BOI'2008

Tasks and Solutions

Gdynia, 2008

Authors:

Szymon Acedański
Zbigniew Czech
Marian M. Kędzierski
Marcin Kubica
Jakub Łącki
Martin Maas
Jimmy Mårdell
Anna Niewiarowska
Martins Opmanis
Paweł Parys
Linas Petrauskas
Michał Pilipczuk
Wolfgang Pohl
Jakub Radoszewski
Laur Tooming
Ahto Truu
Wojciech Tyczyński
Szymon Wąsik
Filip Wolski

Editors:

Marcin Kubica
Jakub Łącki
Jakub Radoszewski

Contents

Preface

Baltic Olympiad in Informatics (BOI) is an annual competition that gathers the best teen-age programmers from countries surrounding the Baltic Sea. The 14-th BOI was held in Gdynia, Poland, between April 17 and April 23, 2008. The competition was organized in Pomeranian Science and Technology Park. Ten countries took part in the competition: Denmark, Estonia, Finland, Germany, Latvia, Lithuania, Norway, Poland, Sweden and Switzerland. There were two competition days preceded with one practice day, with three tasks during each competition day and one task during the practice day.

This booklet presents tasks from BOI'2008 together with the discussion of their solutions. More information about this competition, including the results and test data used during the evaluation of solutions, can be found at `http://b08.oi.edu.pl/`. We hope that this booklet will prove to be useful to both organizers and participants of various programming contests.

Marcin Kubica

Gdynia, Poland, April 2008

Laur Tooming, Ahto Truu
Task idea and formulation

Michał Pilipczuk
Analysis

BOI'2008

Elections

The citizens of Byteland have recently been voting in the parliamentary elections. Now, when the results have been published, the parties have to decide on a coalition to form the government.

Each party received a certain number of seats in the parliament. The coalition must be a subset of the parties such that together they have strictly more than half of all the seats in the parliament. It is desirable for the coalition to have as many seats as possible, to ensure they can still pass their proposed laws even if a few of their members are absent from a parliament session.

A coalition is called **redundant** *if one of its parties can be removed with the remaining ones still having more than half of the seats in the parliament. Of course, such a removable party would effectively have no power — the other members of the coalition would be able to force the laws regardless of its opinion.*

Task

Write a program that:

- *reads the election results from the standard input,*
- *finds a non-redundant coalition that has the maximal possible number of seats in the parliament,*
- *writes the description of this coalition to the standard output.*

Input

The first line of the standard input contains one integer n ($1 \leqslant n \leqslant 300$) — the number of parties that participated in the elections. The parties are numbered from 1 to n.

The second line contains n nonnegative integers $a_1, \ldots, a_n$, separated by single spaces, where a_i is the number of seats received by the i-th party. You may assume that the total number of seats in the parliament will be positive and lower or equal to 100 000.

Additionally, in test cases worth 40% of points, the number of parties will not exceed 20.

Output

The first line of the standard output should contain one integer k — the number of parties in a non-redundant coalition which has the maximal number of seats.

The second line should contain k distinct integers separated by single spaces — the numbers of parties that form the coalition.

If there are several non-redundant coalitions with the maximal number of seats, you may output any of them. The member parties can be listed in any order.

Example

For the input data:	*the correct result is:*
`4` `1 3 2 4`	`2` `2 4`

Solution

Firstly, let us observe that a coalition is non-redundant iff without member party with the smallest number of seats it does not have the majority in the parliament. Obviously, every non-redundant coalition has this property. Moreover, if exclusion of the smallest party breaks the majority, then exclusion of each larger party breaks it as well. Hence, all non-redundant coalitions can be generated from subsets of parties not having majority by adding a party not larger than any party already included.

In the first step of the algorithm the results of parties are to be sorted in non-ascending order of the number of seats. As there are at most 300 parties, this step can be performed using any sorting algorithm, even running in $O(n^2)$ time. From now on, we assume that the numbers of seats of parties $a_1, a_2, \ldots, a_n$ satisfy $a_1 \geqslant a_2 \geqslant \ldots \geqslant a_n$.

In the second step, we apply dynamic programming approach. Let s be the total number of seats in the parliament, $s = a_1 + a_2 + \ldots + a_n$. We use an array `partyUsed`$[0..s]$. The parties are processed in the order from the largest one to the smallest one. Let us assume that we have already considered the first $k-1$ largest parties. For all $i = 1, \ldots, s$ we consider such subsets

$I_i \subseteq \{1,\ldots,k\}$, that $\sum_{j\in I_i} a_j = i$, and either $i \leqslant \lfloor \frac{s}{2} \rfloor$ or I_i represents a non-redundant coalition. In `partyUsed`$[i]$ we store the maximum element that can appear in some I_i (if there is no possible I_i, then `partyUsed`$[i] = -1$). Additionally, `partyUsed`$[0] = 0$.

Let us analyse, how such a data structure can be maintained in consecutive values of k. Initially, for $k = 0$, `partyUsed`$[1..s]$ is filled with -1. Party k should be added to a set of parties having no more than $\lfloor \frac{s}{2} \rfloor$ seats. Hence, for all $i = 0,\ldots,\lfloor \frac{s}{2} \rfloor$, if `partyUsed`$[i] \neq -1$, then the new value of `partyUsed`$[i+a_k]$ is equal k. All values assigned to `partyUsed`$[i]$ for $i > \lfloor \frac{s}{2} \rfloor$ correspond to (the smallest parties in) non-redundant coalitions.

In the third step, we reconstruct the largest non-redundant coalition. The number of seats in such a coalition is the largest such index j, that `partyUsed`$[j] > 0$. The resulting coalition $\{i_1, i_2, \ldots\}$ can be reconstructed as:

$$\begin{aligned} i_1 &= \texttt{partyUsed}[j] \\ i_2 &= \texttt{partyUsed}[j - a_{i_1}] \\ i_3 &= \texttt{partyUsed}[j - a_{i_1} - a_{i_2}] \\ &\vdots \end{aligned}$$

The first step of the algorithm can be implemented in $O(n \log n)$ running time. The second step requires $O(n \cdot s)$ time — for each of n possible values of k we consider $O(s)$ values. The last step has $O(n)$ complexity, since the result cannot be larger. Since $s \geqslant n$, even if we use sorting running in $O(n^2)$ time in the first step, the overall time complexity of the algorithm is still $O(n \cdot s)$. The memory complexity is clearly $O(s)$.

Less Efficient Solutions

Algorithm 1. (40 points) One of the inefficient algorithms considered is a simple back-tracking with time complexity $O(n \cdot 2^n)$. For all 2^n sets of parties, we check (in $O(n)$ time) whether the set is a non-redundant coalition. The solution is straightforward and scores only the guaranteed amount points.

This solution can also be implemented in a randomized way. Instead of considering all the possible subsets, we pick at random as many sets, as the time limit allows, and print the best non-redundant coalition found. If the number of parties is small, the chance of finding the optimal solution is quite

big. However, for larger values of n, the algorithm has problems with finding even a single non-redundant coalition, not to mention the largest one.

Algorithm 2. (50 points) This algorithm is an improved version of the previous one. While recurrently constructing sets of parties, their numbers of seats are computed, and only coalitions having majority are checked. Moreover, after achieving more than a half of all the seats, the search is not continued, since adding any further parties would result in a redundant coalition. The search is also pruned if even including all the remaining parties would not result in majority. These optimizations improve running time, but it is still exponential. However, such a solution scores ten points more.

Algorithm 3. (75 points) In this algorithm dynamic programming is applied, but in a inefficient way. For each party i, we assume that it is the smallest one in some non-redundant coalition. The rest of such a coalition should contain some of the parties $1, \ldots, i$, and the total number of their seats should be greater than $\lfloor \frac{s}{2} \rfloor - a_i$ and not greater than $\lfloor \frac{s}{2} \rfloor$. We can look for such a set of parties using dynamic programming similar to that used in the model solution. Since we apply this step n times, the overall time complexity of this algorithm is $O(n^2 \cdot s)$.

