

Модули на АСМ

Владимир Милосердов, Владимир Шабанов

7 ноября 2014 г.

Оглавление

1	Геометрия	5
2	Графы	7
2.1	Основы	7
2.2	Идентификация графа	8
2.2.1	Поиск цикла и проверка на ацикличность	8
2.2.2	Гамильтонов граф	9
2.2.3	Двудольный граф $O(n)$	9
2.2.4	Компоненты связности $O(n)$	9
2.2.5	Компоненты сильной связности $O(n + m)$	10
2.2.6	Минимальное остовное дерево $O(M * \log(M))$	11
2.2.7	Паросочетания	11
2.2.8	Точки сочленения	13
2.2.9	Эйлеров путь/цикл	13
3	Строки	15
4	ДП	17
5	Структуры данных	19
5.1	Дерево отрезков	19
5.1.1	Операции на отрезке в дереве отрезков	22
6	Декартово дерево	23
6.1	Реализация дерамиды	24
7	Алгебра	29
7.1	Алгоритм Евклида	29
7.1.1	НОД, НОК (gcd, lcm)	29
7.1.2	Расширенный алгоритм Евклида	29
7.1.3	Восстановление в кольце по модулю	29
7.2	Решето Эратосфена	30
7.2.1	Классический вариант	30
7.2.2	Линейное решето	30
8	Разное	31
8.1	Разбор выражений	31

Глава 1

Геометрия

Глава 2

Графы

2.1 Основы

Граф - множество вершин и ребер(заданных явно или не явно). Понятия используемые в дальнейшем:

- Ациклический граф

Граф без циклов

- Влентность/Степень вершины

Количество ребер входящих/выходящих в вершину

- Взвешенный граф

Граф в котором у каждого ребра есть стоимость

- Висячая вершина

Вершина со степенью один

- Гамильтонов путь

Путь в графе содержащий каждую вершину ровно один раз

- Гамильтонов цикл

Цикл содержащий каждую вершину ровно один раз

- Двудольный граф

Граф в котором можно выделить два множества такие, что между любыми двумя вершинами одного множества нет ребер.

- Компонента связности

Множество вершин и ребер графа такое, что из каждой его вершины достижима любая другая вершина этого множества

- Компонента сильной связности

Множество вершин и ребер ориентированного графа такое, что из каждой его вершины достижима любая другая вершина этого множества

- Кратные ребра

Ребра связывающие одну и ту же пару вершин

- Минимальный каркас

Множество ребер соединяющих все вершины графа без циклов и имеющее минимальный суммарный вес

- Паросочетания

Множество попарно не смежных ребер

- Точка сочленения

Вершина после удаления которой количество компонент связности возрастает

- Эйлеров путь

Путь в графе содержащий каждое ребро ровно один раз

- Эйлеров цикл

Цикл содержащий каждое ребро ровно один раз

2.2 Идентификация графа

2.2.1 Поиск цикла и проверка на ацикличность

Воспользуемся поиском в глубину. Окрасим все вершины в белый. Запускаясь от вершины перекрашиваем ее в серый, а выходя из нее красим в черный. Если *dfs* попытается пойти в серую вершину, значит мы нашли цикл, который сможем вывести с помощью массива предков, иначе граф циклов не имеет. Далее реализация на списках смежности.

```

1 bool dfs (int v) {
2     cl[v] = 1; //colors
3     for (int i = 0; i < g[v].size(); i++) {
4         int to = g[v][i];
5         if (cl[to] == 0) {
6             p[to] = v;
7             if (dfs(to)) //if son have cycle then parent have cycle
8                 return true;
9         }
10    }
```



```

10         else if (cl[to] == 1) {
11             cycle_end = v;
12             cycle_st = to;
13             return true;
14         }
15     }
16     cl[v] = 2;
17     return false;
18 }

```

2.2.2 Гамильтонов граф

Пусть n количество вершин графа, а δ минимальная степень вершины в графе, тогда граф имеет гамильтонов цикл если $n \geq 3$ и $\delta \geq \frac{n}{2}$

