# ZCO 2022 Editorial and Statistics

ZCO Problem Setters

December 2021

## 1 Problem 1: Vacation

*Abridged statement:* Given a grid of 0s and 1s, and many queries for two grid squares, find the minimum cost of a path between the two grid squares, where the cost of the path is defined as the product of the values.

The following subtasks are available:

- Subtask 1 [5 points]: All the cells in the grid have a cost of 1.
- Subtask 2 [6 points]: For all trips, $A_i = C_i$ and $B_i = D_i$.
- Subtask 3 [23 points]: $N \leq 8$, $M \leq 8$ and $Q \leq 5$.
- Subtask 4 [20 points]: $Q \leq 5$
- Subtask 5 [19 points]: At most 10 cells in the grid have a cost of 0.
- Subtask 6 [10 points]: $N = 1$
- Subtask 7 [8 points]: $N \leq 5$
- Subtask 8 [9 points]: No additional constraints.

### 1.1 Subtask 1: All cells in the grid have a cost of 1

For this subtask, since all cells in the grid have a cost of 1, it naturally follows that all squares in any query will have a cost of 1. Then, all squares in any path between two points must have a cost of 1. Now, consider the product of costs on the path. Since all individual costs are 1, the product of all the costs must also be 1. Therefore, the answer to all trips is 1, and you can output 1, $Q$ times to solve this subtask. Each query takes $O(1)$ time to solve.

### 1.2 Subtask 2: For all trips, $A_i = C_i$ and $B_i = D_i$

For this subtask, the starting square is the same as the ending square, for all trips. As a result, there is only one possible sequence of moves for each trip, consisting of solely the starting or the ending square. Then, the cost of the trip

is the product of only that single square. If the single square has cost 0, then the cost is 0, while if the square has cost 1, the cost is 1. Therefore, for each trip, it is sufficient to simply check whether the single square is a 0 or a 1, and output that value, to solve this subtask. Each query takes $O(1)$ time to solve.

### 1.3 Subtask 3: $N \leq 8$, $M \leq 8$ and $Q \leq 5$.

This subtask has no special constraints. The only change is that the values of $N$, $M$ and $Q$ have been reduced.

Let us analyse how many paths can exist between the starting square and the ending square on the grid. Naturally, if we choose the starting square to be the top-left hand corner (cell $(1, 1)$) and the ending square to be the bottom-right hand corner (cell $(N, M)$), then we can expect to have the most paths available. Since the maximum limit for $N$ and $M$ is 8, our objective is to count the number of possible paths from $(1, 1)$ to $(8, 8)$.

One way to think about the number of paths is to consider the move made at each step. To get from $(1, 1)$ to $(8, 8)$, we will need to move to the right 7 times, and move down 7 times, for a total of 14 moves. Out of the final sequence of 14 moves, it does not matter exactly which moves are to the right and which moves are downwards, provided that there are exactly 7 of each move type. This means that it is sufficient to count the number of ways to pick 7 downwards moves out of 14 total moves, after which the remaining 7 unpicked moves will become our rightward moves. This can be represented by $\binom{14}{7}$, which has a value of 3432, which is the maximum number of paths between any two queried points.

In general, let us define $X = C_i - A_i$ and $Y = D_i - B_i$. Then, $X$ is the number of downward moves required, and $Y$ is the number of rightward moves required. There are many ways to implement generating all paths. Perhaps the simplest way is to use a recursive function to travel through the queried region. This has been implemented below, and takes $O(\binom{X+Y}{X})$ time per query.

```
// assume that grid[i][j] represents the value in the grid for all
    1 <= i <= N and 1 <= j <= M
int best_path_cost(int posx, int posy, int c, int d) {
    if (posx == c && posy == d) {
        // this is our base case, we are at the end of the trip.
        // return the cost of only this square.
        return grid[posx][posy];
    }
    int best = 1; // to store our answer
    if (posx < c) {
        // then we can go down from here, without leaving our grid
        // remember to never go down below c, since upward moves
    are not allowed!
        best = min(best, grid[posx][posy] * best_path_cost(posx+1,
    posy, c, d));
    }
    if (posy < d) {
        // then we can go right from here, without leaving our grid
        // remember to never go right beyond d, since leftward
    moves are not allowed!
```

```
        best = min(best, grid[posx][posy] * best_path_cost(posx, posy+1, c, d));
    }
    return best;
}
// call best_path_cost(a, b, c, d);
```

A different way to implement this is to generate all numbers from 0 to $2^{X+Y}-1$, both inclusive. Then, a 1 at position $i$ of the binary representation of this number can represent moving downwards, and a 0 can represent moving to the right. Since our path should have exactly $X$ downwards moves, we should now only consider those numbers with $X$ set bits.

