Задача D

1) Алгоритм

Для решения задачи была выбрана структура данных — декартово дерево с неявными ключами, т.к. позволяет эффективно реализовать операции вставки, удаления, поиска и разделения.

Декартово дерево позволяет нам хранить элементы в отсортированном порядке и поддерживать балансировку. Мы использовали две декартовы структуры: fst_tree и scnd_tree:

- fst_tree хранит элементы массива, которые имеют четные индексы.
- scnd_tree хранит элементы массива, которые имеют нечетные индексы.

Операция "Обмен значений на отрезке" – для реализации операции обмена значений на четном отрезке мы использовали методы decSplit и decMerge для разделения и объединения поддеревьев. В начале мы определяем, какие значения лежат на четных и нечетных индексах в указанном отрезке:

- Используем метод decSplit, чтобы разделить каждое дерево (fst_tree и scnd_tree) на три части: элементы до начала отрезка, элементы на самом отрезке и элементы после отрезка.
- Меняем местами внутренние отрезки обоих деревьев.
- Собираем обратно все части деревьев с помощью decMerge.

Операция "Вычисление суммы на отрезке" - для вычисления суммы на заданном отрезке:

1. Определяем, какие индексы принадлежат деревьям fst_tree и scnd_tree.
2. С помощью метода decSplit извлекаем поддеревья, содержащие значения на данном отрезке.
3. Складываем значения узлов в извлеченных поддеревьях для получения общей суммы.
4. Восстанавливаем оригинальные деревья путем объединения поддеревьев с помощью decMerge.

Функция для очистки дерева - для предотвращения утечек памяти была добавлена функция clear, которая рекурсивно освобождает память, выделенную для каждого узла дерева.

2) Сложность решения

- Вставка элементов в дерево имеет сложность O(logN) для каждого элемента. Следовательно, общая сложность инициализации составляет O(NlogN).
- Операция обмена значений на отрезке состоит из нескольких операций разделения и объединения, каждая из которых имеет сложность O(logN). Следовательно, сложность операции обмена равна O(logN).

- Операция вычисления суммы на отрезке также требует выполнения операций разделения и объединения, что обеспечивает сложность O(logN) для одного запроса.

Таким образом, обе операции (обмен значений и вычисление суммы) имеют сложность O(logN) для каждого запроса, что позволяет эффективно обрабатывать большое количество запросов.

Используемая память Память используется для хранения двух декартовых деревьев, каждое из которых хранит примерно половину элементов исходного массива. В худшем случае память потребуется для хранения всех узлов, что соответствует O(N).

Обоснование корректности Декартово дерево позволяет нам поддерживать отсортированный порядок элементов и быстро выполнять разделение и объединение. Таким образом, операции, требующие доступа к элементам по их индексам (например, обмен значений и вычисление суммы), реализуются корректно и эффективно. Мы гарантируем, что каждая операция с деревом обновляет его состояние, чтобы сохранять актуальную информацию о размерах и суммах поддеревьев.

3) Распечатка кода

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <random>
#include <utility>

long long even_end, even_start, odd_end, odd_start;

using namespace std;

struct DecartTree {
  static std::minstd_rand prior_gen;

  struct DecNode {
    long long priority;
    long long val, size = 1;
    long long sum = 0;
    DecNode* left_tree = nullptr;
    DecNode* rigth_tree = nullptr;
    DecNode(long long val) : priority(prior_gen()), val(val), sum(val) {
    }
```

```
  };