Jimmy Mårdell
Task idea and formulation

Szymon Wąsik
Analysis

BOI'2008

Game

Two players, A and B, play a game on a square board of size $n \times n$. The squares of the board are either white or black. The game is played only on the white squares — the black ones are excluded from the game. Each player has one piece, initially placed at this player's **starting point** *— one of the white squares on the board. The starting point of A is different than that of B.*

In each move a player moves his piece to one of the neighboring white squares (either up, down, left or right). If the player moves his piece to the square currently occupied by his opponent's piece, he gets an extra move (this way he jumps over the opponent). Note that in this case the direction of the second move can be different than that of the first move.

Player A moves first, then players alternate. The goal of the game is to reach the opponent's starting point. The player whose piece reaches his opponent's starting point first, wins the game. Even if the player's last move consists of two jumps, and he only jumps over his opponent's starting point (since it is occupied by his opponent), the player wins. We want to determine which player has a winning strategy (a player has a winning strategy if he can win regardless of his opponent's moves).

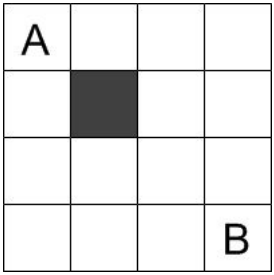


Figure 1. *If A moves to the right on his first three moves, B will move up the first three moves. Thus, on the third move player B will reach the square with A's piece and will be allowed to move again. Because of this, B will reach A's starting point first and will win the game.*

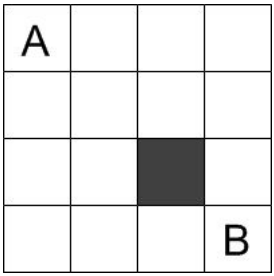


Figure 2. *A can start by moving one step to the right and one step down. Then, depending on the first two moves of B, he will either go down or right and evade B. This way A will reach B's starting point first, thus winning the game. In fact we proved that A has a winning strategy.*

Task

Write a program, that:

- *reads the layout of the grid and the starting points of the two players from the standard input,*
- *finds the player who has a winning strategy,*
- *writes the result to the standard output.*

Input

The first line of the standard input contains one integer t the number of test cases ($1 \leqslant t \leqslant 10$). After it the description of t tests appears. Each test is described as follows. In the first line of the test there is one integer n ($2 \leqslant n \leqslant 300$), the length of the side of the grid. Then next n lines contain the description of the grid. Each line consists of n characters (with no white-spaces between them). Each character is either '.' (a white square), '#' (a black square), 'A' (the starting point of A) or 'B' (the starting point of B).

You may assume that there exists a path of white squares between the starting points of A and B.

Additionally, in test cases worth 60% of points, $n \leqslant 150$ and in test cases worth 40% of points, $n \leqslant 40$.

Output

For each test case exactly one line should be printed to standard output containing a single character `'A'` *or* `'B'`*, indicating the player who has a winning strategy.*

Example

For the input data:	*the correct result is:*

`2`
`4`
`A...`
`.#..`
`....`
`...B`
`4`
`A...`
`....`
`..#.`
`...B`
	`B`
	`A`

Solution

The problem is a typical game theory problem. Let us denote by D the distance between A's and B's starting points. Both players have the same distance D to travel. Therefore, it is obvious that both players should move along a shortest path between starting points. Otherwise a player would make at least $D+1$ moves and his opponent would win. Since player A moves first, he will win if and only if player B is unable to reach the same square as player A in $D/2$ moves. Hence, if D is odd, then A wins. So, from now on, we assume that $D = 2d$.

Since players move along shortest paths it is easy to find for each player all such squares, that can be reached in exactly p moves. We can do this by running the BFS algorithm twice and finding the distances from A's and B's starting points to all the squares. After p moves, player A can be on one of the squares whose distance to A's starting point is p and the distance to B's starting point is $D-p$ (and conversely for player B).

This basic observation is necessary to solve this problem correctly, as it gives us the order in which all configurations of A's and B's positions should be processed.

Model Solution — $O(n^3)$ Time, $O(n^2)$ Memory

This solution uses simple dynamic programming. Let LA_k and LB_k be the lists of such squares which can be reached by players A and B, respectively, in k moves, and can be reached their respective opponent in $D-k$ moves. By $LA_k[i]$ and $LB_k[i]$ we will denote the i-th elements of these lists. Let $T_{k,i,j}$ be **true** if after $d-k$ moves player A has a winning strategy, when his piece is on the square $LA_{d-k}[i]$ and player B is on the square $LB_{d-k}[j]$. If B has a winning strategy, then $T_{k,i,j}$ is **false**. Please note, that LA_d equals LB_d, and $T_{0,i,j}$ is **true** for all i and j such, that $LA_d[i] \neq LB_d[j]$ (that is, the pieces cannot stand on the same square).

Please note that A has a winning strategy, iff he can make such a move, after which B can only make moves, after which A still has a winning strategy. More formally, let $NextA_{k,i}$ be the list of squares from LA_{d-k+1} onto which player A can move from $LA_{d-k}[i]$. Let $NextB_{k,j}$ be a similar list for player B. If for some $i' \in NextA_{k,i}$, for all $j\prime \in NextB_{k,j}$ the value of $T_{k-1,i\prime,j\prime}$ is **true**, then $T_{k,i,j}$ is **true**, otherwise it is **false**.

Using the above observation, we can calculate values $T_{k,i,j}$ for $k = 1,2,..,d$. Player A has a winning strategy in the whole game if and only if $T_{d,1,1}$ is **true**.

The only problem is to compute $NextA$ and $NextB$ lists efficiently. For square (x,y) where one of the players can be after k moves, we can easily find all squares (x',y') where this player can be after $k+1$ moves. The problem is to find the position of (x',y') on the LA_{k+1} and LB_{k+1} lists. But since each square is present on at most two such lists, when we add some square to some list we can also store its position on an appropriate list.

Since each value $T_{k,i,j}$ can be computed in constant time, the total time complexity is $O(n^3)$. Please note, that although there are $O(n^3)$ values $T_{k,i,j}$, we only have to store $T_{k,i,j}$ for two consecutive values of k. Moreover, since each square belongs to at most two lists LA/LB, these lists occupy $O(n^2)$ memory. Therefore, the overall memory complexity is $O(n^2)$.

Slower Solution — $O(n^3 \log n)$ Time, $O(n^3)$ Memory

This solution is similar to the correct solution but instead of *LA* and *LB* lists, and T array it stores configurations of pieces in a dictionary where $T(a_x, a_y, b_x, b_y)$ is **true** if and only if A has a winning strategy, when piece A is on the (a_x, a_y) square and piece B is on the (b_x, b_y) square. The implementation of the dictionary adds a factor of $\log n$ to the time complexity. Since all the configurations must be stored, the memory complexity is $O(n^3)$.

This solution scores 60 % of points.

Slower Solution — $O(n^3)$ Time, $O(n^4)$ Memory

This solution is similar to the previous one, but instead of a dictionary it uses a 4-dimensional array. The time complexity is $O(n^3)$, but the memory used is $O(n^4)$. This solution scores 40 % of points.

