2024 Team Formation Test

Task Overview

ID	Name	Time Limit	Memory Limit	Subtasks
T241	Introvert Seating	1.500 s	1024 MB	3 + 3 + 3 + 3 + 12 + 18 + 14 + 36 + 8
T242	Perfect Tour	1.000 s	1024 MB	7 + 15 + 16 + 10 + 15 + 28 + 9
T243	Round Table Knights	2.000 s	1024 MB	4 + 6 + 10 + 7 + 26 + 24 + 19 + 4
T244	Tree Speculation	1.000 s	1024 MB	100

Notice:

- All tasks are divided into subtasks. You need to pass all test cases in a subtask to get points.
- There is an attachment package that you can download from the contest system, containing sample graders, sample implementations, example test cases, and compile and run scripts.
- When testing your programs with the sample grader, your input should match the format and constraints from the task statement, otherwise, unspecified behaviors may occur.
- In sample grader inputs, every two consecutive tokens on a line are separated by a single space, unless another format is explicitly specified.
- The task statements specify signatures using generic type names `void`, `bool`, `int`, `int64`, `int[]` (array) and `int[][]` (array of array).
- In C++, the graders use appropriate data types or implementations, as listed below:

`void`	`bool`	`int`	`int64`	`int[]`	`int[][]`	length of array `a`
`void`	`bool`	`int`	`long long`	`std::vector<int>`	`std::vector<std::vector<int>>`	`a.size()`

T241 - INTROVERT SEATING

Time Limit: 1.500 s / Memory Limit: 1024 MB

N introverts are attending the first-ever face-to-face HKOI training since the COVID-19 pandemic. As introverts, they prefer to minimize social interaction during trainings, starting from their special strategies in choosing their seats.

The N introverts enter the room one by one. The room has N available seats, arranged in a straight line. The seats are numbered $0, 1, \ldots, N-1$ from left to right. The N introverts choose their seats according to the following rules:

- The first person chooses an arbitrary seat.
- Starting from the second person, each person chooses a seat that maximizes the distance to the nearest occupied seat. If there are multiple seats with the same maximal distance, the person can choose any of them. The distance between two seats numbered a and b is defined as the absolute difference between a and b, i.e. $|a-b|$.

For example, if $N = 20$ and the currently occupied seats are $\{0, 9, 19\}$, the fourth person should choose the seat 14 to achieve the maximal distance of 5.

Now, the N introverts have already chosen their seats. However, you only know that the X-th person (in the order of entering the room, 0-based) is sitting at seat Y. You wish to recover the whole seating process and find out the seat each person is occupying.

IMPLEMENTATION DETAILS

You should implement the following procedure:

`union(bool, int[]) recover_plan(int N, int X, int Y)`

- N: the number of introverts and the number of seats.
- X, Y ($0 \le X, Y < N$): the X-th person (in the order of entering the room, 0-based) is sitting at seat Y.
- If there is no seating plan such that the X-th person is sitting at seat Y, the procedure should return `false`. Otherwise, the procedure should return an array of N integers, the i-th integer denotes the seat taken by the i-th person entering the room ($0 \le i < N$). If there are multiple solutions, the procedure must return the solution with the **smallest lexicographical order**.
- This procedure is called exactly once.

Implementation Notes

In C++, `union` is implemented by `std::variant` defined in the `<variant>` header. A method with the return type `std::variant<bool, std::vector<int>>` can return either a `bool` or an `std::vector<int>`.

The sample code below shows three working examples of functions returning `std::variant`.

```
std::variant<bool, std::vector<int>> foo(int N) {
 return N % 2 == 0;
}
std::variant<bool, std::vector<int>> goo(int N) {
 return std::vector<int>(N, 0);
}
std::variant<bool, std::vector<int>> hoo(int N) {
 if (N % 2 == 0) {
   return false;
 }
 return std::vector<int>(N, 0);
}
```

EXAMPLE

Consider the following call:

```
recover_plan(5, 0, 1)
```

A possible solution would be $[1, 4, 0, 2, 3]$. In this case, the procedure should return the array `[1, 4, 0, 2, 3]`.

Consider the following call:

```
recover_plan(5, 1, 1)
```

It can be shown that no valid seating plan exists such that the second person occupies seat 1. Therefore, the procedure should return `false`.

