

常用算法模板

Wings

2021 年 8 月 25 日

目录

0 说明	1
0.1 约定	1
0.2 模板头	1
1 数据结构	2
1.1 并查集	2
1.1.1 基本操作	2
1.1.2 带边权	2
1.2 树状数组	3
1.3 线段树	3
1.3.1 区间修改 & 区间查询	3
1.3.2 动态开点 & 合并	4
1.4 ST 表	5
1.5 主席树	6
1.6 FHQTreap	7
2 动态规划	9
2.1 背包	9
2.1.1 01 背包	9
2.1.2 完全背包	9
2.1.3 分组背包	9
2.2 最长公共子序列	9
2.3 优化	10
2.3.1 决策单调性	10
3 树相关	10
3.1 点分治	10
4 图论	11
4.1 匈牙利	11
4.2 网络流	12
4.2.1 Dinic	12
4.2.2 费用流	13
5 数学	13
5.1 取模函数	13
5.2 快速幂	13
5.3 多项式	14
5.3.1 拉格朗日插值	14
5.3.2 横坐标连续拉格朗日插值	14
5.3.3 重心拉格朗日插值 (增删点)	15

6	计算几何	15
6.1	误差修正	15
6.2	向量和点	15
6.3	直线和线段	16
6.4	多边形	17
6.5	圆	19
7	杂项	21
7.1	最长单调子序列	21
7.2	莫队	21
7.2.1	普通莫队	21
7.2.2	带修莫队	22
7.2.3	回滚莫队	23
7.3	图论分块	24
8	附录	25
8.1	Vim 配置	25
8.2	Vim 录制宏	25
8.3	GDB 命令	25
8.4	对拍	25
8.4.1	Bash 脚本	25
8.4.2	cmd 批处理脚本	26
8.5	数学公式	26
8.5.1	几何	26

0 说明

0.1 约定

- **const int** MAXN 为最大数据长度
- **const int** MAXV 为 (抽象) 图最大点数
- **const int** MAXM 为第二维数据最大长度或 (抽象) 图最大边数
- **const int** MAXQ 为最大询问
- **#ifdef GDB** 表示 GDB 调试, 从文件读取数据等
- **#ifdef NGCS** 表示输出中间变量调试 (NGCS = Nothing Gold Can Stay)
- 若无特殊情况, 数组从下标 1 开始存储数据
- 图论带边权一般以邻接表建图; 否则用向量存邻接点
- 网络流邻接表边从下标 0 开始, 因为要建反边; 否则从 1 开始
- 为节省篇幅, 代码进行部分压行

0.2 模板头

```

1  #include <cstdio>
2  #include <algorithm>
3  #include <iostream>
4  #include <cstring>
5  #include <cmath>
6  #include <vector>
7  #include <queue>
8  #include <set>
9  #include <map>
10 #include <unordered_map>
11 #include <string>
12 #define lowbit(x) (x&(-x))
13 #define LCH(x) (x<<1)
14 #define RCH(x) (x<<1|1)
15 using namespace std;
16
17 typedef long long LL;
18 typedef unsigned long long ULL;
19 typedef long double LD;
20 typedef pair<int, int> PII;
21 typedef vector<int> VI;
22
23 const int INTINF = 0x3f3f3f3f;
24 const LL INF = 0x3f3f3f3f3f3f3f3f;
25
26 const int P = 998244353;
27 const int MAXN = 1e5 + 10;
28
29 int T, n;
30
31 int main() {
32 #ifdef GDB
33     freopen("X.in", "r", stdin);
34     freopen("X.out", "w", stdout);
35 #endif
36     scanf("%d", &T);
37     while (T--) {
38         scanf("%d", &n);
39     }
40     return 0;
41 }
```

