First Round SOI 2020

Solution Booklet

Swiss Olympiad in Informatics

15 September – 30 November 2019

Marathon

Task Idea	Johannes Kapfhammer
Task Preparation	Erwan Serandour
Description English	Erwan Serandour
Description German	Joël Mathys
Description French	Erwan Serandour
Solution	Jan Schär
Correction	Erwan Serandour

Mouse Binna wants to organize a marathon on a very long road. However, she found out that there are holes in the road. Your task was to help her organize the longest possible marathon without holes on the track.

Subtask 1: A short road (20 Points)

Subtask 2: A Longer Road (20 points)

Subtask 3: A very long Road (20 points)

The first three subtasks are the same, except that the part of the road we see gets longer, so we can use the same solution for all.

As input we are given the length of the road N, and for each of the N pieces of road a number that is 1 if it has a hole, 0 otherwise. We have to calculate the length of the longest section of the road without holes.

The simplest solution is to look at every piece of road in order, and keep a counter that is reset to zero when we see a hole. The answer is then just the maximum value that the counter had.

This algorithm is correct because for each position, we consider the longest possible track ending there.

Running time: $O(N)$, because we look at each piece only once. Space usage: $O(1)$.

Here is a C++ implementation of this solution:

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

int main () {
  // Speed up input
  ios_base::sync_with_stdio(false);
  cin.tie(0);

  int T; cin >> T;
  for (int t = 0; t < T; ++t) {
    int N; cin >> N;
    int length = 0;
    int maxLength = 0;
    for (int i = 0; i < N; ++i) {
      int pi; cin >> pi;
      if (pi == 0) length += 1;
      else length = 0;
      maxLength = max(length, maxLength);
    }
    cout << "Case #" << t << ": " << maxLength << "\n";
  }
}
```

Subtask 4: Fix the road (20 points)

Subtask 5: Marathon (20 points)

Again, the two subtasks only differ in the length of the road.

Our task is similar to before, but we are given an additional number K as input: The number of holes that we can repair. You could think of the previous subtasks as a special case of this with $K = 0$.

We can extend our previous solution by adding a counter for the number of holes in the track we are considering. When this counter gets larger than K, we have to reduce the length of the track at the start until we see a hole, and then reduce it by one more step. Instead of storing the length of the track in a variable, we store its start position (this is actually just a different way of representing the same information).

Running time: Still $O(N)$. Space usage: $O(N)$, because now we are loading the complete input into memory.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  for (int t = 0; t < T; ++t) {
    int N, K; cin >> N >> K;
    vector<int> p(N);
    for (int &pi : p) cin >> pi;
    int start = 0;
    int holes = 0;
    int maxLength = 0;
    for (int end = 0; end < N; ++end) {
      if (p[end] == 1) holes += 1;
      if (holes > K) {
        // too many holes, move start forward
        while (p[start] == 0) start += 1;
        start += 1;
        holes -= 1;
      }
      maxLength = max(end + 1 - start, maxLength);
    }
    cout << "Case #" << t << ": " << maxLength << "\n";
  }
}
```

We can reduce the space usage to $O(K)$, because at any point we only need to know the positions of the last $K + 1$ holes. This optimization was not necessary for the contest.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  for (int t = 0; t < T; ++t) {
    int N, K; cin >> N >> K;
    deque<int> parts;
    parts.push_back(0);
    int maxLength = 0;
    for (int i = 0; i < N; ++i) {
      int pi; cin >> pi;
      if (pi == 1) {
        parts.push_back(i + 1);
        if (parts.size() > K + 1) parts.pop_front();
      }
      maxLength = max(i + 1 - parts.front(), maxLength);
    }
    cout << "Case #" << t << ": " << maxLength << "\n";
  }
}
```

Stickers

Task Idea	Johannes Kapfhammer
Task Preparation	Yunshu Ouyang, Daniel Rutschmann
Description English	Yunshu Ouyang
Description German	Joël Mathys
Description French	Yunshu Ouyang
Solution	Yunshu Ouyang
Correction	Timon Gehr, Daniel Rutschmann, Yunshu Ouyang

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

In the task, we are given N positive integers $a_0, \cdots, a_{N-1}$ and we would like to change them to numbers $b_0, \cdots, b_{N-1}$ such that the set of values $\{b_0, \cdots, b_{N-1}\}$ has size at most k. The problem was presented as having a collection of stickers and exchanging stickers so that all stickers belonged to some families.

Changing one a_i into b_i has a predefined cost. Our goal is to choose the b_i's such that the total cost is minimal.

Subtask 1: Subtask 1 (20 Points)

In this first part, $N = 3$, $K = 1$ and the cost of exchanging a to b is always 1. In this case a brute force solution would work. Since $N = 3$ and we would like all numbers to be equal while minimizing the number of trades, we know that:

- If all three numbers are equal, the answer is 0 as we do not need to change anything.
- If two of the numbers are equal, the answer is 1 as we need to change the third number to be equal to the two others.
- If all three numbers are different, then we change two numbers so that they equal the third one. The answer if therefore 2.

In all three cases we can notice that the answer is always the number of different values minus 1. The solution therefore works in constant time and using constant amount of memory.

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

int main(){
  int T; cin >> T;
  for(int t=0;t<T;t++){
    int n, k;
    cin >> n >> k;
    set<int> S;
    for(int i=0;i<n;i++){
      int x; cin >> x;S.insert(x);
    }
    cout << "Case #" <<t << ": " << S.size()-1 << "\n";
  }
  return 0;
}
```

Subtask 2: Subtask 2 (20 points)

This subtask differs from the first one by only the number of stickers. We would like to change the numbers in such a way that all numbers are equal while using a minimum number of exchanges. In other words, we need to find a number x such that the number of values different from x is minimal. How can we choose such an x? x has to be equal to one of the values with the maximum number of occurrences. In this case the solution would then be N minus the maximum number of occurrences.

In order to find the maximum number of occurrences of a single value, we can use a hash map which stores for every value its number of occurrences. This solution works in $O(N)$ time complexity and $O(N)$ space complexity.

```cpp
int T;
cin >> T;
for(int t=0;t<T;t++){
  int n, k;
  cin >> n >> k;
  map<int,int> M;
  int maxi = 0;
  for(int i=0;i<n;i++){
    int x; cin >> x;
    maxi = max(maxi, ++M[x]);
  }
  cout << "Case #" <<t << ": " << n - maxi << "\n";
```

Subtask 3: Subtask 3 (20 points)

In this subtask only the price of exchange two stickers is always 1. There are no more restrictions on N and k. This means we still want the minimum number of trades possible but we allow more different values (families).

The question is then which K families do we want to have in the end? To minimize the number of exchanges, we should keep the families that have the greatest number of stickers already. This involves sorting the values by decreasing number of occurrences and the answer would then be N minus the sum of all occurrences of the first K values in the list.

Because we are sorting the values by their occurrences, the time complexity is $O(Nlog(N))$ and the space complexity is $O(N)$.

```cpp
int T;
cin >> T;
for(int t=0;t<T;t++){
  int n, k;
  cin >> n >> k;
  map<int,int> M;
  for(int i=0;i<n;i++){
    int x; cin >> x;
    M[x]++;
  }
  vector<int> v;
  for(auto p : M){
    v.push_back({-p.second}); // vector containing the number of occurrences
  }
  int c = 0;
  sort(v.begin(), v.end()); // sort the vector in decreasing order
  for(int i=0;i<min((int)v.size(), k);i++){
    c += v[i];
  }
  cout << "Case #" <<t << ": " << n + c << "\n";
}
```

Subtask 4: Subtask 4 (20 points)

In this subtask, $K = 2$ but the price of changing a to b has become $b - a$. Furthermore, we can only change a into b if and only if $a < b$.

As $N < 100$, a brute force solution with time complexity $O(n^2)$ would work. Since $K = 2$, the sticker with the maximum value has to stay the same so we can only choose at most one other value x such that all stickers less than x get converted to x and all stickers greater than x get converted into the maximum possible value. We need to calculate the cost of the exchange for all possible values of x.

This solution has a time complexity of $O(n^2)$ and a space complexity of $O(n)$.

```
int T; cin >> T;
for(int t=0;t<T;t++){
  int n, k; cin >> n >> k;
  vector<int> v(n);
  for(int i=0;i<n;i++){
    cin >> v[i];
    int x = v[i];
  }
  sort(v.begin(), v.end());
  int minDiff = n*v[n-1];
  for(int i=0;i<n;i++){
    int diff = 0;
    // count all up to me
    for(int j=0;j<i;j++){
      diff += v[i]-v[j];
    }
    for(int j=i+1; j<n;j++){
      diff += v[n-1]-v[j];
    }
    minDiff = min(diff, minDiff);
  }
  cout << "Case #" <<t << ": " << minDiff << "\n";
}
```

Subtask 5: Subtask 5 (20 points)

In this last subtask, there is no restrictions on K anymore. One solution for this problem would be to use dynamic programming. We would like to express the minimum cost for exchanging the first n stickers such that we have in the end the most k stickers.

One problem is that we do not know for each configuration $dp[k][n]$ the final values of the stickers. This means that when we get a new sticker, we do not know whether its value is part of the k final values of the previous stickers or not. The solution to this is to notice that each stickers gets converted to the smallest value that is greater or equal to its original value. Hence, we only need to sort the stickers by their original value and for each newly added sticker, we iterate through all the previous stickers and at the i-th iteration we set $k-1$ final values to be less than the value of the i-th sticker. This means in particular that all stickers with values greater than the i-th sticker get converted to the value of the last sticker.

