i = 0; i < M; ++i) {
76              cin >> X[i] >> Y[i];
77          }
78
79          fill(used, used + M + 2, false);
80
81          int ans = solve_subtask5(N, M, X, Y);
82          cout << "Case #" << test_case << ": " << ans << "\n";
83      }
84
85      return 0;
86  }
```

Alternatively, we can solve this subtask using a Union–Find (disjoint-set) based approach. The modeling of the problem remains exactly the same as before: we still treat every fire source as a node, and we add two special nodes LB and RT representing the start-side and end-side regions of the grid. Altogether we have $M + 2$ nodes, and for every pair of nodes (i, j) we define an edge weight $w(i, j)$, the earliest time when fire spreading from i and j can fully seal the passage between them. The geometric formulas for $w(i, j)$—for fire–fire pairs and for fire–boundary pairs—are identical to those described in the earlier subtasks.

As in the minimax-path interpretation used before, any path from LB to RT becomes a complete firewall precisely when all edges on that path have been "activated," meaning the time has reached at least their $w(i, j)$. Thus, the time when a particular path becomes unusable is the maximum w along that path. We again seek the path whose maximum edge weight is minimized; call that optimal value T. The mouse must start no later than $T - 1$.

The Union–Find viewpoint corresponds to imagining time increasing from 0 upward. As soon as t reaches $w(i, j)$, the "gap" between i and j is burned shut, so the firewall structure now connects i and j. If we sort all edges by their w and process them from smallest to largest, then at time t we unite exactly those pairs (i, j) whose burning threshold $w(i, j)$ is at most t.

More concretely:

- Create all $M + 2$ nodes (the M fires plus LB and RT).
- For every pair (i, j), compute $w(i, j)$ using the same geometric formulas as before, and store the edge (i, j, w) in a list.
- Sort all edges by increasing w.

- Initialize a Union–Find structure so each node starts in its own set.
- Traverse edges from small to large:
  - For an edge (i, j, w), if i and j are in different sets, unite them.
  - After uniting, check whether LB and RT now belong to the same set.
    - If not, continue to the next edge.
    - If this is the first time they become connected, the current edge weight w is exactly the critical value T: T is the minimum, over all paths P, of the maximum $w(e)$ over all edges e on P.

The final answer is $T - 1$. As before, if in theory $T = 0$ were to occur, it would mean the grid is already blocked at time 0, and the correct output would be -1.

In terms of complexity, we have $M + 2$ nodes and about $O(M^2)$ edges. Sorting the edges costs $O(M^2 \log M)$, and each Union–Find operation is effectively constant time. This is more than efficient enough for the largest constraints, and conceptually very natural: it directly simulates the firewall forming gradually as time increases, and detects the precise moment when LB and RT become connected by burned segments.

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

struct DSU {
    vector<int> p, sz;
    DSU(int n=0){ init(n); }
    void init(int n){
        p.resize(n); sz.assign(n,1);
        iota(p.begin(), p.end(), 0);
    }
    int find(int x){ return p[x]==x? x : p[x]=find(p[x]); }
    bool unite(int a, int b){
        a = find(a); b = find(b);
        if(a==b) return false;
        if(sz[a] < sz[b]) swap(a,b);
        p[b]=a; sz[a]+=sz[b];
        return true;
    }
    bool same(int a, int b){ return find(a)==find(b); }
};

struct Edge {
    int u, v;
    long long w;
    bool operator<(const Edge& other) const {
```

```
26          if (w != other.w) return w < other.w;
27          if (u != other.u) return u < other.u;
28          return v < other.v;
29      }
30  };
31
32  static inline long long ceil_div2(long long s) {
33      // ceil(s/2) for non-negative s
34      return (s + 1) / 2;
35  }
36
37  int main(){
38      ios::sync_with_stdio(false);
39      cin.tie(nullptr);
40
41      int T;
42      if(!(cin >> T)) return 0;
43      for(int tc=0; tc<T; ++tc){
44          long long N; int M;
45          cin >> N >> M;
46          vector<long long> x(M), y(M);
47          for(int i=0;i<M;++i){
48              cin >> x[i] >> y[i];
49          }
50
51          const int A = M;       // Super node A (left border ∪ bottom
            border)
52          const int B = M + 1;   // Super node B (right border ∪ top
            border)
53          vector<Edge> edges;
54          edges.reserve((long long)M*(M-1)/2 + 2LL*M + 5);
55
56          // Fire source - fire source edges (earliest 8-neighbor contact
            time)
57          for(int i=0;i<M;++i){
58              for(int j=i+1;j<M;++j){
59                  long long dx = llabs(x[i]-x[j]);
60                  long long dy = llabs(y[i]-y[j]);
61                  long long need_x = (dx>1? dx-1: 0);
62                  long long need_y = (dy>1? dy-1: 0);
63                  long long w = ceil_div2(need_x + need_y);
```

```
64                  edges.push_back({i,j,w});
65              }
66          }
67
68          // Fire source - boundary super nodes
69          // (must burn to the boundary cell itself, i.e. 4-neighbor)
70          for(int i=0;i<M;++i){
71              // left column (y=0) or bottom row (x=N-1)
72              long long wA = min(y[i], (N-1) - x[i]);
73              // top row (x=0) or right column (y=N-1)
74              long long wB = min(x[i], (N-1) - y[i]);
75              edges.push_back({i, A, wA});
76              edges.push_back({i, B, wB});
77          }
78
79          sort(edges.begin(), edges.end());
80
81          DSU dsu(M+2);
82          long long Tblock = 0;
83          for(const auto& e : edges){
84              dsu.unite(e.u, e.v);
85              if(dsu.same(A,B)){
86                  Tblock = e.w;
87                  break;
88              }
89          }
90          long long ans = Tblock - 1;
91          cout << "Case #" << tc << ": " << ans << "\n";
92      }
93      return 0;
94  }
```

Rob or Buy

Task Idea	Ursus Wigger, Linus Lüchinger
Task Preparation	Ursus Wigger
Translation (de)	Ursus Wigger
Translation (fr)	Noémie Jacquemot
Solution Author	Linus Lüchinger

Subtask 1: The road between Samarkand and Bukhara (10 points)

The key to solving this subtask is recognizing the pattern of which merchants are robbed and from which the item is bought in the optimal solution. Firstly, we can realize that if there is a merchant at index i which alarms to the left and we rob that merchant, we cannot rob any merchant that alarms to the right at index j with $j < i$. The reason is that there would be no order in which not either of the merchants alarms the other and robbing them becomes impossible. Thus, the merchants which we rob must all alarm to the left up to some index s and after that index all alarm to the right. Additionally, if we rob a merchant that alarms to the right at position i, it is always optimal to rob all merchants at positions $j > i$ that also alarm to the right. This can be proven by contradiction, however it is intuitive and will thus not be proven here.

Thus, the problem consists of finding the optimal s such that robbing all merchants alarming to the left with $i < s$ and robbing all merchants alarming to the right with $i \geq s$ results in the lowest cost.

This can be done be simply checking all potential s.

Subtask 2: The road across Uzbekistan (15 points)

This subtask is similar to the first subtask. We just need to make sure that we check the s cleverly. For example, we can start by checking $s = 0$, and increment s one by one, where in each step we only have to do a minimal amount of work. We then again take the answer for the best s as our final answer.

An alternative approach which will allow for generalization for later subtasks is to use dynamic programming. We are still looking for solutions of the form where we rob all left alarming merchants in some prefix and then rob all right alarming merchants in the remaining suffix. We can do this with the following dp states:

$\mathrm{dp}[i][b] :=$ answer for the prefix $0 \ldots i$ with an additional boolean dimension b.

b represents whether we are still in the prefix where we rob every left alarming merchant. Meaning $\mathrm{dp}[i][\mathrm{false}]$ contains the answer to the full problem if one only considers merchants 0 to i, while $\mathrm{dp}[i][\mathrm{true}]$ contains the answer if one is not allowed to rob any right alarming

merchants only considering merchants 0 to i. As we established, for $b = \text{true}$, we rob every single left alarming merchant.

Thus the transitions for this dp are:

$$\text{dp}[i][b] = \begin{cases} \text{dp}[i-1][\text{true}] & \text{if } s[i]=\text{l and } b=\text{true} \\ \text{dp}[i-1][\text{true}]+p[i] & \text{if } s[i]=\text{r and } b=\text{true} \\ \min(\text{dp}[i-1][\text{false}]+p[i],\ \text{dp}[i-1][\text{true}]) & \text{if } s[i]=\text{l and } b=\text{false} \\ \text{dp}[i-1][\text{false}] & \text{if } s[i]=\text{r and } b=\text{false} \end{cases}$$

where s is the string of l and r and p is the array containing the costs of buying from the merchants.

We have the trivial base cases:

$\text{dp}[-1][\text{false}] = 0$

$\text{dp}[-1][\text{true}] = 0$

And the final answer is in:

$\text{dp}[N-1][\text{false}]$

Subtask 3: Interrupting the alarms (10 points)

This subtask is of course solved by the solutions to subtask 4 and 5. There is also a solution specific to this subtask:

The key insight is that if we do not use a mercenary, every time there is an l to the right of an r, we need to buy the item from either of those merchants costing at least 1 mouse coin. However, since buying mercenaries is so cheap, it is always at least as good to instead place a mercenary in between the two merchants and rob them both, saving on buying from them. Thus, the optimal solution robs every single merchant and places a mercenary at every position where there is an r followed by an l in the string s.