SAMPLE GRADER

The sample grader reads the input in the following format:

- The first and only line consists of three integers, N, X, Y.

The output of the sample grader is in the following format:

- If the procedure returns `false`, the sample grader outputs `Impossible` on a single line.
- Otherwise, it outputs the contents of the array returned, separated by spaces.

SAMPLE TESTS

	Input	Output
1	`5 0 1`	`1 4 0 2 3`
2	`5 1 1`	`Impossible`

SUBTASKS

For all cases:
$1 \leq N \leq 5 \times 10^5$
$0 \leq X, Y < N$

	Points	Constraints
1	3	$X = 0$ $1 \leq N \leq 1000$
2	3	$X = 0$ N is odd $Y = \lfloor \frac{N}{2} \rfloor$
3	3	$X = 0$
4	3	$X = 1$ $1 \leq N \leq 10^5$
5	12	$1 \leq N \leq 30$
6	18	$1 \leq N \leq 80$
7	14	$1 \leq N \leq 1000$
8	36	$1 \leq N \leq 10^5$
9	8	No additional constraints

T242 - PERFECT TOUR

Time Limit: 1.000 s / Memory Limit: 1024 MB

Recently, Alice has been speedrunning a video game called Jump Frog. The game objective is simple: control a frog to collect all coins in the level, and return to its starting position (because in Japanese, "frog" and "to return" have the same pronunciation!). The game's hardest level, "The Endless Abyss", is a level with a grid layout of stone pillars standing in an abyss, where the frog can safely stay on. The coins, however, are all floating mid-air above the abyss, tempting you to risk your frog's life to collect them.

We can represent the level with a $N \times M$ grid with N rows and M columns, where both N and M are odd. Cell (r, c) is the cell on the r-th row and the c-th column $(1 \le r \le N, 1 \le c \le M)$. A cell (r, c) contains a stone pillar if and only if both r and c are odd. From a stone pillar cell (r, c), the frog can jump to $(r-2, c-2)$, $(r-2, c)$, $(r-2, c+2)$, $(r, c-2)$, $(r, c+2)$, $(r+2, c-2)$, $(r+2, c)$, $(r+2, c+2)$ via a straight line if the target cell is within the grid – basically the closest stone pillar along the eight cardinal and ordinal directions.

Among all cells without any stone pillars, exactly K of them contain a coin floating mid-air. The frog can only collect a coin if the frog jumps past the cell it is located in. For example, if the frog jumps from $(2, 2)$ to $(4, 2)$, it can collect a coin located in $(3, 2)$ if there is one. After a coin is collected, it disappears and cannot be collected again.

To speedrun the game, Alice wants to minimize the number of jumps needed to collect all coins. If there is a sequence of jumps such that she can collect a coin for every jump the frog makes, then such strategy is theoretically optimal. If such sequence also ends at the starting cell, it would be dubbed the name "Perfect Tour".

Specifically, she wants to choose a cell containing a stone pillar for the frog to start at, make some jumps such that every jump collects a coin, and return to the starting cell right after the last coin is collected. You may collect the coins in any order.

Find a Perfect Tour if it exists.

IMPLEMENTATION DETAILS

You should implement the following procedure:

```
bool perfect_tour(int N, int M, int K, int[] X, int[] Y)
```

- N: the total number of rows.
- M: the total number of columns.
- K: the number of coins.
- X, Y: two arrays of length K, representing the coins' locations. For each i $(0 \le i < K)$, a coin is placed at $(X[i], Y[i])$. No two coins share the same location.
- If a Perfect Tour exists, the procedure should make exactly one call to `report` (see below) to report a solution, following which it should return `true`.
- Otherwise, the procedure should return `false`.
- This procedure is called exactly once.

Your implementation can call the following procedure to report a Perfect Tour:

```
void report(int[] U, int[] V)
```

- U, V: two arrays of length $K+1$, representing the sequence of stone pillar cells within a perfect tour. The frog starts in cell $(U[0], V[0])$. Chronologically, for each i $(0 \le i < K)$, the frog should jump from cell $(U[i], V[i])$ to cell $(U[i+1], V[i+1])$ and collect one coin in the middle of the jump. The jump should be valid and the coin should not been collected before. $(U[0], V[0])$ should be equal to $(U[K], V[K])$.