Linas Petrauskas
Task idea and formulation

Wojciech Tyczyński
Analysis

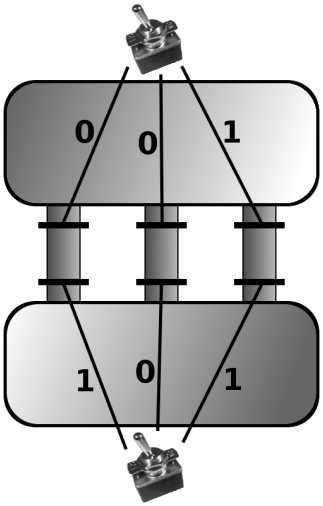
BOI'2008

Gates

After many years of working as a software developer you have decided to try something entirely different, and started looking at random job offers. The one that really caught your eye was a job in fish farming (a form of aquaculture). 'Cool!', you thought, and besides, fish are nice creatures. So you applied, got accepted, and today is your first day at work.

Your boss has already assigned you a task. You have to isolate one water reservoir from another. After looking at some schemes you've been given, here's what you've figured out.

The two water reservoirs are connected by several channels. Each channel has two gates. The channel is open when both gates are open, and is closed otherwise. The gates are controlled using switches. The same switch may operate several gates, but each gate is operated by exactly one switch. It is possible that both gates on a channel are controlled by the same switch and that a switch controls no gates.



Example with three channels and two switches.

The switch may operate the gate in one of two ways:

- *the gate is open when the switch is on, and is closed when the switch is off,*
- *the gate is closed when the switch is on, and is open when the switch is off.*

After playing a bit with the switches you suddenly realize that your programming experience will come in very handy. Write a program that, given the configuration of gates and switches, determines whether it is possible to close all channels, and if it is, then finds a state of every switch in one such valid configuration.

Input

The first line of the standard input contains two integers n *(*$1 \leqslant n \leqslant 250\,000$*) and* m *(*$1 \leqslant m \leqslant 500\,000$*), the number of channels and switches respectively. Switches are numbered from 1 to* m*. Additionally, in test cases worth at least 30 % points,* n *will not exceed 40 and* m *will not exceed 20.*

The following n *lines describe channels, each channel is described by a separate line containing four integers:* a*,* s_a*,* b*,* s_b*. Numbers* a *and* b *represent switches (*$1 \leqslant a, b \leqslant m$*) that operate gates of this channel. Numbers* s_a *and* s_b *can be either 0 or 1 and correspond to the described operation modes:* $s_i = 0$ *means that the gate is closed if and only if the switch* i *is off and* $s_i = 1$ *means that the gate is closed if and only if the switch* i *is on.*

Output

If it is possible to close all the channels, the standard output should contain m *lines. Line* i *should contain 0, if switch* i *should be off, and 1 if switch* i *should be on. If there are many possible solutions, your program may output any of them.*

If it is impossible to close all channels, your program should output one line, containing a single word `IMPOSSIBLE`.

Example

For the input data:	*the correct result is:*

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

and for the input data:	*the correct result is:*

`2 1`
`1 0 1 0`
`1 1 1 1`
	`IMPOSSIBLE`

The first example corresponds to the picture from the task description.

Solution

Let us denote the state of i-th switch by a_i. We can think about a_i as a Boolean variable, which is true when the switch is turned on. Using this approach, all we have to do is to find a valuation of these variables fulfilling task conditions. Moreover, we can easily transform the problem to the language of logic. We show it using the example from the task statement:

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

In this example we have two switches. As a result, we will have two variables a_1 and a_2. If we look at the channel number one we will see that switch number one has to be off or switch number two has to be on in order for this channel to be closed. For each channel, we can create a logical formula:

- channel 1: $\neg a_1 \vee a_2$
- channel 2: $\neg a_1 \vee \neg a_2$
- channel 3: $a_1 \vee a_2$

Since we are looking for a configuration that closes all the channels, we have to make a conjunction of the constructed logical formulas:

$$(\neg a_1 \vee a_2) \wedge (\neg a_1 \vee \neg a_2) \wedge (a_1 \vee a_2)$$

Our task is to find a valuation of variables a_1 and a_2 that satisfies the above formula. In general, the problem of satisfying a given logical formula is known to be NP-complete. Fortunately, the formula obtained in this task is of a very special form:

$$(x_1^1 \vee x_1^2) \wedge (x_2^1 \vee x_2^2) \wedge \ldots \wedge (x_n^1 \vee x_n^2)$$

where x_i^1 and x_i^2 are literals and each of them stands for some variable a_j or negated variable $\neg a_j$. The literals x_i^1 and x_i^2 correspond to the gates on the i-th channel. The formula is a conjunction of alternatives of exactly two literals. This form is called the *second conjunctive normal form* (2-CNF). We show that this problem can be solved in linear time.

For the remaining part of the analysis we assume that our goal is to find such a valuation of variables a_1,a_2,...,a_m, that the following formula is satisfied:

$$(x_1^1 \vee x_1^2) \wedge (x_2^1 \vee x_2^2) \wedge \ldots \wedge (x_n^1 \vee x_n^2)$$

2-CNF Formulas and Graphs

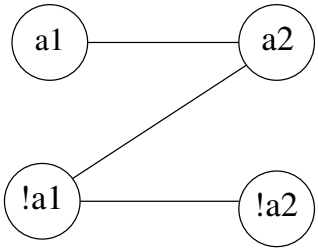
Let us consider an undirected graph $G = (V,E)$ with vertices corresponding to all possible literals:

$$V = \{a_1, \neg a_1, a_2, \neg a_2, \ldots, a_m, \neg a_m\}$$

and edges connecting pairs of literals which appear in alternatives of the formula

$$E = \{(l_i^1, l_i^2) : i = 1,2,\ldots,n\}$$

Graph G is shown on the following figure.



Our goal is to select a subset $W \subseteq V$ containing vertices corresponding to literals which have to be true in order for the formula to be true. For that the following conditions have to be satisfied:

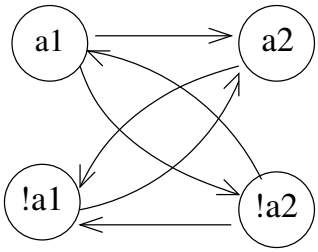
(a) either $a_i \in W$ or $\neg a_i \in W$, for $i = 1, 2, \ldots, m$

(b) for every edge $(u, v) \in E$, $u \in W$ or $v \in W$

Let us construct a directed graph $G_1 = (V, E_1)$ using the definition of graph G:

$$E_1 = \{(\neg u, v) : (u, v) \in E\}$$

We will call it the *inference graph* because it can be used to find which vertices have to belong to W, provided that some given vertex has already been included in it. The inference graph has $2m$ vertices and at most $2n$ edges. The inference for graph G is shown below.



Looking at this graph it is easy to notice that every $W \subseteq V$, which is a correct solution of our problem, must satisfy the condition:

$$(w \in W \wedge (w, v) \in E_1) \Rightarrow v \in W \tag{1}$$

This logical formula is equivalent to condition (b). As a result we can search for $W \subseteq V$ using inference graph.

Let us denote

$$\text{Induced}(u) = \{v : u \mapsto v\}$$

where $u \mapsto v$ means that there exists a path from vertex u to vertex v in graph G_1. Using this definition we can easily rewrite (1) as:

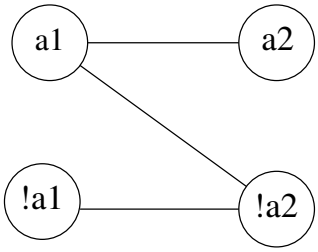
$$w \in W \Rightarrow \text{Induced}(w) \subseteq W$$

If a set Induced(v) contains two opposite vertices ($w \in$ Induced(v) and $\neg w \in$ Induced(v)) we will call the vertex v *problematic*.

Let us introduce another kind of undirected graph: a conflict graph. Using condition (b) we can say that for every edge $(u,v) \in E$ of graph G it is not possible for $\neg u$ and $\neg v$ to be both in W. We can say that these vertices are in conflict. Therefore, we can build a conflict graph $G_2 = (V, E_2)$ where:

$$E_2 = \{(\neg u, \neg v) : ((u,v) \in E) \vee ((v,u) \in E)\}$$

Conflict graph for our example is shown below.