Subtask 4: The mercenaries need money (25 points)

To solve this and the last subtask, we can incorporate the possibility of buying mercenaries in the dynamic programming solution from before. Specifically, buying a mercenary allows for going from $\text{dp}[i-1][\text{false}]$ to $\text{dp}[i][\text{true}]$ at an additional cost of W. This option might be better than $\text{dp}[i-1][\text{true}] + p[i]$ so we need to incorporate it there into the transitions:

$$\text{dp}[i][b] = \begin{cases} \text{dp}[i-1][\text{true}] & \text{if } s[i]=\text{l and } b=\text{true} \\ \min(\text{dp}[i-1][\text{true}]+p[i],\ \text{dp}[i-1][\text{false}]+W) & \text{if } s[i]=\text{r and } b=\text{true} \\ \min(\text{dp}[i-1][\text{false}]+p[i],\ \text{dp}[i-1][\text{true}]) & \text{if } s[i]=\text{l and } b=\text{false} \\ \text{dp}[i-1][\text{false}] & \text{if } s[i]=\text{r and } b=\text{false} \end{cases}$$

Code for this dp solution is:

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

#define int long long
```

```
4
5   using namespace std;
6
7   signed main() {
8     ios_base::sync_with_stdio(false);
9     cin.tie(0);
10    int t;
11    cin >> t;
12    for (int j = 0; j < t; ++j) {
13      int n, w;
14      cin >> n >> w;
15      string s;
16      cin >> s;
17      vector<int> c(n);
18      for (int &ci : c)
19        cin >> ci;
20      // We do not need to store the entire dp table but the last column is
21      // enough. We store dp[i-1][true] in no_rob_right and dp[i-1][false]
        in
22      // rob_right.
23      int rob_right = 0, no_rob_right = 0;
24      for (int i = 0; i < n; ++i) {
25        if (s[i] == 'r') {
26          no_rob_right += c[i];
27        } else {
28          rob_right += c[i];
29        }
30        rob_right = min(rob_right, no_rob_right);
31        no_rob_right = min(no_rob_right, rob_right + w);
32      }
33      cout << "Case #" << j << ": " << rob_right << "\n";
34    }
35
36    return 0;
37  }
```

The time complexity of this algorithm is $\mathcal{O}(n)$ while the memory complexity excluding input is $\mathcal{O}(1)$.

Since the limits allow for quadratic solutions, different dynamic programming states which result in $\mathcal{O}(n)$ transitions would also work to solve this subtask. For example:

$\text{dp}[i] :=$ solution to the full problem for merchants $0 \ldots i$

For these states we need to do transitions by considering placing a mercenary after every merchant j with $j < i$, then for a possible mercenary position we solve $j + 1 \ldots i$ using a strategy similar to subtask 2 and for $0 \ldots j$ we just take $\mathrm{dp}[j]$.

Subtask 5: The ultimate heist (40 points)

The $\mathcal{O}(n)$ dynamic programming solution proposed for subtask 4 is already fast enough for this subtask.

Emirate of Bukhara

Task Idea	Charlotte Knierim, Luca Versari
Task Preparation	Charlotte Knierim, Luca Versari
Translation (de)	who knows...
Translation (fr)	Noémie Jacquemot
Solution Author	Ursus Wigger
Further Credits	Théo von Düring

Subtask 1: Reversing the boxes (7 points)

The main idea is to always swap the first $K = 1000$ elements with the last 1000, which can be reversed on the fly, because we're carrying the elements in RAM. After we swapped the "*outer*" elements, we continue doing this for the "*inner*" elements.

In practice, we need some additional RAM cells to store the current prefix and suffix that we already reversed, maybe also to some cells for swapping variables. So, the actual number $K \approx 995$

Per iteration, we store the first K elements into RAM, go over to the last K elements, swap them out, then go back and insert the last elements of the first K elements. We can reverse the array in RAM, so we do not need additional head moves for that.

Thus, the number of head moves can be approximated for $m = \frac{N}{2K}$ by:

$$(2N + K) + (2(N - 2K) + K) + ... + (2(N - 2m \cdot K) + K)$$
$$\in \mathcal{O}\left(\frac{N^2}{K}\right)$$

Which is sufficient for $N = 40'000, K = 995$

1.1. XOR swap

To store precious RAM cells, you can do the **XOR-swap**. The C++ equivalent code is as follows:

```cpp
void swap(int *a, int *b) {
    *a = *a ^ *b; // a <- a^b
    *b = *a ^ *b; // b <- (a^b) ^ b = a
    *a = *a ^ *b; // a <- (a^b) ^ a = b
}
```

Using this, we don't need to rely on a 2nd RAM cell that stores a temporary value.

Subtask 2: Sorting the boxes (15 points)

In this task, there would've been multiple sorting algorithm you could choose from. The master solution chose a modified quick sort algorithm.

We mark the pivots by using the highest bit of the cell, which is unused because every element has a value between $0 \leq b_i \leq 2^{63} - 1$

To save more head movement, if we detect that a certain range is smaller than the size of RAM, we can move in the entire range into RAM and sort it there and copy it back on the tape.

(Solution by Luca Versari)

```cpp
#include <algorithm>
#include <bits/stdc++.h>
#include <cassert>
#include <cmath>
#include <cstdio>

using namespace std;

#define all(x) x.begin(), x.end()
#define sz(x) (int)x.size()

//--------CODE--------//

const int MOD = 1e9 + 7;

#define MOVE(k) cout << "MOVE " << k << "\n";
#define READ "READ"
#define WRITE(k) cout << "WRITE " << k << "\n"
#define INTERNAL_RAMREAD(i) "RAMREAD " << i
#define INTERNAL_RAMWRITE(i, v) cout << "RAMWRITE " << i << " " << v << "\n"
#define IF(c, BODY)
\

  {                                                                    \
    cout << "IF " << c << "\n";
    \
    BODY;
    \
    cout << "END IF\n";
    \
  }
```

```
27  #define WHILE(c, BODY)
    \

28    {                                                                 \
        cout << "WHILE " << c << "\n";
29      \
        BODY;
30      \
        cout << "END WHILE\n";
31      \
      }
32  #define ADD(a, b) "+ " << a << " " << b
33  #define SUB(a, b) "- " << a << " " << b
34  #define MUL(a, b) "* " << a << " " << b
35  #define DIV(a, b) "/ " << a << " " << b
36  #define MOD(a, b) "MOD " << a << " " << b
37  #define BITAND(a, b) "& " << a << " " << b
38  #define BITOR(a, b) "| " << a << " " << b
39  #define XOR(a, b) "^ " << a << " " << b
40  #define LESS(a, b) "< " << a << " " << b
41  #define GREAT(a, b) "> " << a << " " << b
42  #define LESSEQ(a, b) "<= " << a << " " << b
43  #define GREATEQ(a, b) ">= " << a << " " << b
44  #define EQ(a, b) "== " << a << " " << b
45  #define DIFF(a, b) "!= " << a << " " << b
46  #define NOT(b) "! " << b
47  #define AND(a, b) "&& " << a << " " << b
48  #define OR(a, b) "|| " << a << " " << b
49  #define TOINT(b) "INT " << b
50  #define OUT(b) cout << "OUT " << b << "\n"
51  #define COLOUT(u, c) cout << "COLOUT " << u << " " << c << "\n";
52  #define READCELL(CELL) INTERNAL_RAMREAD(ADD(CELL, NUM_VARS))
53  #define WRITECELL(CELL, VAL) INTERNAL_RAMWRITE(ADD(CELL, NUM_VARS), VAL)
54  #define SVAR(VAR, VAL) INTERNAL_RAMWRITE(VAR_##VAR, VAL)
55  #define VAR(VAR) INTERNAL_RAMREAD(VAR_##VAR)
56  #define INCR(V, VAL) SVAR(V, ADD(VAR(V), VAL))
57  #define ASSERT_TAPE_HEAD(VAL)
    \

58    {                                                                 \
        MOVE(SUB(0, VAL));
59      \
```

```
61        MOVE(VAL);
          \
62        MOVE(SUB(n, VAL));
          \
63        MOVE(SUB(VAL, n));
          \
64      }
65
66    #define CGAP INTERNAL_RAMREAD(SUB(RAM - 1, VAR(GAP)))
67
68    enum {
69      VAR_SORTED_PREFIX = 0,
70      VAR_PIVOT,
71      VAR_TO_STORE = VAR_PIVOT,
72      VAR_PIVOT_POSITION,
73      VAR_PIVOT_FOUND,
74      VAR_SHORT_SEGMENT = VAR_PIVOT_FOUND,
75      VAR_NEW_PIVOT_POSITION = VAR_SHORT_SEGMENT,
76      VAR_I,
77      VAR_PTR_BEFORE = VAR_I,
78      VAR_J,
79      VAR_PTR_AFTER = VAR_J,
80      VAR_STORED,
81      VAR_REPLACED,
82      VAR_TEMP = VAR_REPLACED,
83      VAR_PTR_BEFORE_MEM,
84      VAR_GAP = VAR_PTR_BEFORE_MEM,
85      NUM_VARS
86    };
87
88    void solve() {
89      int n;
90      cin >> n;
91
92      const int gaps[] = {701, 301, 132, 57, 23, 10, 4, 1};
93      const int NUM_GAPS = sizeof gaps / sizeof *gaps;
94
95      const int RAM = 1000;
96
97      int ram = std::min(RAM - NUM_VARS, n);
98      int ramsort_thres = RAM - NUM_VARS - NUM_GAPS;
```