  DecNode* root = nullptr;

  static long long takeIndex(DecNode* node, long long cur_value = 0) {
    if (node->left_tree != nullptr) {
      return node->left_tree->size + cur_value + 1;
    } else {
      return cur_value + 1;
    }
  }

  static long long getSize(DecNode* node) {
    return node ? node->size : 0;
  }

  static long long getsum(DecNode* node) {
    return node ? node->sum : 0;
  }

  static void update(DecNode* node) {
    if (node) {
      node->size = getSize(node->left_tree) + 1 + getSize(node-
>rigth_tree);
      node->sum = getsum(node->left_tree) + getsum(node->rigth_tree) +
node->val;
    }
  }

  static void decSplit(DecNode* root, long long key, DecNode*& fst_root,
DecNode*& scnd_root) {
    if (!root) {
      fst_root = scnd_root = nullptr;
      return;
    }

    if (takeIndex(root) <= key) {
```

```
      decSplit(root->rigth_tree, key - takeIndex(root), root->rigth_tree,
scnd_root);
      fst_root = root;
    } else {
      decSplit(root->left_tree, key, fst_root, root->left_tree);
      scnd_root = root;
    }
    update(fst_root);
    update(scnd_root);
  }

  static DecNode* decMerge(DecNode* fst_node, DecNode* scnd_node) {
    if (!fst_node || !scnd_node) {
      return fst_node ? fst_node : scnd_node;
    }
    if (fst_node->priority > scnd_node->priority) {
      fst_node->rigth_tree = decMerge(fst_node->rigth_tree, scnd_node);
      update(fst_node);
      return fst_node;
    }
    scnd_node->left_tree = decMerge(fst_node, scnd_node->left_tree);
    update(scnd_node);
    return scnd_node;
  }

  static void pushBack(DecNode*& root, long long value) {
    root = decMerge(root, new DecNode(value));
  }

  static DecNode* searchForNode(DecNode* root, long long key_val, long
long value = 0) {
    if (root == nullptr) {
      return nullptr;
    }
    DecNode* res = nullptr;
    if (takeIndex(root, value) == key_val) {
      return root;
    }
```

```
    if (takeIndex(root, value) < key_val) {
      res = searchForNode(root->rigth_tree, key_val, takeIndex(root,
value));
    } else {
      res = searchForNode(root->left_tree, key_val, value);
    }
    return res;
  }
};

std::pair<DecartTree::DecNode*, DecartTree::DecNode*> replaceOp(
    DecartTree::DecNode* fst_root,
    DecartTree::DecNode* scnd_root,
    long long lft_bound,
    long long rght_bound
) {
  even_end = even_start = odd_end = odd_start = 0;

  if (rght_bound % 2 != 0) {
    even_start = lft_bound / 2;
    even_end = floor(rght_bound / 2.0);
    odd_start = even_start + 1;
    odd_end = ceil(rght_bound / 2.0);
  } else {
    if (lft_bound == 1) {
      even_start = 1;
      odd_start = 1;
    } else {
      even_start = ceil(lft_bound / 2.0);
      odd_start = ceil(lft_bound / 2.0);
    }
    even_end = odd_end = rght_bound / 2;
  }

  DecartTree::DecNode *less_fst_lft, *grtr_fst_lft, *grtr_fst_rght,
*less_fst_rght;
  DecartTree::DecNode *less_scnd_lft, *grtr_scnd_lft, *grtr_scnd_rght,
*less_scnd_rght;
```

```
  DecartTree::decSplit(fst_root, even_end, less_fst_rght, grtr_fst_rght);
  DecartTree::decSplit(less_fst_rght, even_start - 1, less_fst_lft,
grtr_fst_lft);

  DecartTree::decSplit(scnd_root, odd_end, less_scnd_rght,
grtr_scnd_rght);
  DecartTree::decSplit(less_scnd_rght, odd_start - 1, less_scnd_lft,
grtr_scnd_lft);

  return {
      DecartTree::decMerge(less_fst_lft,
DecartTree::decMerge(grtr_scnd_lft, grtr_fst_rght)),
      DecartTree::decMerge(less_scnd_lft,
DecartTree::decMerge(grtr_fst_lft, grtr_scnd_rght))
  };
}

void sumCalculateOp(
    DecartTree::DecNode* nod_tree,
    DecartTree::DecNode* odd_tree,
    long long left_bound,
    long long right,
    long long& res
) {
  even_end = even_start = odd_end = odd_start = 0;

  if (left_bound == right) {
    DecartTree::DecNode* node = nullptr;
    if (left_bound % 2 == 0) {
      even_start = left_bound == 1 ? 1 : left_bound / 2;
      node = DecartTree::searchForNode(nod_tree, even_start);
    } else {
      even_start = left_bound == 1 ? 1 : ceil(left_bound / 2.0);
      node = DecartTree::searchForNode(odd_tree, even_start);
    }
    res = node->val;
    return;
```