As we iterate through all possible k and n and that each iteration takes a time complexity of $O(n)$, the total time complexity is $O(K * N^2)$. Memoization forces the space complexity to be $O(K * N)$. Below is the code for DP. Do not forget to first sort the array of sticker values.

```
int f(int pos, int k){
  if(k < 0){return -1;}
  if(pos < 0){return 0;}
  if(dp[pos][k] != -2){return dp[pos][k];}
  int minCost = -1;
  int diff = 0;
  for(int i=pos;i>=0;i--){
    diff += v[pos] - v[i];
    int ff = f(i-1, k-1);
    if(ff == -1){continue;}
    int cost = diff + ff;
    if(minCost == -1){minCost = cost;}
    minCost = min(minCost, cost);
  }
  return dp[pos][k] = minCost;
}
```

Bargain

Task Idea	Johannes Kapfhammer
Task Preparation	Florian Gatignon
Description English	Florian Gatignon
Description German	Joël Mathys
Description French	Florian Gatignon
Solution	Florian Gatignon
Correction	Florian Gatignon

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

The problem involved two arrays of N positive integers $b_0, \cdots, b_{N-1}$ and $s_0, \cdots, s_{N-1}$, presented in the story as the buying and selling prices of a loaf of cheese in N cities on a participant's way to several events of the Mouse Olympiad in Informatics. The stock of cheese is unlimited in every city, as well as the quantity of cheese which local merchants are ready to buy from the participant; it is guaranteed that $b_i > s_i$ for every i. The goal was to output the maximal profit that this participant could obtain on the road by buying and selling cheese while going through the N cities in order. Each subtask imposed additional rules on how many cities could be visited and how much cheese could be bought.

Subtask 1: Workshop (10 points)

In this first part, there were only two cities and you could buy at most one loaf of cheese in total. Therefore, the answer was determined by a choice between buying a loaf of cheese in the first city to resell it in the second one and not buying or selling anything at all. It is evident that in order to maximize profit one must buy cheese (and resell it) if and only if the buying price in the first city is larger than the price in the second city. If the prices are equal or if one has to sell the cheese at a loss, then one might as well perform no transaction and end up with a profit of 0.

This translates into the formula $sol = \max(s_1 - b_0, 0)$, which can be computed in constant time and using a constant amount of memory. A possible complete solution to this subtask abiding by this idea could resemble the following:

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

signed main() {
	int T; scanf("%lld",&T);
	for(int t=0; t<T; t++) {
		int b0, s0, b1, s1;
		scanf("%lld%lld%lld%lld", &b0, &s0, &b1, &s1);
		printf("Case #%lld: %lld\n", t, max(0,s1-b0));
	}
}
```

Subtask 2: Winter Camp (25 points)

This subtask differs from the first one by only the number of cities, which can now reach 1000.

A trivial and sufficient though non-optimal approach is to try all pairs of cities, computing the difference between the selling price in the second and the buying price in the first and keeping track of the largest possible profit encountered. One must still, of course, make sure to output 0 and not a negative value in cases where it is impossible to make a profit. This is obviously correct, since all possible candidate solutions are examined.

This can be done using two nested loops: for each city, one looks at all cities that follow it. This translates into a solution in $O(N^2)$. An example may be found below – from now on, I shall not include general code common to all subtasks, only the computation of the answer to a particular test case of the studied subtask.

```
int N, sol=0; scanf("%lld", &N);
vector<int> b(N), s(N);
for(int i=0; i<N; i++) {
        scanf("%lld%lld", &b[i], &s[i]);
}
for(int i=0; i<N; i++) {
        for(int j=i+1; j<N; j++) {
                sol=max(sol,s[j]-b[i]);
        }
}
printf("Case #%lld: %lld\n", t, sol);
```

Subtask 3: IMOI (25 points)

Two changes distanced this subtask from the previous one: N is now limited to 500 000 – in other words, 500 times more than before – and the participant may now buy K loafs of cheese.

First, it is necessary to see that the second change makes little difference to the computation. If one has the solution for $K = 1$, then multiplying it by the actual value of K returns the full solution. The reason behind it is that making the most profitable available transactions always generates the most profit in the end. Furthermore, it is always best to make K transactions between the two same cities rather than diversifying one's buying and selling locations, because stock is not limited: one can always make K optimal transactions if there is one.

Therefore, using the same solution as the one presented in Subtask 2 and multiplying the output by K would give correct results. However, the increased number of cities is too large and one should not be able to produce the output fast enough to meet the time limit.

This issue is meant to force one to find a $O(N)$ solution. A way to reduce the runtime can be found, as often, by looking for unnecessary computations. Looking at all pairs of cities to find the best one is inefficient because one repeatedly tries pairs of cities where the buying price is higher than the one in some previous city. It serves no purpose to try these, for it is always better to buy the cheese in that earlier city, and the solution for that one was already computed.

In fact, one can avoid most computations by keeping track of the minimal buying price in the $i-1$ first cities and computing the profit made by buying at that cost and reselling in the i-th city. The solution, therefore, simply goes through all cities, and at each step, updates the maximal profit overall on the basis of the possible profit made by selling cheese in that city, and updates the minimal buying price if necessary. The order of those two operations does not matter, since it is never profitable to buy and sell cheese in the same city ($b_i > s_i, \forall i$). A constant number of operations is used for each city and there is no need to store all buying and selling prices, so the running time is in $O(N)$ and the memory usage in $O(1)$.

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

signed main() {
        int T; scanf("%lld",&T);
        for(int t=0; t<T; t++) {
                int N, K, lowPrice=1e18, sol=0; scanf("%lld%lld", &N, &K);
                for(int i=0; i<N; i++) {
                        int b, s;
                        scanf("%lld%lld", &b, &s);
                        sol=max(sol,s-lowPrice);
                        lowPrice=min(lowPrice,b);
                }
                printf("Case #%lld: %lld\n", t, K*sol);
        }
}
```

Subtask 4: Leader Binna (15 points)

In this subtask and the next one, the rules about the quantity of cheese that the participant is allowed to buy change substantially. It is now unlimited overall, but the participant can carry at most C loafs of cheese at any point in time.

Some general facts should be discussed before turning to the solution of the individual subtasks. It is always possible to construct an optimal solution using some number of disjoint buy-sell intervals. If two buy-sell intervals overlap, it is possible to get a better solution by buying all the cheese in the interval with the best profit per cheese loaf – of course, if both profits are equal, it does not matter, but only using one interval does get an optimal solution.

In Subtask 4, the limit on N is small again. For that reason, a solution using the same principle as that for Subtask 2 could be accepted. One just had to modify the code to take into account the possibility of exploiting several and not just one buy-sell interval. Since one does not have to output the way to get the optimal profit but just its value, the only necessary change is that the buying price in city i can be "reduced" by the maximal profit that one can make on the way to i. To that effect, this value should always be computed before one needs to use it to compute a new result.

In order to operate this change, one can modify the order in which one goes through all pairs of cities: instead of trying every possible selling city for each buying city, one tries every possible buying city for each selling city. In that way, one can store into memory the maximal possible profit for any first i cities and use that to compute it for the first $i + 1$ cities. Repeating this until one gets to the result for all cities allows to calculate the overall maximal profit.

Like in Subtask 2, the runtime is in $O(N^2)$ and the memory usage in $O(N)$.

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

signed main() {
	int T; scanf("%lld",&T);
	for(int t=0; t<T; t++) {
		int N, C; scanf("%lld%lld", &N, &C);
		vector<int> b(N), s(N), sol(N);
		for(int i=0; i<N; i++) {
			scanf("%lld%lld", &b[i], &s[i]);
		}
		for(int j=0; j<N; j++) {
			for(int i=0; i<j; i++) {
				sol[j]=max(sol[j],max(0ll,s[j]-b[i])+sol[i]);
			}
		}
		printf("Case #%lld: %lld\n", t, C*sol[N-1]);
	}
}
```

Subtask 5: Back at IMOI (25 points)

In Subtask 5, N becomes large again. Here too, one may recycle a previous solution, that of Subtask 3, while taking into account the possibility of multiple buy-sell intervals. To that effect, we do not keep track of the best profit and the lowest buying price, but of the best profit if one leaves the i-th city with C loafs of cheese and if one leaves it with no cheese at all. The solution, then, is the best profit if one leaves the last city with no cheese at all.

Let $bought_i$ and $sold_i$ be the max profit when leaving with and without cheese from the i-th city. Let $C = 1$ – if C is not 1, one can just multiply the solution for $C = 1$ by the actual C at the end. Then $bought_i = \max(bought_{i-1}, soldi - 1 - b_i)$ and $sold_i = \max(sold_{i-1}, boughti - 1 + s_i$: either we don't do any transaction in the i-th city, or we make one. Of course, we need a base case: $bought_0 = -b_0$ and $sold_0 = 0$.

Like for Subtask 3, the solution only requires a constant number of operations per city, so the running time is in $O(N)$. Since everything is computed in order, there is no need to store more than the current values of b_i, s_i, $bought_i$, and $sold_i$, allowing one to use only a constant amount of memory.

Below is a solution that follows that principle.

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

signed main() {
        int T; scanf("%lld",&T);
        for(int t=0; t<T; t++) {
                int N, C, bought=-1e18, sold=0; scanf("%lld%lld", &N, &C);
                for(int i=0; i<N; i++) {
                        int b, s;
                        scanf("%lld%lld", &b, &s);
                        sold=max(sold,bought+s);
                        bought=max(bought,sold-b);
                }
                printf("Case #%lld: %lld\n", t, C*sold);
        }
}
```

Reversi

Task Idea	Johannes Kapfhammer
Task Preparation	Bibin Muttappillil
Description English	Bibin Muttappillil
Description German	Joël Mathys
Description French	Yunshu Ouyang
Solution	Bibin Muttappillil
Correction	Bibin Muttappillil, Simon Meinhard

We have arrangement of N continuous discs that are either light (1) or dark (0). Additionally we have the operation of adding a disc of either color to the left or right of the arrangement. If we add a light disc on the left side then every dark disc between the added and the next light disc would be flipped and become light. A similar thing applies to the other color and the other side. Some examples:

- $1000 \uplus 1 \longrightarrow 11111$
- $0 \uplus 0101010 \longrightarrow 00101010$
- $1 \uplus 000011111100100 \longrightarrow 1111111111100100$

For easier understanding we will define a *group* as a subsequence where every disc has the same color. Furthermore are on the left and right of a group either discs of the other color or the end. e.g. 000011111100100 has five groups: 0000, 111111, 00, 1, 00

Subtask 1: One Turn (10 points)

In subtask 1 we need to simulate one operation on a given string. S is the initial arrangement as a bitstring. $N = |S|$ is the number of disc. D (either L or R) and B (either 0 or 1) describe the operation: add a disc of color B the the D side.