```
99
100   const long long PIVOT_MARK = 1ll << 63;
101
102   // Quick sort.
103   // General idea: mark pivots by setting the highest bit. Keep track of
      a
104   // sorted prefix. While the sorted prefix is not everything, scan for
      the
105   // next pivot, then if the subarray to be sorted is small enough move
106   // everything to memory and do some sorting there. Otherwise, choose a
      pivot
107   // and partition.
108
109   SVAR(SORTED_PREFIX, 0);
110   WHILE(LESS(VAR(SORTED_PREFIX), n), {
111     // Tape head here is at VAR(SORTED_PREFIX).
112     SVAR(PIVOT_POSITION, VAR(SORTED_PREFIX));
113     // Find the next pivot (or N).
114     SVAR(PIVOT_FOUND, 0);
115     WHILE(AND(LESS(VAR(PIVOT_POSITION), n), NOT(EQ(VAR(PIVOT_FOUND),
      1))), {
116       IF(NOT(EQ(BITAND(READ, PIVOT_MARK), 0)), { SVAR(PIVOT_FOUND,
        1); });
117       INCR(PIVOT_POSITION, 1);
118       MOVE(1);
119     });
120     IF(EQ(VAR(PIVOT_FOUND), 1), {
121       INCR(PIVOT_POSITION, -1);
122       MOVE(-1);
123     });
124     // Move the tape head back to SORTED_PREFIX.
125     MOVE(SUB(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION)));
126     // Now we know we need to sort [SORTED_PREFIX, PIVOT_POSITION).
127     SVAR(SHORT_SEGMENT,
128          TOINT(LESSEQ(SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX)),
129                       ramsort_thres)));
130     // COLOUT(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION));
131     IF(EQ(VAR(SHORT_SEGMENT), 1), {
132       SVAR(I, VAR(SORTED_PREFIX));
133       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
134         WRITECELL(SUB(VAR(I), VAR(SORTED_PREFIX)), READ);
```

```
135         MOVE(1);
136         INCR(I, 1);
137       });
138       SVAR(STORED, SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX)));
139       // In-memory shell sort.
140       for (int i = 0; i < NUM_GAPS; i++) {
141         INTERNAL_RAMWRITE(RAM - 1 - i, gaps[i]);
142       }
143       SVAR(GAP, 0);
144       WHILE(LESS(VAR(GAP), NUM_GAPS), {
145         SVAR(I, CGAP);
146         WHILE(LESS(VAR(I), VAR(STORED)), {
147           SVAR(TEMP, READCELL(VAR(I)));
148           SVAR(J, VAR(I));
149           WHILE(AND(GREATEQ(VAR(J), CGAP),
150                     GREAT(READCELL(SUB(VAR(J), CGAP)), VAR(TEMP))),
151                 {
152                   WRITECELL(VAR(J), READCELL(SUB(VAR(J), CGAP)));
153                   INCR(J, SUB(0, CGAP));
154                 });
155           WRITECELL(VAR(J), VAR(TEMP));
156           INCR(I, 1);
157         });
158         INCR(GAP, 1);
159       });
160       // We now have an in-memory copy of the relevant part of the tape.
          Copy it
161       // back.
162       MOVE(SUB(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION)));
163       SVAR(I, VAR(SORTED_PREFIX));
164       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
165         WRITE(READCELL(SUB(VAR(I), VAR(SORTED_PREFIX))));
166         MOVE(1);
167         INCR(I, 1);
168       });
169
170       // Now the entire prefix up to PIVOT_POSITION is sorted.
171       SVAR(SORTED_PREFIX, VAR(PIVOT_POSITION));
172       // If PIVOT_POSITION was an actual pivot (i.e. not n), remove the
          mark bit
173       // and declare that position to also be sorted.
```

```
174       IF(LESS(VAR(PIVOT_POSITION), n), {
175         WRITE(BITAND(READ, ~PIVOT_MARK));
176         INCR(SORTED_PREFIX, 1);
177         MOVE(1);
178       });
179     });
180     IF(EQ(VAR(SHORT_SEGMENT), 0), {
181       // Pick the new pivot.
182       // 9223372036854775783 = 2^63-25 is prime. This should work
          relatively
183       // well to pick a random element to be the pivot.
184       SVAR(I, MOD(9223372036854775783,
185                   SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX))));
186       // Swap the pivot with the first position in the subarray.
187       SVAR(PIVOT, READ);
188       MOVE(VAR(I));
189       SVAR(PIVOT, XOR(READ, VAR(PIVOT)));
190       WRITE(XOR(READ, VAR(PIVOT)));
191       SVAR(PIVOT, XOR(READ, VAR(PIVOT)));
192       MOVE(SUB(0, VAR(I)));
193       WRITE(VAR(PIVOT));
194
195       // Count the number of elements lower than the pivot.
196       SVAR(I, VAR(SORTED_PREFIX));
197       SVAR(NEW_PIVOT_POSITION, VAR(SORTED_PREFIX));
198       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
199         IF(LESS(READ, VAR(PIVOT)), { INCR(NEW_PIVOT_POSITION, 1); });
200         INCR(I, 1);
201         MOVE(1);
202       });
203       // Write the pivot (with the marker). Temporarily store the value
          in the
204       // cell in I, and write it in cell 0 of the subarray.
205       MOVE(SUB(VAR(NEW_PIVOT_POSITION), VAR(PIVOT_POSITION)));
206       SVAR(I, READ);
207       WRITE(BITOR(VAR(PIVOT), PIVOT_MARK));
208       MOVE(SUB(VAR(SORTED_PREFIX), VAR(NEW_PIVOT_POSITION)));
209       WRITE(VAR(I));
210
211       // Partition.
```

```
212       // Read elements >= pivot from the first half, swap them with
          elements <
213       // pivot in the second half, then go back to the first half and
          swap
214       // elements back.
215       SVAR(PTR_BEFORE, VAR(SORTED_PREFIX));
216       // The new pivot is a negative number, so we *must* skip it.
217       SVAR(PTR_AFTER, ADD(VAR(NEW_PIVOT_POSITION), 1));
218       WHILE(AND(LESS(VAR(PTR_BEFORE), VAR(NEW_PIVOT_POSITION)),
219                 LESS(VAR(PTR_AFTER), VAR(PIVOT_POSITION))),
220             {
221               // At the start of each loop, tape head is at PTR_BEFORE.
222               SVAR(PTR_BEFORE_MEM, VAR(PTR_BEFORE));
223               SVAR(STORED, 0);
224               // Read large elements.
225               WHILE(AND(LESS(VAR(PTR_BEFORE), VAR(NEW_PIVOT_POSITION)),
226                         LESS(VAR(STORED), ram)),
227                     {
228                       IF(GREATEQ(READ, VAR(PIVOT)), {
229                         WRITECELL(VAR(STORED), READ);
230                         INCR(STORED, 1);
231                       });
232                       INCR(PTR_BEFORE, 1);
233                       MOVE(1);
234                     });
235               // Move to PTR_AFTER.
236               MOVE(SUB(VAR(PTR_AFTER), VAR(PTR_BEFORE)));
237               // Swap out small elements in second half.
238               SVAR(REPLACED, 0);
239               WHILE(LESS(VAR(REPLACED), VAR(STORED)), {
240                 IF(LESS(READ, VAR(PIVOT)), {
241                   WRITECELL(VAR(REPLACED), XOR(READCELL(VAR(REPLACED)),
                      READ));
242                   WRITE(XOR(READCELL(VAR(REPLACED)), READ));
243                   WRITECELL(VAR(REPLACED), XOR(READCELL(VAR(REPLACED)),
                      READ));
244                   INCR(REPLACED, 1);
245                 });
246                 INCR(PTR_AFTER, 1);
247                 MOVE(1);
248               });
```