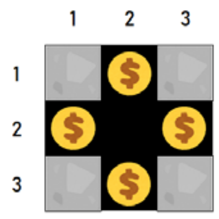
EXAMPLE

Example 1

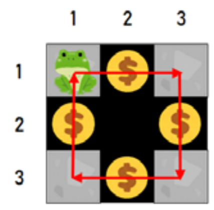
Consider the following call:

```
perfect_tour(3, 3, 4, [1, 2, 2, 3], [2, 1, 3, 2])
```

This means that there are 4 coins located in cells $(1,2)$, $(2,1)$, $(2,3)$ and $(3,2)$. The following figure illustrates the level's layout:



It is possible to construct a Perfect Tour, one of the possible ways to do so is: $(1,1) \to (1,3) \to (3,3) \to (3,1) \to (1,1)$. This solution corresponds to the following figure:



To report this solution, `perfect_tour` should make the following call:

- `report([1, 1, 3, 3, 1], [1, 3, 3, 1, 1])`

It should then return `true`.

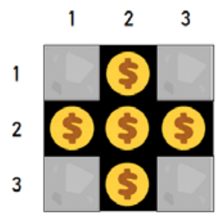
Note that in this case, there are multiple possible Perfect Tour, all of which would be considered correct. For example, it is also correct to call `report([3, 3, 1, 1, 3], [1, 3, 3, 1, 1])` and then return `true`.

Example 2

Consider the following call:

```
perfect_tour(3, 3, 5, [1, 2, 2, 2, 3], [2, 1, 2, 3, 2])
```

The following figure illustrates the level's layout:



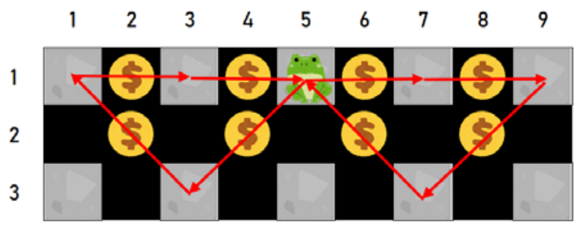
Since there is no way to construct a Perfect Tour such that a coin is collected in each jump and the last jump ends at the starting cell, `perfect_tour` should return `false` without making any calls to `report`.

Example 3

Consider the following call:

```
perfect_tour(3, 9, 8, [1, 2, 1, 2, 1, 2, 1, 2], [2, 2, 4, 4, 6, 6, 8, 8])
```

It is possible to construct a Perfect Tour, one of the possible ways to do so is: $(1,5) \to (1,7) \to (1,9) \to (3,7) \to (1,5) \to (3,3) \to (1,1) \to (1,3) \to (1,5)$. This solution corresponds to the following figure:



To report this solution, `perfect_tour` should make the following call:

- `report([1, 1, 1, 3, 1, 3, 1, 1, 1], [5, 7, 9, 7, 5, 3, 1, 3, 5])`

It should then return `true`.

Note that the Perfect Tour can pass through a cell multiple times, e.g. $(1,5)$ in this example.

SAMPLE GRADER

The sample grader reads the input in the following format:

- The first line consists of three integers, N, M, K.
- The next K line each consists of two integers, $X[i]$ and $Y[i]$.

The output of the sample grader is in the following format:

- If your program makes any erroneous call, e.g. make more than one call to `report`, the output of the grader will be a single line in this format `Error: [MSG]`, where `[MSG]` is the error message.
- Else, the first line consists of a single word, `Yes`, or `No`, depending on the return value of `perfect_tour`.
- If the return value of `perfect_tour` is `true` and `build(U, V)` is called, the grader than additionally prints $K+1$ lines: the j-th line consists of two integers, $U[j]$ and $V[j]$ $(0 \le j \le K)$.