2.2.3 Двудольный граф $O(n)$

Так как граф является двудольным тогда и только тогда, когда все циклы четны, определить двудольность можно за один проход в глубину. На каждом шаге обхода в глубину помечаем вершину. Допустим мы пошли в первую вершину — помечаем её как 1. Затем просматриваем все смежные вершины и если не помечена вершина, то на ней ставим пометку 2 и рекурсивно переходим в нее. Если же она помечена и на ней стоит та же пометка, что и у той, из которой шли (в нашем случае 1), значит граф не двудольный.

```

1 bool dfs (int v, int c) {
2     cl[v] = c; //colors
3     for (int i = 0; i < g[v].size(); i++) {
4         int to = g[v][i];
5         if (cl[to] == 0) {
6             return dfs(to, max(1, (c + 1) % 2));
7         }
8         else
9             return cl[to] != c;
10    }
11 }

```

2.2.4 Компоненты связности $O(n)$

Поиск компонент связности можно осуществить многими методами, в том числе и поиском в глубину. Окрасим все вершины в индивидуальные цвета. Запуская $[dfs]$ от каждой вершины будем перекрашивать в ее цвет все вершины в этой компоненте. В конце алгоритма у нас останется столько цветов, сколько компонент связности в графе, а цвет каждой вершины будет идентифицировать ее компоненту.

```

1 void dfs (int v, int c) {
2     if (!used[v])
3         return;
4     cl[v] = c; //colors
5     for (int i = 0; i < g[v].size(); i++) {
6         int to = g[v][i];
7         if (cl[to] != cl[v]) {
8             dfs(to, c);
9             return;

```

```

10         }
11     }
12     int main(){
13         ...
14         for (int i = 0; i < N; i++){
15             dfs(i, i);
16         }
17     }

```

2.2.5 Компоненты сильной связности $O(n + m)$

Решим эту задачу за несколько обходов в глубину. Сначала топологически (по времени выхода) отсортируем граф, затем обойдем транспонированный(инвертированный) граф в этом порядке. Найденные компоненты связности этого графа и будут компонентами сильной связности исходного графа.

```

1  vector < vector<int> > g, gr; // граф и транспортированный граф
2  vector<char> used;
3  vector<int> order, component; // порядок топ сорта и компонента сильной связности
4
5  void dfs1 (int v) {
6      used[v] = true;
7      for (int i = 0; i < g[v].size(); i++)
8          if (!used[g[v][i]])
9              dfs1(g[v][i]);
10     order.push_back(v);
11 }
12
13 void dfs2 (int v) {
14     used[v] = true;
15     component.push_back(v);
16     for (int i = 0; i < gr[v].size(); i++)
17         if (!used[gr[v][i]])
18             dfs2(gr[v][i]);
19 }
20
21 int main() {
22     int n, m;
23     cin >> n >> m;
24     for (int i = 0; i < m; i++){
25         int a, b;
26         cin >> a >> b;
27         g[a].push_back(b);
28         gr[b].push_back(a);
29     }
30
31     used.assign(n, false);
32     for (int i = 0; i < n; i++)
33         if (!used[i])
34             dfs1(i);
35     used.assign(n, false);
36     for (int i = 0; i < n; i++) {
37         int v = order[n - 1 - i];
38         if (!used[v]) {
39             dfs2 (v);
40             //... вывод component ...

```

```

41         component.clear();
42     }
43 }
44 }

```

2.2.6 Минимальное остовное дерево $O(M * \log(M))$

Алгоритм Кр(a)yскала. Составим список ребер, отсортируем их по возрастанию весов. Распределим все вершины в соответствующие им множества, для работы с ними воспользуемся СНМ. Будем добавлять ребра по порядку, но только, если они соединяют вершины из разных множеств. После этого объединим их множества.

```

1  vector<int> p (n); //Множества
2
3  int dsu_get(int v) {
4      return(v == p[v]) ? v : (p[v] = dsu_get(p[v]));
5  }
6
7  void dsu_unite(int a, int b) {
8      a = dsu_get(a);
9      b = dsu_get(b);
10     if (rand() & 1)
11         swap(a, b);
12     if (a != b)
13         p[a] = b;
14 }
15
16 int main(){
17     .....
18     sort (g.begin(), g.end()); // список ребер <pair<вес, pair<начало, конец> > >
19     p.resize (n);
20     for (int i = 0; i < n; i++)
21         p[i] = i;
22     for (int i = 0; i < m; i++) {
23         int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
24         if (dsu_get(a) != dsu_get(b)) {
25             cost += l; // вес каркаса
26             res.push_back(g[i].second); // каркас
27             dsu_unite(a, b);
28         }
29     }