```
249               // assert that we're still before PIVOT_POSITION?
250               // Move back to PTR_BEFORE_MEM.
251               MOVE(SUB(VAR(PTR_BEFORE_MEM), VAR(PTR_AFTER)));
252               // Write back all the elements that were replaced.
253               SVAR(REPLACED, 0);
254               WHILE(LESS(VAR(REPLACED), VAR(STORED)), {
255                 IF(GREATEQ(READ, VAR(PIVOT)), {
256                   WRITE(READCELL(VAR(REPLACED)));
257                   INCR(REPLACED, 1);
258                 });
259                 INCR(PTR_BEFORE_MEM, 1);
260                 MOVE(1);
261               });
262               MOVE(SUB(VAR(PTR_BEFORE), VAR(PTR_BEFORE_MEM)));
263             });
264
265       // Move the head back to SORTED_PREFIX.
266       MOVE(SUB(VAR(SORTED_PREFIX), VAR(PTR_BEFORE)));
267     });
268   });
269 }
270
271 signed main() {
272   ios_base::sync_with_stdio(0);
273   cin.tie(0);
274
275   int t;
276   cin >> t;
277
278   for (int i = 0; i < t; i++) {
279     cout << "Case #" << i << ":\n";
280     solve();
281   }
282 }
```

Subtask 3: Is the graph connected? (18 points)

The main issue that appears is that we cannot do a normal DFS or BFS. Instead, the idea is that we **contract** the edges in a specific way: We iterate through every edge from left to right, and store up to K edges (u, v) such that $u \neq v$ into RAM. Since for every edge, we found that u and v are connected, we can consider them to be the same vertex or component. We can achieve this by renaming $v \to u$, which we can do on the way back from right to left, where we rename all occurrences of $v \to u$.

This way, we contract all edges until every edge has the form (u, u). Note that per iteration an edge (u, v) with $u \neq v$ will be renamed to (u, u), thus this algorithm will end in at most $2 \cdot \frac{M}{K}$ sweeps.

The graph is connect if and only if every edge has the exact same form (u, u) for some $0 \leq u < N$. If there are two edges (u, u) and (v, v) where $u \neq v$, that would mean there didn't appear a renaming rule from $v \to u$, which is the same as saying there is no path between those vertices.

However, the time will not be sufficient if we rename them naively, i.e. on each edge apply each rule sequentially.

This can be solved by implementing a hash map in RAM. For every new rule $B \to A$ that we pick up, we need some case distinction:

- If there already exists a rule $B \to C$, then instead rename $A \to C$
- If there already exists a rule $A \to C$, then also rename $B \to C$

Then, we use the values in our hash map to save time upon renaming all edges.

(Solution by Luca Versari)

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

using namespace std;

//--------CODE--------//

const int MOD = 1e9 + 7;

#define MOVE(k) cout << "MOVE " << k << "\n";
#define READ "READ"
#define WRITE(k) cout << "WRITE " << k << "\n";
#define INTERNAL_RAMREAD(i) "RAMREAD " << i
```

```
14  #define INTERNAL_RAMWRITE(i, v) cout << "RAMWRITE " << i << " " << v <<
    "\n"
15  #define IF(c, BODY)
    \
16    {                                                                    \
17      cout << "IF " << c << "\n";
        \
18      BODY;
        \
19      cout << "END IF\n";
        \
20    }
21  #define WHILE(c, BODY)
    \
22    {                                                                    \
23      cout << "WHILE " << c << "\n";
        \
24      BODY;
        \
25      cout << "END WHILE\n";
        \
26    }
27  #define ADD(a, b) "+ " << a << " " << b
28  #define SUB(a, b) "- " << a << " " << b
29  #define MUL(a, b) "* " << a << " " << b
30  #define DIV(a, b) "/ " << a << " " << b
31  #define MOD(a, b) "MOD " << a << " " << b
32  #define BITAND(a, b) "& " << a << " " << b
33  #define BITOR(a, b) "| " << a << " " << b
34  #define XOR(a, b) "^ " << a << " " << b
35  #define LESS(a, b) "< " << a << " " << b
36  #define GREAT(a, b) "> " << a << " " << b
37  #define LESSEQ(a, b) "<= " << a << " " << b
38  #define GREATEQ(a, b) ">= " << a << " " << b
39  #define EQ(a, b) "== " << a << " " << b
40  #define DIFF(a, b) "!= " << a << " " << b
41  #define NOT(b) "! " << b
42  #define AND(a, b) "&& " << a << " " << b
43  #define OR(a, b) "|| " << a << " " << b
44  #define SHL(a, b) "<< " << a << " " << b
```

```
45  #define TOINT(b) "INT " << b
46  #define OUT(b) cout << "OUT " << b << "\n"
47  #define COLOUT(u, c) cout << "COLOUT " << u << " " << c << "\n";
48  #define READCELL(CELL) INTERNAL_RAMREAD(ADD(CELL, NUM_VARS))
49  #define WRITECELL(CELL, VAL) INTERNAL_RAMWRITE(ADD(CELL, NUM_VARS), VAL)
50  #define SVAR(VAR, VAL) INTERNAL_RAMWRITE(VAR_##VAR, VAL)
51  #define VAR(VAR) INTERNAL_RAMREAD(VAR_##VAR)
52  #define INCR(V, VAL) SVAR(V, ADD(VAR(V), VAL))
53  #define ASSERT_TAPE_HEAD(VAL)
    \
54    {                                                                 \
55      MOVE(SUB(0, VAL));
      \
56      MOVE(VAL);
      \
57      MOVE(SUB(m, VAL));
      \
58      MOVE(SUB(VAL, m));
      \
59    }
60  #define A(VAL) MOD(VAL, EDGE_MOD)
61  #define B(VAL) DIV(VAL, EDGE_MOD)
62  #define EDGE(A, B) ADD(A, MUL(B, EDGE_MOD))
63  #define HTP(VAL) MOD(MUL(VAR(HASHP), VAL), hashtable_size)
64  #define HTSET(POS, SIDE)
    \
65    WRITECELL(                                                        \
66        ADD(hashtable_size + SIDE * mask_size, DIV(POS, 64)),
        \
67        BITOR(READCELL(ADD(hashtable_size + SIDE * mask_size, DIV(POS,
        64))),    \
68              SHL(1, MOD(POS, 64))))
69  #define HTTEST(POS, SIDE)
    \
70    NOT(EQ(                                                           \
71        BITAND(READCELL(ADD(hashtable_size + SIDE * mask_size, DIV(POS,
        64))),   \
72               SHL(1, MOD(POS, 64))),
               \
73        0))
```

```
74
75  enum {
76    VAR_NUM_CONTRACTING_EDGES = 0,
77    VAR_TOTAL_CONTRACTED,
78    VAR_I,
79    VAR_A,
80    VAR_B,
81    VAR_HASHP,
82    VAR_LAST_EDGE = VAR_HASHP,
83    NUM_VARS
84  };
85
86  const int EDGE_MOD = 100000;
87
88  void solve() {
89    int n, m;
90    cin >> n >> m;
91
92    // Collect many edges to contract in a hash map (as long as the source
      node
93    // does not collide with any node used before).
94    // Then, contract edges in memory, and then contract all the edges in
      the
95    // edge list.
96    // This guarantees that all nodes are merged properly, and fills the
      hash map
97    // in *most* pass. This implementation does O(1) operations per edge
      per
98    // pass.
99    // Once all edges are (a, a) (i.e. we obtain an empty hashmap), we are
      done
100   // and we output 1 IFF all edges are equal.
101
102   int ram = 1000 - NUM_VARS;
103   int hashtable_size = 0;
104   for (int i = 0; i + 2 * ((hashtable_size + 63) / 64) < ram; i++) {
105     bool is_prime = true;
106     for (int j = 2; j * j <= i; j++) {
107       if (i % j == 0)
108         is_prime = false;
109     }
```

```
110     if (is_prime)
111       hashtable_size = i;
112   }
113
114   int mask_size = (hashtable_size + 63) / 64;
115
116   SVAR(TOTAL_CONTRACTED, 0);
117   SVAR(HASHP, 1);
118   WHILE(EQ(0, 0), {
119     // Clear hashmap.
120     SVAR(I, 0);
121     WHILE(LESS(VAR(I), 2 * mask_size), {
122       WRITECELL(ADD(VAR(I), hashtable_size), 0);
123       INCR(I, 1);
124     });
125     // Change hash function.
126     SVAR(HASHP, MOD(MUL(VAR(HASHP), 16807), 1LL << 32));
127
128     // ASSERT_TAPE_HEAD(0);
129
130     // Find edges to contract.
131     SVAR(NUM_CONTRACTING_EDGES, 0);
132     SVAR(I, 0);
133     WHILE(LESS(VAR(I), m), {
134       SVAR(A, A(READ));
135       SVAR(B, B(READ));
136       IF(NOT(EQ(VAR(A), VAR(B))), {
137         // If B is present in the hashmap, swap A and B.
138         IF(HTTEST(HTP(VAR(B)), 1), {
139           SVAR(A, XOR(VAR(A), VAR(B)));
140           SVAR(B, XOR(VAR(A), VAR(B)));
141           SVAR(A, XOR(VAR(A), VAR(B)));
142         });
143
144         // If B is not present in the hash map, record a B -> A
           replacement.
145         IF(NOT(HTTEST(HTP(VAR(B)), 1)), {
146           // First check if we already have a A -> C mapping, and store
             B -> C.
147           IF(AND(HTTEST(HTP(VAR(A)), 0), EQ(B(READCELL(HTP(VAR(A)))),
             VAR(A))),
```

```
148             {
149               WRITECELL(HTP(VAR(B)), EDGE(A(READCELL(HTP(VAR(A)))),
                  VAR(B)));
150             });
151           // Otherwise, store B -> A.
152           IF(NOT(AND(HTTEST(HTP(VAR(A)), 0),
153                      EQ(B(READCELL(HTP(VAR(A)))), VAR(A)))),
154             { WRITECELL(HTP(VAR(B)), EDGE(VAR(A), VAR(B))); });
155           HTSET(HTP(VAR(B)), 0);
156           HTSET(HTP(VAR(B)), 1);
157           HTSET(HTP(VAR(A)), 1);
158           INCR(NUM_CONTRACTING_EDGES, 1);
159           INCR(TOTAL_CONTRACTED, 1);
160         });
161       });
162       INCR(I, 1);
163       MOVE(1);
164     });
165     // ASSERT_TAPE_HEAD(m);
166     // All edges are (a, a).
167     IF(EQ(VAR(NUM_CONTRACTING_EDGES), 0),
168        { OUT(EQ(VAR(TOTAL_CONTRACTED), n - 1)); });
169     // For every edge, check if either side is present in the hashmap
       (as a
170     // source), and if so replace them.
171     WHILE(GREAT(VAR(I), 0), {
172       INCR(I, -1);
173       MOVE(-1);
174       SVAR(A, A(READ));
175       SVAR(B, B(READ));
176       IF(AND(HTTEST(HTP(VAR(A)), 0), EQ(B(READCELL(HTP(VAR(A)))),
          VAR(A))),
177          { SVAR(A, A(READCELL(HTP(VAR(A))))); });
178       IF(AND(HTTEST(HTP(VAR(B)), 0), EQ(B(READCELL(HTP(VAR(B)))),
          VAR(B))),
179          { SVAR(B, A(READCELL(HTP(VAR(B))))); });
180       WRITE(EDGE(VAR(A), VAR(B)));
181     });
182   });
183 }
184
```

```
185 signed main() {
186   ios_base::sync_with_stdio(0);
187   cin.tie(0);
188 
189   int t;
190   cin >> t;
191 
192   for (int i = 0; i < t; i++) {
193     cout << "Case #" << i << ":\n";
194     solve();
195   }
196 }
```

Subtask 4: Is the graph bipartite? (27 points)

We use the solution from subtask 3 and modify it to solve this subtask. The important observation is that per edge we only use $\lceil 2 \cdot \log_2((10^6)) \rceil = 34$ bits to store the edge. Thus, we can use the remaining 30 bits to store some additional information.

