Модули на ACM

Владимир Милосердов, Владимир Шабанов

29 ноября 2014 г.

Оглавление

Глава 1

Геометрия

1.1 Примитивы

Для начала введём геометрические структуры, такие как точка, отрезок, вектор, прямая, окружность...

Точка

```
1 struct point {
2     double x, y;
3     point (double _x, double _y) : x(_x), y(_y) {}
4 };
```

Отрезок

```
1 struct segment {
2     point a, b;
3     segment () {}
4     segment (point _a, point _b) : a(_a), b(_b) {}
5 };
```

Вектор

```
1 struct _vector {
2     point v;
3     _vector (double x, double y) : v(point(x, y)) {}
4     _vector (point a, point b) : v(point(a.x - b.x, a.y - b.y)) {}
5 };
```

Прямая

```
1 struct line {
2     double a, b, c;
3     line (point a, point b) : a(a.y - b.y), b(b.x - a.x), c(a.x * b.y - b.
          x * a.y) {}
4 };
```

Окружность

```
1 struct circle {
2     point c;
3     double r;
4     circle (point a, double _r) : c(a), r(_r) {}
5 };
```

1.2 Полезности и точность

Нам понадобятся полезные функции: поиск расстояния между точками, сравнение чисел с плавающей точкой и т.д.
Стандартные функции содержатся в <cmath>

```
1 #include <cmath>
2 double dist(point a, point b) {
3     return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
4 }
```

Для сравнения вещественных чисел следует использовать ***EPS***:

```
1 const double EPS = 1e-6;
2 bool is_equal(double a, double b) {
3     return (abs(a - b) <= EPS) ? 1 : 0;
4 }
```

Все стандартные функции C++ (как и многих других языков) считают углы в радианах. Перевод в градусы:

```
1 double to_degree(double an) {
2     return an * 180.0 / M_PI;
3 }
```

(*Что к чему это здесь?*)Проверка точки на принадлежность прямоугольнику:

```
1 bool in_square(point a, point b, point c) {
2     return (min(a.x, b.x) <= c.x && c.x <= max(a.x, b.x)
3         && min(a.y, b.y) <= c.y && c.y <= max(a.y, b.y));
4 }
```

1.3 Типовые операции в геометрии

Рассмотрим реализацию различных геометрических ситуаций.

1.3.1 Пересечение прямых

Примечание: *как угловой коэффициент линейной функции, это* tan *угла наклона прямой относительно положительной полуоси Ox, то у него существует значение, при котором он не существует:* 90 *градусов. Поэтому возникают деления на нуль и это объясняет большое количество случаев.*
Проверка на факт пересечения

```
1 bool cross_lines(line l1, line l2) {
2     if (abs(l1.b - 0.0) < EPS && abs(l2.b - 0.0) < EPS) //если параллельны oY
3         return 0;
4     else if (abs((l1.a / l1.b) - (l2.a / l2.b)) < EPS) //если угловые совпали
5         return 0;
6     else
7         return 1;
8 }
```

Теперь давайте найдём точку пересечения
Примечание: *Можно использовать, только проверив, что пересечение существует*

```
point cross_lines2(line l1, line l2) {
    double k1 = -1.0 * (l1.a / l1.b); //угловой коэфициент 1 прямой
    double b1 = -1.0 * (l1.c / l1.b); //свободный коэфициент 1 прямой
    double k2 = -1.0 * (l2.a / l2.b); //угловой коэфициент 2 прямой
    double b2 = -1.0 * (l2.c / l2.b); //свободный коэфициент 2 прямой
    point ans;
    if (is_equal(l1.b, 0.0) && !is_equal(l2.b, 0.0)) // если 1 прямая || Oy
        ans = point((-1.0 * (l1.c / l1.a)), (-1.0 * (l1.c / l1.a)) * k2 +
           b2);
    else if (!is_equal(l1.b, 0.0) && is_equal(l2.b, 0.0)) // если 2 прямая ||
       Oy
        ans = point((-1.0 * (l2.c / l2.a)), (-1.0 * (l2.c / l2.a)) * k1 +
           b1);
    else // если пересекаются
        ans = point((b2 - b1) / (k1 - k2), ((b2 - b1) / (k1 - k2)) * k1 +
           b1);
    return ans;
}
```