Then, for each such number, we can generate the path it would follow, and accumulate the product along the way, and take the minimum of these products. This has been implemented below, and takes $O((N+M)2^{N+M})$ time per query:

```
// again, assume that grid[i][j] represents the value in the grid for all 1 <= i <= N and 1 <= j <= M
int best_path_cost(int a, int b, int c, int d) {
    int x = c - a; // the total number of downward moves
    int y = d - b; // the total number of rightward moves
    int sum = x + y; // the total number of moves
    vector<int> valid_paths;
    for (int i=0; i<=(1<<sum)-1; i++) {
        // generate all numbers from 0...2^(sum)-1, both inclusive.
        int set_bits = __builtin_popcount(i);
        if (set_bits == x) {
            // there are exactly x set bits. this is a valid path description from (a, b) to (c, d).
            valid_paths.push_back(i);
        }
    }
    int best_product = 1;
    for (auto path: valid_paths) {
        // what is the product of the numbers on this path?
        int product = grid[a][b]; // since the path begins at (a, b)
        int current_x = a;
        int current_y = b;
        for (int move=0; move<sum; move++) { // for every move in the path
            if (path & (1 << move)) { // this checks whether the 'move'-th bit is set in the path
                // since the 'move'-th bit is set in the path, we are now travelling down.
                current_x++;
            }
            else {
                // or, since it's not set, we are now travelling right
                current_y++;
            }
            product *= grid[current_x][current_y]; // account for the product of this square.
        }
        assert(current_x == c);
```

```
        assert(current_y == d);
        best_product = min(best_product, product); // and update
    this with the global minimum product of all paths.
    }
    return best_product;
}
// call best_path_cost(a, b, c, d);
```

### 1.4 Subtask 4: $Q \leq 5$

The values of $N$ and $M$ are no longer low, but the low value of $Q$ remains for this subtask. Recall that the number of paths in a trip is $\binom{X+Y}{X}$. If all queries were from $(1, 1)$ to $(N, M)$, for $N = 400$ and $M = 500$, then a total of $\binom{399+499}{399}$ paths would exist. This is approximately $2 \times 10^{266}$. It should be clear that trying all possible paths should no longer be a feasible option.

Instead, this subtask asks for further observation. From here on, we consider the subgrid defined by the corners $(A, B)$ and $(C, D)$ to be the query subgrid. This entire region is the set of squares which is on at least one path from $(A, B)$ to $(C, D)$. From Subtask 1, we already know that if a grid consists of only 1s, all possible products of costs are 1 too. The opposite question remains: what if all the squares in the query subgrid are *not* 1? Then, there must be at least one cell in the query subgrid that is 0.

Of course, if it is possible to travel between $(A, B)$ and $(C, D)$ while also visiting a square with 0 cost, the cost of such a path would automatically be zero. It remains to check whether such a construction is always possible. Let us call $(X, Y)$ the position of *any* zero in the query subgrid. Then, via the following construction, we can always travel between $(A, B)$ and $(C, D)$ via $(X, Y)$:

- $(A, B)$ (from the starting square, move downwards)
- $(A + 1, B)$
- $\ldots$
- $(X - 1, B)$
- $(X, B)$ (now, move to the right)
- $(X, B + 1)$
- $\ldots$
- $(X, Y - 1)$
- $(X, Y)$ (zero square)
- $(X, Y + 1)$
- $\ldots$
- $(X, D - 1)$

- $(X, D)$ (now, move down again)
- $(X + 1, D)$
- ...
- $(C - 1, D)$
- $(C, D)$ (and we have reached the ending square)

Then, it follows that the necessary and sufficient condition for the optimal cost to be 0 is the existence of a 0 in the queried subgrid. This has been implemented below, and takes $O(NM)$ time per query:

```
// again, assume that grid[i][j] represents the value in the grid
    for all 1 <= i <= N and 1 <= j <= M

int best_path_cost(int a, int b, int c, int d) {
    for (int i=a; i<=c; i++) {
        for (int j=b; j<=d; j++) {
            if (grid[i][j] == 0) {
                // there's a zero in range! the answer is then 0.
                return 0;
            }
        }
    }
    // there is no zero in the grid, the answer is now 1.
    return 1;
}
```

## 1.5 Subtask 5: At most 10 cells in the grid have a cost of 0.

Now have a subtask where at most 10 cells have a cost of 0. Recall that our objective is to check, for each query whether there exists a 0 in the query box.

Since there are far too many queries and the grids queried can be large, it is no longer possible to search through each query subgrid to verify whether there exists a 0 in the subgrid.

However, the rather low number of 0s in the entire grid is very promising. To solve this subtask, we simply reverse the process: rather than scanning through each square in each grid and checking if the value is 0, we check for each zero, whether it is in the query subgrid. We can store the positions of the zeroes in the grid, and check if they exist in the query subgrid. This has been implemented below, and takes $O(1)$ time per query:

```
// assume that zeroes is a vector of pairs, where each pair is the
    coordinate of one of the zero positions in the grid.
// by the specification of this subtask, the size of zeroes is at
    most 10.
int best_path_cost(int a, int b, int c, int d) {
   for (auto pos: zeroes) {
       if (a <= pos.first && pos.first <= c) {
           if (b <= pos.second && pos.second <= d) {
```

```
                // this zero is in the subgrid!
                return 0;
            }
        }
    }
    // no zeroes in the subgrid, answer is 1.
    return 1;
}
```

### 1.6 Subtask 6: $N = 1$

This subtask has $N = 1$, which means that the input is no longer a grid, but an array. The task remains how to check whether there exists a 0 within a queried subarray. For simplicity, we will denote $L = B$ and $R = D$ in this explanation, to remain consistent with the variables used in arrays.