We want to reuse the last solution, but we also want to color the graph with two colors at the same time. We initially consider all vertices to be colored black, and we will color some of them white. We store 1 bit for each edge (u, v), which indicates whether u and v have the same color or not; i.e. we store their parity. Initially, all endpoints of the edges have different colors.

If we contract an edge (u, v) with $u \neq v$, we create the rule $v \to u$ as usuals. But when we rename a specific endpoint of an edge, we also change the parity if u, v have different colors.

If we ever contract an edge to the form (u, u) we do nothing if the extra bit indicates that they have the same color, but if they have different color, then we have a contradiction and output `FALSE`.

After we contracted everything without issues, the graph is bipartite. Now we need to reconstruct the coloring. We can do this during the contraction phase: Instead of contracting edges to the form $(u, v) \to (u, u)$, we store an additional bit indicating whether this edge has already been fully contracted. We will still apply the rules of the form $u \to a$, but we will not apply rules $v \to a$ on that edge. Eventually, every edge in the same component has the form (u, v) where u is the same among all the edges in the same component. Thus, we color u black and then check the parity between u and v.

This algorithm has the same running time as in subtask 3 up to a constant factor.

(Solution by Luca Versari)

cpp

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

using namespace std;

//--------CODE--------//

const int MOD = 1e9 + 7;

#define MOVE(k) cout << "MOVE " << k << "\n";
#define READ "READ"
#define WRITE(k) cout << "WRITE " << k << "\n";
#define INTERNAL_RAMREAD(i) "RAMREAD " << i
#define INTERNAL_RAMWRITE(i, v) cout << "RAMWRITE " << i << " " << v << "\n"
```

```
15  #define IF(c, BODY)
    \
16    {                                                                    \
17      cout << "IF " << c << "\n";
        \
18      BODY;
        \
19      cout << "END IF\n";
        \
20    }
21  #define WHILE(c, BODY)
    \
22    {                                                                    \
23      cout << "WHILE " << c << "\n";
        \
24      BODY;
        \
25      cout << "END WHILE\n";
        \
26    }
27  #define ADD(a, b) "+ " << a << " " << b
28  #define SUB(a, b) "- " << a << " " << b
29  #define MUL(a, b) "* " << a << " " << b
30  #define DIV(a, b) "/ " << a << " " << b
31  #define MOD(a, b) "MOD " << a << " " << b
32  #define BITAND(a, b) "& " << a << " " << b
33  #define BITOR(a, b) "| " << a << " " << b
34  #define XOR(a, b) "^ " << a << " " << b
35  #define LESS(a, b) "< " << a << " " << b
36  #define GREAT(a, b) "> " << a << " " << b
37  #define LESSEQ(a, b) "<= " << a << " " << b
38  #define GREATEQ(a, b) ">= " << a << " " << b
39  #define EQ(a, b) "== " << a << " " << b
40  #define DIFF(a, b) "!= " << a << " " << b
41  #define NOT(b) "! " << b
42  #define AND(a, b) "&& " << a << " " << b
43  #define OR(a, b) "|| " << a << " " << b
44  #define SHL(a, b) "<< " << a << " " << b
45  #define TOINT(b) "INT " << b
46  #define OUT(b) cout << "OUT " << b << "\n"
```

```
47  #define COLOUT(u, c) cout << "COLOUT " << u << " " << c << "\n";
48  #define READCELL(CELL) INTERNAL_RAMREAD(ADD(CELL, NUM_VARS))
49  #define WRITECELL(CELL, VAL) INTERNAL_RAMWRITE(ADD(CELL, NUM_VARS), VAL)
50  #define SVAR(VAR, VAL) INTERNAL_RAMWRITE(VAR_##VAR, VAL)
51  #define VAR(VAR) INTERNAL_RAMREAD(VAR_##VAR)
52  #define INCR(V, VAL) SVAR(V, ADD(VAR(V), VAL))
53  #define ASSERT_TAPE_HEAD(VAL)
    \
54    {                                                                   \
55      MOVE(SUB(0, VAL));
      \
56      MOVE(VAL);
      \
57      MOVE(SUB(m, VAL));
      \
58      MOVE(SUB(VAL, m));
      \
59    }
60
61  #define A(VAL) MOD(VAL, EDGE_MOD)
62  #define B(VAL) MOD(DIV(VAL, EDGE_MOD), EDGE_MOD)
63  #define COLA(VAL) MOD(DIV(VAL, COLA_MOD), 2)
64  #define COLB(VAL) MOD(DIV(VAL, COLB_MOD), 2)
65  #define MARKER(VAL) DIV(VAL, MARKER_MOD)
66  #define EDGE(A, COLA, B, COLB, MARKER)
    \
67    ADD(ADD(MUL(MARKER, MARKER_MOD), MUL(COLA, COLA_MOD)),
      \
68        ADD(MUL(COLB, COLB_MOD), ADD(A, MUL(B, EDGE_MOD))))
69  #define HTP(VAL) MOD(MUL(VAR(HASHP), VAL), hashtable_size)
70  #define HTSET(POS)
    \
71    WRITECELL(ADD(hashtable_size, DIV(POS, 64)),
      \
72              BITOR(READCELL(ADD(hashtable_size, DIV(POS, 64))),
              \
73                    SHL(1, MOD(POS, 64))))
74  #define HTTEST(POS)
    \
75    NOT(EQ(BITAND(READCELL(ADD(hashtable_size, DIV(POS, 64))),
      \
```

```
76                  SHL(1, MOD(POS, 64))),
                    \
77          0))
78
79  enum { VAR_NUM_CONTRACTING_EDGES = 0, VAR_I, VAR_HASHP, NUM_VARS };
80
81  const long long EDGE_MOD = 100000;
82  const long long COLA_MOD = EDGE_MOD * EDGE_MOD;
83  const long long COLB_MOD = COLA_MOD * 2;
84  const long long MARKER_MOD = COLB_MOD * 2;
85
86  void solve() {
87    int n, m;
88    cin >> n >> m;
89
90    // This follows the general scheme of the full solution of sub3.
91    // Initially, we color all nodes white (0).
92    // Whenever we contract an edge, we remove the edge from the graph and
      replace
93    // it with a "marker" edge that records that the removed node was
      white when
94    // this operation happened. If the edge is monochromatic, we swap the
      colours
95    // of all the nodes in the same connected component (as of now) of B.
      Finally,
96    // when we perform the contraction, if we encounter a same-color edge
      we
97    // return FALSE. If we don't have any more contractible edges, we
      return TRUE.
98
99    int ram = 1000 - NUM_VARS;
100   int hashtable_size = 0;
101   for (int i = 0; i + ((hashtable_size + 63) / 64) < ram; i++) {
102     bool is_prime = true;
103     for (int j = 2; j * j <= i; j++) {
104       if (i % j == 0)
105         is_prime = false;
106     }
107     if (is_prime)
108       hashtable_size = i;
109   }
110
```

```
111   int mask_size = (hashtable_size + 63) / 64;
112
113   // Colour everything white.
114   SVAR(I, 0);
115   WHILE(LESS(VAR(I), n), {
116     COLOUT(VAR(I), 0);
117     INCR(I, 1);
118   });
119
120   SVAR(HASHP, 1);
121   WHILE(EQ(0, 0), {
122     // Clear hashmap.
123     SVAR(I, 0);
124     WHILE(LESS(VAR(I), mask_size), {
125       WRITECELL(ADD(VAR(I), hashtable_size), 0);
126       INCR(I, 1);
127     });
128     // Change hash function.
129     SVAR(HASHP, MOD(MUL(VAR(HASHP), 16807), 1LL << 32));
130
131     // ASSERT_TAPE_HEAD(0);
132
133     // Find edges to contract.
134     SVAR(NUM_CONTRACTING_EDGES, 0);
135     SVAR(I, 0);
136     WHILE(LESS(VAR(I), m), {
137       IF(NOT(EQ(A(READ), B(READ))), {
138         // If B is collides in the hashmap, swap A and B.
139         IF(HTTEST(HTP(B(READ))), {
140           WRITE(EDGE(B(READ), COLB(READ), A(READ), COLA(READ),
              MARKER(READ)));
141         });
142
143         // If B does not collide in the hash map, and A is not already
          being
144         // replaced, record a B -> A replacement.
145         IF(AND(NOT(HTTEST(HTP(B(READ)))),
146                NOT(AND(HTTEST(HTP(A(READ))),
147                        EQ(B(READCELL(HTP(A(READ)))), A(READ))))),
148
149           {
```

```
150             // If COLB == COLA, we need to flip the color of B. Record
                that
151             // as a marker of 1 if swapping is needed, 0 otherwise.
152             WRITECELL(HTP(B(READ)), EDGE(A(READ), 0, B(READ), 0,
153                                          TOINT(EQ(COLA(READ),
                                             COLB(READ)))));
154             HTSET(HTP(B(READ)));
155             HTSET(HTP(A(READ)));
156             INCR(NUM_CONTRACTING_EDGES, 1);
157             // Write the marker to remember this is a contracted edge.
158             // At this point in time, B actually has color 0. So, to
                avoid
159             // confusion let's directly write down this edge as (b, b)
160             // with color (1, 0). It doesn't actually matter that we
                end
161             // up not checking this edge properly, as this is a tree
                edge
162             // and we make it correct by construction.
163             WRITE(EDGE(B(READ), 1, B(READ), 0, ADD(B(READ), 1)));
164           });
165       });
166       INCR(I, 1);
167       MOVE(1);
168     });
169     // ASSERT_TAPE_HEAD(m);
170     // All edges are (a, a).
171     // If we didn't fail yet, the entire graph is two-colorable and we
      output
172     // TRUE, after going over all contracted edges to apply the proper
      color.
173     IF(EQ(VAR(NUM_CONTRACTING_EDGES), 0), {
174       WHILE(GREAT(VAR(I), 0), {
175         INCR(I, -1);
176         MOVE(-1);
177         IF(GREAT(MARKER(READ), 0),
178            { COLOUT(SUB(MARKER(READ), 1), COLB(READ)); });
179       });
180
181       OUT(EQ(0, 0));
182     });
183     // For every edge, check if either side is present in the hashmap
      (as a
```