1.3.2 Пересечение отрезков

Примечание: ищет только точку пересечения. В случае наложения отрезков срабатывает условия, что они параллельны

```
point cross_segments(segment s1, segment s2) {
    //строим линии по двум точкам отрезков
    line l1 = line(s1.a, s1.b);
    line l2 = line(s2.a, s2.b);
    if (cross_lines(l1, l2)) {//убеждаемся, что прямые пересекаются
                point crs = cross_lines2(l1, l2);//точка пересечения прямых
    //проверяем, что точка принадлежит отрезкам
                if (in_square(point(s1.a.x, s1.a.y), point(s1.b.x, s1.b.y)
                    , crs) &&                          in_square(point(s2.a.
                    x, s2.a.y), point(s2.b.x, s2.b.y), crs))
                     return crs;
        }
}
```

1.4 Операции над векторами

Выше приведенная структура вектора в своём конструкторе формирует из двух точек радиус-вектор. Чаще всего удобнее работать именно с такими векторами.

1.4.1 Скалярное произведение

Скалярное произведение векторов – скалярная величина численно равная сумме произведений соответствующих координат
$(a, b) = a_x b_x + a_y b_y$
или прозведению модулей векторов на cos угла между ними
$(a, b) = |a| \cdot |b| \cdot \cos \phi$

```
1 double cross_product(_vector a, _vector b) {
2     return (a.v.x * b.v.x + a.v.y * b.v.y);
3 }
```

Векторным произведением двух трехмерных векторов $a(a_x, a_y, a_z)$ и $b(b_x, b_y, b_z)$ является вектор c с координатами $(a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$. Этот вектор перпендикулярен плоскости, в которой расположены вектора a и b. Но мы рассматриваем геометрию на плоскости, поэтому результатом век- торного произведения двух векторов, лежащих в плоскости XOY будет вектор, коллинеарный оси OZ, и поэтому его можно представить в виде одного числа — координаты c_z

1.4.2 Поворот вектора

Поворот вектора Для того чтобы повернуть вектор на заданный угол, нужно:

$x' = x\cos\alpha - y\sin\alpha$
$y' = x\sin\alpha + y\cos\alpha$
Из этого следует код.
Примечание: *следует помнить, что использование sin и cos ведёт к частичной потери точности*

```
1 _vector turn_vector(_vector a, double angle) {
2         double new_x = a.v.x * cos(angle) - a.v.y * sin(angle);
3         double new_y = a.v.x * sin(angle) - a.v.y * cos(angle);
4         return _vector(new_x, new_y);
5 }
```

Примечание: угол полярный, в радианах.

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 соеденяющих 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 Определение свойств 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.

```
bool dfs (int v) {
    cl[v] = 1; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to] == 0) {
            p[to] = v;
            if (dfs(to)) //if son have cycle then parent have cycle
                return true;
        }
        else if (cl[to] == 1) {
            cycle_end = v;
            cycle_st = to;
            return true;
        }
    }
    cl[v] = 2;
    return false;
}
```

2.2.2 Гамильтонов граф

Пусть n количесттво 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.

```
bool dfs (int v, int c) {
    cl[v] = c; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to] == 0) {
            return dfs(to, max(1, (c + 1) % 2));
        }
        else
            return cl[to] != c;
    }
}
```

2.2.4 Компоненты связности O(n)

Поиск компонент свзязности 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.

```
void dfs (int v, int c) {
    if (!used[v])
        return;
    cl[v] = c; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to]!= cl[v]) {
            dfs(to, c);
            return;
        }
}
int main(){
    ...
    for (int i = 0; i < N; i++){
        dfs(i, i);
    }
}
```