*Approach 1:* First, we will describe an approach which builds on from the preceding subtask. Again, store a vector of all the positions which have a value of 0. Then, sort this vector. For each query from $L$ to $R$, we binary search for the earliest item in our vector which is greater than or equal to $L$. Then there are two cases:

- There is no such item greater than or equal to $L$. This means that there is no 0 value on or after position $L$ in the array, and therefore there is no 0 in the range $[L, R]$. Then, the answer to the query must be 1.
- There is an earliest item which is greater than or equal to item $L$. Let's call this item $X$. We further subdivide the analysis into two more cases:
  - Either $X \leq R$, in which case $X$ occurs in the range $[L, R]$. Then, we have found a 0 in the range, and the answer to our query is 0.
  - Otherwise, $X > R$. Then, as $X$ is also the earliest 0 after $L$, there cannot be any other zeroes between $L$ and $R$. Then the answer to the query must be 1.

Here is some code implementing this approach, which takes $O(\log M)$ time per query:

```
// assume that zeroes is a vector of zero positions
int query(int l, int r) {
    auto x = lower_bound(zeroes.begin(), zeroes.end(), l);
    if (x == zeroes.end()) return 0;
    if (*x > r) return 1;
    return 0;
}
```

*Approach 2:* Second, we will describe an approach which builds on easily to the full solution. Consider making a new array $pf$, of length $M + 1$ where all indices are initially set to zero. Then, for every 0 in the input array, set the corresponding index to 1. Finally, iterate over the indices 1 to $M$ in this array, and at each step, add the value of the preceding index to the value of the current index.

This resulting structure is known as a prefix sum array. In particular, $pf[X]$ represents the number of indices $i \leq X$ such that the value at position $i$ of the array is 0. This makes it extremely easy to answer queries about the number of 0s in a range - we simply find $pf[R]$ to find the number of zeroes in the first $R$ positions, and subtract $pf[L-1]$, which represents the number of zeroes strictly before the $L$th position. If this value of $pf[R] - pf[L-1] > 0$, we know that there must be a 0 between the two indices, and therefore the answer to our query is 0, and otherwise the answer to our query must be 1. Here is some code implementing this approach, which takes $O(1)$ time per query:

```
vector<int> pf;

void construct(int arr[], int m) {
    pf = vector(m+1, 0);
    for (int i=1; i<=m; i++) {
        if (arr[i] == 0) pf[i] = 1;
    }
    for (int i=1; i<=m; i++) {
        pf[i] = pf[i-1] + pf[i];
    }
}

int query(int l, int r) {
    if (pf[r] - pf[l-1] > 0) return 0;
    return 1;
}
```

## 1.7 Subtask 7: $N \leq 5$

This subtask again presents a very low value of $N$. To solve this subtask, it is sufficient to pick any method to solve Subtask 6, and then use the method $N$ times (once for each row) per query. If you choose to perform approach 1 multiple times, the resulting time complexity is $O(N \log M)$ per query, and if you choose to perform approach 2 multiple times, the resulting time complexity is $O(N)$, per query.

## 1.8 Subtask 8: No additional constraints.

After solving the first 7 subtasks, there are many ways to arrive at the full solution.

*Approach 1:* The most convenient way to solve this subtask is to generalise the second approach in Subtask 6. What we did in Subtask 6 was to construct prefix sums in one dimension, and Subtask 8 can be solved by constructing prefix sums in two dimensions. This time, $pf[X][Y]$ will represent the number of 0s in the rectangle on the top left corner from $(1, 1)$ to $(X, Y)$. Again, we begin by setting all entries in the $pf$ array as 0, and setting all 0 positions in the 2 dimensional input as 1. Then, you can use $pf[X][Y] = pf[X-1][Y] + pf[X][Y-1] - pf[X-1][Y-1]$ to calculate the resepctive values in the prefix sum array. Answering a query involves checking whether $pf[C][D] - pf[A -$

$1][D] - pf[C][B-1] + pf[A-1][B-1]$ is strictly positive. If it is, then the answer is 0, otherwise the answer is 1. This solution works in $O(1)$ per query, and is implemented below:

```
vector<vector<int>> pf;

void construct(vector<vector<int>> arr, int n, int m) {
    pf = vector<vector<int>>(n+1, vector<int>(m+1));
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=m; j++) {
            pf[i][j] = pf[i-1][j] + pf[i][j-1] - pf[i-1][j-1];
            if (arr[i][j] == 0) pf[i][j]++;
        }
    }
}

int query(int a, int b, int c, int d) {
    if (pf[c][d] - pf[a-1][d] - pf[c][b-1] + pf[a-1][b-1] > 0)
    return 0;
    return 1;
}
```

A different way to construct prefix sums in two dimensions is to first construct them horizontally in one dimension, and then to use the columns of the resulting prefix sums to again construct a one dimensional prefix sums, vertically.

*Approach 2:* Another way to solve the full problem is to build onto the ideas from Subtask 7. Subtask 7 is rather easy to solve since we are guaranteed that $N$ is quite small. But in the full problem, $N$ can be large. Notice that despite $N$ being large, $NM$ is still constrained by $2 \times 10^5$. This means that for $N$ to be large, $M$ must become small to compensate!