We can solve this task by changing the discs of the first group to B (which would mean flipping them) if $D == L$. If $D == R$ we set the last group to B.

We can do this by iterating from start (respectively end) to the point where we find a disc of the same color as B (the added disc). The existence of such a disc is guaranteed by the task statement.

In the end we print out the changed string and the new disc in the correct order, depending on D.

The runtime is $O(N)$ as we iterate once through the string. We need $O(N)$ space to save the string, every other variable is constant.

```cpp
/*
Author:         Bibin Muttappillil
Task:           SOI 2020 Reversi Subtask 1: One Turn
Performance:    O(n) runtime, O(n) space
*/

#include <bits/stdc++.h>

using namespace std;

signed main() {

    // fast in/output
    ios_base::sync_with_stdio(false);
    cin.tie(0);

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

        // input
        int N; cin >> N;
        string S; cin >> S;
        char D, B; cin >> D >> B;

        // first part of output
        cout << "Case #" << t << ": ";

        if(D == 'L'){
            // change the first group of disc to the other color
            // the group ends when the next character is the same color as the added disc
            // the  i < N  isn't necessary, but good practice ;)
                for(int i = 0; i < N && S[i] != B; i++){
                        S[i] = B;
                }

            // don't forget to add the added disc
                cout << B << S << "\n";
        }else{
            // change the last group of disc to the other color
            // similar to the  D == 'L'  case
                for(int i = N-1; i >= 0 && S[i] != B; i--){
                        S[i] = B;
                }

                cout << S << B << "\n";
        }
    }
}
```

Subtask 2: Few Turns (20 points)

Given an arrangement of discs, what is the minimum number of operations you need such that all discs will be of the same color?

In other words we want to have just one group.

We make a case distinction on what happens to the number of groups when we add a disc:

- if we add a disc of the same color the number of groups doesn't change.
- if we add a disc of the other color the number of groups goes down by one.

The same thing happens on the right side.

We see here that there is no way to reduce the number of groups by two.

So we can always reduce the number of groups by one by adding a disc of the opposite color (doesn't matter which side). That means the minimum number of operations is the number of groups -1.

We can count this by counting the transitions ...01... and ...10... as the mark the end and the beginning of a group. The number of groups then is the number of transitions +1

The solution therefore is just the number of transitions.

We can implement this by iterating through the string and always comparing the current disc with the disc before (comparing $S[i]$ and $S[i-1]$). Just be careful to start on the second element to avoid off-by-one errors. To save memory we can just read the current character of the string ($S[i]$) and remember the last one ($S[i-1]$).

The runtime is $O(N)$ as we iterate once through the string. We have $O(1)$ space as every variable is constant (no strings, arrays or vectors).

```
/*
Author:         Bibin Muttappillil
Task:           SOI 2020 Reversi Subtask 2: Few Turns
Performance:    O(n) runtime, O(1) space
*/

#include <bits/stdc++.h>

using namespace std;
```

```

signed main() {

    ios_base::sync_with_stdio(false);
    cin.tie(0);

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

        // input
        int N; cin >> N;
        char last; cin >> last; // setting the first previous

        int sol = 0;
        for(int i = 1; i < N; i++){
            char current; cin >> current;

            // count '...01...' and '...10...'
            if(last != current){
                sol++;
            }

            // setting the next previous
            last = current;
        }

        // output
        cout << "Case #" << t << ": " << sol << "\n";
    }
}
```

Subtask 3: Low Cost (20 points) & In Theory... (50 points)

We now define a *cost* for each operation as the number of discs that get flipped +1. What is the minimum cost you need such that all discs will be the same color?

Subtask 3 and 4 are the same except that in subtask 4 we don't just implement the algorithm but we need to reason why it is correct. The first part can be seen as a solution to subtask 4 and the implementation as the solution to subtask 3.

There are several ideas to solve this task (e.g. dynamic programming, scanline, ...). One of them is greedy, where on each step we take the side with the smaller group, e.g. in 000011111100100 we would add a disc to the right side because the last group 00 is smaller than the first one 0000.

This might seem intuitively correct as we try to minimize the cost in each step but this reason is not enough to justify the correctness of our algorithm.

To be definitely sure we need to show that every other strategy of adding discs is not better than our algorithm. We can't check every strategy by hand, as there are potentially infinite, so we need some helping steps to reduce this amount: Lemma A: discs are only flipped from either the left or the right side If a disc gets flipped it is because a disc was added to the left or right, but it can only be flipped from the left side if all the discs on its left are already the same color as it (and the same for right). Let's assume that we have a disc A that is at some moment flipped from the left side and at another moment from the right side. This means that all the disc on the left AND on the right of A are the same color as A $\rightarrow$ one of the addings wasn't necessary. Lemma B: every solution with the same number of left addings and right addings has the same total cost With Lemma A we can split our arrangement in three parts: a part where its discs get only flipped by the left side, a part for the right side and a middle part where its discs don't get flipped. Now its clear that the left part doesn't affect the right side and vice versa, meaning the order of adding discs doesn't matter $\rightarrow$ Lemma B. Lemma C: we need to add at least as many discs as there are transitions See subtask 2. Lemma D: adding a disc of the same color is useless From Lemma B and C we know that every transition in the left part needs to be dealt with by adding discs on the left. To achieve the minimum number of discs we need to always add one of the opposite color. Adding one of the same color doesn't get rid of one transition and only increases the cost for the adding in the future, so by leaving it away we achieve a lower total cost.

Now with these helping tools we don't need to look at every strategy but only at the strategies that add the same number of discs as our solution (the number of transitions) and we can choose the order, as it doesn't affect the cost (Lemma B).

Let's assume there exist a better optimal strategy that adds l' discs on the left and r' on the right, where our solution adds l on the left and r on the right. We know $l' + r' = l + r$.

Without loss of generality we can assume that the better strategy adds more on the left (this means the argument also works if you swap 'left' and 'right' in the following part). This means $l' > l$ and $r' < r$. After adding l disc on the left and r' on the right the better strategy and our strategy have the same situation, but the better strategy now only adds to the left, where as our strategy only adds to the right after this point. Since we always the the smaller one we know that the size of the current group on the right side is smaller or equal to the current one on the left side. Special case, if there is only one adding left: Then it is clear that the better strategy can't be better (is its last flips cost the same or more). Otherwise we let the better strategy add all but the last left disc. Now we now that the group on the left is strictly bigger than the one on the right, as the right side didn't change but the left had discs added to its side. This means we could now add a disc on the right side and achieve a lower total cost than the better optimal strategy. This contradicts our assumption that this strategy is optimal, meaning that such a strategy can't exits $\longrightarrow$ our strategy is optimal.

The easiest implementation would be to just simulate it with an array, a vector or a deque, but this would require $O(N^2)$ runtime. Instead of that we could calculate the groups and only update the outermost groups, every time we do an adding, but this would need $O(N)$ space. Rather than calculating the groups explicitly we keep a pointer to a position of the outermost group in the original string and the numbers of added disc. With that we can calculate the sizes of the outermost group and update them in constant time. PICTURE

```cpp
/*
Author:         Bibin Muttappillil
Task:           SOI 2020 Reversi Subtask 3: Low Cost
Performance:    O(n) runtime, O(1) space (without input)
*/

#include <bits/stdc++.h>

using namespace std;

int solve(int N, string S){

    int left_index = 0;
    while(left_index < N && S[0] == S[left_index]){
        left_index++;
    }
    // left_index is the size of the first group

    int right_index = N-1;
    while(right_index >= 0 && S[N-1] == S[right_index]){
        right_index--;
    }
    // (N-1) - right_index is the size of the last group

    int left_added = 0; // #disc added on the left side
    int right_added = 0; // #disc added on the right side
    int total_cost = 0; // solution

    while(left_index <= right_index + 1){ // +1 is needed for the last operation

        int left_size = left_added + left_index; // size of the leftest group
        int right_size = right_added + (N-1) - right_index; // size of the rightest group

        if(left_size < right_size){ // choose greedy

            total_cost += left_size + 1;

            // update by adding next group
            char new_color = S[left_index];
            while(left_index < N && new_color == S[left_index]){
                left_index++;
            }

            left_added++;

        }else{

            total_cost += right_size + 1;

            // update by adding next group
            char new_color = S[right_index];
            while(right_index >= 0 && new_color == S[right_index]){
                right_index--;
            }

            right_added++;
        }
    }

    return total_cost;
}

signed main() {

    ios_base::sync_with_stdio(false);
    cin.tie(0);

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

        int N; cin >> N;
        string S; cin >> S;

        cout << "Case #" << t << ": " << solve(N, S) << "\n";
    }
}
```

The space is in $O(1)$ as we only have constant variables (int, char). On the first look the runtime seems like $O(N^2)$ because of the nested loop, but it actually is $O(N)$. Everything before the first while loop is run once $\rightarrow O(1)$. The outer loop can iterate at most N times, since in every iteration either `left_index` is increased or `right_index` is decreased and they are bounded by $0 \leq$ `left_index` $\leq N$ and $N \geq$ `right_index` ≥ 0. Most things in the loop (except the inner while loop) run in constant time so in total it needs $O(N)$. The first inner loops can in *total* only do at most N iterations as in every iteration `left_index` is increased (and never decreased) and it is bound by $0 \leq$ `left_index` $\leq N$. The same thing holds for the second loop with `right_index`. Since every part runs in $O(N)$ the whole algorithm runs also in $O(N)$.

