バグの解読 (Unscrambling a Messy Bug)

Ilshat はデータ構造の開発に携わるソフトウェアエンジニアである．ある日，彼は新しいデータ構造を開発した．このデータ構造は n ビットの*非負整数*の集合を格納する．ただし，n は 2 の冪である，すなわち，ある非負整数 b に対し $n = 2^b$ が成り立つ．

このデータ構造は，初めは空である．このデータ構造を使うプログラムは以下のルールに従わなければならない．

- 関数 `add_element(x)` を使い，いくつかの n ビットの整数を 1 つずつデータ構造に追加する．もし，すでにデータ構造の中に格納されている追加しようとした場合は，何も起こらない．
- 最後の整数を追加したら，関数 `compile_set()` をちょうど 1 回呼び出さなければならない．
- 最後に，関数 `check_element(x)` を使い整数 x がデータ構造に格納されているか確認することができる．この関数は複数回使うことができる．

Ilshat がこのデータ構造を実装したとき，彼は `compile_set()` にバグを埋め込んでしまった．このバグは，集合の要素の 2 進数での桁を，すべての要素について同じ方法で並べ替えてしまう．Ilshat はこのバグによる並べ替え方を知りたい．

より正確に記述する．0 から $n-1$ のそれぞれの数が 1 回ずつ現れる数列 $p = [p_0, \ldots, p_{n-1}]$ を*順列*と呼ぶ．集合のある要素の 2 進表記を $a_0, \ldots, a_{n-1}$ としよう．ただし，a_0 は最上位ビットとする．関数 `compile_set()` が呼ばれると，この要素は $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$ に置き換えられる．

集合のそれぞれの要素に対し，並べ替えには同じ順列 p が用いられる．この順列としては，すべての $0 \leqq i \leqq n-1$ に対し $p_i = i$ となる順列を含む，任意のものが考えられる．

例えば，$n = 4$，$p = [2, 1, 3, 0]$ で，2 進表記が `0000`，`1100`，`0111` となる整数を集合に追加した場合を考えよう．関数 `compile_set` を呼び出すことで，これらの要素はそれぞれ `0000`，`0101`，`1110` に変化する．

あなたの仕事は，このデータ構造を使うことによって，順列 p を見つけることである．あなたのプログラムは，以下の順に処理を行わなければならない．

1. n ビットの整数の集合を選ぶ．
2. それらの要素をデータ構造に追加する．
3. 関数 `compile_set` を呼び出すことでバグを引き起こす．
4. バグの起こった後の集合においていくつかの要素が存在するか確認する．
5. その情報を使い，順列 p を返す．

`compile_set` は 1 回だけ呼ばなければならないことに注意せよ．

さらに，データ構造のライブラリ関数を呼ぶ回数には制限がある．プログラムは

- `add_element` を高々 w 回（w は "writes" を意味する），
- `check_element` を高々 r 回（r は "reads" を意味する）

までしか呼び出せない．

実装の詳細 (Implementation details)

あなたは以下の関数を実装しなければならない．

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: 集合に追加できる要素の 2 進表記でのビット数．これは p の長さでもある．
  - `w`: `add_element` を呼び出すことのできる回数の上限．
  - `r`: `check_element` を呼び出すことのできる回数の上限．
  - この関数は復元した順列 p を返さなくてはならない．

C言語の場合，引数及び戻り値の形式が少し異なる．

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w`, `r` は上記のものと同じである．
  - この関数は，引数として与えられた配列 `result` に復元した順列 p を格納しなければならない．すなわち，それぞれの i に対し p_i を `result[i]` に格納しなければならない．

ライブラリ関数 (Library functions)

データ構造と通信するため，あなたのプログラムは以下の関数を使わなければならない．

- `void add_element(string x)`
  この関数は集合に `x` で表される要素を追加する．
  - `x`: `'0'` と `'1'` からなる，集合に追加する整数の 2 進表記となる文字列．`x` の長さは n でなければならない．

- `void compile_set()`
  この関数はちょうど 1 回呼ばなければならない．この関数を呼んだ後に，`add_element()`を呼んではならない．この関数を呼ぶ前に，`check_element()`を呼んではならない．
- `boolean check_element(string x)`
  この関数はバグの起こった後の集合に `x` で表される要素が存在するか確認する．
  - `x`: `'0'` と `'1'` からなる，集合に存在するか確認する整数の 2 進表記となる文字列．`x` の長さは n でなければならない．
  - `x` がバグの起こった後の集合に存在するなら `true` を，そうでなければ `false` を返す．

あなたのプログラムが上記の制約のいずれかを満たさないとき，採点結果は "Wrong Answer" となる．

すべての文字列について，最初の文字が文字列の表す整数の最上位ビットとなる．

採点プログラムは，関数 `restore_permutation` が呼ばれる前に順列 p を固定する．

実装の詳細については，各言語のテンプレートファイルを参照せよ．

例 (Example)

採点プログラムが以下の関数呼び出しをする場合，

- `restore_permutation(4, 16, 16)`

$n = 4$ であり，あなたのプログラムは高々 16 回の ”writes” と，高々 16 回の ”reads” を行うことができる．

あなたのプログラムが以下の関数呼び出しをしたとする．

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

これらの関数呼び出しと矛盾しない順列は $p = [2, 1, 3, 0]$ だけである．よって，`restore_permutation` は `[2, 1, 3, 0]` を返さなければならない．

小課題 (Subtasks)

1. (20 点): $p_i \neq i$ （$0 \leqq i \leqq n-1$）なる i は高々 2 個 かつ $n = 8$ かつ $w = 256$ かつ $r = 256$ を満たす．
2. (18 点): $n = 32$ かつ $w = 320$ かつ $r = 1024$ を満たす．
3. (11 点): $n = 32$ かつ $w = 1024$ かつ $r = 320$ を満たす．
4. (21 点): $n = 128$ かつ $w = 1792$ かつ $r = 1792$ を満たす．
5. (30 点): $n = 128$ かつ $w = 896$ かつ $r = 896$ を満たす．

採点プログラムのサンプル (Sample grader)

採点プログラムのサンプルは，以下のフォーマットで入力を読み込む:

- 1 行目: 整数 n，w，r．
- 2 行目: 順列 p を表す n 個の整数．