This observation motivates us to find a rather special property of these constraints. In particular, the value of $min(N, M)$ can be at most 447. It is easy to see why this is true: first assume that the minimum of $N$ and $M$ is not at most 447. then the minimum must be at least 448. Then, we can say that $N \geq 448$ and $M \geq 448$. Then, $NM \geq 448 \times 448 = 200704$. However, $NM \leq 2 \times 10^5$, according to the problem statement. This is a contradiction, and therefore the minimum of $N$ and $M$ must be at most 447.

Using this, we can apply the same solution to Subtask 7 to solve Subtask 8. If $N \leq M$, then we can construct prefix sums over rows, and solve for Subtask 7. Otherwise, we can instead construct prefix sums over columns, and solve for Subtask 7 again, but instead of iterating over rows in each query, we iterate over columns. This solution works in $O(\sqrt{NM})$ per query. Alternatively, we can also use the binary search solution in the same manner: construct zero position lists over each column or each row, depending on which is fewer. This solution works in $O(\sqrt{NM} \log(NM))$ per query.

The prefix sum version of this solution is implemented below:

```
vector<vector<int>> pf;

void construct(vector<vector<int>> arr, int n, int m) {
    pf = vector<vector<int>>(n+1, vector<int>(m+1));
```

```
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=m; j++) {
            if (arr[i][j] == 0) pf[i][j] = 1;
        }
    }
    if (n <= m) {
        // do row-wise prefix sums
        for (int i=1; i<=n; i++) {
            for (int j=1; j<=m; j++) {
                pf[i][j] += pf[i][j-1];
            }
        }
    }
    else {
        // do column-wise prefix sums
        for (int j=1; j<=m; j++) {
            for (int i=1; i<=n; i++) {
                pf[i][j] += pf[i-1][j];
            }
        }
    }
}

int query(int a, int b, int c, int d) {
    if (n <= m) {
        // use row-wise
        int sum = 0;
        for (int i=a; i<=c; i++) sum += pf[i][d] - pf[i][b-1];
        if (sum == 0) return 1;
        return 0;
    }
    else {
        // use column wise
        int sum = 0;
        for (int i=b; i<=d; i++) sum += pf[c][i] - pf[a-1][i];
        if (sum == 0) return 1;
        return 0;
    }
}
```

# 2 Problem 2: Messages

*Abridged statement:* There are $M = 2^N$ messages, each of length $N$. Message 0 is the original message. For all other messages, message $i$ is the same as message 0 in columns corresponding to the '0' bits of the binary representation of $i$, and different from message 0 in other columns. The messages are then arbitrarily shuffled. Find which message(s) could have been message 0, or that none exist.

The following subtasks are available:

- Subtask 1 [11 points]: $N = 1$, $M = 2$
- Subtask 2 [16 points]: $N \leq 7$, $M \leq 128$, $\sum M \leq 128$
- Subtask 3 [15 points]: $N \leq 11$, $M \leq 2048$, $\sum M \leq 2048$
- Subtask 4 [10 points]: $0 \leq A_{i,j} \leq 1$ for all $(i, j)$, $K \geq 1$
- Subtask 5 [14 points]: $0 \leq A_{i,j} \leq 1$ for all $(i, j)$
- Subtask 6 [12 points]: $K = 1$
- Subtask 7 [14 points]: $0 \leq K \leq 1$
- Subtask 8 [8 points]: No additional constraints.

## 2.1 Subtask 1: $N = 1$, $M = 2$

For this subtask, we notice that each message has length 1. Moreover, the binary representation of 1 is $1_2$. Therefore, we notice that the single fake message must be different from the original message. Moreover, if the two messages are different, then either one could have been the original message! So, we simply check if the two messages are different, and output both messages if so and "0" otherwise.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  if(a[0][0] == a[1][0]) cout<<"0\n\n"; //both messages same
  else cout<<"2\n0 1\n"; //both messages different
}
```

## 2.2 Subtask 2: $N \leq 7$, $M \leq 128$, $\sum M \leq 128$

This subtask is given for brute force solutions of cubic complexity. Let's check if each message could have been the original message. To do so, we attempt to reconstruct the original list of messages before the shuffling. We can do this by iterating through all $i$ from 1 to $2^N - 1$ and check through all the messages to see whether there is a message that satisfies that $i$. This solution works in $O(M^3N)$ since, for each of the $M$ possible original messages, we are reconstructing the unshuffled list of $M$ messages, and for each message in this list we check all $M - 1$ messages, in $O(N)$ each.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  vector<int> ans; //vector of possible original messages
  for(int i=0; i<m; i++){
    //i is the original message we check
    //message 0 in the unshuffled list already decided
    bool yes=1; //true if i may be the original message
    for(int j=1; j<m; j++){
      bool exists=0;
      //we find message j of the unshuffled list
      for(int k=0; k<m; k++){
        //we check if k was j in the unshuffled list
        bool b[n];
        //b[l]=0 if a[k][l]=a[i][l], 1 otherwise
        for(int l=0; l<n; l++){
          b[l] = (a[k][l]!=a[i][l]);
          //fancy way of doing the above
        }
        int den=0; //convert b[] into denary
        for(int l=0; l<n; l++) den += (1<<l)*b[l];
        //note: 1<<l is 2^l
        if(den==j) exists=1; //we found unshuffled j
      }
      if(!exists) yes=0;
      //if unshuffled j doesn't exist
      //then i cannot be the original message
    }
    if(yes) ans.push_back(i);
    //i could be the original message
  }
  cout<<ans.size()<<'\n'; //# original messages
  for(int i : ans) cout<<i<<' '; //output all
  cout<<'\n';
}
```