```
184     // source), and if so replace them, possibly replacing the color.
185     WHILE(GREAT(VAR(I), 0), {
186       INCR(I, -1);
187       MOVE(-1);
188       IF(AND(HTTEST(HTP(A(READ))), EQ(B(READCELL(HTP(A(READ)))),
          A(READ))), {
189         WRITE(EDGE(A(READCELL(HTP(A(READ)))),
190                    XOR(COLA(READ), MARKER(READCELL(HTP(A(READ))))),
                       B(READ),
191                    COLB(READ), MARKER(READ)));
192       });
193       IF(AND(HTTEST(HTP(B(READ))), EQ(B(READCELL(HTP(B(READ)))),
          B(READ))), {
194         WRITE(EDGE(A(READ), COLA(READ), A(READCELL(HTP(B(READ)))),
195                    XOR(COLB(READ), MARKER(READCELL(HTP(B(READ))))),
196                    MARKER(READ)));
197       });
198       // If we created a (a, a) edge with the same colours, the graph is
          not
199       // 2-colorable.
200       IF(AND(EQ(A(READ), B(READ)), EQ(COLA(READ), COLB(READ))),
201          { OUT(EQ(0, 1)); });
202     });
203   });
204 }
205
206 signed main() {
207   ios_base::sync_with_stdio(0);
208   cin.tie(0);
209
210   int t;
211   cin >> t;
212
213   for (int i = 0; i < t; i++) {
214     cout << "Case #" << i << ":\n";
215     solve();
216   }
217 }
```

Subtask 5: Use as few colors as possible (33 points)

An important note is that we cannot efficiently solve this problem optimally, as this is an NP-Hard problem. Therefore, we just need a very good and fast solution.

We first preprocess each edge (u, v) such that for every edge $u < v$. Then, we sort the edges (u, v) first by u and then by v. We can reuse the algorithm from subtask 2 for doing this. This ensures that all edges from a vertex u are processed after all edges from vertices $n < u$ are visited.

The general algorithm is as follows:

- We initiate a color count $c = 0$
- We go through every node u.
- For each node, find any color $b < c$ such that no node $v < u$ which is attached to u has this color.
- If there exists such a color, color u to b.
- If there doesn't exist such a color, then color u to c and increment c

To find a color not already used by neighboring vertices, we make a heuristic and only count the number of colors occurring in a certain buckets (range) of colors. For each "back edge", we increase the count of the bucket corresponding to the neighboring node. If we realize that one of the buckets has a count less than the number of colors in the bucket, we backtrack to find out which color has not been used.

(Solution by Luca Versari)

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

using namespace std;

#define all(x) x.begin(), x.end()
#define sz(x) (int)x.size()

//--------CODE--------//

const int MOD = 1e9 + 7;

#define MOVE(k) cout << "MOVE " << k << "\n";
#define READ "READ"
#define WRITE(k) cout << "WRITE " << k << "\n"
#define INTERNAL_RAMREAD(i) "RAMREAD " << i
#define INTERNAL_RAMWRITE(i, v) cout << "RAMWRITE " << i << " " << v << "\n"
```

```
19  #define IF(c, BODY)
    \

20    {                                                                    \
        cout << "IF " << c << "\n";
21      \
        BODY;
22      \
        cout << "END IF\n";
23      \
    }
24  #define WHILE(c, BODY)
25  \

26    {                                                                    \
        cout << "WHILE " << c << "\n";
27      \
        BODY;
28      \
        cout << "END WHILE\n";
29      \
    }
30  #define ADD(a, b) "+ " << a << " " << b
31  #define SUB(a, b) "- " << a << " " << b
32  #define MUL(a, b) "* " << a << " " << b
33  #define DIV(a, b) "/ " << a << " " << b
34  #define MOD(a, b) "MOD " << a << " " << b
35  #define BITAND(a, b) "& " << a << " " << b
36  #define BITOR(a, b) "| " << a << " " << b
37  #define XOR(a, b) "^ " << a << " " << b
38  #define LESS(a, b) "< " << a << " " << b
39  #define GREAT(a, b) "> " << a << " " << b
40  #define LESSEQ(a, b) "<= " << a << " " << b
41  #define GREATEQ(a, b) ">= " << a << " " << b
42  #define EQ(a, b) "== " << a << " " << b
43  #define DIFF(a, b) "!= " << a << " " << b
44  #define NOT(b) "! " << b
45  #define AND(a, b) "&& " << a << " " << b
46  #define OR(a, b) "|| " << a << " " << b
47  #define TOINT(b) "INT " << b
48  #define OUT(b) cout << "OUT " << b << "\n"
49  #define COLOUT(u, c) cout << "COLOUT " << u << " " << c << "\n";
50
```

```
51  #define READCELL(CELL) INTERNAL_RAMREAD(ADD(CELL, NUM_VARS))
52  #define WRITECELL(CELL, VAL) INTERNAL_RAMWRITE(ADD(CELL, NUM_VARS), VAL)
53  #define READBUCKET(CELL)
    \
54    INTERNAL_RAMREAD(ADD(CELL, ADD(NUM_VARS, VAR(NUM_COLORS_IN_MEMORY))))
55  #define WRITEBUCKET(CELL, VAL)
    \
56    INTERNAL_RAMWRITE(ADD(CELL, ADD(NUM_VARS, VAR(NUM_COLORS_IN_MEMORY))),
      VAL)
57  #define SVAR(VAR, VAL) INTERNAL_RAMWRITE(VAR_##VAR, VAL)
58  #define VAR(VAR) INTERNAL_RAMREAD(VAR_##VAR)
59  #define INCR(V, VAL) SVAR(V, ADD(VAR(V), VAL))
60  #define ASSERT_TAPE_HEAD(VAL)
    \
61    {                                                                    \
62      MOVE(SUB(0, VAL));
      \
63      MOVE(VAL);
      \
64      MOVE(SUB(m, VAL));
      \
65      MOVE(SUB(VAL, m));
      \
66    }
67
68  #define CGAP INTERNAL_RAMREAD(SUB(RAM - 1, VAR(GAP)))
69
70  #define A(VAL) MOD(VAL, EDGE_MOD)
71  #define B(VAL) MOD(DIV(VAL, EDGE_MOD), EDGE_MOD)
72  #define MARKER(VAL) DIV(VAL, (EDGE_MOD * EDGE_MOD))
73  #define EDGE(A, B, MARKER)
    \
74    ADD(MUL(MARKER, (EDGE_MOD * EDGE_MOD)), ADD(A, MUL(B, EDGE_MOD)))
75
76  enum {
77    VAR_SORTED_PREFIX = 0,
78    VAR_NUM_USED_COLORS = VAR_SORTED_PREFIX,
79    VAR_PIVOT,
80    VAR_NEXT_NODE = VAR_PIVOT,
81    VAR_TO_STORE = VAR_PIVOT,
82    VAR_PIVOT_POSITION,
```

```
83      VAR_NODE_EDGES_START = VAR_PIVOT_POSITION,
84      VAR_PIVOT_FOUND,
85      VAR_NODE_EDGES_END = VAR_PIVOT_FOUND,
86      VAR_SHORT_SEGMENT = VAR_PIVOT_FOUND,
87      VAR_NEW_PIVOT_POSITION = VAR_SHORT_SEGMENT,
88      VAR_I,
89      VAR_PTR_BEFORE = VAR_I,
90      VAR_J,
91      VAR_PTR_AFTER = VAR_J,
92      VAR_STORED,
93      VAR_BUCKET_SIZE = VAR_STORED,
94      VAR_REPLACED,
95      VAR_BUCKET_START = VAR_REPLACED,
96      VAR_TEMP = VAR_REPLACED,
97      VAR_PTR_BEFORE_MEM,
98      VAR_GAP = VAR_PTR_BEFORE_MEM,
99      VAR_NUM_COLORS_IN_MEMORY = VAR_PTR_BEFORE_MEM,
100     VAR_FIRST_NODE_COLOR_IN_MEMORY,
101     NUM_VARS
102 };
103
104 const long long EDGE_MOD = 100000;
105
106 void solve() {
107   int nodes, m;
108   cin >> nodes >> m;
109
110   const int gaps[] = {701, 301, 132, 57, 23, 10, 4, 1};
111   const int NUM_GAPS = sizeof gaps / sizeof *gaps;
112
113   const int RAM = 1000;
114
115   int ram = std::min(RAM - NUM_VARS, m);
116   int ramsort_thres = RAM - NUM_VARS - NUM_GAPS;
117
118   int max_color_ram = ram / 2;
119
120   // General algorithm:
121   // - First, ensure all edges are in form (a, b) with a < b.
122   // - Then, sort all edges in *increasing* order (using the quicksort
      code).
```

```
123   // This guarantees that all edges that only involve nodes below X
      appear
124   // before any edge involving X.
125   // - Then, go over edges. Color nodes one by one, by finding *any*
      unused
126   // color across the backwards edges of that node <= the number of
      colors that
127   // we have so far. If there is none, increase color count.
128   // - We do this in two passes over the backward edges of that node. In
      the
129   // first pass, we count the number of colors used in buckets of size
130   // K ~ #colors/ram. In the second pass, we do the same in the first
      bucket
131   // that is below capacity.
132   // - When we reach capacity for number of nodes we store colors for,
      we go
133   // over all the remaining edges and write down the color of the node
      we have
134   // in memory where it matches a.
135
136   // Step 1: (a, b) -> (min(a, b), max(a, b)).
137   SVAR(I, 0);
138   WHILE(LESS(VAR(I), m), {
139     IF(GREAT(A(READ), B(READ)), { WRITE(EDGE(B(READ), A(READ), 0)); });
140     MOVE(1);
141     INCR(I, 1);
142   });
143   MOVE(-m);
144
145   const long long PIVOT_MARK = 1ll << 63;
146
147   SVAR(SORTED_PREFIX, 0);
148   WHILE(LESS(VAR(SORTED_PREFIX), m), {
149     // Tape head here is at VAR(SORTED_PREFIX).
150     SVAR(PIVOT_POSITION, VAR(SORTED_PREFIX));
151     // Find the next pivot (or N).
152     SVAR(PIVOT_FOUND, 0);
153     WHILE(AND(LESS(VAR(PIVOT_POSITION), m), NOT(EQ(VAR(PIVOT_FOUND),
      1))), {
154       IF(NOT(EQ(BITAND(READ, PIVOT_MARK), 0)), { SVAR(PIVOT_FOUND,
      1); });
155       INCR(PIVOT_POSITION, 1);
```

```
156       MOVE(1);
157     });
158     IF(EQ(VAR(PIVOT_FOUND), 1), {
159       INCR(PIVOT_POSITION, -1);
160       MOVE(-1);
161     });
162     // Move the tape head back to SORTED_PREFIX.
163     MOVE(SUB(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION)));
164     // Now we know we need to sort [SORTED_PREFIX, PIVOT_POSITION).
165     SVAR(SHORT_SEGMENT,
166          TOINT(LESSEQ(SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX)),
167                       ramsort_thres)));
168     // COLOUT(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION));
169     IF(EQ(VAR(SHORT_SEGMENT), 1), {
170       SVAR(I, VAR(SORTED_PREFIX));
171       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
172         WRITECELL(SUB(VAR(I), VAR(SORTED_PREFIX)), READ);
173         MOVE(1);
174         INCR(I, 1);
175       });
176       SVAR(STORED, SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX)));
177       // In-memory shell sort.
178       for (int i = 0; i < NUM_GAPS; i++) {
179         INTERNAL_RAMWRITE(RAM - 1 - i, gaps[i]);
180       }
181       SVAR(GAP, 0);
182       WHILE(LESS(VAR(GAP), NUM_GAPS), {
183         SVAR(I, CGAP);
184         WHILE(LESS(VAR(I), VAR(STORED)), {
185           SVAR(TEMP, READCELL(VAR(I)));
186           SVAR(J, VAR(I));
187           WHILE(AND(GREATEQ(VAR(J), CGAP),
188                     GREAT(READCELL(SUB(VAR(J), CGAP)), VAR(TEMP))),
189                 {
190                   WRITECELL(VAR(J), READCELL(SUB(VAR(J), CGAP)));