Using these definitions and facts we can show a solution for the problem.

Solution

Naive Solution. The first naive solution uses backtracking approach. For every possible evaluation of the logical variables we check if the formula is satisfied. Obviously, this algorithm runs in exponential time.

Polynomial Solution. Using previously introduced definitions we can solve this problem with the following method:

1. $W = \emptyset$
2. while $|W| < m$ repeat
   - let x be a vertex such that $x \notin W$ and $\neg x \notin W$
   - if both x and $\neg x$ are problematic then solution does not exist (stop the algorithm)
   - let v be a non-problematic vertex among x and $\neg x$

- $W := W \cup \text{Induced}(v)$

3. W is a correct solution for our problem

It is not obvious that this algorithm really returns a correct answer. To see that, let us show the following lemma.

Lemma 1. Let A be a set of vertices a, such that $a \notin \text{Induced}(v)$ and $\neg a \notin \text{Induced}(v)$. If vertex v is non-problematic then there is not an edge in the conflict graph between any pair of vertices from A and Induced(v).

Proof: Let a be a vertex from A and u be a vertex from Induced(v). If there was a conflict between a and u there would have to be an edge $(u, \neg a)$ in the inference graph G_2. As a result there would have to be: $\neg a \in \text{Induced}(u) \subseteq \text{Induced}(v)$. But $\neg a \notin \text{Induced}(v)$ by the definition of A. ■

This Lemma shows the correctness of the presented algorithm — setting the value of a certain variable in a way not leading to a contradiction does not affect the vertices not belonging to Induced(v) and thus leads to a proper solution, if one exists. This solution has the overall time complexity $O(m(n+m))$, as checking if a vertex is problematic may take time proportional to the size of the graph.

Model Solution. Let us think about strongly connected components of the inference graph. (Two vertices u and v belong to the same strongly connected component if and only if there is a path from u to v as well as from v to u). Strongly connected components of an inference graph have a very useful property for us: for any component $C \subseteq V$, either $C \subseteq W$, or $C \bigcap W = \emptyset$, as every vertex induces its whole component. As a result, we can consider the graph of components $G_c = (V_c, E_c)$, whose vertices are strongly connected components of the graph G_1 and edges are inherited from that graph in a natural way. There is a simple algorithm computing strongly connected components of a graph in $O(n+m)$ time.

Obviously the graph of components is a directed acyclic graph. We sort it topologically and consider its vertices in non-ascending order. This can be done in linear time as well.

We say that we accept a component when that component is chosen and included in W while performing the algorithm. Similarly, we say that we reject a component if we decide not to choose the component. Note that if we

reject a component, we have to reject all its predecessors as well. Similarly, if we accept a component, we have to accept all the components induced by it.

These observations lead to a more efficient version of the previous algorithm:

1. Generate the inference graph G_1.
2. Find strongly connected components of G_1 and build a graph of components G_c.
3. If there are two opposite vertices in a component, reject this component (and all its predecessors).
4. Sort the components topologically, and process them in non-ascending order:
   - If the current component has not been rejected yet, accept it.
   - For each vertex in the accepted component reject the component containing the opposite vertex (and consequently all its predecessors).
5. If exactly m vertices have been accepted, they form a correct solution. Otherwise a solution does not exist.

Because of the topological ordering of components, each time we accept a component C, all components induced by C have already been accepted. Indeed, if one of those components had been rejected before, then C would also have been rejected as its predecessor. It is also clear that set W does not contain any conflicts. As a result, if the algorithm finds a solution, it is correct. All we have to do is to show that if a solution exists, this algorithm will find it. It is a consequence of the previous algorithm and Lemma 1, as we reject all problematic vertices in the third step of our algorithm.

As we mentioned before, steps 1–3 can be done in linear time. Moreover, we accept or reject each component exactly once. Therefore, the fourth step also runs in linear time. Hence, the running time of the presented algorithm is $O(n+m)$.

Martins Opmanis
Task idea and formulation

Marian M. Kędzierski
Analysis

BOI'2008

Gloves

In the dark basement of chemistry professor Acidrain's house there are two drawers with gloves — one with left hand and other with right hand gloves. In each of them there are gloves of n different colours. Professor knows how many gloves of each colour there are in each drawer (the number of gloves of the same colour may differ in both drawers). He is also sure that it is possible to find a pair of gloves of the same colour.

Professor's experiment may be successful only if he uses gloves of the same colour (it does not matter which one), so before every experiment he goes to the basement and takes gloves from the drawers hoping that there will be at least one pair of the same colour. It is so dark in the basement that there is no possibility to recognize colour of any glove without going out of the basement. Professor hates going to the basement more than once (in case there was no pair of gloves of the same colour), as well as bringing unnecessarily large amounts of gloves to the laboratory.

Task

Write a program that:

- *reads the number of colours and the number of gloves in each colour in each drawer from the standard input,*
- *calculates the smallest total number of gloves which must be taken to be sure that among them it is possible to find at least one pair of gloves of the same colour (it is necessary to specify the exact number of gloves to be taken from each drawer),*
- *writes the result to the standard output.*

Input

The first line of the standard input contains one positive integer n ($1 \leqslant n \leqslant 20$) describing the number of distinct colours. The colours are numbered from 1 to n. The second line of input contains n non-negative integers

$0 \leqslant a_1, a_2, \ldots a_n \leqslant 10^8$, where a_i corresponds to the number of gloves of colour number i in the drawer with left hand gloves. Finally, the third line of input contains n non-negative integers $0 \leqslant b_1, b_2, \ldots, b_n \leqslant 10^8$, where b_i corresponds to the number of gloves of colour number i in the drawer with right hand gloves.

Additionally, in test cases worth 40% of points, $n \leqslant 4$ and $a_i, b_i \leqslant 10$.

Output

The first line of the standard output should contain a single integer — the number of gloves which must be taken from the drawer with left hand gloves. The second line of output should contain a single integer — the number of gloves which must be taken from the drawer with right hand gloves. The sum of these two numbers should be as small as possible. If there are several correct results, your program should output any of them.

Example

For the input data:	*the correct result is:*

`4`
`0 7 1 6`
`1 5 0 6`
	`2`
	`8`

Solution

Let us denote by Z_n the set of all considered colours, i.e:

$$Z_n = \{1, 2, \ldots, n\}$$

For any $X \subseteq Z_n$ let us denote by A_X the number of all left gloves of colours from the set X, and by B_X the number of all right gloves of colours from the set X. Formally:

$$A_X = \sum_{i \in X} a_i \quad \text{and} \quad B_X = \sum_{i \in X} b_i$$

By $L = A_{Z_n} = a_1 + \ldots + a_n$ we will denote the total number of left gloves, and by $R = B_{Z_n} = b_1 + \ldots + b_n$ we will denote the total number of right gloves.

We will call a pair (l, r) *acceptable* if and only if we *can* take l left gloves and r right gloves (i.e. $l \leqslant L$ and $r \leqslant R$) and taking l left gloves and r right gloves guarantees, that we have taken at least one pair of equally-coloured

gloves. We say, that pair (l,r) *dominates* pair (l',r') iff $l \geqslant l'$ and $r > r'$, or $l > l'$ and $r \geqslant r'$.