### 2.3 Subtask 3: $N \leq 11$, $M \leq 2048$, $\sum M \leq 2048$

This subtask requires a quadratic complexity solution. Essentially, what we can aim to do is optimise our previous solution. The easiest part to optimise is the reconstruction of the unshuffled list. To do this, we observe that for each message, given an original message, there is exactly one message in the unshuffled list that it could correspond to. Therefore, for each of $M$ possible original messages, we iterate through all $M-1$ other messages and for a message $i$, we find which message in the unshuffled list $i$ corresponds to in $O(N)$. Then, check if all unshuffled-list messages have been produced. The final time complexity is $O(M^2N)$.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  vector<int> ans; //vector of possible original messages
  for(int i=0; i<m; i++){
    //we check if i could be the original message
    bool exis[m]; //exis[j] true if unshuffled-j exists
    memset(exis,0,sizeof(exis)); //set exis to 0
```

```
    for(int j=0; j<m; j++){
      int b[n];
      //b[k]=0 if a[j][k]=a[i][k], 1 otherwise
      for(int k=0; k<n; k++){
        b[k] = (a[j][k]!=a[i][k]);
        //fancy way of doing the above
      }
      int den=0; //convert b[] into denary
      for(int k=0; k<n; k++) den += (1<<k)*b[k];
      //note: 1<<k is 2^k
      exis[den]=1;
      //j is message "den" in unshuffled list
    }
    bool yes=1;
    //true if message i could be original message
    for(int j=0; j<m; j++){
      yes &= exis[j];
      //check if all exis[j] are true
    }
    if(yes) ans.push_back(i);
  }
  cout<<ans.size()<<'\n'; //# original messages
  for(int i : ans) cout<<i<<' '; //output all
  cout<<'\n';
}
```

### 2.4 Subtask 4: $0 \leq A_{i,j} \leq 1$ for all $i$, $K \geq 1$

We may observe from the sample that in Subtask 4 all messages could be the original message, so we only need to output all messages. Details of why this is the case are described in the Subtask 5 solution.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  cout<<m<<'\n';
  for(int i=0; i<m; i++) cout<<i<<' ';
  cout<<'\n';
  //output all messages because
  //all messages could have been the original message
}
```

### 2.5 Subtask 5: $0 \leq A_{i,j} \leq 1$ for all $i$

Let the original message be $00\ldots00$ and consider the other messages that would be produced. We can see that, if $B_{i,j} = 1$, then $A_{i,j} = 1$, because the only value possible other than 0 is 1, and obviously if $B_{i,j} = 0$, $A_{i,j} = 0$. In other words, we see that in this case, $A_{i,j} = B_{i,j}$ for all $(i, j)$, and therefore the message $i$ in the unshuffled list will correspond to the number $i$ in binary.

Next, let's consider what happens with a different original message. Say the message is $00\ldots001$. This is similar to the previous case, in that $A_{i,j} = B_{i,j}$ for all $(i, j)$ except when $j$ is the last bit. When $j$ is the last bit, $B_{i,j} = \neg A_{i,j}$. In other words, the last bit is flipped. Now, notice that the set of messages that

will be produced is exactly isomorphic to the set of messages that was produced in the previous case. In fact, it can be proven that for any base message, the set of messages produced will be exactly the same, and it is equivalent to the numbers 0 to $2^N - 1$ in binary.

Remaining is the implementation. Simply create an array $exis$ of length $2^N$, with $exis[i]$ denoting if the number $i$ in binary appears in the set of messages. If $exis[i]$ is true for all $i$, then the answer is all messages. Otherwise, there is no answer.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  bool exis[m];
  //exis[i] is true if unshuffled message i exists
  memset(exis,0,sizeof(exis)); //set all to 0
  for(int i=0; i<m; i++){
    //convert a[i] to denary
    int den=0;
    for(int j=0; j<n; j++) den += (1<<j)*a[i][j];
    //note: 1<<j is 2^j
    exis[den]=1;
  }
  bool all=1; //true if exis[i] is true for all i
  for(int i=0; i<m; i++){
    all &= exis[i];
  }
  if(all){
    cout<<m<<'\n';
    for(int i=0; i<m; i++){
      cout<<i<<' ';
    }
    cout<<'\n';
    //all bitmasks exist
    //and so we output all messages
    //due to mirroring
  }
  else{
    cout<<"0\n\n";
    //all bitmasks do not exist
    //so the message list is invalid
  }
}
```

## 2.6 Subtask 6: $K = 1$

In this subtask, we are guaranteed that there is only one original message. The way to think of this is that we only need to find one original message. So what would the original message be?

If we look at the process to produce the messages once again, we notice that either $A_{i,j} = A_{0,j}$ or $A_{i,j} \neq A_{0,j}$. Intuitively, we notice that under the latter case, $A_{i,j}$ can be one of many values, whereas under the former case, $A_{i,j}$ can be only one value. So, if we consider the values $A_{i,j}$ for fixed $j$, it seems reasonable to guess that $A_{0,j}$ will be the most common of these values. Therefore, our algorithm is simple: for each position $j$, the value that appears most times is

the $j$-th position of the original message. After finding the original message, we only need to find which message in the shuffled sequence corresponds to the original message.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  int freq[n][n+1];
  //freq[i][j] is the frequency of value j in column i
  //note that values are in range [0,n]
  memset(freq,0,sizeof(freq)); //set frequencies to 0
  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
      freq[j][a[i][j]]++;
    }
  }

  pair<int,int> mc[n];
  //most common value in each column
  //mc[i].first is the frequency
  //mc[i].second is the value
  memset(mc,-1,sizeof(mc));

  for(int i=0; i<n; i++){
    for(int j=0; j<=n; j++){
      mc[i]=max(mc[i],{freq[i][j],j});
      //if freq[i][j] >= mc[i].first
      //then set mc[i].first to freq[i][j]
      //and set mc[i].second to j
    }
  }

  int ans=-1; //the original message
  for(int i=0; i<m; i++){
    bool yes=1;
    //true if i is the original message
    for(int j=0; j<n; j++){
      if(a[i][j] != mc[j].second) yes=0;
      //the value must be the most common
    }
    if(yes) ans=i;
  }
  cout<<"1\n"; //only one answer, as k=1
  cout<<ans<<'\n';
}
```