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 транспонированый(инвертированый) граф в этом порядке. Найденые компоненты связности этого графа и будут компонентами сильной связности исходного графа.

```
vector < vector<int> > g, gr; // граф и транспортированый граф
vector<char> used;
vector<int> order, component; // порядок топ сорта и компонента сильной связности

void dfs1 (int v) {
    used[v] = true;
    for (int i = 0; i < g[v].size(); i++)
        if (!used[g[v][i]])
            dfs1(g[v][i]);
    order.push_back(v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back(v);
    for (int i = 0; i < gr[v].size(); i++)
        if (!used[gr[v][i]])
            dfs2(gr[v][i]);
}

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++){
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        gr[b].push_back(a);
    }

    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($M \cdot log(M)$)

Алгоритм Кр(а|у)скала. 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 Точки сочленения

Алгоритм для связного графа. Запустим поиск в глубину от произвольной вершины. Основная идея алгоритма: вершина является точка сочлинения, 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, выевленного 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 }
```

2.2.8 Эйлеров путь/цикл

Необходимым условием Эйлерово циклаи 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 Эйлерова 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:

```
findPath(v):
   S.clear()
   S.add(v)
   while not S.isEmpty():
     w := S.top()
     if E contains(w, u):
       S.add(u)
       remove(w, u)
     else:
       S.pop()
       print w
```

2.3 Кратчайший путь

Все дальнейшие алгоритмы находят кратчайшие пути в одной компоненте связности при отсутствии циклов отрицательного веса в них, которые лекго обнаружить запустив алгоритм Форд-Беллмана один лишний раз, и если он обновит расстояние хоть до одной вершины, то в этой компоненте связности присутствует цикл отрицательного веса.

2.3.1 Дейкстра

Ищет путь от одной вершины до всех остальных. Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a. На каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

O(N^2)

```
def Dijkstra(N, S, matrix):
    valid = [True]*N
    weight = [1000000]*N
    weight[S] = 0
    for i in range(N):
        min_weight = 1000001
        ID_min_weight = -1
        for i in range(len(weight)):
            if valid[i] and weight[i] < min_weight:
                min_weight = weight[i]
                ID_min_weight = i
        for i in range(N):
            if weight[ID_min_weight] + matrix[ID_min_weight][i] < weight[i
               ]:
                weight[i] = weight[ID_min_weight] + matrix[ID_min_weight][
                   i]
        valid[ID_min_weight] = False
    return weight
```

O($Mlog(N)$)

```
const int INF = 1000000000;

int main() {
```

```
4       int n;
5       ...
6       vector < vector < pair<int,int> > > g (n);
7       ...
8       int s = ...; // стартовая вершина
9
10      vector<int> d (n, INF),  p (n);
11      d[s] = 0;
12      set < pair<int,int> > q;
13      q.insert (make_pair (d[s], s));
14      while (!q.empty()) {
15          int v = q.begin()->second;
16          q.erase (q.begin());
17
18          for (size_t j=0; j<g[v].size(); ++j) {
19              int to = g[v][j].first,
20                  len = g[v][j].second;
21              if (d[v] + len < d[to]) {
22                  q.erase (make_pair (d[to], to));
23                  d[to] = d[v] + len;
24                  p[to] = v;
25                  q.insert (make_pair (d[to], to));
26              }
27          }
28      }
29  }
```

2.3.2 Флой-Уоршелл O(N^3)

Пути между всеми парами вершин. Работает всегда.

```
1  for k = 1 to n
2    for i = 1 to n
3      for j = 1 to n
4        W[i][j] = min(W[i][j], W[i][k] + W[k][j])
```

2.3.3 Форд-Беллман O($M \cdot N$)

Находит путь от одной вершины до всех. Работает всегда.

```
1  struct edge {
2      int a, b, cost;
3  };
4
5  int n, m, v;
6  vector<edge> e;
7  const int INF = 1000000000;
8
9  void solve() {
10     vector<int> d (n, INF);
11     d[v] = 0;
12     for (int i=0; i<n-1; ++i)
13         for (int j=0; j<m; ++j)
14             if (d[e[j].a] < INF)
15                 d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
16     // вывод d, например, на экран
17 }
```

