Paint By Numbers

Paint By Numbers is a well-known puzzle game. We consider a simple one-dimensional version of this puzzle. In this puzzle, the player is given a row of n cells. The cells are numbered 0 through $n-1$ from the left to the right. The player has to paint each cell black or white. We use 'X' to denote black cells and '_' to denote white cells.

The player is given a sequence $c = [c_0, \ldots, c_{k-1}]$ of k positive integers: the *clues*. He has to paint the cells in a way such that the black cells in the row form exactly k blocks of consecutive cells. Moreover, the number of black cells in the i-th block (0-based) from the left should be equal to c_i. For example, if the clues are $c = [3, 4]$, the solved puzzle must have exactly two blocks of consecutive black cells: one of length 3 and then another of length 4. Hence, if $n = 10$ and $c = [3, 4]$, one solution satisfying the clues is "_XXX__XXXX". Note that "XXXX_XXX__" does not satisfy the clues because the blocks of black cells are not in the correct order. Also, "__XXXXXXX_" does not satisfy the clues because there is a single block of black cells, not two separate blocks.

You are given a partially solved Paint By Numbers puzzle. That is, you know n and c, and additionally you know that some cells must be black and some cells must be white. Your task is to deduce additional information about the cells.

Specifically, a *valid solution* is one that satisfies the clues, and also agrees with the colors of the known cells. Your program should find cells that are painted black in every valid solution, and cells that are painted white in every valid solution.

You may assume that the input is such that there is at least one valid solution.

Implementation details

You should implement the following function (method):

- `string solve_puzzle(string s, int[] c)`.
  - `s`: string of length n. For each i ($0 \le i \le n-1$) character i is:
    - 'X', if cell i must be black,
    - '_', if cell i must be white,
    - '.', if there is no information about cell i.
  - `c`: array of length k containing clues, as defined above,
  - the function should return a string of length n. For each i ($0 \le i \le n-1$) character i of the output string should be:
    - 'X', if cell i is black in every valid solution,
    - '_', if cell i is white in every valid solution,

    - '?', otherwise (i.e., if there exist two valid solutions such that cell i is black in one of them and white in the other one).

In the C language the function signature is a bit different:

- `void solve_puzzle(int n, char* s, int k, int* c, char* result)`
    - `n`: length of the string `s` (number of cells),
    - `k`: length of the array `c` (number of clues),
    - the other parameters are the same as above,
    - instead of returning a string of n characters, the function should write the answer to the string `result`.

The ASCII codes of characters used in this problem are:

- 'X': 88,
- '_': 95,
- '.': 46,
- '?': 63.

Please use the provided template files for details of implementation in your programming language.

Examples

Example 1

```
solve_puzzle("..........", [3, 4])
```

These are all possible valid solutions of the puzzle:

- "XXX_XXXX__",
- "XXX__XXXX_",
- "XXX___XXXX",
- "_XXX_XXXX_",
- "_XXX__XXXX",
- "__XXX_XXXX".

One can observe that the cells with (0-based) indices 2, 6, and 7 are black in each valid solution. Each of the other cells can be, but does not have to be black. Hence, the correct answer is "??X???XX??".

Example 2

```
solve_puzzle("........", [3, 4])
```

In this example the entire solution is uniquely determined and the correct answer is "XXX_XXXX".

Example 3

```
solve_puzzle("..._._....", [3])
```

In this example we can deduce that cell 4 must be white as well — there is no way to fit three consecutive black cells between the white cells at indices 3 and 5. Hence, the correct answer is "???___????".

Example 4

```
solve_puzzle(".X........", [3])
```

There are only two valid solutions that match the above description:

- "XXX_______",
- "_XXX______".

Thus, the correct answer is "?XX?______".

Subtasks

In all subtasks $1 \leq k \leq n$, and $1 \leq c_i \leq n$ for each $0 \leq i \leq k-1$.