This is only one of the possibly many solutions, here are some others:

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

using namespace std;

signed main() {

    ios_base::sync_with_stdio(false);
    cin.tie(0);

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

        int N; cin >> N;
        string stones; cin >> stones;

        vector<int> group;

        int c = 1;
        for(int i = 1; i < N; i++){
            if(stones[i-1] != stones[i]){
                group.push_back(c);
                c = 0;
            }

            c++;
        }

        group.push_back(c);

        vector<int> original = group;

        int G = group.size();

        if(G == 1){
            cout << "Case #" << t << ": " << 0 << "\n";
            continue;
        }

        vector<int> costLeft(G);
        for(int i = 0; i+1 < G; i++){

            // adding left cost
            if(i > 0){
                costLeft[i] = costLeft[i-1] + group[i] + 1;
            }else{
                costLeft[i] = group[i] + 1;
            }

            // update datastructure
            group[i+1] += group[i] + 1;
            group[i] = 0;
        }

        // reset datastructure
        group = original;

        vector<int> costRight(G);
        for(int i = G-1; i > 0; i--){

            // adding left cost
            if(i < G-1){
                costRight[i] = costRight[i+1] + group[i] + 1;
            }else{
                costRight[i] = group[i] + 1;
            }

            // update datastructure
            group[i-1] += group[i] + 1;
            group[i] = 0;
        }

        int sol = min(costRight[1], costLeft[G-2]);

        // try every split
        for(int split = 1; split < G-1; split++){

            if(sol == -1 || sol > costLeft[split-1] + costRight[split+1]){
                sol = costLeft[split-1] + costRight[split+1];
            }
        }

        cout << "Case #" << t << ": " << sol << "\n";
    }
```

```

}
```

Stofl's mountain railways

Task Idea	Johannes Kapfhammer
Task Preparation	Martin Raszyk
Description English	Yunshu Ouyang
Description German	Martin Raszyk
Description French	Yunshu Ouyang
Solution	Martin Raszyk
Correction	Martin Raszyk

Subtask 1: Unit Prices (25 points)

The railway network can be modeled as an undirected graph with N vertices which model the stations and $M + S$ edges which model the routes and StoflTickets. The length of an edge is the length of the route or the price of the StoflTicket, respectively.

```
int N, M, S;
vector<vector<int> > adj; // adjacency lists

// read the graph from the input
scanf("%d%d%d", &N, &M, &S);
adj.resize(N);

for (int i = 0; i < M + S; i++) {
    int a, b, l;
    scanf("%d%d%d", &a, &b, &l); // l == 1 in the first subtask
    adj[a].push_back(b);
    adj[b].push_back(a);
}
```

In the first subtask, all edges have length one and we have to compute the length of a shortest path between each pair of stations. Because the graph is unweighted (the weight, or length, of each edge is one), we can run breadth-first search (BFS) successively from each vertex to determine its distance from every other vertex.

```
void bfs(const vector<vector<int> > &adj, vector<int> &d, int start) {
    int N = adj.size(); // the number of vertices
    queue<int> q;
    q.push(start);
    d.resize(N, -1);
    d[start] = 0;
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        for (int i = 0; i < adj[cur].size(); i++) {
            int next = adj[cur][i];
            if (d[next] == -1 || d[cur] + 1 < d[next]) {
                d[next] = d[cur] + 1;
                q.push(next);
            }
        }
    }
}
```

The time complexity of running BFS from a single vertex is $O(N + M)$. Since we have to run BFS from each vertex, the overall time complexity is $O(N^2 + NM)$. Because the number of edges M is $O(N^2)$, we can also bound the time complexity by $O(N^3)$. The space complexity is $O(N + M)$.

Subtask 2: Arbitrary Prices (25 points)

In the second subtask, the length of an edge is no longer one whence we cannot use BFS anymore. We can use Dijkstra's algorithm to compute the length of a shortest path from a single vertex to every other vertex if the length of each edge is nonnegative (it is indeed nonnegative in this task, in fact even positive).

Dijkstra's algorithm is similar in spirit to BFS, but it uses a priority queue instead of a FIFO queue to explore the vertices of the graph.

```
typedef pair<int, int> adj_t; // pair (neighbour, distance) in adjacency list
typedef pair<int, int> q_t; // pair (distance, vertex) in priority queue

void dijkstra(const vector<vector<adj_t> > &adj, vector<int> &d, int start) {
    int N = adj.size(); // the number of vertices
    // priority queue overriding the default comparator to get smallest element first
    priority_queue<q_t, vector<q_t>, greater<q_t> > q;
    q.push(make_pair(0, start));
    d.resize(N, -1);
    d[start] = 0;
    while (!q.empty()) {
        int cur = q.top().second;
        int dist = q.top().first;
        q.pop();
        if (dist > d[cur]) continue; // shorter path already found
        for (int i = 0; i < adj[cur].size(); i++) {
            int next = adj[cur][i].first;
            int edge = adj[cur][i].second;
            if (d[next] == -1 || d[cur] + edge < d[next]) {
                d[next] = d[cur] + edge;
                q.push(make_pair(d[next], next));
            }
        }
    }
}
```

The time complexity of the above implementation of Dijkstra's algorithm is $O(N + M \log N)$. Using a Fibonacci heap instead of a binary heap would improve the time complexity to $O(N \log N + M)$. Although using a FIFO queue instead of a priority queue in the above code snippet would not affect the algorithm's correctness, it would lead to a bad time complexity in the worst-case and is strongly discouraged.

Since we have to run Dijkstra's algorithm from each vertex, the overall time complexity is $O(N^2 + NM \log N) \subseteq O(N^3 \log N)$, or $O(N^2 \log N + NM) \subseteq O(N^3)$ if Fibonacci heap is used. The space complexity is $O(N + M)$.

A much simpler algorithm with a time complexity of $O(N^3)$ and space complexity of $O(N^2)$ to compute the length of a shortest path between every pair of vertices is Floyd-Warshall algorithm. When implementing the algorithm, mind the order of the three nested for-loops. Permuting the order yields an incorrect algorithm unless it is repeated multiple times[1]. We strongly encourage you to use the standard algorithm as presented next.

```
// INFTY satisfies the following:
// - it is (strictly) higher than the shortest distance between every pair of vertices
// - INFTY + INFTY does not overflow
const int INFTY = 1000000000;

void solve(int t) { // t: test case number
    int N, M, S;
    vector<vector<int> > d; // distance matrix to be computed

    // read the input graph
    scanf("%d%d%d", &N, &M, &S);
    d.resize(N, vector<int>(N, INFTY));
    for (int i = 0; i < N; i++) d[i][i] = 0;
```

[1] https://arxiv.org/abs/1904.01210

```

    for (int i = 0; i < M + S; i++) {
        int a, b, l;
        scanf("%d%d%d", &a, &b, &l);
        d[a][b] = min(d[a][b], l);
        d[b][a] = min(d[b][a], l);
    }

    // Floyd-Warshall algorithm
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                // d[i][k] + d[k][j] could be up to INFTY + INFTY and must not overflow!
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }

    // output the distance matrix
    printf("Case #%d: %d\n", t, N);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%s%d", j > 0 ? " " : "", d[i][j]);
        }
        printf("\n");
    }
}
```

Subtask 3: Reconstruct Unit Prices Network (25 points)

In the third subtask, we are given a distance matrix and seek to find an input for the first subtask that yields the given distance matrix or determine that no such input exists. If there are multiple solutions, we have to output one minimizing the number of StoflTickets and then the total length of the routes. Since any StoflTicket can always be turned into a route, we equivalently seek to find an undirected graph with the minimum total length of all its edges.

Because all edges of a graph in the first subtask have length one, we can use the following observation to solve the third subtask.

Observation 1 *Suppose that there exists a graph $G = (V, E)$ with all edges of length one that yields the given distance matrix. Then there is an edge $\{u, v\} \in E$ if and only if $d_{u,v} = 1$, i.e., if and only if the distance between u and v is one.*

Proof. We prove the equivalence by proving the two implications separately.

$\Longrightarrow$ If there is an edge $\{u, v\} \in E$, then this edge is a path from u to v of length one, so $d_{u,v} = 1$ (the distance between u and v cannot be zero because the length of each edge is strictly positive).

$\Longleftarrow$ We prove the implication by contradiction. If $d_{u,v} = 1$ and there was no edge $\{u, v\} \notin E$, then the shortest path from u to v must be of length at least two — contradiction. □

The above observation provides us with a unique candidate graph $G = (V, E)$ which has an edge $\{u, v\} \in E$ if and only if $d_{u,v} = 1$. But we still need to verify that this graph yields the given distance matrix. To see this at a concrete counterexample, consider the following distance matrix:

$$\begin{pmatrix} 0 & 1 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

Observation 1 yields a path 0–1–2–3 as the candidate graph, but its distance matrix is different:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$$

```
const int INFTY = 1000000000;

void solve(int t) { // t: test case number
    int N, M, S;
    vector<vector<int> > d; // distance matrix to be computed
    vector<vector<int> > in; // input distance matrix

    // read input distance matrix
    scanf("%d", &N);
    in.resize(N, vector<int>(N, INFTY));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &in[i][j]);
        }
    }

    // compute the candidate graph
    M = S = 0;
    d.resize(N, vector<int>(N, INFTY));
    for (int i = 0; i < N; i++) d[i][i] = 0;
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (in[i][j] == 1) {
                d[i][j] = d[j][i] = 1;
                M++;
            }
        }
    }

    // Floyd-Warshall in the candidate graph
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }

    // check if the candidate graph yields the input distance matrix
    for (int i = 0; i < N; i++) {
        // the input distance matrix is symmetric with zeros on the main diagonal
        // ==> we start at j == i + 1
        for (int j = i + 1; j < N; j++) {
            if (d[i][j] != in[i][j]) {
                printf("Case #%d: INCONSISTENT\n", t);
                return;
            }
        }
    }