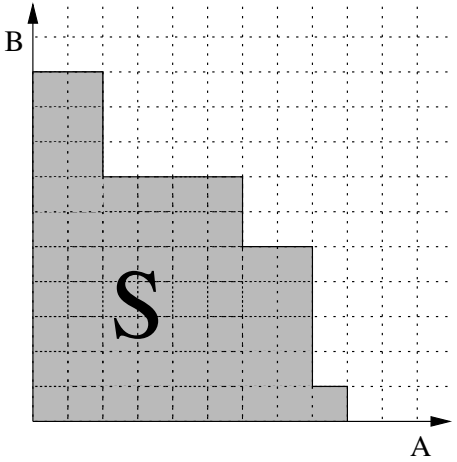
Model Solution

Let X be a subset of Z_n and let $Y = Z_n \setminus X$. For any $0 \leqslant l \leqslant A_X$ and $0 \leqslant r \leqslant B_Y$, choosing l gloves from the first drawer and r gloves from the second drawer does not guarantee getting one pair of equally-coloured gloves. On the other hand, if we take l left gloves and r right gloves from respective drawers, and for every $X \subseteq Z_n$ and $Y = Z_n \setminus X$ we have:

$$l > A_X \vee r > B_Y, \tag{1}$$

then we have taken at least one pair of equally-coloured gloves. To prove it, it is enough to consider any choice of l left gloves and r right gloves, which does not contain any equally-coloured pair and observe that for the partition of Z_n: $X = \{i \in Z_n : a'_i > 0\}, Y = \{i \in Z_n : a'_i = 0\}$ (where a'_i denotes the number of gloves of colour i taken from the first drawer) the condition (1) is not satisfied.

Basing on the above observation we can design the following algorithm: Let us consider all the subsets $X \subseteq Z_n$. For each such X, we compute the numbers $A = A_X$ and $B = B_{Z_n \setminus X}$. We can view a rectangle on the plane $[0,A] \times [0,B]$, representing all the points (l,r), that do not satisfy (1). We can generate all such rectangles, by considering all subsets $X \subseteq Z_n$. Their sum (as subsets of $\mathbb{R}^2$) is a 'staircase shaped' polygon, as shown below:



This figure shows all the points (l,r), which are *not* acceptable. Now we only need to find the minimal (in the sense of the sum of coordinates) point with integer coordinates *outside* the staircase polygon, but inside the rectangle $[0,L] \times [0,R]$.

The model solution considers all 2^n pairs (A,B) and finds these pairs that are not dominated by other pairs, that is are the vertices of the 'staircase' polygon. The pairs are processed in the lexicographical order using a stack. The stack contains these pairs among processed so far, that are not dominated. Whenever we add a pair, the pairs dominated by it are on the top of the stack and can be easily removed. Then such a pair is put on top of the stack. Since each pair can be put and removed from the stack only once, the amortized cost of processing the pairs is $O(2^n)$. Finally, the result is the 'north-east' neighbour of one of concave vertices of the 'staircase' polygon.

The overall time complexity of the algorithm is $O(2^n \cdot n)$, since the dominating phase is sorting 2^n pairs.

Alternative Solution

There is one alternative solution worth mentioning. It is efficient for large n and small number of gloves. It uses dynamic programming similar to the one used for the knapsack problem. We compute values $t[0..L]$, where $t[u]$ is the minimal value of $b_{i_1} + \ldots + b_{i_k}$, over all such sets $\{i_1, \ldots, i_k\} \subseteq Z_n$ for which $a_{i_1} + \ldots + a_{i_k} = u$ (if no such subset exists then $t[u] = \infty$). In other words, pairs $(0, R - t[0]), (1, R - t[1]), \ldots, (L, R - t[L])$, for which $t[i] \neq \infty$, correspond to some points on the boundary of the 'staircase' polygon, including all convex vertices (with positive coordinates). Values of the array t can be computed exactly as in the knapsack problem, for objects with weights a_i and prices b_i, and minimizing the total price.

Zbigniew Czech
Task idea and formulation

Jakub Łącki
Analysis

BOI'2008

Grid

The map of Byteland is drawn on a grid of size $n \times m$ (n is the vertical dimension, m is the horizontal dimension). The horizontal lines marking the division are called parallels*, and are numbered from 0 to n, while the vertical lines of the division are called* meridians*, and are numbered from 0 to m (see figure on the next page).*

Weather forecasting is a serious issue in Byteland. For each unit square of the grid a certain amount of computation time is required to prepare the forecast. Due to terrain conditions and other factors this time may vary from square to square. Until very recently the forecasting system was processing the unit squares one after another, so it took as long as the sum of all the unit times to prepare the complete forecast.

You have been asked to design a new system, running on a multiprocessor computer. To share the computations among processors, the area of Byteland should be divided by r parallels and s meridians into $(r+1)(s+1)$ smaller rectangles. Each processor will cover one rectangle of this division and will process the squares of this rectangle one after another. This way the computation time for such rectangle will be the sum of all computation times of the unit squares contained in this rectangle. The computation time of the complete forecast will be the maximum among computation times of the individual processors.

Your task is to find the minimal possible computation time for some choice of r parallels and s meridians.

Task

Write a program, that:

- *reads the dimensions of the map of Byteland, the required number of parallels and meridians and the unit computation times from the standard input,*
- *finds the minimal time required to compute the complete forecast,*
- *writes the obtained value to the standard output.*

Input

The first line of the input contains four integers n, m, r and s, separated by single spaces ($1 \leqslant r < n \leqslant 18$, $1 \leqslant s < m \leqslant 18$). The following n lines contain the computation times of the unit squares. The j-th number in the $(i+1)$-st line is $c_{i,j}$ — the time required to prepare the weather forecast for the unit square located between the $(i-1)$-st and i-th parallel and between the $(j-1)$-st and j-th meridian ($1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, $0 \leqslant c_{i,j} \leqslant 2\,000\,000$).

Additionally, in test cases worth 40% of points, n and m will not exceed 10.

Output

Your program should write exactly one line. It should contain one integer — the optimal computation time.

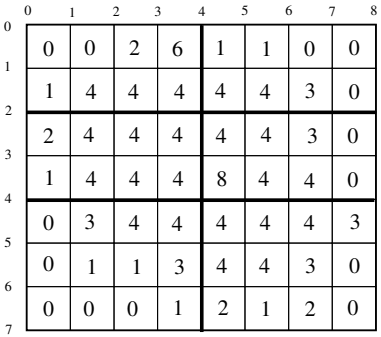
Example

For the input data:

```
7 8 2 1
0 0 2 6 1 1 0 0
1 4 4 4 4 4 3 0
2 4 4 4 4 4 3 0
1 4 4 4 8 4 4 0
0 3 4 4 4 4 4 3
0 1 1 3 4 4 3 0
0 0 0 1 2 1 2 0
```

the correct result is:

```
31
```



The 2-nd and 4-th parallel and the 4-th meridian divide the country into 6 rectangles with computation times 21, 13, 27, 27, 17, 31. The computation time of the complete forecast is 31.

Solution

The simplest solution is to consider all $\binom{n}{r}\binom{m}{s}$ placings of lines. However, if we fix placing of lines going in one direction, then the optimal placing of

lines going in the other direction can be computed much faster. There are two possible approaches:

- Let us fix a placing of all horizontal lines. We can use binary search to find the optimal computation time. Given a candidate for the optimal computation time we can place the vertical lines from left to right using a greedy approach: each vertical line should be put as far to the right as possible. If such placing is impossible, then the candidate value is less then the minimal computation time. Using this criterion we can find the exact minimal computation time using binary search.

  Of course, we have to consider all possible placings of all horizontal lines. Hence, this solution runs in $O(\binom{n}{r} nm\ log(S))$ time, where S is the sum of all computation times for individual squares.

- For each placing of horizontal lines, we can calculate the minimal computation time using dynamic programming. Let $min[i][j]$ be the minimal computation time of the first i columns divided by j meridians. Clearly, we have:

  $$min[i][j] = \min_{j \leqslant t \leqslant i} \max(min[t][j-1], calc\_time[t][i])$$

  where $calc\_time[t][i]$ denotes the maximum among the computation times of rectangles cut by meridians t and i (for the given placing of horizontal lines). Indeed, each of the inner maxima describes a choice of the meridian number t as the last one.