1. (7 points) $n \leq 20$, $k = 1$, s contains only '.' (empty puzzle),
2. (3 points) $n \leq 20$, s contains only '.',
3. (22 points) $n \leq 100$, s contains only '.',
4. (27 points) $n \leq 100$, s contains only '.' and '_' (information only about white cells),
5. (21 points) $n \leq 100$,
6. (10 points) $n \leq 5\,000$, $k \leq 100$,
7. (10 points) $n \leq 200\,000$, $k \leq 100$.

Sample grader

The sample grader reads the input in the following format:

- line 1: string s,
- line 2: integer k followed by k integers $c_0, \ldots, c_{k-1}$.

Unscrambling a Messy Bug

Ilshat is a software engineer working on efficient data structures. One day he invented a new data structure. This data structure can store a set of *non-negative* n-bit integers, where n is a power of two. That is, $n = 2^b$ for some non-negative integer b.

The data structure is initially empty. A program using the data structure has to follow the following rules:

- The program can add elements that are n-bit integers into the data structure, one at a time, by using the function `add_element(x)`. If the program tries to add an element that is already present in the data structure, nothing happens.
- After adding the last element the program should call the function `compile_set()` exactly once.
- Finally, the program may call the function `check_element(x)` to check whether the element x is present in the data structure. This function may be used multiple times.

When Ilshat first implemented this data structure, he made a bug in the function `compile_set()`. The bug reorders the binary digits of each element in the set in the same manner. Ilshat wants you to find the exact reordering of digits caused by the bug.

Formally, consider a sequence $p = [p_0, \ldots, p_{n-1}]$ in which every number from 0 to $n-1$ appears exactly once. We call such a sequence a *permutation*. Consider an element of the set, whose digits in binary are $a_0, \ldots, a_{n-1}$ (with a_0 being the most significant bit). When the function `compile_set()` is called, this element is replaced by the element $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.

The same permutation p is used to reorder the digits of every element. Any permutation is possible, including the possibility that $p_i = i$ for each $0 \leq i \leq n-1$.

For example, suppose that $n = 4$, $p = [2, 1, 3, 0]$, and you have inserted into the set integers whose binary representations are `0000`, `1100` and `0111`. Calling the function `compile_set` changes these elements to `0000`, `0101` and `1110`, respectively.

Your task is to write a program that finds the permutation p by interacting with the data structure. It should (in the following order):

1. choose a set of n-bit integers,
2. insert those integers into the data structure,

3. call the function `compile_set` to trigger the bug,
4. check the presence of some elements in the modified set,
5. use that information to determine and return the permutation p.

Note that your program may call the function `compile_set` only once.

In addition, there is a limit on the number of times your program calls the library functions. Namely, it may

- call `add_element` at most w times (w is for "writes"),
- call `check_element` at most r times (r is for "reads").

Implementation details

You should implement one function (method):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: the number of bits in the binary representation of each element of the set (and also the length of p).
  - `w`: the maximum number of `add_element` operations your program can perform.
  - `r`: the maximum number of `check_element` operations your program can perform.
  - the function should return the restored permutation p.

In the C language, the function prototype is a bit different:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` and `r` have the same meaning as above.
  - the function should return the restored permutation p by storing it into the provided array `result`: for each i, it should store the value p_i into `result[i]`.

Library functions

In order to interact with the data structure, your program should use the following three functions (methods):

- `void add_element(string x)`
  This function adds the element described by `x` to the set.
  - `x`: a string of `'0'` and `'1'` characters giving the binary representation of an integer that should be added to the set. The length of `x` must be n.

- `void compile_set()`
  This function must be called exactly once. Your program cannot call `add_element()` after this call. Your program cannot call `check_element()` before this call.
- `boolean check_element(string x)`
  This function checks whether the element `x` is in the modified set.
  - `x`: a string of `'0'` and `'1'` characters giving the binary representation of the element that should be checked. The length of `x` must be n.
  - returns `true` if element `x` is in the modified set, and `false` otherwise.

Note that if your program violates any of the above restrictions, its grading outcome

will be "Wrong Answer".

For all the strings, the first character gives the most significant bit of the corresponding integer.

The grader fixes the permutation p before the function `restore_permutation` is called.