```

2.2.7 Паросочетания

Наибольшее паросочетание $O(N * M)$

Введем новое понятие понятия:

- Увеличивающая цепь

Простой путь, ребра которого поочередно входят/не входят в паросочетание, а первая и последняя вершина не принадлежат паросочетанию

Теорема Паросочетание является максимальным тогда и только тогда, когда не существует увеличивающих его цепей

В основе алгоритма лежит эта теорема, позволяющая "улучшать" любое паросочетание. Будем считать, что граф уже разбит на две доли. Для каждой вершины из первой доли делаем следующее: если у текущей вершины уже есть ребро включенное в паросочетание, то переходим к следующей вершине, иначе запускаем поиск увеличивающей цепи из этой вершины. Поиск увеличивающей цепи.

Для каждого ребра из этой вершины делаем следующее: пусть смежная с текущим ребром вершина не принадлежит паросочетанию, значит увеличивающая цепь найдена, добавим это ребро к паросочетанию, иначе ищем увеличивающую цепь из новой вершины. Максимальное паросочетание будет найдено после проверки всех вершин из первой доли.

Т.к. на каждом шаге алгоритм улучшает текущее паросочетание, то асимптотику можно значительно улучшить, если за начальное паросочетание брать не пустое, а любое другое, полученное любым эвристическим методом.

Следует заметить, что номера вершин разных долей в данной реализации независимы.

```

1  int n, k; // число вершин в первой и второй доли соответственно
2  vector < vector<int> > g; // список ребер из вершин первой доли
3  vector<int> mt; // mt[i] номер вершины первой доли связанной ребром с вершиной второй
   доли под номером i
4  vector<char> used;
5
6  bool try_kuhn (int v) {
7      if (used[v]) return false;
8      used[v] = true;
9      for (int i = 0; i < g[v].size(); i++) {
10         int to = g[v][i];
11         if (mt[to] == -1 || try_kuhn (mt[to])) {
12             mt[to] = v;
13             return true;
14         }
15     }
16     return false;
17 }
18 int main() {
19     //... чтение графа ...
20
21     mt.assign (k, -1); // текущее паросочетание пусто
22     vector<char> used1 (n);
23     for (int i = 0; i < n; i++)
24         for (int j = 0; j < g[i].size(); j++)
25             if (mt[g[i][j]] == -1) {
26                 mt[g[i][j]] = i; // строим какое-нибудь паросочетание
27                 used1[i] = true;
28                 break;
29             }
30     for (int i = 0; i < n; i++) {
31         if (used1[i]) continue;
32         used.assign (n, false);
33         try_kuhn (i);
34     }
35
36     for (int i = 0; i < k; i++)
37         if (mt[i] != -1)
38             printf ("%d_ %d\n", mt[i]+1, i+1);
39 }

```

2.2.8 Точки сочленения

Алгоритм для связного графа. Запустим поиск в глубину от произвольной вершины. Основная идея алгоритма: вершина является точкой сочленения, тогда и только тогда, когда невозможно попасть в предков этой вершины из её сыновей. Исключение составляет лишь вершина старта *dfs*, ведь у нее нет предков. Она будет точкой сочленения, если у нее больше одного сына, вывешенного поиском в глубину

```

1  vector<int> g[MAXN];
2  bool used[MAXN];
3  int timer, tin[MAXN], fup[MAXN];
4
5  void dfs (int v, int p = -1) {
6      used[v] = true;
7      tin[v] = fup[v] = timer++;
8      int children = 0;
9      for (int i = 0; i < g[v].size(); i++) {
10         int to = g[v][i];
11         if (to == p) continue;
12         if (used[to])
13             fup[v] = min (fup[v], tin[to]);
14         else {
15             dfs (to, v);
16             fup[v] = min (fup[v], fup[to]);
17             if (fup[to] >= tin[v] && p != -1)
18                 IS_CUTPOINT(v);
19             children++;
20         }
21     }
22     if (p == -1 && children > 1)
23         IS_CUTPOINT(v);
24 }
25
26 int main() {
27     int n;
28
29     timer = 0;
30     for (int i = 0; i < n; i++)
31         used[i] = false;
32     dfs (0);
33 }
```