    // output the candidate graph
    printf("Case #%d: OK\n", t);
    printf("%d %d %d\n", N, M, S);
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (in[i][j] == 1) printf("%d %d %d\n", i, j, 1);
        }
    }
}
```

The time complexity of the solution is dominated by Floyd-Warshall algorithm and is thus $O(N^3)$. The space complexity is $O(N^2)$.

Subtask 4: Reconstruct Arbitrary Network (25 points)

In the last subtask, we are given a distance matrix and seek to find an input for the second subtask that yields the given distance matrix or determine that no such input exists. If there are multiple solutions, we have to output one minimizing the number of StoflTickets and then the total length of the routes. Since any StoflTicket can always be turned into a route, we equivalently seek to find an undirected graph with the minimum total length of all its edges.

We first verify the triangle inequality in the given distance matrix to determine if a graph yielding the distance matrix exists.

Observation 2 *There exists a graph $G = (V, E)$ that yields the given distance matrix if and only if the distance matrix satisfies the triangle inequality, i.e., $d_{u,v} \leq d_{u,w} + d_{w,v}$ holds for all $u, v, w \in V$.*

Proof. We prove the equivalence by proving the two implications separately.

$\Longrightarrow$ We prove the implication by contradiction. Suppose that the graph $G = (V, E)$ yields the given distance matrix, but the triangle inequality does not hold, i.e., there exist $u, v, w \in V$ such that $d_{u,v} > d_{u,w} + d_{w,v}$. Then there exists a path from u to w of length $d_{u,w}$ and a path from w to v of length $d_{w,v}$. Combining these two paths yields a path from u to v (via w) of length $d_{u,w} + d_{w,v}$ that is shorter that the shortest path from u to v of length $d_{u,v}$ — contradiction.

$\Longleftarrow$ If the distance matrix satisfies the triangle inequality, then the complete graph with every edge $\{u, v\} \in E$ of length $d_{u,v}$ yields the given distance matrix. To see this, we observe that Floyd-Warshall algorithm never updates any entry $d_{u,v}$ initialized to the length of the edge $\{u, v\}$ because of the triangle inequality. □

Observation 2 allows us to identify an inconsistent distance matrix, but does not necessarily provide us with a graph minimizing the total length of all its edges. To construct an optimal graph, we make use of the following two observations.

Observation 3 *Suppose that the graph $G = (V, E)$ yields the given distance matrix and minimizes the total length of all its edges. Then there is an edge $\{u, v\} \in E$ if and only if $d_{u,v} < d_{u,w} + d_{w,v}$ for all $w \in V \setminus \{u, v\}$.*

Proof. We prove the equivalence by proving the two implications separately.

$\Longrightarrow$ We prove the implication by contradiction. Suppose that the edge $e = \{u, v\} \in E$, but $d_{u,v} \geq d_{u,w} + d_{w,v}$ for some $w \in V \setminus \{u, v\}$. From Observation 2, we obtain $d_{u,v} \leq d_{u,w} + d_{w,v}$ whence $d_{u,v} = d_{u,w} + d_{w,v}$. It follows that there exists a shortest path P from u to v passing through w. Because the length of each edge is positive, this shortest path P cannot contain the edge e. Let us now consider the graph G' obtained by removing the edge $e = \{u, v\} \in E$ from the graph G. The graph G' also yields the given distance matrix, because the edge e can be replaced by the subpath P of the same length on any path in G. But the total length of all edges in G is strictly smaller than that in G — contradiction to the optimality of G.

$\Longleftarrow$ We prove the implication by contradiction. Suppose that $d_{u,v} < d_{u,w} + d_{w,v}$ for all $w \in V \setminus \{u, v\}$, but $\{u, v\} \notin E$. Then the shortest path from u to v passes through some vertex $w \in V \setminus \{u, v\}$ and thus $d_{u,v} = d_{u,w} + d_{w,v}$ — contradiction. □

Observation 4 *Suppose that the graph $G = (V, E)$ yields the given distance matrix and minimizes the total length of all its edges. Then the length of any edge $\{u, v\} \in E$ equals $d_{u,v}$.*

Proof. We prove the implication by contradiction. Suppose that there is an edge $\{u, v\} \in E$ whose length l is not $d_{u,v}$. We proceed by a case distinction on whether $l < d_{u,v}$ or $l > d_{u,v}$.

$l < d_{u,v}$ Then the edge $\{u, v\} \in E$ is also a path from u to v of length l that is shorter than the shortest path from u to v of length $d_{u,v}$ — contradiction.

$l > d_{u,v}$ By definition of $d_{u,v}$, there exists a path P from u to v of length $d_{u,v}$. Because $l > d_{u,v}$, the edge $e = \{u, v\}$ cannot be part of the path P. Moreover, the edge e cannot be part of any shortest path in the graph G, because the edge e could be replaced by the subpath P which is strictly shorter. It follows that removing the edge e from the graph G gives a graph G' that also yields the given distance matrix, but has a strictly smaller total length of all its edges (because the length l of the removed edge e is positive) — contradiction. □

To summarize, we can determine if a graph yielding the given distance matrix exists using Observation 2 and if it does, we can determine an optimal one uniquely using Observation 3 and 4. Unlike in the previous subtask, we do not have to recompute the distance matrix because the condition in Observation 2 guarantees the existence of a graph that yields the given distance matrix.

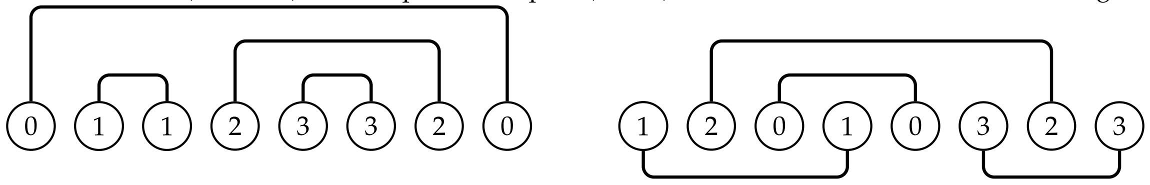
```cpp
const int INFTY = 1000000000;

void solve(int t) { // t: test case number
    int N, M, S;
    vector<vector<int> > d;
    vector<vector<int> > in; // input distance matrix

    // read input distance matrix
    scanf("%d", &N);
    in.resize(N, vector<int>(N, INFTY));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &in[i][j]);
        }
    }

    // check triangle inequality
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                if (in[i][j] > in[i][k] + in[k][j]) {
                    printf("Case #%d: INCONSISTENT\n", t);
                    return;
                }
            }
        }
    }

    // compute the graph
    M = S = 0;
    d.resize(N, vector<int>(N, INFTY));
    for (int i = 0; i < N; i++) d[i][i] = 0;
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            int take_edge = 1;
            for (int k = 0; k < N; k++) {
                if (k == i || k == j) continue;
                if (in[i][j] == in[i][k] + in[k][j]) {
                    take_edge = 0;
                    break;
                }
            }
            if (take_edge) {
                d[i][j] = d[j][i] = in[i][j];
                M++;
            }
        }
    }

    // output the graph
    printf("Case #%d: OK\n", t);
    printf("%d %d %d\n", N, M, S);
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (d[i][j] != INFTY) printf("%d %d %d\n", i, j, d[i][j]);
        }
    }
}
```

The time complexity is $O(N^3)$ and the space complexity is $O(N^2)$.

Arc Match

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Bibin Muttappillil
Description French	Yunshu Ouyang
Solution	Johannes Kapfhammer
Correction	Johannes Kapfhammer

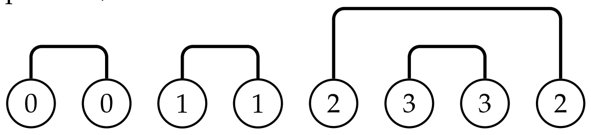
In this task you were given a street with n pairs of houses (so $2n$ houses in total) and you have to connect (to match) the each pair with a path (an arc), such that the arcs are non-intersecting.



There are essentially two variants: In subtasks 1 and 2 the street lies on a shore and you can only connect the pairs on one side of the street (see left picture). This restriction is lifted for subtasks 3, 4 and 5, where you can connect the pairs on both sides of the street (see right picture).

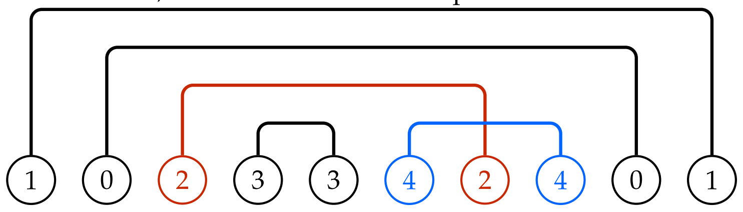
Subtask 1: Lim Chu Kang on the shore (10 points)

To understand this task, it is helpful to look at a few examples. The input "0 0 1 1 2 3 3 2" is possible, because the arcs can be drawn like this:



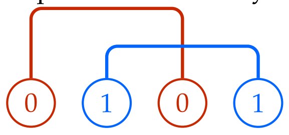
The first observation one can make here is that it doesn't really matter in which shape the arcs are drawn, as long as sufficient care is taken.

In contrast, "1 0 2 3 3 4 2 4 0 1" is impossible



Notice how 2 (red) and 4 (blue) always intersect, no matter how you draw the arcs.

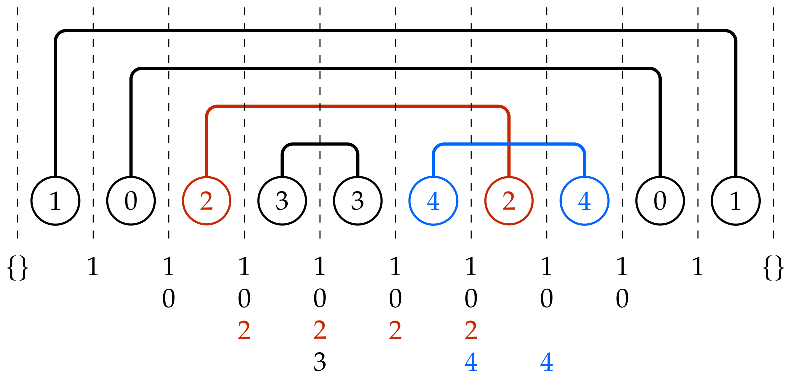
Closely looking at this example we can already come up with a hypothesis: the situation is impossible if and only if there is an "*interleaving pair*" like this:



A proof for this fact is provided at the end of this section, but first, we look at the solution.

Going through with a stack

The simplest solution is to go through the input from left to right and keep track of which arcs are currently open.



When a new arc starts, we append it to the bottom of the list. When an arc ends, we remove it from the list. Note that the black arcs (1, 0, 3) are removed when they are at the bottom, but the red one (2) is removed, while having blue (4) below it! This means that blue (4) has been opened later, but has not yet been closed. So are in a situation where red and blue are interleaved.

More generally if we don't close the most recently opened arc, we must have interleaving pairs. If we always close the most recently opened arc, then it is possible to match them.

The data structure we use to keep track of the open arcs is called a "stack", because we only append elements to the back, remove elements from the back and need to check the last element.

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

bool solve_sub1(int n, vector<int> const& ids) {
  vector<bool> is_open(n); // true if the number has already occured
  vector<int> stack; // stack of open brackets

  for (auto x : ids) {
    if (!is_open[x]) {
      stack.push_back(x);
      is_open[x] = true;
    } else {
      if (stack.back() != x)
        return false;
      stack.pop_back();
    }
  }
  return true;
}

int main() {
  int T; cin >> T;
  for (int i=0; i<T; ++i) {
    int n; cin >> n;
    vector<int> ids;
    copy_n(istream_iterator<int>(cin), 2*n, back_inserter(ids));
    bool ans = solve_sub1(n, ids);
    cout << "Case #" << i << ": "
         << (ans ? "Possible" : "Impossible") << '\n';
  }
}
```