We may also observe further, by the properties of binary numbers (which were used when solving Subtask 5), that $A_{0,j}$ appears exactly $\frac{M}{2}$ times in the list of values $A_{i,j}$ for fixed $j$. This may simplify implementation for this subtask and help in the solving of Subtasks 7 and 8.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  int freq[n][n+1];
  //freq[i][j] is the frequency of value j in column i
  //note that values are in range [0,n]
  memset(freq,0,sizeof(freq)); //set frequencies to 0
  for(int i=0; i<m; i++){
```

```
    for(int j=0; j<n; j++){
      freq[j][a[i][j]]++;
    }
  }

  int ans=-1; //the original message
  for(int i=0; i<m; i++){
    bool yes=1;
    //true if i is the original message
    for(int j=0; j<n; j++){
      if(freq[j][a[i][j]] != m/2) yes=0;
      //for a[i][j] to be the most commmon value
      //frequency of a[i][j] in column j
      //must be exactly m/2
    }
    if(yes) ans=i;
  }
  cout<<"1\n"; //only one answer, as k=1
  cout<<ans<<'\n';
}
```

## 2.7 Subtask 7: $0 \leq K \leq 1$

In this subtask, we need to find one original message if it exists, or find that the list of messages is not valid. It seems reasonable, therefore, to build upon our solution for Subtask 6.

Let's say we've already done the "most common" check to find the original message. Now, we need to check if this message really is the original message. It is clear that we should use the brute force check from Subtask 3 to do this. Since we are only checking one message, this runs in $O(MN)$ (i.e., linear time).

Another way to characterise this method is that it is a generalisation of the Subtask 5 solution. We observe that even in general $A_{i,j}$ there is a sort of duality between $B_{i,j}$ and $A_{i,j}$. Essentially, we can reduce each message to the binary number by setting $B_{i,j} = 1$ if $A_{i,j} \neq A_{0,j}$, and $B_{i,j} = 0$ otherwise. Now it is clear that this solution is essentially the same as the process which was used to construct the messages in the first place, and so it should work.

Note: We do not know of fundamentally different solution methods to this subtask from the one we describe above. For example, the method "check if all messages (rather than binary reductions) are distinct, then check that each value appears exactly $\frac{M}{2}$ times" fails on such a case:

```
1
3 8
0 0 0
0 0 2
0 1 0
0 0 1
2 2 0
1 0 1
1 1 0
```

```
1 1 1
```

The actual output of this testcase is 0 (i.e., the list of messages is invalid), but most incorrect solutions will claim that message 0 was the original message. The reason for this is that this case maintains that the most common element in each position has frequency $\frac{M}{2}$, and that all messages are distinct. However, the binary reductions are not all distinct, so this message could never be produced by the process described. "0 0 2" and "0 0 1" are equivalent, and "2 2 0" and "1 1 0" are equivalent.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  int freq[n][n+1];
  //freq[i][j] is the frequency of value j in column i
  //note that values are in range [0,n]
  memset(freq,0,sizeof(freq)); //set frequencies to 0
  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
      freq[j][a[i][j]]++;
    }
  }

  pair<int,int> mc[n];
  //most common value in each column
  //mc[i].first is the frequency
  //mc[i].second is the value
  memset(mc,-1,sizeof(mc));

  for(int i=0; i<n; i++){
    for(int j=0; j<=n; j++){
      mc[i]=max(mc[i],{freq[i][j],j});
      //if freq[i][j] >= mc[i].first
      //then set mc[i].first to freq[i][j]
      //and set mc[i].second to j
    }
  }

  int val[n];
  for(int i=0; i<n; i++) val[i] = mc[i].second;

  //we have now found the value of each position
  //of the original message and stored in val[]

  //now we find if all bitmasks exist
  //this is done using the exis[] method
  //previously used in subtasks 3 and 5

  bool exis[m];
  memset(exis,0,sizeof(exis));
  for(int i=0; i<m; i++){
    int b[n];
    //b[j]=0 if a[i][j]=val[j]
    //b[j]=1 if a[i][j]!=val[j]
    for(int j=0; j<n; j++){
      b[j] = (a[i][j]!=val[j]);
      //fancy method of doing the above
```

```
    }
    int den=0; //convert b[] into denary
    for(int j=0; j<n; j++) den += (1<<j)*b[j];
    //note: 1<<j is 2^j
    exis[den]=1;
    //j is message "den" in unshuffled list
  }

  //check if all bitmasks exist
  bool yes=1; //true if all bitmasks exist
  for(int i=0; i<m; i++){
    yes &= exis[i];
    //check if exis[i] is true
  }

  if(!yes){
    cout<<"0\n\n"; return;
    //all bitmasks do not exist
    //so the list is invalid
  }

  int ans=-1; //the original message
  for(int i=0; i<m; i++){
    bool yes=1;
    //true if i is the original message
    for(int j=0; j<n; j++){
      if(a[i][j] != val[j]) yes=0;
      //the value must be the most common
    }
    if(yes) ans=i;
  }
  cout<<"1\n"; //only one answer, as k=1
  cout<<ans<<'\n';
}
```