2.2.9 Эйлеров путь/цикл

Необходимым условием Эйлера цикла и пути является наличие не более одной компоненты связности с ненулевым числом ребер. Если выполняется это условие и степень каждой вершины четна, то в графе существует Эйлеров цикл, а если степень всех вершин, кроме двух четна, то существует Эйлеров путь. Алгоритм напоминает поиск в глубину. Главное отличие состоит в том, что пройденными помечаются не вершины, а ребра графа. Начиная со стартовой вершины *v* строим путь, добавляя на каждом шаге не пройденное еще ребро, смежное с текущей вершиной. Вершины пути накапливаются в стеке *S*. Когда наступает такой момент, что для текущей вершины *w* все инцидентные ей ребра уже пройдены, записываем вершины из *S* в ответ, пока не встретим вершину,

которой инцидентны не пройденные еще ребра. Далее продолжаем обход по не посещенным ребрам. Для поиска Эйлера пути необходимо начинать с вершины у которой степень нечетна, и добавить ребро между вершинами с нечетными степенями, а из найденного цикла удалить добавленное ребро. Псевдокод:

```
1 findPath(v):
2   S.clear()
3   S.add(v)
4   while not S.isEmpty():
5     w := S.top()
6     if E contains(w, u):
7       S.add(u)
8       remove(w, u)
9     else:
10      S.pop()
11      print w
```

Глава 3

Строки

Глава 4

ДП

Глава 5

Структуры данных

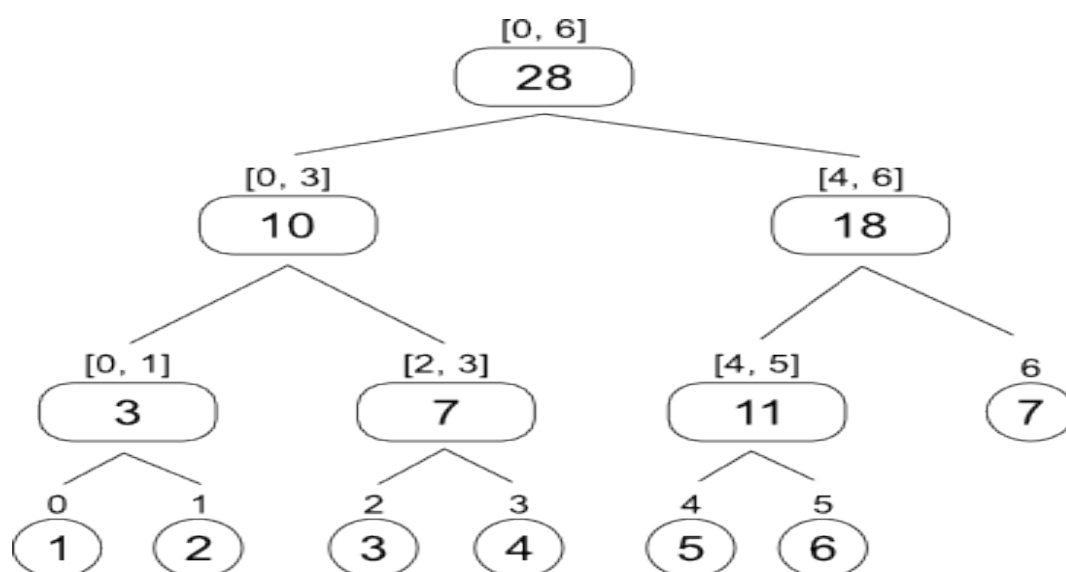
5.1 Дерево отрезков

Дерево отрезков (segment tree) - структура данных, позволяющая за $O(\log n)$ времени и $O(n)$ памяти выполнять следующие операции:

- Подсчёт функции для отрезка $l \dots r$
- Изменение одного элемента
- Изменение элементов на отрезке $l \dots r$

Структура дерева отрезков

Для данного отрезка $a = [0 \dots n - 1]$ начнём делить его пополам: мы получим 2 подотрезка: $a_1 = [0 \dots \lfloor \frac{|a|}{2} \rfloor]$ и $a_2 = [\lfloor \frac{|a|}{2} \rfloor + 1 \dots n - 1]$. Посчитаем сумму на них, разобьём каждый ещё на две части и проделаем всё тоже самое для полученных подотрезков. Будем делать так, пока не получим отрезки длины 1.



