НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерных технологий

Продвинутые алгоритмы и структуры данных

Отчет о задаче № D

Выполнила
студентка

Ершова А. И.

Группа № P4115

Преподаватель: Косяков Михаил Сергеевич

г. Санкт-Петербург

2024

Условие задачи

H. Темпоральный катаклизм

Ограничение времени	1 секунда
Ограничение памяти	50.0 Мб
Ввод	стандартный ввод или input.txt
Вывод	стандартный вывод или output.txt

Доктор Кто заметил, что каждое решение, принятое человечеством, создает новую временную ветвь, разделяя поток времени на бесчисленные альтернативные реальности. Эти ветви переплетаются в сложное древо, где каждая развилка - это новый вариант будущего.

Однако недавний темпоральный катаклизм нарушил естественный ход времени, и теперь все эти ветви начинают нестабильно взаимодействовать друг с другом, угрожая привести к разрушению временного континуума. Чтобы предотвратить распад реальности, Доктору Кто необходимо перемещаться между ключевыми временными точками и восстанавливать порядок.

Но есть одна загвоздка: заряд ТАРДИС - машины времени Доктора - ограничен, и каждый прыжок по временным линиям требует определённого количества энергии. Доктор должен оценить, хватит ли заряда для перемещения от одной точки к другой.

Помогите Доктору Кто справиться с последствиями темпорального катаклизма, путешествуя по ветвям времени и восстанавливая баланс в альтернативных реальностях!

Формат ввода

На вход подаётся следующее:

Первой строкой задаётся число N - количество временных точек (вершин) в древе, такое что $1 \leq N \leq 10^5$

Следующие N−1 строк описывают связи (рёбра) между временными точками. Каждая строка содержит два числа u и v, которые обозначают соединение между временными точками u и v. Эти идентификаторы - натуральные числа от 0 до N-1, уникальные для каждой временной точки.

После этого задаётся число Q - количество запросов на перемещение, такое что $Q \leq 10^6$ Следующие Q строк содержат описание запросов. Каждый запрос представлен тремя числами: два идентификатора временных точек u и v, между которыми Доктор хочет переместиться, и величина заряда T, доступного для этого перемещения, т.е. максимально возможное расстояние на которое можно переместиться, измеряемое в числе пройденных ребер, данная величина не превосходит N.

Формат ввода

На вход программе подается один или более тестовых сценариев. Каждый ценарий представляет из себя запросы к системе. Теперь рассмотрим каждый тестовый набор. Сначала строчка с 2 числами: длина последовательности чисел N и количество запросов R ($1 \leq N, R \leq 100000$). Во второй строке содержится N челых чисел $-10^6 \leq a_i \leq 10^6$ (это наш массив входной массив). Следом идут R строк. Запросы первого типа имеют формат 1 $x_i y_i$, а второго типа имеют формат 2 $a_i b_i$. Гарантируется, что $\sum_{i=1}^{N} N_i + \sum_{i=1}^{R} R_i \leq 200000$ и $x_i < y_i$, а $a_i \leq b_i$. В конце файла будет строка, содержащая два нуля.

Формат вывода

Для каждого набора тестов сперва выведите строчку "Suite N:", где вместо N должен стоять номер набора тестов. Далее выводите ответы на запросы второго типа, каждый ответ в новой строке. Ответы на наборы тестов необходимо разделить пустой строкой.