SAMPLE TESTS

	Input	Output

1	3 3 4
	1 2
	2 1
	2 3
	3 2
		Yes
		1 1
		1 3
		3 3
		3 1
		1 1

2	3 3 5
	1 2
	2 1
	2 2
	2 3
	3 2
		No

	Input	Output

3	3 9 8
	1 2
	2 2
	1 4
	2 4
	1 6
	2 6
	1 8
	2 8
		Yes
		1 5
		1 7
		1 9
		3 7
		1 5
		3 3
		1 1
		1 3
		1 5

4	5 3 6
	1 2
	2 1
	2 2
	4 2
	4 3
	5 2
		No

SUBTASKS

For all cases:
$3 \le N, M < 10^5$
Both N and M are odd.
$1 \le K \le min(10^5, N \times M - (N+1)(M+1)/4)$
$1 \le X[i] \le N$ for $0 \le i < K$
$1 \le Y[i] \le M$ for $0 \le i < K$
$X[i]$ and $Y[i]$ cannot both be odd for $0 \le i < K$
$(X[i], Y[i])$ is distinct for $0 \le i < K$

	Points	Constraints
1	7	$1 \le K \le 8$
2	15	$3 \le N, M < 1000$ It is guaranteed that if a Perfect Tour exists, a Perfect Tour that consists of K distinct cells exists. It is guaranteed that there do not exist coins at $(X[i], Y[i])$ where $X[i]$ and $Y[i]$ are both even
3	16	It is guaranteed that if a Perfect Tour exists, a Perfect Tour that consists of K distinct cells exists. It is guaranteed that there do not exist coins at $(X[i], Y[i])$ where $X[i]$ and $Y[i]$ are both even
4	10	$3 \le N, M < 1000$ It is guaranteed that there do not exist coins at $(X[i], Y[i])$ where $X[i]$ and $Y[i]$ are both even
5	15	$3 \le N, M < 1000$ It is guaranteed that there is at most one coin at $(X[i], Y[i])$ where $X[i]$ and $Y[i]$ are both even
6	28	$3 \le N, M < 1000$ It is guaranteed that if a Perfect Tour exists, a Perfect Tour that consists of K distinct cells exists.
7	9	No additional constraints

T243 - ROUND TABLE KNIGHTS

Time Limit: 2.000 s / Memory Limit: 1024 MB

As the strongest knight in Hackerland, you are invited to a round-table combat tournament hosted by Hooverland! You will fight against other King Arthur's Round Table Knights and strive to bring glory to your homeland!

Before you enter, there were $2^N - 1$ knights arranged in a circle, each facing the center of the circle. These knights are numbered from 0 to $2^N - 2$ (inclusive) in clockwise order. Knight $(i+1)$ is to the left of knight i for $0 \le i \le 2^N - 3$, and knight 0 is to the left of knight $2^N - 2$.

You, as the guest participant, will be inserted **randomly** between two adjacent knights, picked with equal probabilities. There are a total of $2^N - 1$ such positions.

The tournament consists of N knockout rounds. Since you are the guest, in each round you will choose to fight either the knight to your left or the knight to your right. Note that once the decision is made, there is exactly one way to group the remaining knights into adjacent pairs.

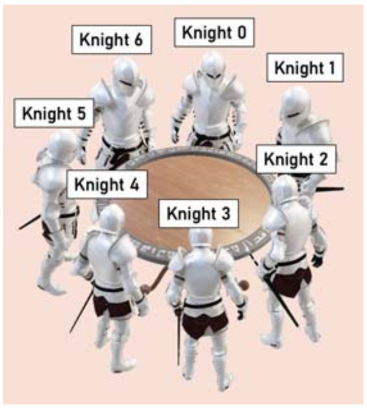


Figure 1: When $N = 3$, seven knights stand in a circle.

Once the pairs are formed, the knights in the same pair fight each other, and the knight with a larger strength level wins. If there is a tie between you and your opponent, you lose. If there is a tie between two knights other than you, they flip a coin to decide who progresses. The loser(s) leave the circle.

In the beginning, the knight i has a strength level $S[i]$. The winner in each round increases their strength by that of the knight they defeat. For example, if knight 3 who has a strength level of 7 fights with knight 4 who has a strength level of 6, knight 3 will win. Knight 4 will leave the circle while knight 3's strength level will increase to $7 + 6 = 13$ and he will advance to the next tournament round.

A knight that wins k rounds in the tournament receives a prize money of $V[k]$, where $V[0] = 0$. To earn more prize money, you will always try to maximize the number of rounds you win by strategically choosing to fight the knight on your left or on your right each round.