Реализация дерева отрезков

```

1  #include <iostream>
2  #include <cstdio>
3  #include <vector>
4
5  template <typename T>
6  class SegmentTree {
7      struct Node {
8          int maxv_i, minv_i;
9          T sum, maxv, minv;
10         int l, r, left, right;
11     };
12
13     std::vector<T> inp;
14     std::vector<Node> arr;
15     size_t sz;
16
17     void build(int l, int r, int ind) {
18         Node v;
19         v.l = l, v.r = r;
20         if (v.r - v.l <= 1) {
21             v.maxv = v.minv = inp[l];
22             v.maxv_i = v.minv_i = l;
23             arr[ind] = v;
24         }
25         else {
26             v.left = 2 * ind, v.right = 2 * ind + 1;
27             build(l, (l + r) / 2, 2 * ind);
28             build((l + r) / 2, r, 2 * ind + 1);
29
30             // max
31             if (arr[2 * ind].maxv > arr[2 * ind + 1].maxv) {
32                 v.maxv = arr[2 * ind].maxv;
33                 v.maxv_i = arr[2 * ind].maxv_i;
34             }
35             else {
36                 v.maxv = arr[2 * ind + 1].maxv;
37                 v.maxv_i = arr[2 * ind + 1].maxv_i;
38             }
39
40             // min
41             if (arr[2 * ind].minv < arr[2 * ind + 1].minv) {
42                 v.minv = arr[2 * ind].minv;
43                 v.minv_i = arr[2 * ind].minv_i;
44             }
45             else {
46                 v.minv = arr[2 * ind + 1].minv;
47                 v.minv_i = arr[2 * ind + 1].minv_i;
48             }
49             arr[ind] = v;
50         }
51     }
52
53     int getmax(int ind, int l, int r) {
54         if (arr[ind].r <= l || r <= arr[ind].l)
55             return -1;

```

```

56         if (l <= arr[ind].l && arr[ind].r <= r && arr[ind].l < arr[ind].r)
57             return arr[ind].maxv_i;
58         int ind1 = getmax(arr[ind].left, l, r);
59         int ind2 = getmax(arr[ind].right, l, r);
60         if (ind1 == -1)
61             return ind2;
62         if (ind2 == -1)
63             return ind1;
64         return (inp[ind1] > inp[ind2] ? ind1 : ind2);
65     }
66
67     int getmin(int ind, int l, int r) {
68         if (arr[ind].r <= l || r <= arr[ind].l)
69             return -1;
70         if (l <= arr[ind].l && arr[ind].r <= r && arr[ind].l < arr[ind].r)
71             return arr[ind].minv_i;
72         int ind1 = getmin(arr[ind].left, l, r);
73         int ind2 = getmin(arr[ind].right, l, r);
74         if (ind1 == -1)
75             return ind2;
76         if (ind2 == -1)
77             return ind1;
78         return (inp[ind1] < inp[ind2] ? ind1 : ind2);
79     }
80
81 public:
82     SegmentTree() {
83         sz = 0;
84         // ??
85     }
86     template<typename Iter> SegmentTree(Iter from, Iter to) {
87         sz = (to - from) + 1;
88         arr.resize(5 * sz + 1);
89         inp = std::vector<T>(from, to);
90         build(0, sz, 1);
91     }
92     T getMaxInd(int l = 1, int r = -1) {
93         if (r == -1)
94             r = sz;
95         return getmax(1, l - 1, r);
96     }
97     T getMinInd(int l = 1, int r = -1) {
98         if (r == -1)
99             r = sz;
100        return getmin(1, l - 1, r);
101    }
102    T getMaxValue(int l = 1, int r = -1) {
103        if (r == -1)
104            r = sz;
105        return inp[getmax(1, l - 1, r)];
106    }
107    T getMinValue(int l = 1, int r = -1) {
108        if (r == -1)
109            r = sz;
110        return inp[getmin(1, l - 1, r)];
111    }
112 };