  The array $calc\_time[i][j]$ can be easily preprocessed in $O(m^2 n)$ time. Having it, one can compute the array $min[i][j]$ in the same time complexity using the above formula. The overall running time of the algorithm is $O(\binom{n}{r} m^2 n))$.

It is noteworthy, that in both solutions we should first preprocess the input to be able to determine the sum of unit computation times in rectangle rectangle in a constant time. Let $sum[a][b]$ be the the sum of unit computation times in a rectangle cut by parallels $0, a$ and meridians $0, b$. This array can be easily filled, using dynamic programming, in $O(nm)$ time, as:

$$sum[a][b] = cost[a][b] + sum[a-1][b] + sum[a][b-1] - sum[a-1][b-1]$$

where $cost[a][b]$ denotes the computation time of a corresponding unit square. Now, the sum of unit computation times in a rectangle cut

by parallels a and c ($a > c$), and meridians b and d ($b > d$) is equal $sum[a][b] - sum[c][b] - sum[a][d] + sum[c][d]$.

Other solutions

Brute-Force Solution. In this solution we consider all $\binom{n}{r}\binom{m}{s}$ placings of lines. It runs in $O(\binom{n}{r}\binom{m}{s}nm)$ time and scores 40 points.

Genetic Algorithm. This algorithm maintains a pool of candidate solutions. In each step it:

- picks a solution, copies it and then mutates the copy, by introducing random changes,
- creates a new solution by crossing over some other solutions (for example, by taking meridians from one solutions and parallels from the other),
- discards the worst solutions, so that the pool does not grow too large.

It outputs the best solution found and scores 40–60 points, as it can give incorrect results.

Brute-Force Solution with Pruning. This solution is similar to the brute-force approach. It places the meridians and parallels starting from the upper left corner and remembers the best computing time obtained so far. For each partial placing, if for any rectangle we have:

$$\left\lceil \frac{\text{sum of all computing times}}{\text{number of parts this rectangle is going to be divided into}} \right\rceil \geqslant \text{best result obtained so far}$$

we know that completing this partial placing would not give us a better result. This solution is in many cases faster than optimal solutions. However, in some specific situations it can be even worse than the brute-force algorithm. It scores c.a. 50 points.

Brute-Force Solution with Pruning and Binary Search. It is very similar to the above solution, but instead of improving the best result in each step, it looks for the optimal solution using binary search. It scores 50 points.

Correcting Algorithm. This algorithm chooses at random many divisions and tries to correct each of them. The correction step consists in moving one of the dividing parallels or meridians by one in a direction resulting in the reduction of the total computation time. The corrections are performed until no improvement is possible. The algorithm tries to consider as many initial divisions as the time limit allows and outputs the best obtained computation time. This solution scores 50–60 points.

Martin Maas, Wolfgang Pohl
Task idea and formulation

Filip Wolski
Analysis

BOI'2008

Mafia

The police in Byteland got an anonymous tip that the local mafia bosses are planning a big transport from the harbour to one of the secret warehouses in the countryside. The police knows the date of the transport and they know that the transport will use the national highway network.

The highway network consists of two-way highway segments, each segment directly connecting two distinct toll stations. A toll station may be connected with many other stations. A vehicle can enter or exit the highway network at toll stations only. The mafia transport is known to enter the highways at the toll station closest to the harbour and leave it at the toll station closest to the warehouse (it will not leave and re-enter the highways in between). Special police squads are to be located in selected toll stations. When the transport enters a toll station under surveillance, it will be caught by the police.

From this point of view, the easiest choice would be to place the police squad either at the entry point or the exit point of the transport. However, controlling each toll station has a certain cost, which may vary from station to station. The police wants to keep the overall cost as low as possible, so they need to identify a **minimal controlling set** *of toll stations, which satisfies the two conditions:*

- *all traffic from the harbour to the warehouse must pass through at least one station from that set,*
- *the cost of monitoring these stations (i.e. the sum of their individual monitoring costs) is minimal.*

You may assume that it is possible to get from the harbour to the warehouse using the highways.

Task

Write a program, that:

- *reads the description of the highway network, the monitoring costs and the locations of the entry and exit points of the transport from the standard input,*

- *finds a minimal controlling set of toll stations,*
- *writes this set to the standard output.*

Input

The first line of the standard input contains two integers n and m ($2 \leqslant n \leqslant 200$, $1 \leqslant m \leqslant 20000$) — the number of toll stations and the number of direct highway segments. The toll stations are numbered from 1 to n.

The second line contains two integers a and b ($1 \leqslant a,b \leqslant n$, $a \neq b$) — the numbers of the toll stations closest to the harbour and to the warehouse, respectively.

The following n lines describe the monitoring costs. The i-th of these lines (for $1 \leqslant i \leqslant n$) contains one integer — the monitoring cost of the i-th station (which is positive number not exceeding 10 000 000).

The following m lines describe the highway network. The j-th of these lines (for $1 \leqslant j \leqslant m$) contains two integers x and y ($1 \leqslant x < y \leqslant n$), indicating that there is a direct highway segment between toll stations numbered x and y. Each highway segment is listed once.

Additionally, in test cases worth about 40 points, $n \leqslant 20$.

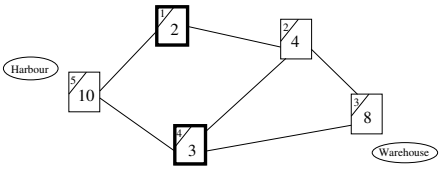
Output

The only line of the output should contain the numbers of toll stations in a minimal controlling set, given in increasing order, separated by single spaces. If there is more than one minimal controlling set, your program may output anyone of them.

Example

For the input data:

```
5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4
```



the correct result is:

```
1 4
```

The figure shows the highway network with the toll station numbers (in the upper-left corners) and the monitoring costs. Stations number 1 and 4 constitute the minimal controlling set with total controlling cost 5.

Solution

Let us have a look at a similar problem, in which every highway has a certain monitoring cost and we have to select the highways to monitor, so that every path from harbour to warehouse leads through at least one monitored highway. Of course the total monitoring cost has to be minimal.

This problem is known as a *minimal cut* problem. The Max-flow min-cut theorem states that finding a minimal cut is equivalent to finding a maximal flow in a graph. Once we find the maximal flow, finding the minimal cut is pretty simple. We compute the set of vertices that are reachable from the source with augmenting paths of a positive residual capacity. The minimal cut is the set of saturated edges (edges with residual capacity equal 0), leading from reachable vertices to unreachable ones. More on maximal flows and the proof of Max-flow min-cut theorem can be found in [1].

The problem described in the problem statement can be transformed into a minimal cut problem in the following way. For every vertex v in an original

graph, we create its copy — v'. For any vertex v, there's an edge from v to v', with flow limit equal to the cost of that vertex being observed by police. Additionally, every unidirectional edge $u \to v$ in the original graph is represented by an edge leading from u' to v, with infinite flow limit. Every bidirectional edge is treated as a pair of unidirectional edges going in both directions.

Now, if we find minimal cut in such network (between vertices a and b'), then edges belonging to that cut (every one of them would be leading from v to v', for some original vertex v) will form an optimal "police covering" of the graph.

Model Solutions

The model solution uses Edmonds-Karp algorithm to find a maximal flow. In general, its running time is $O(VE^2)$ (where V is the number of vertices and E is the number of edges), but in this case, there are only V edges with limited flow in our graph. As a result this algorithm runs in $O(V^2E)$ time. The maximal flow can also be found using many other algorithms. All solutions that run faster that the simplest implementation of the Ford-Fulkerson method should score 100 points.