This solution runs in $O(n)$ time and uses $O(n)$ memory.

There is an even shorter $O(n)$ solution based on a similar observation:

```cpp
bool solve_sub1(int n, vector<int> const& ids) {
  vector<int> stack{-1}; // stack of open brackets, -1 is sentinel

  for (auto x : ids) {
    if (stack.back() == x)
      stack.pop_back();
    else
```

```
        stack.push_back(x);
    }
    return stack.size() == 1;
}
```

This code closes the current arc if possible and otherwise opens a new one. If there are interleaving pairs, some arc may be opened twice. We check for this situation at the end.

Bonus: Proving "interleaving pairs ⇔ impossible"

Although you weren't required to prove that statement, we show here how such a proof could look like. The typical way to prove "⇔" (read: "if and only if") is by proving both directions (⇒ and ⇐) separately. It is sometimes helpful to rewrite $A \Rightarrow B$ into the logically equivalent to $\neg B \Rightarrow \neg A$ (called the contraposition), and prove that instead. The bullet points below are all equivalent formulations:

- interleaving pairs ⇔ impossible
- (interleaving pairs ⇒ impossible) AND (impossible ⇒ interleaving pairs)
- (interleaving pairs ⇒ impossible) AND (no interleaving pairs ⇒ possible)

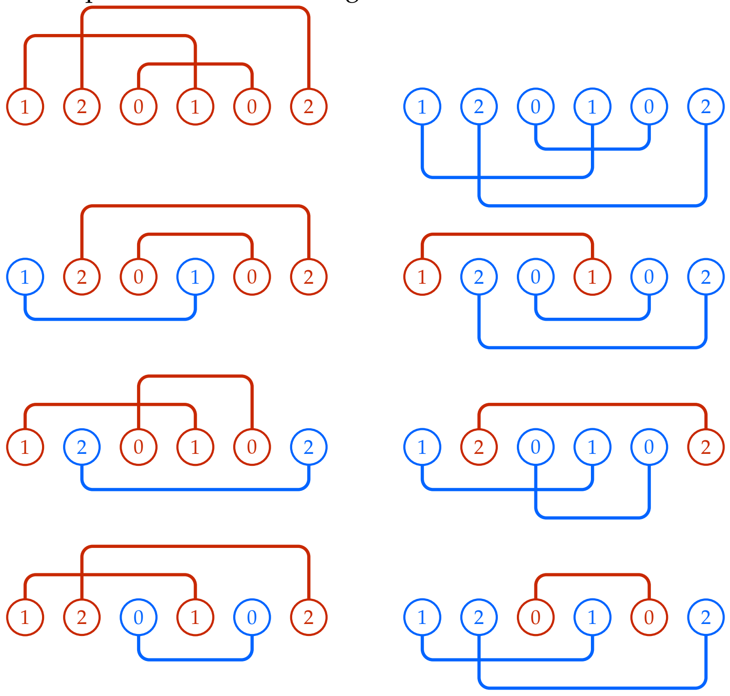
The last one can be proven as follows: (a) having interleaving pairs clearly makes it impossible (see example) and (b) from an input without interleaving pairs, we can construct a solution using the algorithm from the previous section.

Subtask 2: Kallang on the shore (10 points)

The solution shown in the previous subtask still works.

Subtask 3: Tengah (10 points)

This time, arcs can be drawn on *both* sides of the street, so for an input like "1 2 0 1 0 2" there are $2^n = 8$ possibilities of drawing the arcs:



Assuming we know whether each arc goes top (red) or bottom (blue), we can check whether this leads to a correct solution using the solution of the first subtask: we check whether all top arcs are good, and then that all bottom arcs are good.

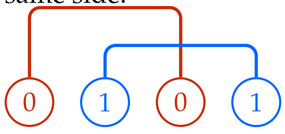
```cpp
bool solve_sub1(int, vector<int> const& ids) { ... }  // as before

vector<bool> solve_sub3(int n, vector<int> const& ids) {
  for (int mask=0; mask < (1<<n); ++mask) {
    // try all possible assignments
    vector<bool> ans(n);
    for (int i=0; i<n; ++i)
      ans[i] = (mask >> i)&1;

    // filter out left and right side
    array<vector<int>, 2> side;
    for (auto x : ids)
      side[ans[x]].push_back(x);  // append to side[0] if ans[x]=false, and side[1] otherwise

    // check if the left AND the right side are good
    if (solve_sub1(n, side[0]) && solve_sub1(n, side[1]))
      return ans;
  }
  // return an empty array to indicate that there is no solution
  return {};
}

int main() {
  int T; cin >> T;
  for (int i=0; i<T; ++i) {
    int n; cin >> n;
    vector<int> ids;
    copy_n(istream_iterator<int>(cin), 2*n, back_inserter(ids));
    vector<bool> ans = solve_sub3(n, ids);
    cout << "Case #" << i << ": ";
    if (ans.empty())
      cout << "Impossible";
    else
      for (auto x : ans)
        cout << (x ? "L" : "R");
    cout << '\n';
  }
}
```

As this solution checks all 2^n assignments, it requires $O(n \cdot 2^n)$ time, which was enough for this subtask.

A solution like this that tries all possibilities is called "bruteforce", and usually quite quick to code, but too slow for the larger subtasks.

Subtask 4: River Valley (30 points)

The key observation for this subtask was once again that interleaving pairs are impossible on the same side:



So if arcs i and j are interleaving, they must have different colors.

Two-Coloring

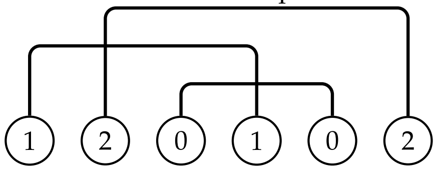
A helpful concept to know is two-coloring of a graph. Given a graph, we want to color each vertex with one of two colors, such that neighboring vertices have different colors. This can be solved using a modified DFS algorithm. See `https://soi.ch/wiki/dfs/` for all details and `https://grader.soi.ch/2h/` for a task that is about two-coloring. We use the following implementation for the solution:

```cpp
bool dfs(vector<vector<int>> const& graph, vector<int>& colors, int v, bool v_color) {
  if (colors[v] != -1) // already visited
    return colors[v] == v_color;
  colors[v] = v_color;
  for (auto u : graph[v])
    if (!dfs(graph, colors, u, !v_color)) // color of u must be opposite
      return false;
  return true;
}

vector<bool> two_color(vector<vector<int>> const& graph) {
  vector<int> colors(graph.size(), -1);
  for (size_t v=0; v<graph.size(); ++v)
    if (colors[v] == -1 &&              // if v hasn't been colored yet
        !dfs(graph, colors, v, false)) // we try to color its component
      return {};                        // and if we fail, the graph is not bipartite

  // when done, return coloring
  return vector<bool>(colors.begin(), colors.end());
}
```

Bruteforce Graph Construction

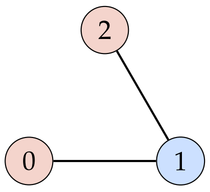
So what does this have to do with our problem? We can loop over all arcs i and j and check whether they are interleaved. If yes, we add an edge between them.

Let's look at an example.



- arc 1 clashes with arc 2 and arc 0
- arc 2 clashes with arc 1
- arc 0 clashes with arc 1

Thus we can draw the following graph:



A two-coloring gives us that arcs 0 and 2 should be on one side, and arc 1 should be on the other side.