1 数据结构

1.1 并查集

1.1.1 基本操作

```

1 // 可以维护其他数据，如集合大小，集合最值。注意仅代表元的值有效
2 int n, fa[MAXN];
3 /* 初始化并查集 */
4 void init(int n) {
5     for (int i = 1; i <= n; i++) fa[i] = i;
6 }
7 /* 查询并返回 x 所在集合的代表元；路径压缩 */
8 int find(int x) {
9     return x == fa[x] ? x : fa[x] = find(fa[x]);
10 }
11 /* 合并 x, y 所在集合 (x→y)；返回合并前是否在不同集合中；无按秩合并 */
12 bool uni(int x, int y) {
13     if (find(x) == find(y))
14         return false;
15     fa[find(x)] = find(y);
16     return true;
17 }
18 /* 查询是否 x, y 是否在同一集合中 */
19 bool query(int x, int y) {
20     return find(x) == find(y);
21 }

```

1.1.2 带边权

```

1 int n, fa[MAXN], d[MAXN]; // d[x] 表示 x 到 fa[x] 的距离
2 /* 初始化并查集 */
3 void init(int n) {
4     for (int i = 1; i <= n; i++) fa[i] = i, d[i] = 0;
5 }
6 int find(int x) {
7     if (x == fa[x])
8         return x;
9     int fx = fa[x]; //先记录一下 x 当前的 fa
10    fa[x] = find(fa[x]);
11    d[x] += d[fx]; //x 到现在的 fa (即代表元) 的路径为两段之和
12    return fa[x];
13 }
14 /* 把 x 所在的集合连接到 y 所在的集合上，且 x 到 y 的距离为 dis */
15 /* 返回合并前是否在不同集合中 */
16 bool uni(int x, int y, int dis) {
17     int fx = find(x), fy = find(y);
18     if (fx == fy)
19         return false;
20     d[fx] = dis + d[y] - d[x];
21     fa[fx] = fy;
22     return true;
23 }
24 /* 查询是否 x, y 是否在同一集合中 */
25 bool query(int x, int y) {
26     return find(x) == find(y);
27 }

```

1.2 树状数组

```

1  int n, t[MAXN];
2  /* 单点修改 (加) */
3  void update(int pos, int x) {
4      for (int i = pos; i <= n; i += lowbit(i))
5          t[i] += x;
6  }
7  /* 查询前缀和 */
8  int presum(int pos) {
9      int res = 0;
10     for (int i = pos; i; i -= lowbit(i))
11         res += t[i];
12     return res;
13 }
14 /* 查询区间 [l, r] 的和 */
15 int query(int l, int r) {
16     return presum(r) - presum(l-1);
17 }

```

1.3 线段树

1.3.1 区间修改 & 区间查询

```

1  struct Node {                // 线段节点
2      int l, r, mid, len;
3      LL val, tag;            // val 为线段和的值, tag 为标记值
4  } t[MAXN << 2]; // 线段节点要开 4 倍数据点的大小
5  int n;
6  /* 更新一条完整的线段 */
7  void updateNode(int u, LL d) {
8      t[u].val += d * t[u].len;
9      t[u].tag += d; // 打上标记, 不再往下
10 }
11 /* 将标记下放 */
12 void pushDown(int u) {
13     updateNode(LCH(u), t[u].tag);
14     updateNode(RCH(u), t[u].tag);
15     t[u].tag = 0; // 取消标记
16 }
17 /* 由左右儿子线段合并更新当前线段 */
18 void pushUp(int u) {
19     t[u].val = t[LCH(u)].val + t[RCH(u)].val;
20 }
21 /* 递归建树 */
22 void build(LL data[], int l, int r, int u) {
23     t[u] = Node{l, r, (l+r)>>1, r-l+1};
24     if (l == r) t[u].val = data[l]; // 该线段只有一个点
25     else {                // 分成左右两边递归求和
26         build(data, l, t[u].mid, LCH(u));
27         build(data, t[u].mid+1, t[u].r, RCH(u));
28         pushUp(u);
29     }
30 }
31 /* 初始化一棵线段树 */
32 void init(int _n, LL data[]) {
33     build(data, 1, _n, 1);
34 }
35 /* 区间修改 */
36 void update(int l, int r, LL d, int u = 1) {
37     if (t[u].l == l && t[u].r == r)