Other Solutions

Ford-Fulkerson Method. This solution uses plain Ford-Fulkerson algorithm to find a maximal flow. In each step, it tries to find an augmenting path using DFS. Its time complexity can be as high as $O(Ef)$, where f is the maximum flow in the graph. It scores c.a. 80 points.

Brute-Force Solution. This solution considers all 2^V subsets of vertices and tries to block each one. It runs in $O(2^VE)$ time and scores around 40 points.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 2001.

Paweł Parys,
Szymon Acedański
Task idea and formulation

Anna Niewiarowska,
Jakub Radoszewski
Analysis

BOI'2008

Magical stones

Famous stones Xi-n-k can only be found in Wonderland. Such a stone is simply a granite board with an inscription consisting only of letters `X` *and* `I`*. Each board contains exactly n letters. There are not more than k positions in each board where letters* `X` *and* `I` *are next to each other.*

The top and bottom sides of the stones are not fixed, so the stones can be rotated upside-down. For instance two figures below depict exactly the same stone:

`IXXIIXXX` `XXXIIXXI`

Fig. 1: Two ways of looking at the same stone. This stone is of type Xi-*8*-*3*, but also Xi-*8*-*4* (and also of any type Xi-*8*-*k* for $k \geqslant 3$).

No two magic stones in Wonderland are the same, i.e. no two stones contain the same inscription (remember that the upside-down rotation of a stone is allowed).

If it is possible to read the inscription of some stone in two different ways (using the upside-down rotation) then the **canonical representation** *of the stone is defined as the lexicographically less*[1] *of these two ways of reading the inscription.*

If a stone's inscription is symmetrical, i.e. the upside-down rotation does not change it, then its canonical representation is defined as the unique way of reading this inscription.

Example: *There are exactly 6 stones of type Xi-3-2. Their canonical representations written in lexicographical order are:* `III`*,* `IIX`*,* `IXI`*,* `IXX`*,* `XIX` *and* `XXX`*.*

Alice is a well-known expert on the Xi-n-k stones from Wonderland. She would like to create a lexicographical index of the canonical representations of

[1] We say that inscription *A* is lexicographically less than *B* (assuming that lengths of *A* and *B* are the same) if *A* contains letter `I` and *B* contains letter `X` at the first position where the inscriptions differ.

all stones of type Xi-n-k (for some specific values of n and k). What inscription should be written at position i of the index, for a given value of i?

Task

Write a programme which:

- *reads numbers n, k and i from the standard input,*
- *determines the i-th (in the lexicographical order) canonical representation of a Xi-n-k stone,*
- *writes the result to the standard output.*

Input

The first and only line of the standard input contains three integers n, k and i ($0 \leqslant k < n \leqslant 60$, $0 < i < 10^{18}$) separated by single spaces.

Output

The first and only line of the standard output should contain the i-th (in the lexicographical order) canonical representation of a Xi-n-k stone.

If the number of Xi-n-k stones is less than i then the first and only line of output should contain expression `NO SUCH STONE`.

Example

For the input data:	*the correct result is:*
`3 2 5`	`XIX`
and for the input data:	*the correct result is:*
`3 2 7`	`NO SUCH STONE`

Solution

The task is, described alternatively, to find the i-th word in the lexicographical order, of length n, composed of letters `I` and `X`, satisfying two conditions:

(a) letters `I` and `X` are next to each other at at most k positions,

(b) the word read backwards is not lexicographically less than the original word.

From now on, we will call each word satisfying the above conditions a *correct* word. Every two neighbouring positions of a word which contain different letters are called a *letter change*.

We will find the letters of the i-th word one by one, from left to right. We start with an empty prefix of the word, which we will call ε. In the m-th step of the algorithm (starting from $m = 0$), knowing a prefix $a_1 \ldots a_m$ of the requested word, our task is to find its next letter. In order to perform this operation, we compute the number of correct words that start with the prefix $a_1 \ldots a_m$`I`; if i is not greater than this number then — thanks to the lexicographical ordering of correct words — the $(m+1)$-st letter of the requested word is `I`. Otherwise, it is `X`.

To help us precisely formulate the described idea of the solution, let us introduce the following notation: $\sigma(a_1 \ldots a_m)$ denotes the number of correct words that start with the prefix $a_1 \ldots a_m$.

With those notations, model solution can described by the following simple pseudo-code:

```
1:    prefix := ε;
2:    j := i;
3:    for m := 0 to n − 1 do
4:    begin
5:      if (j ⩽ σ(prefixI)) then prefix := prefixI;
6:      else
7:      begin
8:        prefix := prefixX;
9:        j := j − σ(prefixI);
10:     end;
11:   end;
12:   if (j > 1) then return NO SUCH STONE;
13:   else return prefix;
```

The case when the correct result is `NO SUCH STONE` requires some explanation. This happens only if $i > \sigma(\varepsilon)$. In such case the final value of $prefix$ in the above algorithm will be `XXX...X` and j will be greater than 1. However, we notice that it is sufficient to check the latter condition, because it can be satisfied if and only if $i > \sigma(\varepsilon)$.

How to Compute Values of σ

Firstly let us introduce one more notation: $\sigma(a_1 \ldots a_m, b_l \ldots b_1)$ denotes the number of correct words that start with the prefix $a_1 \ldots a_m$ and end with the suffix $b_l \ldots b_1$. If $m+l > n$ then the prefix and suffix overlap. Each correct word that starts with the prefix $a_1 \ldots a_m$ is of exactly one of the following forms:

(1) l final letters of the word (where $0 \leqslant l < m$) read backwards are exactly the same as l starting letters of the word, `I` is the $(l+1)$-st letter from the beginning of the word and `X` is the $(l+1)$-st letter from the end of the word.

(2) m final letters of the word read backwards are the same as m starting letters of the word (read from left to right).

This leads directly to the following formula:

$$\sigma(a_1 \ldots a_m) = \sum_{l=0}^{m-1} \left([a_{l+1} = \mathtt{I}] \cdot \sigma(a_1 \ldots a_m, \mathtt{X} a_l \ldots a_1) \right) + \sigma(a_1 \ldots a_m, a_m \ldots a_1) \tag{1}$$

where $[a_{l+1} = \mathtt{I}]$ is equal to 1 if $a_{l+1} = \mathtt{I}$ and 0 otherwise. In general, expression $[condition]$, where *condition* is a logical statement, has a value of 1 if the *condition* is true and 0 otherwise.

We now need to show how to compute the summands from the above equation. Note that we can assume that $a_m = \mathtt{I}$, as the pseudo-code only needs to know the value of $\sigma(prefix\mathtt{I})$. Let us assume that there are exactly k_m letter changes in the word $a_1 \ldots a_m$ and k_l letter changes in the word $\mathtt{X} a_l \ldots a_1$.

Lemma 1. If $2m \leqslant n$ (this simply implies that prefix $a_1 \ldots a_m$ and suffix $a_m \ldots a_1$ do not overlap) then:

$$\sigma(a_1 \ldots a_m, a_m \ldots a_1) =$$

$$= \frac{1}{2} \cdot \sum_{i=0}^{\lfloor (k-2k_m)/2 \rfloor} \binom{n-2m+1}{2i} + \frac{1}{2} \cdot \sum_{i=0}^{\lfloor (k-2k_m)/2 \rfloor} \binom{\lfloor (n-2m+1)/2 \rfloor}{i}$$

Proof: The *total number* of words of the form $a_1 \ldots a_m \ldots a_m \ldots a_1$ over the alphabet $\{\mathtt{I}, \mathtt{X}\}$ that satisfy condition (a) (i.e. contain at most k letter changes)

equals:

$$S = \sum_{i=0}^{\lfloor (k-2k_m)/2 \rfloor} \binom{n-2m+1}{2i}$$

This holds, because there are $n-2m$ free positions in the middle of the word, so a letter change could take place at any of the $n-2m+1$ pairs of consecutive positions. This explains the binomial coefficients in the formula. We also know that the number of letter changes in the middle part of the word is not greater than $k-2k_m$ and even, as the middle part of the word is bounded by a letter a_m. This explains why the formula contains a sum over $\{0,\ldots,\lfloor (k-2k_m)/2 \rfloor\}$.

