**Задача A**

**1) Алгоритм**

Для решения данной задачи был выбран алгоритм корневой декомпозиции. Исходя из числа максимального количества чисел в массиве, размер бакета был проинициализирован числом 256 (степень двойки для оптимизации деления).

Была реализована структура SqrtDecomp в которой хранится общее кол-во нулей zero_count, а также два вектора: в src_arr хранятся исходные числа, в value_count хранятся числа кол-ва нулей для каждого блока.

Конструктор структуры инициализирует исходный вектор, а также за O(n) считает кол-во нулей для каждого блока (counterGroup()). Для удобного нахождения границ начала и конца бакета были реализованы методы grbegin() и grend().

Для обновления элемента по индексу реализован метод update(), который изменяет элемент по заданному индексу за O(1), а также сразу за O(1) обновляет общее кол-во нулей zero_count и кол-во нулей для блока.

Поиск k-го элемента на заданном отрезке осуществляется с помощью метода get_index(). Сперва определяются бакеты в которых лежат левая и правая граница. Далее в зависимости от нахождения данных блоков считаем кол-во нулей находящихся в их пределах. Если находятся в одном блоке, то за O(N) находим k-ое. Если же между границами существуют целые блоки, берем информацию о кол-ве нулей из value_count[блока]. Также реализован случай, когда сумма нулей с блоком больше k – перебором O(N) от начала данного блока считаем кол-во нулей.

**2) Сложность**

Этап предпосчета структуры – O(N). Обновление элемента по заданному индексу осуществляется за O(1) .Поскольку размер блока был выбран ≈ $\sqrt{N}$, то для вычисления k-го значения на отрезке [l…r] понадобится O($\sqrt{N}$) операций.

**3) Распечатка кода**

```cpp
#include <cmath>
#include <cstddef>
#include <ios>
#include <iostream>
#include <vector>

const int gr_size = 256;

struct SqrtDecomp {
  std::vector<int> value_count;
```

```
  std::vector<int>& src_arr;
  int zero_count = 0;

  SqrtDecomp(std::vector<int>& init_arr) : src_arr(init_arr) {
    value_count.assign((init_arr.size() + gr_size - 1) / gr_size, 0);
    for (int i = 0; i < static_cast<int>(value_count.size()); ++i) {
      counterGroup(i);
    }
  }

  int grbegin(int group) {
    return group * gr_size;
    ;
  }

  int grend(int group) {
    return std::min(grbegin(group) + gr_size,
static_cast<int>(src_arr.size()));
  }

  void counterGroup(int group) {
    value_count[group] = 0;
    for (int i = grbegin(group); i < grend(group); ++i) {
      if (src_arr[i] == 0) {
        value_count[group] += 1;
      }
    }
    zero_count += value_count[group];
  }

  void update(int index, int new_value) {
    if (src_arr[index] == 0) {
      if (new_value != 0) {
        value_count[index / gr_size] -= 1;
        zero_count -= 1;
        src_arr[index] = new_value;
      }
    } else {
```

```
    if (new_value == 0) {
      value_count[index / gr_size] += 1;
      zero_count += 1;
      src_arr[index] = new_value;
    }
  }
}

int get_index(int left_bound, int rght_bound, int k) {
  if (k > zero_count) {
    return -1;
  }

  int group_l = left_bound / gr_size;
  int group_r = rght_bound / gr_size;
  int res_index = -1;
  int counter = 0;
  if (group_l == group_r) {
    for (int i = left_bound; i <= rght_bound; ++i) {
      if (src_arr[i] == 0) {
        ++counter;
        if (counter == k) {
          res_index = i;
          return res_index;
        }
      }
    }

  } else {
    for (int i = left_bound; i < grend(group_l); ++i) {
      if (src_arr[i] == 0) {
        ++counter;
        if (counter == k) {
          res_index = i;
          return res_index;
        }
      }
    }
```

```
      for (int i = group_l + 1; i < group_r; ++i) {
        if (counter + value_count[i] < k) {
          counter += value_count[i];
        } else {
          for (int j = grbegin(i); j < grend(i); ++j) {
            if (src_arr[j] == 0) {
              ++counter;
              if (counter == k) {
                res_index = j;
                return res_index;
              }
            }
          }
        }
      }
      for (int i = grbegin(group_r); i <= rght_bound; ++i) {
        if (src_arr[i] == 0) {
          ++counter;
          if (counter == k) {
            res_index = i;
            return res_index;
          }
        }
      }
    }
    return res_index;
  }
};

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  size_t quant_numb = 0;
  size_t quant_req = 0;
  char operation = '0';
  int fst_value = 0;
  int scnd_value = 0;
  int thd_value = 0;
```

```
  std::cin >> quant_numb;
  std::vector<int> source_arr(quant_numb);
  for (size_t i = 0; i < quant_numb; ++i) {
    std::cin >> source_arr[i];
  }
  std::cin >> quant_req;
  SqrtDecomp dt(source_arr);
  for (size_t i = 0; i < quant_req; ++i) {
    std::cin >> operation;
    if (operation == 's') {
      std::cin >> fst_value >> scnd_value >> thd_value;
      --fst_value;
      --scnd_value;
      int answer = dt.get_index(fst_value, scnd_value, thd_value);
      if (answer != -1) {
        answer += 1;
      }
      std::cout << answer << " ";

    } else if (operation == 'u') {
      std::cin >> fst_value >> scnd_value;
      --fst_value;
      dt.update(fst_value, scnd_value);
    }
  }

  return 0;
}
```