2.4 Паросочетания

2.4.1 Наибольшее паросочетание O($N \cdot M$)

Введем новое понятие понятия:

- Увеличивающая цепь

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

Теорема *Просочетание 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.

```
1  int n, k;
2  vector < vector<int> > g;
3  vector<int> mt;
4  vector<char> used;
5
6  bool try_kuhn (int v) {
7      if (used[v])  return false;
8      used[v] = true;
9      for (size_t 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     ...  ...
20
21     mt.assign (k, -1);
22     vector<char> used1 (n);
23     for (int i=0; i<n; ++i)
24         for (size_t j=0; j<g[i].size(); ++j)
25             if (mt[g[i][j]] == -1) {
```

```
26                  mt[g[i][j]] = i;
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  }
```

Глава 3

Строки

3.1 Алгоритмы, которых нет на e-maxx

3.1.1 Наибольшая общая подпоследовательность $|S_1| \cdot |S_2|$

Задача решается ДП. Состояние будет характеризовать два параметра: длина первой и второй строки. Изначально $dp(n_1, n_2) = 0$. Переход к следующему состоянию

$$dp(n_1, n_2) = \begin{cases} 0, n_1 = 0 \ or \ n_2 = 0 \\ dp(n_1 - 1, n_2 - 1) + 1, s[n_1] = s[n_2] \\ max(dp(n_1 - 1, n_2), dp(n_1, n_2 - 1)), s[n_1] \neq s[n_2] \end{cases}$$

Чтобы востановить ответ нужно запоминать из какого состояния мы попали в текущее, а в конце пройти из ячейки $dp(|S_1|, |S_2|)$ до $dp(0, 0)$ по этому маршруту.

3.1.2 Наибольшая общая подстрока

Первой строкой входного файла следует цифра обозначающая количество строк в которых следует искать подстроку. Строки для поиска следуют дальше.

```
1  import sys
2  list = sys.argv[1:]
3  fileName = list[0]
4  data = open(fileName).read().splitlines()
5  numOfStrings = int(data.pop(0))-1
6  data = filter(None, data)
7  data.sort(key=len)
8  leastStr = data.pop(0)
9  maxSharedStr = ''
10 while len(leastStr) > len(maxSharedStr):
11     robTestStr = leastStr
12     while len(robTestStr) > len(maxSharedStr):
13         numOfConcidence = 0
14         for compatStr in data:
15             if robTestStr in compatStr:
16                 numOfConcidence += 1
17             else:
18                 break
19         if numOfConcidence == numOfStrings and len(robTestStr) > len(
               maxSharedStr):
20             maxSharedStr = robTestStr
21         robTestStr = robTestStr[:-1]
```

```
22      leastStr = leastStr[1:]
23  print maxSharedStr
24  sys.exit(0)
```