The number of *palindromic* words of the form $a_1 \ldots a_m \ldots a_m \ldots a_1$ that satisfy condition (a) is:

$$S_p = \sum_{i=0}^{\lfloor (k-2k_m)/2 \rfloor} \binom{\lfloor (n-2m+1)/2 \rfloor}{i}$$

This formula holds for even n, because one half of the word can be filled arbitrarily using no more than $\lfloor \frac{k-2k_m}{2} \rfloor$ letter changes. The remaining part of the word is uniquely determined and contains the same number of letter changes. If n is odd, the leading $\lfloor n/2 \rfloor$ letters define $\lfloor n/2 \rfloor$ trailing letters, as well as the middle letter, as only one letter can be inserted in the middle of the word without adding new letter changes.

The number of words of the form $a_1 \ldots a_m \ldots a_m \ldots a_1$ that satisfy condition (a) and *are not palindromes* is $S - S_p$. Exactly one half of such words are correct, as if we reverse a correct word w it becomes incorrect and non-palindromic. On the other hand, all *palindromic* words of the form $a_1 \ldots a_m \ldots a_m \ldots a_1$ are correct. This gives us the following formula for the *total number* of correct words of the form $a_1 \ldots a_m \ldots a_m \ldots a_1$:

$$\frac{1}{2} \cdot (S - S_p) + S_p = \frac{1}{2} \cdot S + \frac{1}{2} \cdot S_p$$

which is equivalent to the formula from Lemma 1. ■

Lemma 2. If $m+l+1 \leqslant n$ and $a_{l+1} = \mathtt{I}, a_m = \mathtt{I}$ then:

$$\sigma(a_1 \ldots a_m, \mathtt{X} a_l \ldots a_1) = \sum_{i=0}^{\lfloor (k-k_m-k_l-1)/2 \rfloor} \binom{n-m-l}{2i+1}$$

Proof: Here we can choose letter changes from $n-m-l$ positions, knowing that the total number of letter changes is odd (because $a_m = \texttt{I} \neq \texttt{X}$) and not greater than $k - k_m - k_l$. Notice that the condition $a_{l+1} = \texttt{I}$ already implies that every possible choice of middle letters of the word $a_1 \ldots a_m \ldots \texttt{X} a_l \ldots a_1$ leads to a correct word — this is the reason why the formula from Lemma 2 is much simpler than the previous one (from Lemma 1). ■

Finally, if the prefix and the suffix overlap ($2m > n$ or $m+l+1 > n$) then the number of correct words with the desired prefix and suffix is equal to 0 or 1 and this number can be computed in a straightforward manner. This, together with formulas from Lemmas 1 and 2, concludes the algorithm for computing the values of the σ function.

Time complexity of the algorithm

Let us analyse the time complexity of the whole algorithm. In the pseudo-code of the algorithm values of the σ function are computed $O(n)$ times (line 5). Every such computation is done using Formula (1) and requires $O(n)$ computations of values of $\sigma(\text{some\_prefix}, \text{some\_suffix})$. Finally every such value can be computed using Lemma 1, Lemma 2 or direct computation if some_prefix and some_suffix overlap — each of these requires $O(n)$ operations, provided that all values of binomial coefficients $\binom{a}{b}$ for $0 \leqslant a, b \leqslant n$ are precomputed. This preprocessing can be done in $O(n^2)$ time complexity with a well-known formula:

$$\binom{a}{b} = \binom{a-1}{b-1} + \binom{a-1}{b} \text{ for } a > b > 0$$

and simple formulas for border cases like $a < b$ or $a = b$, or $b = 0$. The total time complexity of the algorithm is therefore $O(n \cdot n \cdot n) + O(n^2) = O(n^3)$. The upper limit for n in our task is equal to 60, so this solution is fast enough to receive perfect score.

An improvement of time complexity of the algorithm is, however, possible. It can be achieved by:

- computation of partial sums $\sum_{i=0}^{j} \binom{c}{2i}$ and $\sum_{i=0}^{j} \binom{c}{2i+1}$, for $j = 0, 1, \ldots, \lfloor n/2 \rfloor$,
- computation of the numbers k_m and k_l in constant time (using values computed for shorter prefixes and suffixes),

- computation of values of σ function for overlapping prefixes and suffixes in $O(1)$ time (using values computed for shorter prefixes and suffixes).

These optimizations lead to computation of values of σ(some_prefix, some_suffix) in constant time and therefore reduce the time complexity of the algorithm to $O(n^2)$. Because this improvement is not very elegant to describe and does not change the running time of the solution for $n \leqslant 60$ significantly, detailed formulation of the $O(n^2)$ algorithm is left for $n \leqslant 60$ to the Reader.

How to Avoid Binomial Coefficients

There is also another way to compute values $\sigma(\text{some\_prefix}, \text{some\_suffix})$. It does not involve any binomial coefficients but only dynamic programming technique. Here we also assume that some_prefix and some_suffix do not overlap.

Let us notice that the value $\sigma(a_1 \ldots a_m, a_m \ldots a_1)$ (the one computed in Lemma 1) is simply equal to the number of *correct words* of length $n-2m+2$ that start with letter a_m and end with a_m, in which the number of letter changes is not greater than $k-2k_m$. On the other hand, value $\sigma(a_1 \ldots a_m, \mathtt{X} a_l \ldots a_1)$ (the one computed in Lemma 2) is equal to the number of *any words* of length $n-(m-1)-l$ that start with letter a_m and end with letter X, in which the number of letter changes is not greater than $k-k_l-k_m$. In the dynamic programming solution we compute values of cells of two integer arrays: C and A (letter C corresponds to correct words, and A — to any words), each of which is 4-dimensional. The dimensions represent:

- the number of letters of the word,
- the bound on the number of letter changes in the word,
- the first letter of the word,
- the last letter of the word.

$C[n][k][b][e]$ represents the number of correct words of length n, containing at most k letter changes, starting with letter b and ending with e. $A[n][k][b][e]$ is defined similarly.

From now on we, concentrate on filling array C; computing values of cells of array A is quite similar but easier and is therefore left to the Reader. Border

cases of the computation are:

$$C[n][k][b][e] = 0 \text{ for } n \leqslant 0 \text{ or } k < 0$$

$$C[1][k][b][e] = [b = e]$$

$$C[n][0][b][e] = [b = e]$$

$$C[2][k][b][e] = [b \neq e \wedge b \leqslant e]$$

Let us now assume that none of the border conditions holds, so $n \geqslant 3$ and $k \geqslant 1$. If $b > e$ (i.e. $b = \texttt{X}, e = \texttt{I}$) then

$$C[n][k][b][e] = 0$$

else if $b < e$ then

$$C[n][k][b][e] = A[n][k][b][e]$$

because this condition already implies condition (b). Finally if $b = e$ then

$$C[n][k][b][e] = \sum_{b' \in \{\texttt{I},\texttt{X}\}} \sum_{e' \in \{\texttt{I},\texttt{X}\}} C[n-2][k-[b \neq b']-[e \neq e']][b'][e']$$

Let us now analyze the time complexity of the dynamic programming solution. The size of each of the arrays A and C is $O(n^2)$. Computing the value of any cell of A and C can be done in constant time. Therefore the time complexity of the solution is either $O(n^3)$ or $O(n^2)$, depending on whether the previously described tricks are applied or not.