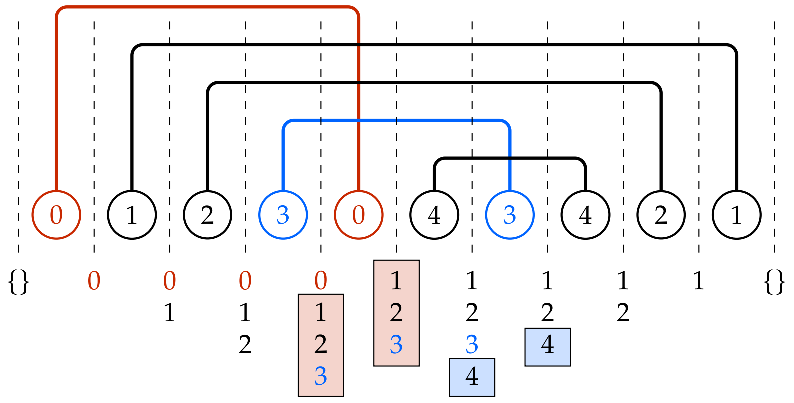
```cpp
bool interleaved(int a, int b, int x, int y) {
  return (a < x && x < b && b < y) || // case [a (x b] y)
         (x < a && a < y && y < b);   // case (x [a y) b]
}

vector<bool> solve_sub4(int n, vector<int> const& ids) {
  vector<int> start(n, -1), end(n, -1);
  for (int i=0; i<(int)ids.size(); ++i) {
    int x = ids[i];
    if (start[x] == -1)
      start[x] = i;
    else
      end[x] = i;
  }

  vector<vector<int>> graph(n);
  for (int i=0; i<n; ++i)
```

```
    for (int j=0; j<i; ++j)
      if (interleaved(start[i], end[i],
                      start[j], end[j])) {
        graph[i].push_back(j);
        graph[j].push_back(i);
      }

  return two_color(graph);
}
```

This solution clearly requires $O(n^2)$ time because we try each pair over arcs.

Stack Graph Construction

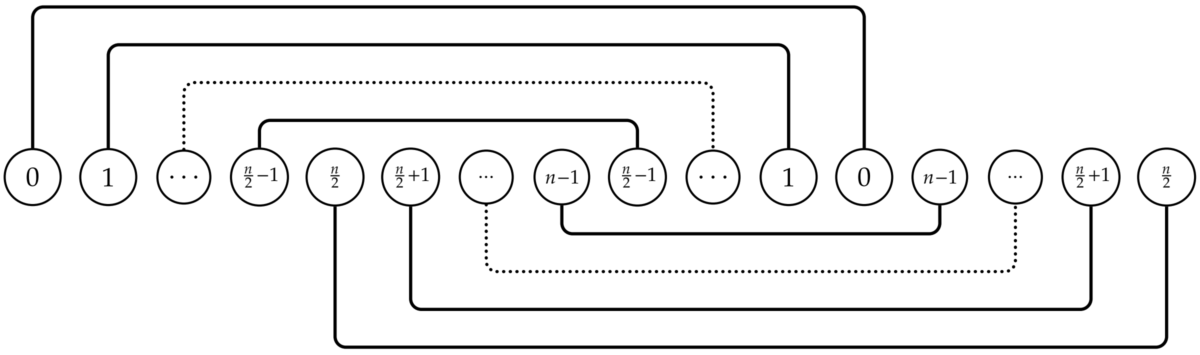
There's a nicer way to construct the graph using the stack solution from earlier. As we'll see later, this can be extended to the full solution for the next task. Consider the following example:



We notice that when we close arc 0, we have to move elements 1, 2, 3 from the end as well. This means 0 clashes with 1, 2 and 3. And when we remove arc 3, we just have to move element 4. Thus we build the graph with edges $\{(0,1),(0,2),(0,3),(3,4)\}$. A possible coloring would be having 0 and 4 red, and the rest blue.

```
vector<bool> solve_sub4(int n, vector<int> const& ids) {
  vector<bool> is_open(n); // true if the number has already occured
  vector<int> stack; // stack of open brackets

  vector<vector<int>> graph(n);

  for (auto x : ids) {
    if (!is_open[x]) {
      stack.push_back(x);
      is_open[x] = true;
    } else {
      // find the opening of x in the stack
      auto it = find(stack.begin(), stack.end(), x);

      // all elements after x must have a different color
      for_each(it+1, stack.end(), [&](int y) {
        graph[y].push_back(x);
        graph[x].push_back(y);
      });

      // remove x from the stack
      stack.erase(it);
    }
  }
  return two_color(graph);
}
```

Note that this solution can add up to $O(n^2)$ edges. This happens for example in the following input:



Every of the $\frac{n}{2}$ lower vertices $\{0, 1, \ldots, \frac{n}{2} - 1\}$ is interleaving with every of the $\frac{n}{2}$ upper vertices $\{\frac{n}{2}, \frac{n}{2} + 1, \ldots, n - 1\}$, so we have $\left(\frac{n}{2}\right)^2 = O(n^2)$ edges in total.

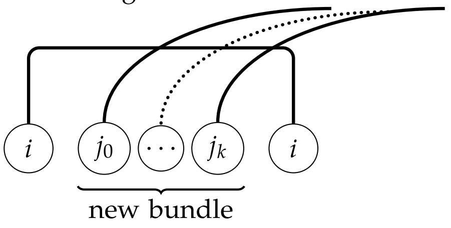
Therefore the running time is $O(n^2)$, and this bound is tight, which means we need something smarter for the next subtask.

Subtask 5: Bedok (40 points)

To solve this last subtask one had to come up with an algorithm that runs in $O(n)$. Our goal is to "sparsify" the graph, which means we want to add fewer edges to it.

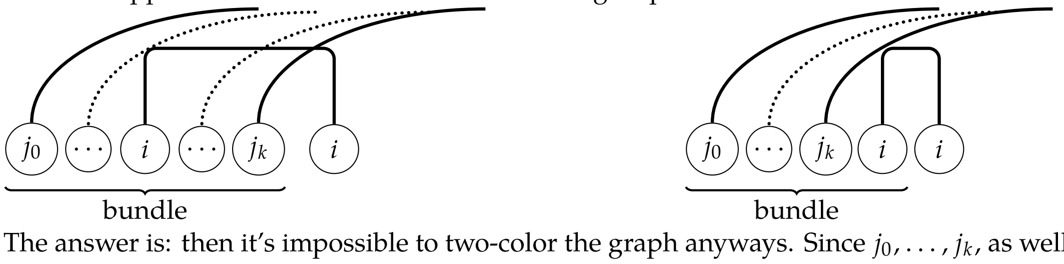
Where were we wasteful in our construction? Consider the previous worst-case example. When we close arc $\frac{n}{2} - 1$, we add edges from it to all of $\{\frac{n}{2}, \ldots, n - 1\}$. Then we know that in any valid coloring of the graph, the vertices $\{\frac{n}{2}, \ldots, n - 1\}$ have always the same color. If vertex $\frac{n}{2}$ is red, then vertex $\frac{n}{2} - 1$ must be blue, and then vertex $\frac{n}{2} + 1$ must be red again, etc. So when we close arc $\frac{n}{2} - 1$, we again add edges to all of $\{\frac{n}{2}, \ldots, n - 1\}$, even though that's completely redundant! It would suffice to add an edge to just one vertices of that set.

We can generalize this observation by "bundling" vertices when we add edges to them.



After we added edges for i, we "bundle" arcs $j_0, \ldots, j_k$ into one group. Inside a group, all colors must be the same. From then on, we act as if that group is only a single arc.

What happens if we want to close an arc from a group?



The answer is: then it's impossible to two-color the graph anyways. Since $j_0, \ldots, j_k$, as well as i are in the same bundle, they must have the same color. The one special case is where i is at the end of the bundle. Then there is no conflict and we can safely remove i from the bundle.

The question now is how to implement the operations for the bundles efficiently.

- we need to merge two bundles
- we need to access and remove the last element

This is something a doubly linked list can do for us. In C++, `list<int>` has the operation `splice` which inserts another list at some position in $O(1)$. A doubly linked list also provides access to the first and last element in $O(1)$.

So we change our stack from `vector<int>` (stack of arcs) to `vector<list<int>>` (stack of bundles of arcs) and implement the operations:

```
vector<bool> solve_sub5(int n, vector<int> const& ids) {
  vector<int> start(n, -1);
  vector<list<int>> stack; // stack of (non-empty) bundles of open arcs
  vector<vector<int>> graph(n);

  for (int i=0; i<(int)ids.size(); ++i) {
    int x = ids[i];
    if (start[x] == -1) {
      stack.push_back({x}); // start a new bundle with only the element x
      start[x] = i;
    } else {
      list<int> between; // bundle of all open arcs between start[x] and i
      while (start[x] < start[stack.back().front()]) { // iterate over bundles from the end
        int y = stack.back().back(); // pick a representant of the bundle
        graph[y].push_back(x);       // add edges between representant and x
        graph[x].push_back(y);
        between.splice(between.begin(), stack.back()); // merge it into "between" in O(1)
        stack.pop_back();                              // and remove the now empty bundle
      }
      // now we know that x is element of the bundle stack.back()

      if (stack.back().back() != x) // if x is the last element of the bundle it works
        return {};                  // otherwise it's impossible

      stack.back().pop_back();  // remove x from the bundle
      if (stack.back().empty()) // and remove the bundle altogether if it gets empty
        stack.pop_back();

      if (!between.empty())             // if there were elements in between
        stack.push_back(move(between)); // move them onto the stack as a single bundle
    }
  }
  return two_color(graph);
}
```

Running time analysis is a bit tricky, since it looks like the while loop could require $O(n)$ iterations in the worst case. While that's true, in total we will only ever do $O(n)$ operations and we can argue with amortized running time. Whenever we pop something from the stack, we must have pushed something onto it before. Now in each iteration of the for-loop, we push *at most once* onto the stack. Thus we can pop from the stack $2n$ at most times. Since the operation `between.splice(...)` runs in $O(1)$, and `stack.push_back(move(between))` also runs in $O(1)$ (we move the list, not copy it), we will only ever do $O(n)$ operations in total. Since `two_color` also runs in $O(n)$, this final solution has a running time and space usage of $O(n)$.