## 2.8 Subtask 8: No additional constraints.

For this subtask, we can once again build upon our previous solution. From our Subtask 7 solution, we know how to find one of the possible original messages, and also how to verify if the list of messages is valid. Now we just need to find any other possible original messages.

One useful observation was discussed previously in Subtask 5: that $A_{0,j}$ appears exactly $\frac{M}{2}$ times in the list of values $A_{i,j}$ for fixed $j$. So, we can notice that there may be a maximum of two values that both appear exactly $\frac{M}{2}$ times. Now, we can see that *either* of these values may have been the value of the $j$-th position in the original message. Similar to the "mirroring" observation of Subtask 5, we may note that any message which has the most common value in each position will work the same as the message that we brute-force verified. Here is a rough proof: consider any position $P$ with 2 candidate values $A$ and $B$ such that there exists a valid original message with $A$ and not $B$. Then you can take the same original message for other values, and consider $B$ at that position. Now looking at the 'difference' bitmasks, you just flipped the $P$-th bit

in each of them. So now, you still have all the bitmasks from $0 \ldots 2^N - 1$, and so this is also a valid solution.

```
//let a[i][j] be position j of message i (all 0-indexed)
void original_messages(int n, int m){
  //Step 1: Find most common value in each position
  //1.1: Find frequencies of each value in each column
  int freq[n][n+5]; //frequency of value j in column i
  //note j is in the range [0,n]
  memset(freq,0,sizeof(freq)); //set all to 0
  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
      freq[j][a[i][j]]++;
      //increment the frequency of value a[i][j]
      //in column j
    }
  }

  //1.2: Find most common value in each position
  //it must have frequency m/2
  //if there are multiple, choose arbitrarily
  int mc[n];
  memset(mc,-1,sizeof(mc)); //set all to -1
  //if there is no element with frequency m/2 in pos i
  //mc[i] = -1, and the message-list is invalid
  for(int i=0; i<n; i++){
    for(int j=0; j<=n; j++){
      if(freq[i][j]==m/2) mc[i]=j;
    }
  }

  //1.3: Check if the most common value in each position
  //has frequency m/2
  for(int i=0; i<n; i++){
    if(mc[i]==-1){
      cout<<"0\n\n"; return;
      //the message-list is invalid
    }
  }

  //Step 2: Check if all bitmasks exist.
  //if so, the message-list is valid
  //2.1: Reverse-engineer the B array from the statement.
  int b[m][n];
  //if a[i][j] = a[orig][j], b[i][j]=0
  //if a[i][j] != a[orig][j], b[i][j]=1
  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
      b[i][j] = (a[i][j] != mc[j]);
    }
  }

  //2.2: Check if B contains all binary numbers 0...(2^N)-1
  //2.2.1: Create exis[] array
  bool exis[m]; //exis[i] true if message i exists
  memset(exis,0,sizeof(exis)); //initialise all to false
  for(int i=0; i<m; i++){
    //convert b[i] to denary
```

```
    int den=0;
    for(int j=0; j<n; j++){
      den += (1<<j)*b[i][j];
      //this adds 2^j * j-th bit
    }
    exis[den] = 1;
  }
  //2.2.2: Ensure exis[i] true for all i
  int yes=1; //true if all exis[i] true
  for(int i=0; i<m; i++){
    if(exis[i]==0) yes=0;
    //if exis[i] is false
    //then not all exis[i] are true
    //and the message-list is invalid
  }

  if(yes==0){ //message list invalid
    cout<<"0\n\n";
    return;
  }

  //Step 3: Find all possible original messages.
  vector<int> ans; //list of answers
  for(int i=0; i<m; i++){
    bool can=1;
    //true if message i could be original
    //must have most common value in each pos
    for(int j=0; j<n; j++){
      if(freq[j][a[i][j]] != m/2) can=0;
      //frequency of a[i][j] in column j
      //must be exactly m/2
      //for it to be most common
    }
    if(can) ans.push_back(i);
    //add i to list of answers
  }
  cout<<ans.size()<<'\n'; //output # answers
  for(int i : ans) cout<<i<<' '; //output all
  cout<<'\n';
}
```

# 3 Contest Statistics

## 3.1 Subtask distribution

### 3.1.1 Problem 1: Vacation

A total of 378 contestants submitted at least one solution to this problem.