Глава 4

Структуры данных

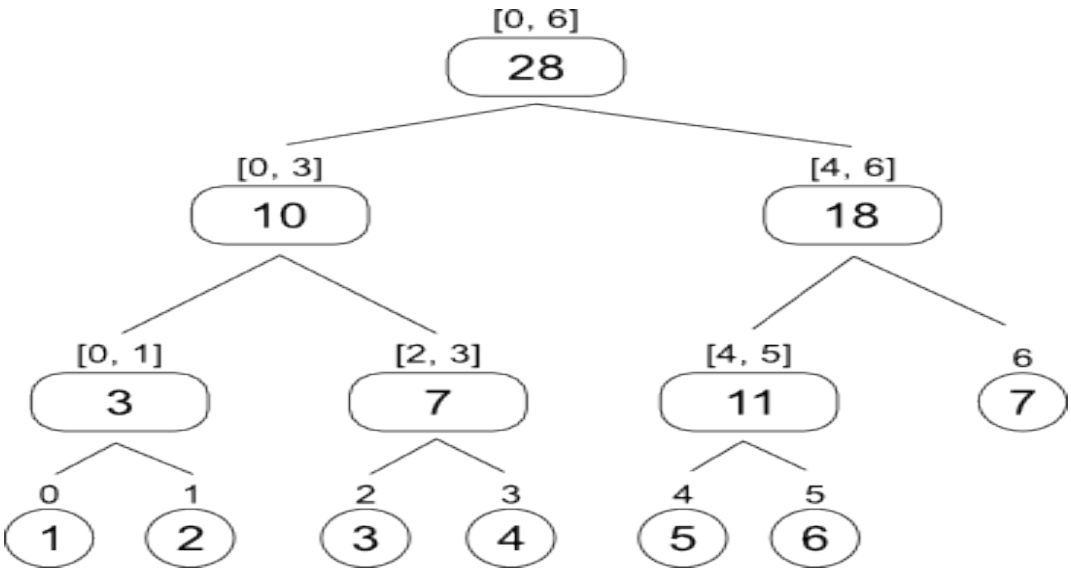
4.1 Дерево отрезков

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

- Подсчётфункции 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: $a_1 = [0 \cdots \frac{|a|}{2}]$ и $a_2 = [\frac{|a|}{2} + 1 \cdots n-1]$. 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

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

template <typename T>
class SegmentTree {
    struct Node {
        int maxv_i, minv_i;
        T sum, maxv, minv;
        int l, r, left, right;
    };

    std::vector<T> inp;
    std::vector<Node> arr;
    size_t sz;

    void build(int l, int r, int ind) {
        Node v;
        v.l = l, v.r = r;
        if (v.r - v.l <= 1) {
            v.maxv = v.minv = inp[l];
            v.maxv_i = v.minv_i = l;
            arr[ind] = v;
        }
        else {
            v.left = 2 * ind, v.right = 2 * ind + 1;
            build(l, (l + r) / 2, 2 * ind);
            build((l + r) / 2, r, 2 * ind + 1);

            // max
            if (arr[2 * ind].maxv > arr[2 * ind + 1].maxv) {
                v.maxv = arr[2 * ind].maxv;
                v.maxv_i = arr[2 * ind].maxv_i;
            }
            else {
                v.maxv = arr[2 * ind + 1].maxv;
                v.maxv_i = arr[2 * ind + 1].maxv_i;
            }

            // min
            if (arr[2 * ind].minv < arr[2 * ind + 1].minv) {
                v.minv = arr[2 * ind].minv;
                v.minv_i = arr[2 * ind].minv_i;
            }
            else {
                v.minv = arr[2 * ind + 1].minv;
                v.minv_i = arr[2 * ind + 1].minv_i;
            }
            arr[ind] = v;
        }
    }

    int getmax(int ind, int l, int r) {
        if (arr[ind].r <= l || r <= arr[ind].l)
            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 }
```

4.1.1 Операции на отрезке в дереве отрезков

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

4.2 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 пропорцианальна 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 $O(\log N)$.

4.3 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
                    : 0)),
34                  (this->r != NULL ? this->r->gcd_val : 0));
35          }
36      };
37      Node *root;
38
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 5

Сортировки

5.1 Сортировка пузырьком $O(N^2)$

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов.
Алгоритм **устойчив**

```
1  #include <algorithm>
2  template<typename Iter>
3  void bubble_sort(Iter first, Iter last) {
4      while(first < --last) {
5          for(Iter it = first; it < last; it++) {
6              if (*(it + 1) < *it) {
7                  std::iter_swap(it, it + 1);
8              }
9          }
10     }
11 }
```

5.2 Сортировка выбором $O(N^2)$

Алгоритм может быть **как устойчивым, так и неустойчивым** Ниже представлены листинги **устойчивого** алгоритма:

1. Берём минимальный элемент в массиве
2. Вставляем значение этого элемента на первую неотсортированную позицию, при этом не меняя порядок остальных элементов
3. Сортируем "хвост"массива, убрав уже отсортированный промежуток

```
1  template <typename T, typename C = less<typename T::value_type> >
2  void select_sort(T first, T last, C c = C()) {
3      if (first != last) {
4          while (first != last - 1) {
5              T mn = first;
6              for (T i = first + 1; i != last; i++) {
7                  if (c(*i, *mn)) {
```

```
8                       typename T::value_type tmp = *mn;
9                       *mn = *i;
10                      *i = tmp;
11                  }
12              }
13              first++;
14          }
15      }
16  }
```

5.3 Сортировка вставками $O(N^2)$

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке, до тех пор, пока набор входных данных не будет исчерпан.
Алгоритм **стабилен**, особо эффективен при небольших входных данных (несколько десятков).

