Министерство образования и науки Российской Федерации федеральное государственное
автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

# Алгоритмы и структуры данных

## Лабораторная работа №3
## Timus

**Выполнил**
Григорьев Давид Владимирович
Группа: P3215
**Преподаватели**
Косяков Михаил Сергеевич
Тараканов Денис Сергеевич

Санкт-Петербург 2025

# Содержание

**1 1494. Монобильярд 1**

**2 1521. Военные учения 2 2**

# 1 1494. Монобильярд

## Пояснение к примененному алгоритму

Задача заключается в проверке, могла ли последовательность извлечения шаров ревизором быть получена при правильном порядке закатывания шаров (1, 2, ..., N). Для решения используется симуляция стека:

1. **Проверка корректности входных данных**: - Последовательность должна быть перестановкой чисел от 1 до N. - Наличие повторяющихся чисел или чисел вне диапазона приводит к немедленному ответу "Cheater".

2. **Симуляция стека**: - Последовательно добавляем шары в стек в порядке 1, 2, ..., N. - При каждом извлечении проверяем, соответствует ли вершина стека текущему элементу последовательности. - Если в любой момент извлечение невозможно или стек не пуст после обработки всех элементов, ответ "Cheater".

## Код алгоритма

```
#include <iostream>
#include <stack>

int main() {
  int sequence_length = 0;
  std::cin >> sequence_length;

  std::stack<int> expected_sequence;
  int highest_ball = 0;

  for (int step = 0; step < sequence_length; ++step) {
    int current_ball = 0;
    std::cin >> current_ball;

    if (current_ball > highest_ball) {
      // Populate stack with intermediate values between previous max and current
      for (int missing_ball = highest_ball + 1; missing_ball < current_ball; ++missing_b
        expected_sequence.push(missing_ball);
      }
      highest_ball = current_ball;
    } else {
      // Verify current ball matches the expected next in sequence
      if (!expected_sequence.empty() && current_ball == expected_sequence.top()) {
        expected_sequence.pop();
      } else {
        std::cout << "Cheater";
        return 0;
      }
    }
  }

  std::cout << "Not a proof";
  return 0;
}
```

# 2 1521. Военные учения 2

## Пояснение к примененному алгоритму

Задача требует определения порядка выбывания солдат из круга, где каждый следующий исключаемый находится на $K$ позиций впереди предыдущего. Для решения используется модификация задачи Иосифа с применением **дерева отрезков**:

### Основные шаги алгоритма

1. **Инициализация дерева отрезков**: - Дерево строится для $N$ элементов, где каждый лист соответствует солдату (значение 1 — активен, 0 — удален). - Внутренние узлы хранят сумму поддеревьев, что позволяет быстро находить количество активных солдат. - Размер дерева выбирается как ближайшая степень двойки $\geq 2N$.

2. **Поиск следующего солдата**: - Начиная с текущей позиции, ищется $K$-ый активный солдат: - Если в левом поддереве достаточно элементов — переход влево. - Иначе — вычитание размера левого поддерева и переход вправо. - Обход дерева выполняется за $O(\log N)$.

3. **Обновление дерева**: - После исключения солдата, соответствующий лист обнуляется, и суммы в родительских узлах пересчитываются.

4. **Эффективность**: - Сложность алгоритма: $O(N \log N)$, что позволяет обрабатывать $N \leq 10^5$.

## Пример работы

Для $N = 3, K = 2$:

1. Первым исключается солдат 2.
2. Отсчет от 2 на 2 позиции: 3 → 1. Исключается 1.
3. Остается 3. Исключается последним.

Итоговый порядок: `2 1 3`.

## Код алгоритма

```cpp
#include <cstddef>
#include <cstdio>
#include <iostream>
#include <vector>

class JosephusTree {
  std::vector<int> segment_tree_;
  int total_soldiers_;
  int last_internal_node_;
  int elimination_step_;
  int current_position_;
  int remaining_count_;

  static unsigned NextPowerOfTwo(unsigned n) {
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return n + 1;
```

```
  }

  static constexpr int Parent(int node) {
    return node / 2;
  }
  static constexpr int Left(int node) {
    return node * 2;
  }
  static constexpr int Right(int node) {
    return (node * 2) + 1;
  }

  bool Valid(int node) const {
    return node <= last_internal_node_ + total_soldiers_ && node > 0;
  }

public:
  JosephusTree(int n, int k)
      : total_soldiers_(n), elimination_step_(k), current_position_(k), remaining_count_
    // Build tree structure
    unsigned req_size = NextPowerOfTwo(n * 2);
    segment_tree_.resize(req_size);

    // Calculate tree height
    unsigned m = (n * 2) - 1;
    int height = 31 - __builtin_clz(m);  // Equivalent to bit_width-1
    last_internal_node_ = (1 << height) - 1;

    // Initialize leaves
    for (int i = last_internal_node_ + 1; i <= last_internal_node_ + n; ++i) {
      segment_tree_[static_cast<size_t>(i)] = 1;
    }

    // Build internal nodes
    for (int i = last_internal_node_; i >= 1; --i) {
      segment_tree_[static_cast<size_t>(i)] =
          (Valid(Left(i)) ? segment_tree_[static_cast<size_t>(Left(i))] : 0) +
          (Valid(Right(i)) ? segment_tree_[static_cast<size_t>(Right(i))] : 0);
    }
  }

  int Remaining() const {
    return remaining_count_;
  }

  int EliminateNext() {
    if (remaining_count_-- == total_soldiers_) {
      return current_position_;
    }

    // Update tree counts
    size_t node = last_internal_node_ + current_position_;
    while (node > 0) {
      segment_tree_[node]--;
      node = Parent(node);
```

```
    }

    // Find next position
    node = last_internal_node_ + current_position_;
    int k = elimination_step_;
    enum class Dir { FromLeft, FromRight, FromAbove };
    Dir dir = Dir::FromRight;

    while (k > 0) {
      const size_t l = static_cast<size_t>(Left(node));
      const size_t r = static_cast<size_t>(Right(node));

      if (dir == Dir::FromAbove) {
        if (Valid(l) && k > segment_tree_[l]) {
          k -= segment_tree_[l];
          node = r;
        } else if (!Valid(l) && k == segment_tree_[node]) {
          k--;
        } else {
          node = l;
        }
      } else if (dir == Dir::FromRight) {
        dir = (node == Right(Parent(node))) ? Dir::FromRight : Dir::FromLeft;
        node = Parent(node);
      } else {  // FROM_LEFT
        if (Valid(r) && k > segment_tree_[r]) {
          k -= segment_tree_[r];
          dir = (node == Right(Parent(node))) ? Dir::FromRight : Dir::FromLeft;
          node = Parent(node);
        } else {
          node = r;
          dir = Dir::FromAbove;
        }
      }

      if (!Valid(node)) {
        node = last_internal_node_ + 1;
        k -= segment_tree_[node];
        dir = Dir::FromRight;
      }
    }

    current_position_ = node - last_internal_node_;
    return current_position_;
  }
};

int main() {
  int N = 0;
  int K = 0;
  std::cin >> N >> K;

  JosephusTree jt(N, K);
  while (jt.Remaining()) {
    int n = jt.EliminateNext();
```

```
    char buf[16];
    int p = 0;
    do
      buf[p++] = n % 10 + '0';
    while (n /= 10);
    while (p--) {
      putchar(buf[p]);
    }
    putchar(' ');
  }
}
```