191                   INCR(J, SUB(0, CGAP));
192                 });
193           WRITECELL(VAR(J), VAR(TEMP));
194           INCR(I, 1);
195         });
```

```
196         INCR(GAP, 1);
197       });
198       // We now have an in-memory copy of the relevant part of the tape.
          Copy it
199       // back.
200       MOVE(SUB(VAR(SORTED_PREFIX), VAR(PIVOT_POSITION)));
201       SVAR(I, VAR(SORTED_PREFIX));
202       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
203         WRITE(READCELL(SUB(VAR(I), VAR(SORTED_PREFIX))));
204         MOVE(1);
205         INCR(I, 1);
206       });
207
208       // Now the entire prefix up to PIVOT_POSITION is sorted.
209       SVAR(SORTED_PREFIX, VAR(PIVOT_POSITION));
210       // If PIVOT_POSITION was an actual pivot (i.e. not n), remove the
          mark bit
211       // and declare that position to also be sorted.
212       IF(LESS(VAR(PIVOT_POSITION), m), {
213         WRITE(BITAND(READ, ~PIVOT_MARK));
214         INCR(SORTED_PREFIX, 1);
215         MOVE(1);
216       });
217     });
218     IF(EQ(VAR(SHORT_SEGMENT), 0), {
219       // Pick the new pivot.
220       // 9223372036854775783 = 2^63-25 is prime. This should work
          relatively
221       // well to pick a random element to be the pivot.
222       SVAR(I, MOD(9223372036854775783,
223                   SUB(VAR(PIVOT_POSITION), VAR(SORTED_PREFIX))));
224       // Swap the pivot with the first position in the subarray.
225       SVAR(PIVOT, READ);
226       MOVE(VAR(I));
227       SVAR(PIVOT, XOR(READ, VAR(PIVOT)));
228       WRITE(XOR(READ, VAR(PIVOT)));
229       SVAR(PIVOT, XOR(READ, VAR(PIVOT)));
230       MOVE(SUB(0, VAR(I)));
231       WRITE(VAR(PIVOT));
232
233       // Count the number of elements lower than the pivot.
```

```
234       SVAR(I, VAR(SORTED_PREFIX));
235       SVAR(NEW_PIVOT_POSITION, VAR(SORTED_PREFIX));
236       WHILE(LESS(VAR(I), VAR(PIVOT_POSITION)), {
237         IF(LESS(READ, VAR(PIVOT)), { INCR(NEW_PIVOT_POSITION, 1); });
238         INCR(I, 1);
239         MOVE(1);
240       });
241       // Write the pivot (with the marker). Temporarily store the value
          in the
242       // cell in I, and write it in cell 0 of the subarray.
243       MOVE(SUB(VAR(NEW_PIVOT_POSITION), VAR(PIVOT_POSITION)));
244       SVAR(I, READ);
245       WRITE(BITOR(VAR(PIVOT), PIVOT_MARK));
246       MOVE(SUB(VAR(SORTED_PREFIX), VAR(NEW_PIVOT_POSITION)));
247       WRITE(VAR(I));
248
249       // Partition.
250       // Read elements >= pivot from the first half, swap them with
          elements <
251       // pivot in the second half, then go back to the first half and
          swap
252       // elements back.
253       SVAR(PTR_BEFORE, VAR(SORTED_PREFIX));
254       // The new pivot is a negative number, so we *must* skip it.
255       SVAR(PTR_AFTER, ADD(VAR(NEW_PIVOT_POSITION), 1));
256       WHILE(AND(LESS(VAR(PTR_BEFORE), VAR(NEW_PIVOT_POSITION)),
257                 LESS(VAR(PTR_AFTER), VAR(PIVOT_POSITION))),
258             {
259               // At the start of each loop, tape head is at PTR_BEFORE.
260               SVAR(PTR_BEFORE_MEM, VAR(PTR_BEFORE));
261               SVAR(STORED, 0);
262               // Read large elements.
263               WHILE(AND(LESS(VAR(PTR_BEFORE), VAR(NEW_PIVOT_POSITION)),
264                         LESS(VAR(STORED), ram)),
265                     {
266                       IF(GREATEQ(READ, VAR(PIVOT)), {
267                         WRITECELL(VAR(STORED), READ);
268                         INCR(STORED, 1);
269                       });
270                       INCR(PTR_BEFORE, 1);
271                       MOVE(1);
```

```
272                   });
273               // Move to PTR_AFTER.
274               MOVE(SUB(VAR(PTR_AFTER), VAR(PTR_BEFORE)));
275               // Swap out small elements in second half.
276               SVAR(REPLACED, 0);
277               WHILE(LESS(VAR(REPLACED), VAR(STORED)), {
278                 IF(LESS(READ, VAR(PIVOT)), {
279                   WRITECELL(VAR(REPLACED), XOR(READCELL(VAR(REPLACED)),
                      READ));
280                   WRITE(XOR(READCELL(VAR(REPLACED)), READ));
281                   WRITECELL(VAR(REPLACED), XOR(READCELL(VAR(REPLACED)),
                      READ));
282                   INCR(REPLACED, 1);
283                 });
284                 INCR(PTR_AFTER, 1);
285                 MOVE(1);
286               });
287               // assert that we're still before PIVOT_POSITION?
288               // Move back to PTR_BEFORE_MEM.
289               MOVE(SUB(VAR(PTR_BEFORE_MEM), VAR(PTR_AFTER)));
290               // Write back all the elements that were replaced.
291               SVAR(REPLACED, 0);
292               WHILE(LESS(VAR(REPLACED), VAR(STORED)), {
293                 IF(GREATEQ(READ, VAR(PIVOT)), {
294                   WRITE(READCELL(VAR(REPLACED)));
295                   INCR(REPLACED, 1);
296                 });
297                 INCR(PTR_BEFORE_MEM, 1);
298                 MOVE(1);
299               });
300               MOVE(SUB(VAR(PTR_BEFORE), VAR(PTR_BEFORE_MEM)));
301             });
302
303       // Move the head back to SORTED_PREFIX.
304       MOVE(SUB(VAR(SORTED_PREFIX), VAR(PTR_BEFORE)));
305     });
306   });
307   MOVE(SUB(0, VAR(SORTED_PREFIX)));
308
309   // Step 3: assign colors.
310   SVAR(NEXT_NODE, 0);
```

```
311   SVAR(NODE_EDGES_START, 0);
312   SVAR(NUM_USED_COLORS, 0);
313   SVAR(NUM_COLORS_IN_MEMORY, 0);
314   SVAR(FIRST_NODE_COLOR_IN_MEMORY, 0);
315
316   // COLOR 0 is unused.
317   WHILE(LESS(VAR(NEXT_NODE), nodes), {
318     // Tape is at NODE_EDGES_START.
319
320     // Flush colors of nodes if needed.
321     IF(GREATEQ(VAR(NUM_COLORS_IN_MEMORY), max_color_ram), {
322       SVAR(I, VAR(NODE_EDGES_START));
323       WHILE(LESS(VAR(I), m), {
324         IF(AND(LESS(A(READ), ADD(VAR(FIRST_NODE_COLOR_IN_MEMORY),
325                                  VAR(NUM_COLORS_IN_MEMORY))),
326                GREATEQ(A(READ), VAR(FIRST_NODE_COLOR_IN_MEMORY))),
327            {
328              WRITE(
329                  EDGE(A(READ), B(READ),
330                       READCELL(SUB(A(READ),
                             VAR(FIRST_NODE_COLOR_IN_MEMORY)))));
331            });
332         MOVE(1);
333         INCR(I, 1);
334       });
335       MOVE(SUB(VAR(NODE_EDGES_START), m));
336       INCR(FIRST_NODE_COLOR_IN_MEMORY, VAR(NUM_COLORS_IN_MEMORY));
337       SVAR(NUM_COLORS_IN_MEMORY, 0);
338     });
339
340     SVAR(NODE_EDGES_END, VAR(NODE_EDGES_START));
341     WHILE(AND(LESS(VAR(NODE_EDGES_END), m), EQ(B(READ),
      VAR(NEXT_NODE))), {
342       INCR(NODE_EDGES_END, 1);
343       MOVE(1);
344     });
345
346     SVAR(
347         BUCKET_SIZE,
348         ADD(DIV(VAR(NUM_USED_COLORS), SUB(ram,
          VAR(NUM_COLORS_IN_MEMORY))), 1));
```

```
349
350     // Zero out buckets.
351     SVAR(I, 0);
352     WHILE(LESS(VAR(I), SUB(ram, VAR(NUM_COLORS_IN_MEMORY))), {
353       WRITEBUCKET(VAR(I), 0);
354       INCR(I, 1);
355     });
356
357     // Go backwards and fill buckets.
358     SVAR(I, VAR(NODE_EDGES_END));
359     WHILE(GREAT(VAR(I), VAR(NODE_EDGES_START)), {
360       INCR(I, -1);
361       MOVE(-1);
362
363       // Read node color (possibly from memory).
364       SVAR(J, MARKER(READ));
365       IF(EQ(VAR(J), 0),
366          { SVAR(J, READCELL(SUB(A(READ),
           VAR(FIRST_NODE_COLOR_IN_MEMORY)))); });
367       INCR(J, -1); // move colors to start from 0
368       // Map that into a bucket index.
369       SVAR(J, DIV(VAR(J), VAR(BUCKET_SIZE)));
370       IF(GREAT(ADD(VAR(J), ADD(NUM_VARS, VAR(NUM_COLORS_IN_MEMORY))),
         999), {
371         COLOUT(0, VAR(NUM_USED_COLORS));
372         COLOUT(1, VAR(NUM_COLORS_IN_MEMORY));
373         COLOUT(2, VAR(BUCKET_SIZE));
374         COLOUT(3, VAR(J));
375         COLOUT(4, VAR(NODE_EDGES_START));
376         COLOUT(5, VAR(NODE_EDGES_END));
377       });
378
379       WRITEBUCKET(VAR(J), ADD(READBUCKET(VAR(J)), 1));
380     });
381
382     // Tape head is at NODE_EDGES_START.
383
384     SVAR(BUCKET_START, 0);
385     WHILE(GREATEQ(READBUCKET(VAR(BUCKET_START)), VAR(BUCKET_SIZE)),
386           { INCR(BUCKET_START, 1); });
387     SVAR(BUCKET_START, MUL(VAR(BUCKET_START), VAR(BUCKET_SIZE)));
```

```
388
389     // Zero out buckets.
390     SVAR(I, 0);
391     WHILE(LESS(VAR(I), SUB(ram, VAR(NUM_COLORS_IN_MEMORY))), {
392       WRITEBUCKET(VAR(I), 0);
393       INCR(I, 1);
394     });
395
396     // Go backwards and fill buckets.
397     SVAR(I, VAR(NODE_EDGES_START));
398     WHILE(LESS(VAR(I), VAR(NODE_EDGES_END)), {
399       // Read node color (possibly from memory).
400       SVAR(J, MARKER(READ));
401       IF(EQ(VAR(J), 0),
402          { SVAR(J, READCELL(SUB(A(READ),
           VAR(FIRST_NODE_COLOR_IN_MEMORY)))); });
403
404       INCR(J, -1); // move colors to start from 0
405       // Write it down if it is in the bucket.
406       SVAR(J, SUB(VAR(J), VAR(BUCKET_START)));
407       IF(AND(GREATEQ(VAR(J), 0), LESS(VAR(J), VAR(BUCKET_SIZE))),
408          { WRITEBUCKET(VAR(J), ADD(READBUCKET(VAR(J)), 1)); });
409
410       INCR(I, 1);
411       MOVE(1);
412     });
413
414     // Tape head is at NODE_EDGES_END.
415
416     SVAR(J, 0);
417     WHILE(GREAT(READBUCKET(VAR(J)), 0), { INCR(J, 1); });
418     SVAR(J, ADD(VAR(BUCKET_START), VAR(J)));
419     // Assign color J.
420     COLOUT(VAR(NEXT_NODE), VAR(J));
421     IF(GREATEQ(VAR(J), VAR(NUM_USED_COLORS)), { INCR(NUM_USED_COLORS,
     1); });
422     INCR(J, 1); // Move back colors in [1, ..) range
423
424     WRITECELL(VAR(NUM_COLORS_IN_MEMORY), VAR(J));
425     INCR(NUM_COLORS_IN_MEMORY, 1);
426     SVAR(NODE_EDGES_START, VAR(NODE_EDGES_END));
```

```
427
428     INCR(NEXT_NODE, 1);
429   });
430 }
431
432 signed main() {
433   ios_base::sync_with_stdio(0);
434   cin.tie(0);
435
436   int t;
437   cin >> t;
438
439   for (int i = 0; i < t; i++) {
440     cout << "Case #" << i << ":\n";
441     solve();
442   }
443 }
```