```

```

38     updateNode(u, d); // 找到对应线段更新
39     else {
40         pushDown(u); // 访问 u 的儿子线段, 需要先下放标记更新
41         if (l > t[u].mid)
42             update(l, r, d, RCH(u)); //更新的线段全在该区间右边
43         else if (r <= t[u].mid)
44             update(l, r, d, LCH(u)); // 全在左边
45         else { // 跨越了左右两边
46             update(l, t[u].mid, d, LCH(u));
47             update(t[u].mid+1, r, d, RCH(u));
48         }
49         pushUp(u); // 由儿子线段的更新后的值计算当前线段值
50     }
51 }
52 /* 区间查询 */
53 LL query(int l, int r, int u = 1) {
54     if (t[u].l == l && t[u].r == r) return t[u].val;
55     pushDown(u);
56     if (l > t[u].mid) return query(l, r, RCH(u));
57     if (r <= t[u].mid) return query(l, r, LCH(u));
58     else return query(l, t[u].mid, LCH(u)) +
59             query(t[u].mid+1, r, RCH(u));
60 }

```

1.3.2 动态开点 & 合并

```

1 struct Node {
2     int l, r, mid, len; // 不再利用完全二叉树的下标性质
3     int lch, rch;      // 而是直接分配下标, 从而动态开点
4     LL val, tag;        // 初始值需要满足可以在开点时赋值, 如全 0
5 } t[MAXM]; // 预先估计一下点的个数
6 int n, idx = 0, root = -1;
7 /* 动态开点, 左右儿子没开, 为 0; 返回新建节点标号 */
8 int newNode(int l, int r) {
9     t[++idx] = Node{l, r, (l+r)>>1, r-l+1};
10    return idx;
11 }
12 /* 初始化线段树, 建立一个 [1, n] 区间的节点 */
13 void init(int _n) {
14     root = newNode(1, _n);
15 }
16 /* 更新一条完整的线段 */
17 void updateNode(int u, LL d) {
18     t[u].val += d * t[u].len;
19     t[u].tag += d; //打上标记, 不再往下
20 }
21 /* 将标记下放 */
22 void pushDown(int u) {
23     if (!t[u].lch) // 第一次访问, 开点
24         t[u].lch = newNode(t[u].l, t[u].mid);
25     updateNode(t[u].lch, t[u].tag);
26     if (!t[u].rch)
27         t[u].rch = newNode(t[u].mid+1, t[u].r);
28     updateNode(t[u].rch, t[u].tag);
29     t[u].tag = 0;
30 }
31 /* 由左右儿子线段合并更新当前线段 */
32 void pushUp(int u) {
33     t[u].val = t[t[u].lch].val + t[t[u].rch].val;
34 }

```

```

35  /* 区间修改 */
36  void update(int l, int r, LL d, int u = -1) {
37      if (u == -1) u = root;
38      if (t[u].l == l && t[u].r == r)
39          updateNode(u, d); // 找到对应线段更新
40      else {
41          pushDown(u); // 访问 u 的儿子线段, 需要先下放标记更新
42          if (l > t[u].mid)
43              update(l, r, d, t[u].rch); // 更新的线段全在该区间右边
44          else if (r <= t[u].mid)
45              update(l, r, d, t[u].lch); // 全在左边
46          else { // 跨越了左右两边
47              update(l, t[u].mid, d, t[u].lch);
48              update(t[u].mid+1, r, d, t[u].rch);
49          }
50          pushUp(u); // 由儿子线段的更新后的值计算当前线段值
51      }
52  }
53  /* 区间查询 */
54  LL query(int l, int r, int u = -1) {
55      if (u == -1) u = root;
56      if (t[u].l == l && t[u].r == r) return t[u].val;
57      pushDown(u);
58      if (l > t[u].mid) return query(l, r, t[u].rch);
59      if (r <= t[u].mid) return query(l, r, t[u].lch);
60      else return query(l, t[u].mid, t[u].lch) +
61                  query(t[u].mid+1, r, t[u].rch);
62  }
63
64  /* 新建一棵线段树为 x, y 两棵线段树的合并, 返回合并后的节点标号 */
65  int merge(int x, int y, int l, int r) {
66      if (!x || !y) return x | y;
67      int rt = newNode(l, r);
68      if (l == r) { // 按需合并
69          t[rt].val = t[x].val + t[y].val, t[rt].tag = t[x].tag + t[y].tag;
70          return rt;
71      }
72      t[rt].lch = merge(t[x].lch, t[y].lch, l, t[rt].mid);
73      t[rt].rch = merge(t[x].rch, t[y].rch, t[rt].mid+1, r);
74      pushUp(rt);
75      return rt;
76  }
77
78  /* 将 y 合并到 x 上, 返回合并后的节点标号 (即 x) */
79  int merge(int x, int y, int l, int r) {
80      if (!x || !y) return x || y;
81      if (l == r) {
82          t[x].val += t[y].val, t[x].tag += t[y].tag;
83          return x;
84      }
85      t[x].lch = merge(t[x].lch, t[y].lch, l, t[x].mid);
86      t[x].rch = merge(t[x].rch, t[y].rch, t[x].mid+1, r);
87      pushUp(x);
88      return x;
89  }