Earthworm Surgery

Task Idea	Johannes Kapfhammer
Task Preparation	Daniel Rutschmann
Description English	Daniel Rutschmann
Description German	Timon Gehr
Description French	Yunshu Ouyang
Solution	Daniel Rutschmann
Correction	Daniel Rutschmann

Subtask 1: Earthworm Jane (20 points)

After unwrapping the story, this task boils down to: You are given a binary string s of length N and may do the following operations on it

1. (citric acid) Replace a zero by two ones or a one by two zeros.
2. (surgery) Remove three consecutive zeroes or three consecutive ones.

You should determine whether it's possible to get the binary string t of length M this way.
As the limits are small, there are many possible solutions that work here:

- Some dynamic programming approach like: $DP[i][j][z]$ = "can you turn $s_1, \ldots, s_i$ into $t_1, \ldots, t_j$ and have z zeros leftover (between s_i and s_{i+1}.) This is fairly slow, running in $O((N+M)^3)$ or $O((N+M)^2)$ depending on the details.
- A greedy algorithm that makes the two strings match from left to right: If the first string is shorter, we may use citric acid to make it longer. Then we compare the digits from left to right and use citric acid whenever they don't match. In the end, we have t followed by some junk, then we turn all the junk into zeros with citric acid and see if we can remove it with surgery. This given a correct algorithm, but it's not immediately obvious why it works.
- In the previous solution, we see that the junk will consist of zero, one or two zeros. Hence it's plausible for there to be three classes of strings where you can turn s into t if and only if both are in the same class. This is indeed the case and the class of s can be computed by

$$c(s) = \text{zero}(s) + 2 \cdot \text{one}(s) \mod 3$$

where $\text{zero}(s)$ and $\text{one}(s)$ are the number of zeros and ones in s respectively. Intuitively, $c(s)$ is the length of the shortest string consisting only of zeros that s can be turned into. $c(s)$ and $c(t)$ can very easily be computed in $O(N+M)$ time. It's easy to check that our citric acid and surgery operations will never change $c(.)$. Such quantities that never change are also called *invariants*. They are very useful for proving correctness of solutions.

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

int calc_invariant(string const&s){
    int ret = 0;
    for(char const &e : s){
        if(e == '0'){
            ret += 1;
        } else {
            ret += 2;
        }
        ret %= 3;
    }
    return ret;
}

```

```
void solve(){
    int n, m;
    cin >> n >> m;
    string s, t;
    cin >> s >> t;
    if(calc_invariant(s) == calc_invariant(t)){
        cout << "YES";
    } else {
        cout << "NO";
    }
}

int main(){
    int T;
    cin >> T;
    for(int cas=0; cas<T; ++cas){
        cout << "Case #" << cas << ": ";

        solve();

        cout << "\n";
    }
    return 0;
}
```

Subtask 2: Earthworm Jim (20 points)

In this subtask, you're again given binary strings s and t. This time, citric acid turns a zero into X ones or a one into X zeros and surgery can only remove Y consecutive equal digits.

As in the previous subtask, you can still use dynamic programming or some greedy construction. Cleaning up the junk is more difficult (you might have to use citric acid, see next section), but as the limits are small, you can run some dfs or bfs to see which number of zeros you can get as junk. There is a new special case $X = 1$, in which case you can't make the string longer, so you have to watch out for that.

The third solution, the one with the $c(s)$ invariant, turns out to be the fastest and easiest to prove correctness for. The straight-forward way of generalizing the invariant would be

$$\text{zero}(s) + X \cdot \text{one}(s) \mod Y$$

This does *not* work however, as applying citric acid to "0" gives us X ones, which would change the invariant from 1 to $X^2 \mod Y$, which in general will not be equal. (This was no issue in the first subtask, as $2^2 \mod 3 = 1$.) To fix this, we instead take

$$c(s) = \text{zero}(s) + X \cdot \text{one}(s) \mod \gcd(Y, X^2 - 1)$$

In the next section, we'll show that this indeed works, i.e. that you only need to check whether $c(s)$ and $c(t)$ are equal (and handle $X = 1$). For now, we'll just note that $\gcd(Y, X^2 - 1)$ can be computed in $O(\log(Y))$ with the Euclidean algorithm, so $c(s)$ and $c(t)$ can be computed in $O(N + M + \log(Y))$.

```
#include <bits/stdc++.h>
using namespace std;
using ll = int64_t;

ll calc_invariant(string const&s, ll const X, ll const Y){
    const LL Z = gcd(X, Y);
    ll ret = 0;
    for(char const &e : s){
        if(e == '0'){
            ret += 1;
        } else {
            ret += X;
        }
        ret %= Z;
```

```cpp
    }
    return ret;
}

void solve(){
    int n, m;
    ll X, Y;
    cin >> n >> m >> X >> Y;
    string s, t;
    cin >> s >> t;
    if(X == 1){
        if(n >= m && (n-m)%Y == 0){
            cout << "YES";
        } else {
            cout << "NO";
        }
    } else {
        if(calc_invariant(s, X, Y) == calc_invariant(t, X, Y)){
            cout << "YES";
        } else {
            cout << "NO";
        }
    }
}

int main(){
    int T;
    cin >> T;
    for(int cas=0; cas<T; ++cas){
        cout << "Case #" << cas << ": ";

        solve();

        cout << "\n";
    }
    return 0;
}
```

Subtask 3: (Theoretical): Even longer earthworms (60 points)

For now assume that $X > 1$. Let $Z = \gcd(Y, X^2 - 1)$. We first show that $c(s) = \text{zero}(s) + X \cdot \text{one}(s) \mod Z$ is indeed an invariant, i.e. that performing our operations on s will never change it.

- If we use acid on a one, we get X zeros, so one(s) decreases by one and zero(s) increases by X. Hence zero(s) + X · one(s) stays the same.
- If we use acid on a zero, we get X ones. In this case, zero(s) decreases by one and one(s) increases by X, so zero(s) + X · one(s) increases by $X^2 - 1$. Hence $c(s)$ stays unchanged as we mod out by Z and Z divides $X^2 - 1$.
- If we perform a surgery operation, zero(s) or one(s) decreases by Y, so zero(s) + X · one(s) decreases by a multiple of Y. Again $c(s)$ stays unchanged as we mod out by Z and Z divides Y.

Hence $c(s)$ stays unchanged under all operations. In particular, if $c(s) \neq c(t)$, then we can't turn s into t. Hence it's impossible for Binna to achieve her pattern if our algorithm prints "NO".

We now show that if $c(s) = c(t)$, then we can indeed turn s into t. As $X > 1$, we may use acid to make s longer than t. Then we proceed from left to right, i.e. with i from 0 to $M - 1$. If $s_i \neq t_i$, then we apply acid to s_i to make s_i equal to t_i. After these steps, $s_0, \ldots, s_{M-1}$ is equal to t, but we might have some more digits $s_M, \ldots, S_{L-1}$ where L is the length of the new s. Let's call these digits the junk. Our goal is to get rid of them. We first apply acid to all ones of the junk. After that, the junk only consists of zeros, so let k be the number of zeros. We can now do the following steps with these zeros:

- Perform surgery to reduce the number of zeros by Y.

- Apply acid to one zero, then apply acid to all the resulting ones. This turns one zero into X^2 zeros, so the total number of zeros in the junk increases by $X^2 - 1$.

If we apply the second step b times and then do the first step a times, we end up with $k - aY + b(X^2 - 1)$ zeros. We want this to be zero, so we want to find integers $a, b \geq 0$ with $aY - b(X^2 - 1) = k$. As $c(.)$ is an invariant and initially $c(s) = c(t)$, we now again have $c(t) = c(s)$. s now consists of t followed by k zeros (the junk). Hence $c(s) \equiv c(t) + k \pmod{Z}$, so $k \equiv 0 \pmod{Z}$, hence $Z = \gcd(Y, X^2 - 1)$ divides k. By Bezout's identity, there are (possibly negative) integers a' and b' with $a'Y - b'(X^2 - 1) = k$. If a' or b' is negative, we may increase a' by $(X^2 - 1)$ and increase b' by Y without destroying this equality. Hence we may assume that a' and b' are both positive, at which point we set $a = a'$ and $b = b'$, which shows that we can indeed get rid of the junk. This shows that if $c(s) = c(t)$, then we can indeed turn s into t. Hence Binna can achieve her goal if our algorithm prints "YES".

If $X = 1$, then we can use acid to change individual ones into zeros and vice versa, so we only need to care about the lengths of s and t. We can't make s longer and can only make it shorter by multiples of Y. Hence we just have to check that $N \geq M$ and that Y divides $M - N$.

SOIway

Task Idea	Benjamin Schmid
Task Preparation	Benjamin Schmid
Description English	Bibin Muttappillil
Description German	Benjamin Schmid
Description French	Yunshu Ouyang
Solution	Benjamin Schmid
Correction	Benjamin Schmid

Subtask 1: Town (25 Points)

We are in a static setting where we have all the stations and a single route right from the start. In a first step we have to find a route which is as short as possible. As we always have the same input we can just solve this by hand and hard code the route. In a second step we then just have to simulate the train driving around. We can simply take the passengers greedily if they fit into the train and deliver them to their destination.

Subtask 2: City (25 Points)

We are again in a static setting but this time we have multiple train routes. There are many possible ways to build the routes. One option is to build clusters with stations which are close together. For each cluster we use a train route and we add some overlap between the clusters so the passengers can change route. Within a cluster we use some approximation for the traveling salesman problem (even a bad one suffices) to find a viable route. However, simulation gets more tricky as we have to implement path finding for the passengers so they know which routes to take and at which stations to change train. It is also necessary to look slightly into the future so we can leave some space on trains to take passengers from stations which are almost overflowing. We also adjust the number of trains per route from time to time so routes with many passengers get more trains.

There are many more possible solutions to this subtask which work just as well.

Subtask 3: Capital (25 Points) & Metropolis (25 Points)

In these two subtask new stations and train routes appear from time to time. As this is the creative task there is not necessarily an optimal solution. We can use the building blocks from subtask 2 and adapt it to the dynamic setting. This mostly involves calculating new routes from time to time and to redistribute the trains between the routes.