```

```
113
114 int main() {
115     freopen("index-max.in", "r", stdin);
116     freopen("index-max.out", "w", stdout);
117     int n;
118     std::cin >> n;
119     std::vector<double> inp(n);
120     for (int i = 0; i < n; i++) {
121         std::cin >> inp[i];
122     }
123     SegmentTree<int> t(inp.begin(), inp.end());
124     int k;
125     std::cin >> k;
126     for (int i = 0; i < k; i++) {
127         int l, r;
128         std::cin >> l >> r;
129         std::cout << t.getMaxInd(l, r) + 1 << '\n';
130     }
131
132     return 0;
133 }
```

5.1.1 Операции на отрезке в дереве отрезков

```
/* TODO
* + Надо уметь МЕНЯТЬ элементы
*/
```

Глава 6

Декартово дерево

Декартово дерево (дуча, декамида, treap, etc..) - структура данных, в вершине которого содержится ключ x и приоритет y которая обладает свойством двоичного дерева поиска по ключам и свойством кучи по приоритетам. При случайных значениях приоритетов в вершинах высота дерева составит $O(\log n)$.

Рассмотрим необходимые операции:

Объединение деревьев

Пусть необходимо получить дерево T , объединив деревья T_1 и T_2 . Корнем результирующего дерева станет корень одного из данных деревьев с наибольшим приоритетом. Пусть y корня T_1 больше значения приоритета корня T_2 . Тогда левое поддерево T_1 будет левым поддеревом T , правое дерево в таком случае будет результатом объединения правого поддерева T_1 с T_2 (т.е. описанная функция будет рекурсивной). Если на текущем шаге левое поддерево пусто, то вернем правое, аналогично и с правым. Если приоритет корня T_2 больше, чем у T_1 , то делаем симметрично по аналогии. Для полученного T пересчитываем нужные для запросов значения. Очевидно, глубина рекурсивных вызовов пропорциональна высоте дерева. В декартовом дереве с большой (почти 100%) вероятностью $h \approx \log_2 N$, где h – высота. Тогда итоговая сложность для объединения составит $O(\log N)$.

Деление деревьев по ключу

Пусть дано дерево T и ключ x_0 . Необходимо получить такие дучи T_l, T_r , что в T_l все ключи будут меньше x_0 , а в T_r – больше. Пусть $k \geq x_0$, где k – ключ корня T , тогда левое поддерево T_l совпадет с левым поддеревом T . Правое дерево T_r будет результатом деления правого поддерева T по ключу x_0 . Также пересчитаем нужные для запросов значения для T_r . При $k > x_0$ действуем симметрично по аналогии. По аналогии с объединением получим сложность $O(\log N)$.

Добавление узла в дучу

Для добавления узла x в дерево T разделим это дерево по $x.key$ на деревья T_l и T_r , создадим дерево T_m из одного узла с ключом x и случайным приоритетом y . Объединим

деревья T_l, T_m , получив T' , исходное дерево T будет результатом объединения T' и T_r . Исходя из используемых операций сложность составит $O(\log N)$ с большой константой.

Удаление узла

Для удаления узла с ключом x из исходного дерева T необходимо разделить T по $x - 1$, таким образом узел с ключом x окажется в T_r – правом дереве после деления. Разделим T_r по x на T_r^l и T_r^r . Заметим, что T_r^l – дерево, состоящие из одного узла с ключом, равным x . Исходное дерево без узла будет объединением T_l и T_r^r . Сложность, аналогично с добавлением, составит $O(\log N)$.