```
1  template<class T>
2  void InsertionSort(T *A, int N) {
3      if (N < 2)
4          return;
5      for (int i = 1; i < N; i++)
6          for (int j = i; j && A[j] < A[j-1]; j--)
7              std::swap(A[j], A[j-1]);
8  }
```

5.4 Сортировка слиянием $O(N \cdot \log N)$

В основе сортировки слиянием лежит принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино. Алгоритм **стабилен**. Кушает $O(N) + O(\log N)$ памяти. Ниже представлен листинг:

1. Массив рекурсивно разбивается пополам, если размер куска $|a| > 1$
2. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив
3. Элементы перезаписываются из результирующего массива в исходный

На следующей странице представлена реализация алгоритма:

```
template<typename T>
void mergeSort(vector<T>& buf, size_t left, size_t right) {
    if(left >= right)
        return;

    size_t mid = (left + right) / 2;
    MergeSort(buf, left, mid);
    MergeSort(buf, mid + 1, right);
    merge(buf, left, right, mid);
}

template<typename T>
static void merge(vector<T>& buf, size_t left, size_t right, size_t mid) {
    if (left >= right || mid < left || mid > right)
        return;
    if (right == left + 1 && buf[left] > buf[right]) {
        swap(buf[left], buf[right]);
        return;
    }

    vector<T> tmp(&buf[left], &buf[left] + (right + 1));
    for (size_t i = left, j = 0, k = mid - left + 1; i <= right; i++) {
        if (j > mid - left) {
            buf[i] = tmp[k++];
        } else if(k > right - left) {
            buf[i] = tmp[j++];
        } else {
            buf[i] = (tmp[j] < tmp[k]) ? tmp[j++] : tmp[k++];
        }
    }
}
```

.

Глава 6

Алгебра

6.1 Алгоритм Евклида

6.1.1 НОД, НОК (gcd, lcm)

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

6.1.2 Расширенный алгоритм Евклида

```
int gcdex (int a, int b, int & x, int & y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd (b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

Функция возвращает нод и коеффициенты x, y по сслыкам

6.1.3 Восстановление в кольце по модулю

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
    cout << "no solution";
else {
    x = (x % m + m) % m;
    cout << x;
}
```

6.2 Решето Эратосфена

6.2.1 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;
```

6.2.2 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 7

Разное

7.1 Разбор выражений

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

```
def E1():
    res = E2()
    while True:
        c = getc()
        if c == '+':
            res += E2()
        elif c == '-':
            res -= E2()
        else:
            putc(c)
            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 '/').

```
def E2():
    res = E3()
    while True:
        c = getc()
        if c == '*':
            res *= E3()
        elif c == '/':
            res /= E3()
        else:
            putc(c)
            return res
```

Функция работает аналогично $E1()$. Таким образом мы можем поддерживать сколько угодно операций различных приоритетов, добавляя функции.
Перейдем к обработке скобок:

```
def E3():
    if c == '(':
        res = E1()
        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
```

7.2 Перевод между системами счистления

На Питоне очень просто:

```
1  def metabase(n, to):
2      def baseN(num,b,numerals="0123456789abcdefghijklmnopqrstuvwxyz"):
3          return ((num == 0) and numerals[0]) or (baseN(num // b, b,
               numerals).lstrip(numerals[0]) + numerals[num % b])
4      return baseN(int(str(n[0]), n[1]), to)
5
6  print(metabase((2147483647, 10), 16))
7  # Output: 7fffffff
```