Please use the provided template files for details of implementation in your programming language.

Example

The grader makes the following function call:

- `restore_permutation(4, 16, 16)`. We have $n = 4$ and the program can do at most 16 "writes" and 16 "reads".

The program makes the following function calls:

- `add_element("0001")`
- `add_element("0011")`
- `add_element("0100")`
- `compile_set()`
- `check_element("0001")` returns `false`
- `check_element("0010")` returns `true`
- `check_element("0100")` returns `true`
- `check_element("1000")` returns `false`
- `check_element("0011")` returns `false`
- `check_element("0101")` returns `false`
- `check_element("1001")` returns `false`
- `check_element("0110")` returns `false`
- `check_element("1010")` returns `true`
- `check_element("1100")` returns `false`

Only one permutation is consistent with these values returned by `check_element()`: the permutation $p = [2, 1, 3, 0]$. Thus, `restore_permutation` should return `[2, 1, 3, 0]`.

Subtasks

1. (20 points) $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ for at most 2 indices i ($0 \leq i \leq n - 1$),
2. (18 points) $n = 32$, $w = 320$, $r = 1024$,
3. (11 points) $n = 32$, $w = 1024$, $r = 320$,
4. (21 points) $n = 128$, $w = 1792$, $r = 1792$,
5. (30 points) $n = 128$, $w = 896$, $r = 896$.

Sample grader

The sample grader reads the input in the following format:

- line 1: integers n, w, r,
- line 2: n integers giving the elements of p.

Aliens

Our satellite has just discovered an alien civilization on a remote planet. We have already obtained a low-resolution photo of a square area of the planet. The photo shows many signs of intelligent life. Our experts have identified n points of interest in the photo. The points are numbered from 0 to $n-1$. We now want to take high-resolution photos that contain all of those n points.

Internally, the satellite has divided the area of the low-resolution photo into an m by m grid of unit square cells. Both rows and columns of the grid are consecutively numbered from 0 to $m-1$ (from the top and left, respectively). We use (s,t) to denote the cell in row s and column t. The point number i is located in the cell (r_i, c_i). Each cell may contain an arbitrary number of these points.

Our satellite is on a stable orbit that passes directly over the *main* diagonal of the grid. The main diagonal is the line segment that connects the top left and the bottom right corner of the grid. The satellite can take a high-resolution photo of any area that satisfies the following constraints:

- the shape of the area is a square,
- two opposite corners of the square both lie on the main diagonal of the grid,
- each cell of the grid is either completely inside or completely outside the photographed area.

The satellite is able to take at most k high-resolution photos.

Once the satellite is done taking photos, it will transmit the high-resolution photo of each photographed cell to our home base (regardless of whether that cell contains some points of interest). The data for each photographed cell will only be transmitted *once*, even if the cell was photographed several times.

Thus, we have to choose at most k square areas that will be photographed, assuring that:

- each cell containing at least one point of interest is photographed at least once, and
- the number of cells that are photographed at least once is minimized.

Your task is to find the smallest possible total number of photographed cells.

Implementation details

You should implement the following function (method):

- `int64 take_photos(int n, int m, int k, int[] r, int[] c)`
  - `n`: the number of points of interest,
  - `m`: the number of rows (and also columns) in the grid,

- k: the maximum number of photos the satellite can take,
- r and c: two arrays of length n describing the coordinates of the grid cells that contain points of interest. For $0 \leq i \leq n-1$, the i-th point of interest is located in the cell (r[i], c[i]),
- the function should return the smallest possible total number of cells that are photographed at least once (given that the photos must cover all points of interest).

Please use the provided template files for details of implementation in your programming language.