6.1 Реализация дерамиды

```

1 // Vladimir Miloserdov (c) 2014
2 #include <iostream>
3 #include <cstdio>
4 #include <cstdlib>
5 #include <ctime>
6
7 using namespace std;
8
9 inline int gcd(int u, int v) {
10     while (v != 0) {
11         int r = u % v;
12         u = v;
13         v = r;
14     }
15     return u;
16 }
17
18 class Treap {
19     struct Node {
20         int x, y;
21         int gcd_val, cnt;
22         Node *l, *r;
23
24         Node(const int x, const int y) {
25             this->l = this->r = 0;
26             this->cnt = 0;
27             this->x = x;
28             this->y = y;
29             this->gcd_val = x;
30         }
31
32         void recalc() {
33             this->gcd_val = gcd(gcd(x, (this->l != NULL ? this->l->gcd_val
34                 : 0)),
35                 (this->r != NULL ? this->r->gcd_val : 0));
36         }
37     };
38     Node *root;
39     Node* merge(Node* l, Node* r) {

```



```

40         if (l == NULL) {
41             return r;
42         }
43         else if (r == NULL) {
44             return l;
45         }
46         else if (l->y > r->y) {
47             l->r = merge(l->r, r);
48             l->recalc();
49             return l;
50         }
51         else {
52             r->l = merge(l, r->l);
53             r->recalc();
54             return r;
55         }
56     }
57
58     pair<Node*, Node*> split(Node* T, int x0) {
59         if (T == NULL) {
60             return make_pair((Node*)0, (Node*)0);
61         }
62         pair<Node*, Node*> res;
63         if (T->x >= x0) {
64             res = split(T->l, x0);
65             T->l = res.second;
66             T->recalc();
67             res.second = T;
68             return res;
69         }
70         else {
71             res = split(T->r, x0);
72             T->r = res.first;
73             T->recalc();
74             res.first = T;
75             return res;
76         }
77     }
78
79     Node* insert(Node* T, Node* t0) {
80         if (!T) {
81             return t0;
82         }
83         else if (t0->y > T->y) {
84             pair<Node*, Node*> ret = split(T, t0->x);
85             t0->l = ret.first;
86             t0->r = ret.second;
87             t0->recalc();
88             return t0;
89         }
90         else if (t0->x < T->x) {
91             T->l = insert(T->l, t0);
92             T->recalc();
93             return T;
94         }
95         else {
96             T->r = insert(T->r, t0);

```

```

97         T->recalc();
98         return T;
99     }
100 }
101
102 Node* erase(Node* T, int x0) {
103     if (T->x == x0) {
104         if (T->cnt == 0) {
105             return merge(T->l, T->r);
106         }
107         else {
108             T->cnt--;
109             return T;
110         }
111     }
112     else if (x0 < T->x) {
113         T->l = erase(T->l, x0);
114         T->recalc();
115         return T;
116     }
117     else {
118         T->r = erase(T->r, x0);
119         T->recalc();
120         return T;
121     }
122 }
123
124 bool check(const Node* T, const int x0) {
125     if (!T)
126         return 0;
127     else if (T->x == x0)
128         return 1;
129     else if (x0 < T->x)
130         return check(T->l, x0);
131     else
132         return check(T->r, x0);
133 }
134
135 Node* find(Node* t, int x0) {
136     Node* cur = t;
137     while (cur && cur->x != x0) {
138         if (cur->x > x0) {
139             cur = cur->l;
140         }
141         else if (cur->x < x0){
142             cur = cur->r;
143         }
144     }
145     return cur;
146 }
147
148 public:
149     Treap() : root((Node*)0) { }
150
151     void insert(const int x0) {
152         Node *cur = find(root, x0);
153         if (!cur) {

```

```
154         Node* t0 = new Node(x0, rand());
155         root = insert(root, t0);
156     }
157     else {
158         cur->cnt++;
159     }
160 }
161
162 void erase(const int x0) {
163     root = erase(root, x0);
164 }
165
166 bool check(const int x0) {
167     return check(root, x0);
168 }
169
170 int get_gcd() {
171     return (root ? root->gcd_val : 1);
172 }
173 };
174
175 int main() {
176     srand(time(0));
177     Treap t;
178     int n;
179     scanf("%d", &n);
180     for (int i = 0; i < n; i++) {
181         char c;
182         int tmp;
183         scanf("%c%d", &c, &tmp);
184         if (c == '+')
185             t.insert(tmp);
186         else
187             t.erase(tmp);
188         printf("%d\n", t.get_gcd());
189     }
190
191     return 0;
192 }
```