You have not yet decided how much effort you want to put towards training. Therefore, you started an investigation. You ask Q questions in the format: "If my strength level is $T[j]$, what is the expected amount of prize money I will earn given that I use the optimal strategy to decide my opponents?" You want to find that expectation multiplied by $(2^N - 1)$.

IMPLEMENTATION DETAILS

You should implement the following procedures:

```
void init(int N, int[] S, int[] V)
```

- N: the number of rounds in the combat tournament.
- S: an array of length $2^N - 1$. $S[i]$ represents the strength of knight i, which they are standing in clockwise order.
- V: an array of length $N + 1$. $V[i]$ represents the prize money gained by a knight that wins i rounds.
- This procedure is called exactly once, before any calls to `query`.

```
int64 query(int T)
```

- T: the strength level that you are considering.
- The procedure should return the expected amount of prize money if your strength level is T, multiplied by $(2^N - 1)$. It can be proven that this return value is an integer.
- This procedure is called exactly Q times.

EXAMPLE

Consider the following call:

```
init(3, 2, [10, 20, 30, 40, 50, 6, 100], [0, 1, 10, 100])
```

Consider the following query:

```
query(40)
```

In this query, your strength level is 40. Let's consider the case where you are starting between knight 0 and knight 6, the lineup is $\underline{40}, 10, 20, 30, 40, 50, 6, 100$.

To maximize the number of rounds you win, you choose to fight the knight to your left - the remaining knights into adjacent pairs accordingly.

After you beat the knight 0 with the strength level 10, your strength level increased to 50. The lineup then becomes $\underline{50}, 50, 90, 106$. You cannot win any more rounds.

In summary, here are the different results when you are inserted into different positions.

- You can win one round if you start:
  - Between knight 6 and knight 0.
  - Between knight 0 and knight 1.
  - Between knight 4 and knight 5.
  - Between knight 5 and knight 6.
- You can win two rounds if you start:
  - Between knight 1 and knight 2.
  - Between knight 2 and knight 3.
- You cannot win any rounds if you start:
  - Between knight 3 and knight 4.

The answer to the query is then $1 + 1 + 10 + 10 + 0 + 1 + 1 = 24$. Hence, the procedure should return 24.

Consider the following query:

```
query(888)
```

In this query, your strength level is 888, you can win the tournament wherever you start. The answer to the query is then $100 \times 7 = 700$. Hence, the procedure should return 700.

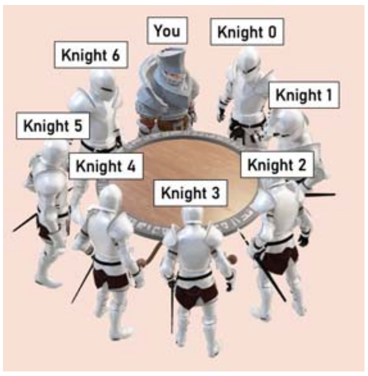


Figure 2: When you are inserted between knight 0 and knight 6.

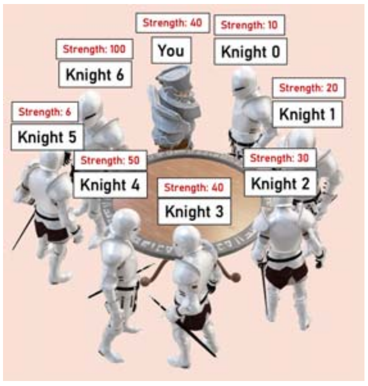


Figure 3: In round 1, you decide to fight knight 0.

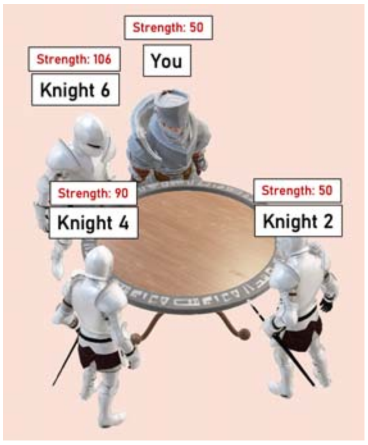


Figure 4: The result of round 1.