Tomb exploration

Task Idea	Noémie Jacquemot, Timon Stampfli, Yaël Arn
Task Preparation	Théo von Düring
Translation (de)	Vivienne Burckhardt
Translation (fr)	Cheng Zhong
Solution Author	???

The intended author of this solution refused to write Typst, sorry about that :).

Subtask 1: Ordered statues (17 points)

Subtask 2: Sports bags (18 points)

Subtask 3: Small bags (24 points)

Subtask 4: The great find (41 points)

Evilruins

Task Idea	Mark Neumann, Yaël Arn, Charlotte Knierim
Task Preparation	Yaël Arn, Ursus Wigger
Translation (de)	Ursus Wigger
Translation (fr)	Théo von Düring
Solution Author	Yaël Arn
Further Credits	Johannes Kapfhammer, Linus Lüchinger

We are given a tree with positive values in the leaves. Two players take turns, trying to claim the highest total value of leaves for themselves. In each turn, a player can claim the root of one of the remaining subtrees, freeing up the direct children of that root for the next turn. Your task is to compute the difference in total leaf values between the two players if they play optimally.

Subtask 1: Direct channels (6 points)

In this subtask, the tree is a star. Thus, the first player must claim the root, and afterwards, the players alternate in claiming leaves.

We now show a statement which looks obvious, but is somewhat lenghty to prove rigorosely.

Lemma 1. *It is always optimal to take the leaf with the largest value.*

Proof. For the proof, we use induction on the number of leafs.

The base case, one leaf, is clear, as the player doesn't have any other choice. Further, for two leafs, if we don't take the bigger one, the opponent will get it, so we are stricly worse off.

Now assume we have shown that for all $m \leq n$ leafs, this stategy is optimal. Now let us have $3 \leq n+1$ leafs, and assume there is a optimal strategy where we take some leaf with value a_i which is not maximal. Consider the set of leafs consisting of the maximal leaf a_{n+1}, the leaf a_i and the leaf a_j which is greatest excluding a_{n+1} and a_i. Now in the optimal strategy starting with a_i, the opponent will take a_{n+1} by the induction hypothesis, and we will take a_j. If instead we took a_{n+1}, the opponent could only take the maximal element of a_i and a_j. If in the next move, we take the remaining element of a_i and a_j, the rest of the game is exactly the same as in the optimal strategy as the remaining elements are the same. But because in the new strategy the opponent didn't get a_{n+1} in their first move, but a smaller one, and got the same leafs afterwards, their total value is not greater as in the optimal strategy. Because the sum of all leafs is the same, this means the value of the first player is not worse as in the optimal strategy, so we can always start with taking the largest value. □

Using Lemma 1, we get that if the values of the leaves a_i are sorted in decreasing manner, the answer is

$$\sum_{i=0}^{\lceil \frac{n}{2} \rceil} a_{2i} - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} a_{2i+1} = \sum_{i=0}^{n} (-1)^i a_i$$

We can compute this with a very short program:

(Solution by Yaël)

```python
def get_input():
  n = int(input())
  w = map(int, input().split())
  edges = [map(int, input().split()) for i in range(n - 1)]
  return n, w, edges

def solve(n, w, edges):
  w = sorted(w)
  return sum([(-1)**i * a for i, a in enumerate(w)])

t = int(input())
for i in range(t):
  print(f"Case #{i}: {solve(*get_input())}")
```

Subtask 2: Transportation channels (10 points)

In this subtask, we have a catarpillar tree, so a path with leaves hanging from it. We can try to generalise the approach from subtask 1.

Subtask 3: Guard chambers (19 points)

Subtask 4: Collapsed chambers (8 points)

Subtask 5: Equally sized entrances (14 points)

Subtask 6: The evil aura (43 points)