- Subtask 1 was solved by 242 participants.
- Subtask 2 was solved by 260 participants.
- Subtask 3 was solved by 249 participants.
- Subtask 4 was solved by 241 participants.
- Subtask 5 was solved by 128 participants.
- Subtask 6 was solved by 87 participants.
- Subtask 7 was solved by 76 participants.
- Subtask 8 was solved by 60 participants.

### 3.1.2 Problem 2: Messages

A total of 207 contestants submitted at least one solution to this problem.

- Subtask 1 was solved by 127 participants.
- Subtask 2 was solved by 39 participants.
- Subtask 3 was solved by 29 participants.
- Subtask 4 was solved by 49 participants.
- Subtask 5 was solved by 33 participants.
- Subtask 6 was solved by 26 participants.
- Subtask 7 was solved by 4 participants.
- Subtask 8 was solved by 0 participants.

## 3.2 Score distribution

### 3.2.1 Problem 1: Vacation

| Score | Contestants at this score | Contestants at or above this score |
|---|---|---|
| 100 | 60 | 60 |
| 91 | 9 | 69 |
| 83 | 7 | 76 |
| 73 | 43 | 119 |
| 72 | 4 | 123 |
| 67 | 11 | 134 |
| 64 | 4 | 138 |
| 54 | 59 | 197 |
| 53 | 1 | 198 |
| 49 | 40 | 238 |
| 48 | 2 | 240 |
| 43 | 2 | 242 |
| 34 | 4 | 246 |
| 28 | 1 | 247 |
| 23 | 2 | 249 |
| 11 | 15 | 264 |
| 6 | 11 | 275 |
| 5 | 25 | 300 |
| 0 | 78 | 378 |

### 3.2.2 Problem 2: Messages

| Score | Contestants at this score | Contestants at or above this score |
|---|---|---|
| 100 | 0 | 0 |
| 92 | 4 | 4 |
| 78 | 7 | 11 |
| 66 | 7 | 18 |
| 63 | 3 | 21 |
| 54 | 1 | 22 |
| 52 | 2 | 24 |
| 51 | 1 | 25 |
| 47 | 7 | 32 |
| 42 | 8 | 40 |
| 37 | 2 | 42 |
| 35 | 4 | 46 |
| 33 | 4 | 50 |
| 27 | 4 | 54 |
| 21 | 6 | 60 |
| 11 | 67 | 127 |
| 10 | 2 | 129 |
| 0 | 78 | 207 |

### 3.2.3 ZCO 2022 Full Score distribution

| Score | Contestants at this score | Contestants at or above this score |
|---|---|---|
| 200 | 0 | 0 |
| 192 | 4 | 4 |
| 178 | 7 | 11 |
| 166 | 7 | 18 |
| 163 | 3 | 21 |
| 154 | 1 | 22 |
| 152 | 1 | 23 |
| 151 | 1 | 24 |
| 147 | 6 | 30 |
| 143 | 1 | 31 |
| 142 | 4 | 35 |
| 135 | 3 | 38 |
| 133 | 3 | 41 |
| 130 | 1 | 42 |
| 127 | 2 | 44 |
| 126 | 1 | 45 |
| 115 | 1 | 46 |
| 111 | 16 | 62 |
| 109 | 1 | 63 |
| 104 | 1 | 64 |
| 102 | 4 | 68 |
| 100 | 5 | 73 |
| 94 | 9 | 82 |
| 91 | 2 | 84 |
| 87 | 2 | 86 |
| 84 | 14 | 100 |
| 83 | 5 | 105 |
| 82 | 1 | 106 |
| 81 | 1 | 107 |
| 78 | 1 | 108 |
| 75 | 1 | 109 |
| 73 | 22 | 131 |
| 72 | 1 | 132 |
| 70 | 1 | 133 |
| 67 | 8 | 141 |
| 65 | 19 | 160 |
| 64 | 3 | 163 |
| 60 | 2 | 165 |
| 59 | 1 | 166 |
| 54 | 36 | 202 |
| 53 | 1 | 203 |

| Score | Contestants at this score | Contestants at or above this score |
|---|---|---|
| 49 | 36 | 239 |
| 48 | 1 | 240 |
| 44 | 1 | 241 |
| 43 | 2 | 243 |
| 34 | 3 | 246 |
| 28 | 1 | 247 |
| 23 | 2 | 249 |
| 22 | 2 | 251 |
| 21 | 1 | 252 |
| 11 | 12 | 264 |
| 6 | 11 | 275 |
| 5 | 25 | 300 |
| 0 | 84 | 384 |

### 3.3 Cutoffs

| Grade Level | Male Cutoff | Female Cutoff |
|---|---|---|
| 12 | 94 | 87 |
| 11 | 83 | 54 |
| 10 | 72 | 48 |
| 9 | 65 | 43 |
| 8 | 49 | 28 |
| 7 and below | 34 | 22 |