```

1.4 ST 表

```

1  int n, lg[MAXN], pw[MAXN], st[MAXN][30]; // st[i][j] 表示 [i, i+2^j) 的最值
2  /* 预处理出 log2x 和 2k */

```



```

3 void init() {
4     lg[0] = -1, pw[0] = 1;
5     for (int i = 1; i <= n; i++) lg[i] = lg[i>>1] + 1;
6     for (int i = 1; i <= lg[n]; i++) pw[i] = pw[i-1] << 1;
7 }
8 /* 建立 ST 表 */
9 void build(int data[]) {
10     //st[i][0] 存数据 st[i][0] = [i, i + 2^0) = data[i]
11     for (int i = 1; i <= n; i++) st[i][0] = data[i];
12     for (int j = 1; j <= lg[n]; j++) // 注意先枚举 j
13         for (int i = 1; i + pw[j] <= n + 1; i++) // 左闭右开, 到 n+1
14             //把 [i, i + 2^j) 分成 [i, i + 2^{j-1}) 和 [i + 2^{j-1}, i + 2^j) 两段
15             st[i][j] = min(st[i][j-1], st[i+pw[j-1]][j-1]);
16 }
17 /* 查询 [l, r] 中的的最值 */
18 int query(int l, int r) {
19     int k = lg[r - l + 1];
20     // 由于  $2^{\lfloor \log 2x \rfloor} > \frac{x}{2}$ , 故前后两个长度为  $2^{\lfloor \log 2x \rfloor}$  区间就可以取完所有的元素
21     return min(st[l][k], st[r+1-pw[k]][k]); // 设  $r' = r + 1$ , 查询  $[l, r')$ 
22 }

```

1.5 主席树 (静态区间 k 小)

```

1 int nn, b[MAXN]; // nn 为不重复个数, b[] 是离散化用的数组
2 /* 先将数组离散化, 返回的是不重复的数字个数 */
3 int discretize(int _n, int data[]) {
4     for (int i = 1; i <= _n; i++) b[i] = data[i];
5     sort(b + 1, b + 1 + _n);
6     return nn = unique(b + 1, b + 1 + _n) - b - 1;
7 }
8 /* 返回 x 离散化以后的 id 值 */
9 int id(int x) { return lower_bound(b + 1, b + 1 + nn, x) - b; }
10 struct Node { // val 保存的是前缀和, 区间要差分一下
11     int l, r, mid, lch, rch, val;
12 } t[MAXM]; // 点一般开  $4n + m \log n$  稍大一点, 如  $4n + m(\log n + 1)$ 
13 int n, root[MAXN], idx = 0;
14 /* 递归建一棵空树, 返回值为节点编号 */
15 int build(int l, int r) {
16     int u = ++idx;
17     t[u] = Node{l, r, (l+r)>>1, 0, 0, 0};
18     if (l < r) {
19         t[u].lch = build(l, t[u].mid);
20         t[u].rch = build(t[u].mid + 1, r);
21     }
22     return u;
23 }
24 /* 从上一棵树的对应节点拉边过来建树, 返回值为节点编号 */
25 int update(int pre, int pos, int x) {
26     int u = ++idx;
27     t[u] = t[pre]; // 复制上一棵树对应节点的所有信息
28     if (t[u].l == t[u].r) t[u].val += x;
29     else {
30         if (pos <= t[u].mid)
31             t[u].lch = update(t[pre].lch, pos, x);
32         else t[u].rch = update(t[pre].rch, pos, x);
33         t[u].val = t[t[u].lch].val + t[t[u].rch].val;
34     }
35     return u;
36 }
37 /* 递归找 [l, r] 对应区间的值 */