Examples

Example 1

```
take_photos(5, 7, 2, [0, 4, 4, 4, 4], [3, 4, 6, 5, 6])
```

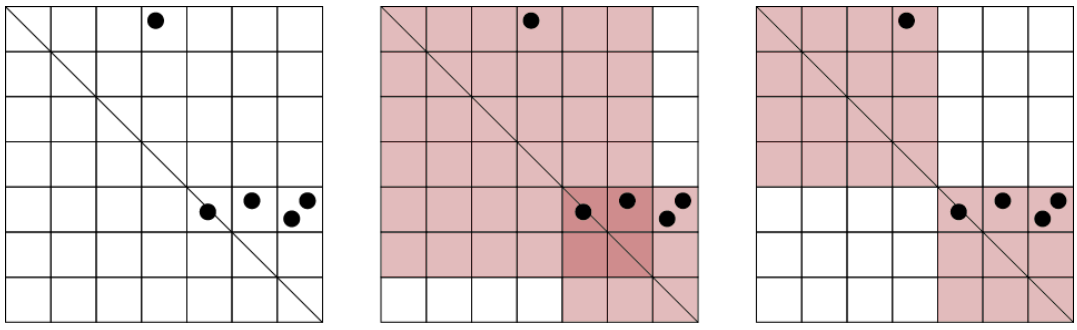
In this example we have a 7×7 grid with 5 points of interest. The points of interest are located in four different cells: $(0, 3)$, $(4, 4)$, $(4, 5)$ and $(4, 6)$. You may take at most 2 high-resolution photos.

One way to capture all five points of interest is to make two photos: a photo of the 6×6 square containing the cells $(0, 0)$ and $(5, 5)$, and a photo of the 3×3 square containing the cells $(4, 4)$ and $(6, 6)$. If the satellite takes these two photos, it will transmit the data about 41 cells. This amount is not optimal.

The optimal solution uses one photo to capture the 4×4 square containing cells $(0, 0)$ and $(3, 3)$ and another photo to capture the 3×3 square containing cells $(4, 4)$ and $(6, 6)$. This results in only 25 photographed cells, which is optimal, so take_photos should return 25.

Note that it is sufficient to photograph the cell $(4, 6)$ once, even though it contains two points of interest.

This example is shown in the figures below. The leftmost figure shows the grid that corresponds to this example. The middle figure shows the suboptimal solution in which 41 cells were photographed. The rightmost figure shows the optimal solution.

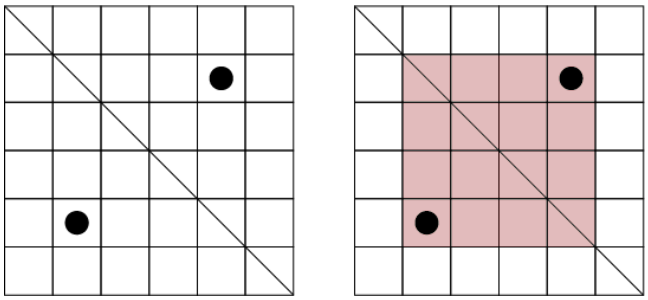


Example 2

```
take_photos(2, 6, 2, [1, 4], [4, 1])
```

Here we have 2 points of interest located symmetrically: in the cells $(1, 4)$ and $(4, 1)$. Any valid photo that contains one of them contains the other one as well. Therefore, it is sufficient to use a single photo.

The figures below show this example and its optimal solution. In this solution the satellite captures a single photo of 16 cells.



Subtasks

For all subtasks, $1 \le k \le n$.

1. (4 points) $1 \le n \le 50$, $1 \le m \le 100$, $k = n$,
2. (12 points) $1 \le n \le 500$, $1 \le m \le 1000$, for all i such that $0 \le i \le n - 1$, $r_i = c_i$,
3. (9 points) $1 \le n \le 500$, $1 \le m \le 1000$,
4. (16 points) $1 \le n \le 4000$, $1 \le m \le 1\,000\,000$,
5. (19 points) $1 \le n \le 50\,000$, $1 \le k \le 100$, $1 \le m \le 1\,000\,000$,
6. (40 points) $1 \le n \le 100\,000$, $1 \le m \le 1\,000\,000$.

Sample grader

The sample grader reads the input in the following format:

- line 1: integers n, m and k,
- line 2 + i ($0 \le i \le n - 1$): integers r_i and c_i.