SAMPLE GRADER

The sample grader reads the input in the following format:

- The first line consists of two integers, N, Q.
- The second line consists of $2^N - 1$ integers, $S[0], S[1], \ldots, S[2^N - 2]$.
- The third line consists of N integers, $V[1], \ldots, V[N]$.
- The next Q lines each describe a query. The j-th line (0-based) consists of one integer, $T[j]$, representing the strength level to be considered in query j.

The output of the sample grader is in the following format:

- Q lines consists of a single integer, where the j-th line (0-based) is the return value of `query(T[j])`.

SAMPLE TESTS

Input / **Output**

1

```
3 2
10 20 30 40 50 6 100
1 10 100
40
888
```

```
24
700
```

2

```
4 5
3 1 4 1 5 9 2 6 5 3 5 8 9 7 0
2 3 5 10
8
6
0
4
7
```

```
88
49
0
23
81
```

SUBTASKS

For all cases:
$2 \leq N \leq 17$
$1 \leq Q \leq 2 \times 10^5$
$0 = V[0] < V[1] < V[2] < \ldots < V[N] \leq 10^5$
$0 \leq S[i], T[j] \leq 10^9$

	Points	Constraints
1	4	$2 \leq N \leq 5$ $1 \leq Q \leq 150$ $S[0] < S[1] = S[2] = \cdots = S[2^N - 2]$
2	6	$2 \leq N \leq 5$ $1 \leq Q \leq 150$ $S[0] > S[1] = S[2] = \cdots = S[2^N - 2]$
3	10	$2 \leq N \leq 5$ $1 \leq Q \leq 150$
4	7	$2 \leq N \leq 9$ $1 \leq Q \leq 150$
5	26	$2 \leq N \leq 9$
6	24	$2 \leq N \leq 14$ $1 \leq Q \leq 150$
7	19	$2 \leq N \leq 16$
8	4	No additional constraints

T244 - TREE SPECULATION

Time Limit: 1.000 s / Memory Limit: 1024 MB

During Alice's competitive programming career, she treasures her intuitions so much that she never *proves* any observations she makes. She calls her problem-solving process *speculation*, which refers to guessing possible observations without having enough information to be certain. Yet, she remains confident in her own conclusions.

Charlie is intrigued by such a problem-solving process so he decides to call Alice and Bob over for an experiment. He takes out a map showing N cities in Hackerland, where $N-1$ distinct pairs of cities are connected by bidirectional roads. The cities are numbered from 1 to N and the i-th road ($0 \le i < N-1$) is described by its two endpoints, U_i and V_i. It is always possible to travel from one city to any other city using one or more roads. In other words, the structure of the roads in Hackerland, when viewed as a graph, is a tree.

Next, Charlie generates the **pre-order traversal** C of the cities, following the rules below:

1. Charlie starts from an arbitrary city of his choice.
2. Repeat the following until all cities have been visited:
    - Let p be the city that Charlie is currently at.
    - If city p has not been visited yet, Charlie marks city p as visited and appends p to the array C.
    - If there is an unvisited city p' that is directly connected by a road from p, he travels to city p' through the road. If there are multiple such cities, Charlie picks any of them.
    - If no such city p' exists, he leaves city p through the road used when he first arrived at city p.

The order of the cities in the pre-order traversal is denoted as $C_1, C_2, \ldots, C_N$, in which the cities 1 to N will each appear exactly once. Charlie would inform Alice of the pre-order traversal $C_1, C_2, \ldots, C_N$.

After knowing the pre-order traversal, Alice could decide on an arbitrary constant k, and then draw a graph with k vertices (labelled from 1 to k) and $k-1$ undirected edges on a paper, such that it is always possible to travel to one vertex to another using one or more edges. In other words, the graph that Alice draws must be a tree. Alice has full control on the structure of the graph, i.e. the labels of the vertices connected by each edge. Alice hands this paper over to Charlie.

After receiving the graph from Alice, Charlie will choose one vertex in the graph as Bob's starting vertex (vertices $1, 2, \ldots, k$ are all possible). At any instant, Bob will only see the index of the vertex he is currently at, as well as the indices of all vertices which are directly connected to that vertex via an edge. Bob could travel to a neighbouring vertex (a vertex v is a neighbour of another vertex u if and only if there is an edge connecting the vertices u and v), but unfortunately, **Bob is only allowed to travel five times**.