Пример

Ввод	Вывод
5 5 1 2 3 4 5 1 2 5 2 2 4 1 1 4 2 1 3 2 4 4 0 0	Suite 1: 10 9 2

Решение:

```
#include <fstream>
#include <iostream>
#include <random>
#include <tuple>
using namespace std;

struct TreapNode {
 int key;
```

```
 int value;
 int priority;
 int size;
 int64_t sum;
 TreapNode* left;
 TreapNode* right;

 TreapNode(int k, int v)
     : key(k), value(v), priority(rand()), size(1), sum(v),
left(nullptr), right(nullptr) {
 }
};

TreapNode* odd_tree;
TreapNode* even_tree;

int get_size(TreapNode* node) {
 return node ? node->size : 0;
}

int64_t get_sum(TreapNode* node) {
 return node ? node->sum : 0;
}

void update(TreapNode* node) {
 if (node) {
   node->size = get_size(node->left) + get_size(node->right) + 1;
   node->sum = get_sum(node->left) + get_sum(node->right) +
node->value;
 }
}
void destroy_tree(TreapNode* node) {
 if (!node)
   return;
 destroy_tree(node->left);
 destroy_tree(node->right);
 delete node;
}

pair<TreapNode*, TreapNode*> split(TreapNode* node, int key) {
 if (!node)
   return {nullptr, nullptr};
 if (key <= get_size(node->left)) {
   auto [left, right] = split(node->left, key);
   node->left = right;
   update(node);
   return {left, node};
 } else {
   auto [left, right] = split(node->right, key - get_size(node->left)
- 1);
```

```
    node->right = left;
    update(node);
    return {node, right};
  }
}

TreapNode* merge(TreapNode* left, TreapNode* right) {
  if (!left || !right)
    return left ? left : right;
  if (left->priority > right->priority) {
    left->right = merge(left->right, right);
    update(left);
    return left;
  } else {
    right->left = merge(left, right->left);
    update(right);
    return right;
  }
}

TreapNode* insert_odd_tree(int key, int value) {
  TreapNode* new_node = new TreapNode(key, value);
  auto [left, right] = split(odd_tree, key);
  return merge(merge(left, new_node), right);
}
TreapNode* insert_even_tree(int key, int value) {
  TreapNode* new_node = new TreapNode(key, value);
  auto [left, right] = split(even_tree, key);
  return merge(merge(left, new_node), right);
}

int64_t range_sum_odd_tree(int l, int r) {
  auto [left, middle] = split(odd_tree, l);
  auto [middle_part, right] = split(middle, r - l + 1);
  int64_t result = get_sum(middle_part);
  odd_tree = merge(merge(left, middle_part), right);
  return result;
}

int64_t range_sum_even_tree(int l, int r) {
  auto [left, middle] = split(even_tree, l);
  auto [middle_part, right] = split(middle, r - l + 1);
  int64_t result = get_sum(middle_part);
  even_tree = merge(merge(left, middle_part), right);
  return result;
}

void swap_segments(int l, int r) {
  int odd_l = (l + 1) / 2;
  int odd_r = r / 2;
```

```
 int even_l = l / 2;
 int even_r = (r - 1) / 2;

 auto [odd_left, odd_middle] = split(odd_tree, odd_l);
 auto [odd_segment, odd_right] = split(odd_middle, odd_r - odd_l + 1);

 auto [even_left, even_middle] = split(even_tree, even_l);
 auto [even_segment, even_right] = split(even_middle, even_r - even_l
+ 1);

 odd_tree = merge(merge(odd_left, even_segment), odd_right);
 even_tree = merge(merge(even_left, odd_segment), even_right);
}

int main() {
 ifstream fin("input.txt");
 int suite_num = 0;

 while (true) {
   int n, r;
   if (!(fin >> n >> r))
     break;
   if (n == 0 && r == 0)
     break;
   odd_tree = nullptr;
   even_tree = nullptr;
   for (int i = 0; i < n; ++i) {
     int val;
     fin >> val;
     if (i % 2 == 0) {
       odd_tree = insert_odd_tree(i / 2, val);
     } else {
       even_tree = insert_even_tree(i / 2, val);
     }
   }

   cout << "Suite " << ++suite_num << ":\n";
   for (int i = 0; i < r; ++i) {
     int q_type, l, rr;
     fin >> q_type >> l >> rr;
     if (q_type == 1) {
       l--;
       rr--;
       swap_segments(l, rr);
     } else if (q_type == 2) {
       l--;
       rr--;
       if (l == rr && l % 2 == 0) {
         int64_t odd_sum = range_sum_odd_tree((l + 1) / 2, rr / 2);
         cout << odd_sum << "\n";
```

```
      } else if (l == rr && l % 2 == 1) {
        int64_t even_sum = range_sum_even_tree(l / 2, (rr - 1) / 2);
        cout << even_sum << "\n";
      } else {
        int64_t odd_sum = range_sum_odd_tree((l + 1) / 2, rr / 2);
        int64_t even_sum = range_sum_even_tree(l / 2, (rr - 1) / 2);
        int64_t sec_req_result = odd_sum + even_sum;
        cout << sec_req_result << '\n';
      }
    }
  }
  cout << "\n";
  destroy_tree(odd_tree);
  destroy_tree(even_tree);
  odd_tree = nullptr;
  even_tree = nullptr;
 }
 return 0;
}
```

Сложность: O(logn)

Объяснение решения:

Для решения данной задачи было использовано 2 декартовых дерева: для чисел с нечетными индексами (even_tree) и для чисел с четными индексами (odd_tree). Каждый узел содержит ключ, значение, приоритет, размер поддерева и сумму всех значений в этой поддереве.

Функция split() разбивает дерево на 2 части по заданному индексу, возвращая указатели на левые и правые части. merge() объединяет 2 кучи. Вставка нового компонента в дерево происходит через комбинацию функций split и merge.

Для выполнения первого типа запросов есть специальная функция swap_segments, она использует функции split и merge для извлечения элементов из двух деревьев, а затем эти сегменты меняются местами. После этого обновленные сегменты снова соединяются с основной частью деревьев.

Для запросов второго типа сначала дерево разбивается таким образом, чтобы выделить заданный сегмент, из него извлекается предвычисленная сумма, затем исходная структура дерева восстанавливается.

После обработки каждого теста память, выделенная под декартовы деревья, освобождается.

Вывод:

Было изучено и рассмотрено использование декартового дерева.