```
  }
  if (right % 2 == 0) {
    odd_end = right / 2;
    even_end = right / 2;
  } else {
    odd_end = ceil(right / 2.0);
    even_end = floor(right / 2.0);
  }
  if (left_bound % 2 == 0) {
    even_start = left_bound / 2;
    odd_start = even_start + 1;
  } else {
    if (left_bound == 1) {
      even_start = 1;
      odd_start = 1;
    } else {
      even_start = ceil(left_bound / 2.0);
      odd_start = ceil(left_bound / 2.0);
    }
  }

  DecartTree::DecNode *less_fst_lft, *grtr_fst_lft, *grtr_fst_rght,
*less_fst_rght;
  DecartTree::DecNode *less_scnd_lft, *grtr_scnd_lft, *grtr_scnd_rght,
*less_scnd_rght;

  DecartTree::decSplit(nod_tree, even_end, less_fst_rght, grtr_fst_rght);
  DecartTree::decSplit(less_fst_rght, even_start - 1, less_fst_lft,
grtr_fst_lft);

  DecartTree::decSplit(odd_tree, odd_end, grtr_scnd_lft, grtr_scnd_rght);
  DecartTree::decSplit(grtr_scnd_lft, odd_start - 1, less_scnd_lft,
less_scnd_rght);

  res = DecartTree::getsum(grtr_fst_lft) +
DecartTree::getsum(less_scnd_rght);
```

```
  nod_tree = DecartTree::decMerge(less_fst_lft,
DecartTree::decMerge(grtr_fst_lft, grtr_fst_rght));
  odd_tree =
      DecartTree::decMerge(less_scnd_lft,
DecartTree::decMerge(less_scnd_rght, grtr_scnd_rght));
}

void clear(DecartTree::DecNode* node) {
  if (node == nullptr) {
    return;
  }
  clear(node->left_tree);
  clear(node->rigth_tree);
  delete node;
}

std::minstd_rand DecartTree::prior_gen;

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);

  long long test_number = 1;
  long long n_val = 0;
  long long r_val = 0;

  std::cin >> n_val >> r_val;
  while ((n_val != 0) && (r_val != 0)) {
    DecartTree* fst_tree = new DecartTree();
    DecartTree* scnd_tree = new DecartTree();

    for (long long i = 1; i <= n_val; i++) {
      long long val = 0;
      std::cin >> val;
      if (i % 2 == 0) {
        DecartTree::pushBack(fst_tree->root, val);
      } else {
        DecartTree::pushBack(scnd_tree->root, val);
```

```
      }
    }
    std::cout << "Suite " << test_number << ":\n";
    for (long long i = 0; i < r_val; i++) {
      long long type = 0;
      long long fst_val = 0;
      long long scnd_val = 0;
      std::cin >> type >> fst_val >> scnd_val;
      if ((fst_val <= 0) || (scnd_val > n_val)) {
        continue;
      }
      if (type == 1) {
        if (fst_val == scnd_val) {
          continue;
        }
        if ((fst_val + scnd_val) % 2 == 0) {
          continue;
        }
        std::pair<DecartTree::DecNode*, DecartTree::DecNode*> res_trees =
            replaceOp(fst_tree->root, scnd_tree->root, fst_val, scnd_val);
        fst_tree->root = res_trees.first;
        scnd_tree->root = res_trees.second;
      } else if (type == 2) {
        long long result = 0;
        sumCalculateOp(fst_tree->root, scnd_tree->root, fst_val, scnd_val,
result);
        std::cout << result << "\n";
      }
    }
    std::cout << "\n";
    std::cin >> n_val >> r_val;
    test_number++;
    clear(fst_tree->root);
    clear(scnd_tree->root);
    delete fst_tree;
    delete scnd_tree;
  }
}
```