Глава 7

Алгебра

7.1 Алгоритм Евклида

7.1.1 НОД, НОК (gcd, lcm)

$$\gcd(a, b) = \begin{cases} a & \text{если } a = 0 \\ b & \text{иначе} \end{cases}$$
$$\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

7.1.2 Расширенный алгоритм Евклида

```
1 int gcdex (int a, int b, int & x, int & y) {
2     if (a == 0) {
3         x = 0; y = 1;
4         return b;
5     }
6     int x1, y1;
7     int d = gcd (b%a, a, x1, y1);
8     x = y1 - (b / a) * x1;
9     y = x1;
10    return d;
11 }
```

Функция возвращает нод и коэффициенты x , y по ссылкам

7.1.3 Восстановление в кольце по модулю

```
1 int x, y;
2 int g = gcdex (a, m, x, y);
3 if (g != 1)
4     cout << "no solution";
5 else {
6     x = (x % m + m) % m;
7     cout << x;
8 }
```

7.2 Решето Эратосфена

7.2.1 Классический вариант

Дано число n . Требуется найти все простые в отрезке $[2; n]$. Решето Эратосфена решает эту задачу за $O(n \log \log n)$

Запишем ряд чисел $1 \dots n$, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем делящиеся на 3, кроме самого числа 3, затем на 5 и так далее ...

```

1  int n;
2  vector<char> prime (n+1, true);
3  prime[0] = prime[1] = false;
4  for (int i=2; i<=n; ++i)
5      if (prime[i])
6          if (i * 1ll * i <= n)
7              for (int j=i*i; j<=n; j+=i)
8                  prime[j] = false;
```

7.2.2 Линейное решето

Чуть быстрее по времени, чем классическое $O(N)$. Цена вопроса - оверхед по памяти. Пусть $lp[i]$ – минимальный простой делитель числа i , $2 \leq i \leq n$.

- $lp[i] = 0$ – число i – простое
- $lp[i] \neq 0$ – число i – составное

```

1  const int N = 10000000;
2  int lp[N+1];
3  vector<int> pr;
4
5  for (int i=2; i<=N; ++i) {
6      if (lp[i] == 0) {
7          lp[i] = i;
8          pr.push_back (i);
9      }
10     for (int j=0; j<(int)pr.size() && pr[j]<=lp[i] && i*pr[j]<=N; ++j)
11         lp[i * pr[j]] = pr[j];
12 }
```

Вектор $lp[]$ можно как-то использовать для факторизации чисел

Глава 8

Разное

8.1 Разбор выражений

Дано выражение. Начинаем парсить его функцией *E1*, которая обрабатывает самые низкоприоритетные операции (в нашем случае '+', '-').

```
1 def E1():
2     res = E2()
3     while True:
4         c = getc()
5         if c == '+':
6             res += E2()
7         elif c == '-':
8             res -= E2()
9         else:
10            putc(c)
11            return res
```

Сначала происходит обработка атома, стоящего слева от знака ('+', '-'), затем обработка каждого атома между знаками. Обработку этих атомов производит функция *E2*(). Это функция более низкого ранга, которая обрабатывает более приоритетные операции (в нашем случае '*' и '/').

```
1 def E2():
2     res = E3()
3     while True:
4         c = getc()
5         if c == '*':
6             res *= E3()
7         elif c == '/':
8             res /= E3()
9         else:
10            putc(c)
11            return res
```

Функция работает аналогично *E1*(). Таким образом мы можем поддерживать сколько угодно операций различных приоритетов, добавляя функции.

Перейдем к обработке скобок:

```
1 def E3():
2     if c == '(':
3         res = E1()
4         c = getc()
```

```
5         return res
6     else:
7         putc(c)
8         return E4()
```

Берём символ, если он скобка, тогда обрабатываем выражение внутри и считываем закрывающую скобку.

Теперь рассмотрим считывание числа. Ничего сложного:

```
1 def E4():
2     res = 0
3     while True:
4         c = getc()
5         if str.isdigit(c):
6             res = res * 10 + int(c)
7         else:
8             putc(c)
9             return res
```