Additionally, Bob can ask Charlie questions on the road network of Hackerland. For each question, Bob specifies two subsets of cities A and B in the network, such that A and B do not share any common city. Charlie would respond with `Yes` if there is a road directly connecting some city in A to some city in B, or `No` otherwise.

From Alice's graph and Charlie's answers to his questions, Bob must deduce the entire road map of Hackerland. Of course, as Charlie is busy, he would hope that the value of k (the number of vertices in Alice's graph) and q (the number of questions asked by Bob) are as small as possible.

Alice would like to collaborate with Bob to solve Charlie's challenge, but they are not smart enough to do so. They are now asking you for help. Devise (hopefully not speculate!) a strategy for both Alice and Bob, so that they can always deduce the road network successfully as a team.

IMPLEMENTATION DETAILS

You should implement the two procedures `Alice` and `Bob`:

```
pair(int, int)[] Alice(int N, int[] C)
```

- This procedure will be called at most 100 times per run. You should implement Alice's strategy here.
- N: The number of cities in Hackerland.
- C: An array of N integers, the pre-order traversal of the cities in Hackerland as defined above.
- This procedure should return an array p with length $k-1$. For each $0 \leq i < k-1$ with $p_i = (x_i, y_i)$, there should be a edge directly connecting vertices x_i and y_i in Alice's graph, where $1 \leq x_i, y_i \leq k$.
- k (the number of vertices in Alice's graph) is arbitrary and can be defined by Alice, but k should be as small as possible and your score will partially be determined by your chosen value of k.

```
pair(int, int)[] Bob(int N, int s, int[] d)
```

- This procedure will be called at most 100 times per run. You should implement Bob's strategy here.
- N: The number of cities in Hackerland.
- s: The city index in Alice's graph that Bob starts at. It is guaranteed that $1 \leq s \leq k$, but note that Bob does not know the value of k.
- d: The list of neighbouring vertices of vertex s, in Alice's graph. It is guaranteed that all numbers in d range from 1 to k, and they are pairwise distinct.
- This procedure should return an array r with length $N-1$. For each $0 \leq i < N-1$ with $r_i = (u_i, v_i)$, there should be a road connecting cities u_i and v_i in Hackerland's road network, where $1 \leq u_i, v_i \leq N$. The road network should be equivalent to Charlie's network, except that the order and direction of the roads do not matter.

In addition, you may call the following two functions in `Bob` any time:

```
int[] travel(int t)
```

- Bob travels to a neighbouring vertex t in Alice's graph. t must be a neighbour of Bob's current location , i.e. t must be in the array d.
- The function returns the array d', the neighbouring vertices of vertex t in Alice's graph. It is your responsibility to update the values s and d accordingly in the procedure `Bob`.
- You may only call this function **at most 5 times** in a single call of `Bob`.

```
bool ask(int[] a, int[] b)
```

- Bob asks Charlie a question with the sets a and b as described above. a and b can only contain integers from 1 to N, and they cannot share common numbers. There are no other restrictions on the sizes of a and b.
- The function returns `true` if Charlie answers the question with `Yes`, and the function returns `false` otherwise.
- You may only call this function **at most 5000 times** in a single call of `Bob`. Moreover, a high number of calls will result in significant penalties in your score (see the "Scoring" section).

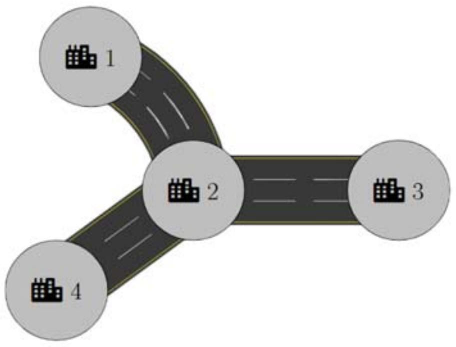
Important Notice

Note that the function `Alice` and `Bob` will be called in **two separate processes**. Contents of the variables in the program cannot be shared between the Alice side and the Bob side.

Besides, remember that there may be **multiple calls** to `Alice` or `Bob` **within the same run**. Ensure that you initialized the state properly in both procedures.

EXAMPLE

Consider the following road network of Hackerland with $N = 4$ cities, which corresponds to Sample Test 1:

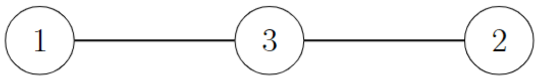


One of the pre-order traversals of the cities would be `[4, 2, 3, 1]`. There are other possible pre-order traversals including `[2, 3, 1, 4]` and `[4, 2, 1, 3]`.

The procedure `Alice` is first called with the pre-order traversal:

```
Alice(4, [4, 2, 3, 1])
```

Then, Alice could return the array consisting of the pairs `[1, 3]` and `[2, 3]`, which corresponds to the following graph:



Assume that Bob starts at vertex 2 in the graph. Then the procedure `Bob` will be called with the following parameters:

```
Bob(4, 2, [3])
```

The parameters denote that Bob is currently at vertex 2 in the graph and its only neighbour is vertex 3.

Then, Bob may call the following functions:

Function Call	Returns	Explanation
`travel(3)`	`[2, 1]`	Bob travels to city 3 in Alice's graph. Now, Charlie reports that vertices 2 and 1 are the neighbours of vertex 3. Note that the vertices come in no particular order.
`travel(2)`	`[3]`	Bob travels back to vertex 2. Now, Charlie reports that its only neighbour is vertex 3.
`ask([4, 3], [1, 2])`	`true`	Bob asks whether there is a road connecting from either city 4 or city 3, to either city 1 or city 2. There are two such roads: $(4, 2)$ and $(3, 2)$. Therefore, Charlie answers the question with `Yes`.
`ask([1, 3], [4])`	`false`	Bob asks whether there is a road connecting from either city 1 or city 3 to city 4. No such road exists, thus Charlie answers the question with `No`.
`travel(3)`	`[2, 1]`	Bob travels to vertex 3 again in Alice's graph.

At the end, the procedure `Bob` may return the array consisting of the pairs `[1, 2]`, `[2, 4]` and `[3, 2]`. There are other acceptable answers, for example, Bob may also return the array consisting of the pairs `[1, 2]`, `[2, 3]` and `[4, 2]`.

The return values and function calls in this example are not necessarily meaningful.

SAMPLE GRADER

The sample grader reads input in the following format:

- The first line consists of one integer N.
- The next $N-1$ lines each consists of two integers U_i and V_i.
- The next line consists of N integers $C_1, C_2, \dots, C_N$.

The sample grader runs your program 5 times, each of which the initial location of Bob is chosen pseudo-randomly from 1 to k.

For each run, if your program is judged as incorrect, the type of the error will be outputted (`Wrong Answer: Reason`).

Otherwise, the program outputs `Accepted` followed by the value of k and q.

The grader does **NOT** check whether the input or Alice's graph is valid. If either is invalid, unexpected behaviours may happen.

SAMPLE TESTS

	Input	Output

1	4
	1 2
	2 4
	3 2
	4 2 3 1
		Accepted: k = 3 and q = 2

SUBTASKS

	Points	Constraints
1	100	$1 \le N \le 100$. Refer to the "Scoring" section for the partial scoring formula.

SCORING

For a test case, you will receive zero score if your program does not exit correctly, returns an incorrect road network, or violates any requirements in the task description. Otherwise, your score will be calculated using k and q as follows:

The **base score** s_0 is calculated according to k using the following formula:

$$s_0 = \begin{cases} 100, & \text{if } k \le 150 \\ 98 - 2(k-150), & \text{if } 150 < k \le 154 \\ 40 + \dfrac{7700}{k}, & \text{if } 154 < k \le 2 \times 10^4 \\ 0, & \text{if } k > 2 \times 10^4 \end{cases}$$

Your **final score** *score* will then be further adjusted according to q as follows:

$$score = \begin{cases} s_0, & \text{if } q \le 400 \\ 62\% \cdot s_0, & \text{if } 400 < q \le 600 \\ 29\% \cdot s_0, & \text{if } 600 < q \le 1000 \\ 21\% \cdot s_0, & \text{if } 1000 < q \le 1600 \\ 10\% \cdot s_0, & \text{if } 1600 < q \le 5000 \\ 0, & \text{if } q > 5000 \end{cases}$$

Among runs within the same test case, the maximum values of k and q will be used to calculate your score of the test case. Your score for this task is the lowest score you get among all test cases.