```

```

38 int qry(int lu, int ru, int k) {
39     if (t[lu].l == t[lu].r) return t[lu].l;
40     int x = t[t[ru].lch].val - t[t[lu].lch].val; // 差分
41     if (k <= x) return qry(t[lu].lch, t[ru].lch, k);
42     else return qry(t[lu].rch, t[ru].rch, k - x);
43 }
44 /* 查询 [l, r] 中第 k 小, 注意返回值 x 是离散化以后的, 需要 b[x] 得到原数 */
45 int query(int l, int r, int k) { return qry(root[l-1], root[r], k); }
46 /* 离散化, 并根据权值建树 */
47 void init(int _n, int data[]) {
48     nn = discretize(_n, data);
49     root[0] = build(1, n);
50     for (int i = 1; i <= _n; i++) // 根据权值建立主席树
51         root[i] = update(root[i-1], id(data[i]), 1);
52 }

```

1.6 FHQTreap(带区间翻转标记)

```

1  struct Node { //key 为排序用的随机键, size 为子树大小
2      int val, key, size, lch, rch, tag;
3  } t[MAXN];
4  int idx = 0, root;
5  /* 初始化, 输入种子 */
6  void init(int seed) {
7      idx = root = 0; // 如果多个树, 则删去 idx = 0
8      srand(seed);
9  }
10 /* 向上更新子树大小 */
11 void pushUp(int u) {
12     t[u].size = t[t[u].lch].size + 1 + t[t[u].rch].size;
13 }
14 /* 标记下放; 这里写的是反转标记, 根据题目修改 */
15 void pushDown(int u) {
16     if (t[u].tag) swap(t[u].lch, t[u].rch);
17     t[t[u].lch].tag ^= t[u].tag;
18     t[t[u].rch].tag ^= t[u].tag;
19     t[u].tag = 0;
20 }
21 /* 合并两棵树 l, r */
22 int merge(int l, int r) {
23     if (!l || !r) return l | r; // 有一个为空, 返回另一个
24     int u;
25     if (t[l].key < t[r].key) {
26         pushDown(u = l); t[l].rch = merge(t[l].rch, r);
27     }
28     else { pushDown(u = r); t[r].lch = merge(l, t[r].lch); }
29     pushUp(u);
30     return u;
31 }
32 /* 根据点的值的排名分割树 u, 前 k 小的在 l 里, 其他在 r 里 */
33 void splitByRank(int k, int &l, int &r, int u = -1) {
34     if (u == -1) u = root;
35     if (!u) { l = r = 0; return; }
36     pushDown(u);
37     if (k <= t[t[u].lch].size) {
38         r = u;
39         splitByRank(k, l, t[u].lch, t[u].lch);
40     }
41     else {
42         l = u;

```

```

43         splitByRank(k-t[t[u].lch].size-1, t[u].rch, r, t[u].rch);
44     }
45     pushUp(u);
46 }
47 /* 根据点的值分割树 u, 小于 k 的在 l 里, 其他的在 r 里 */
48 void splitByValue(int k, int &l, int &r, int u = -1) {
49     if (u == -1) u = root;
50     if (!u) { l = r = 0; return; }
51     pushDown(u);
52     if (t[u].val <= k) {
53         l = u;
54         splitByValue(k, t[u].rch, r, t[u].rch);
55     }
56     else {
57         r = u;
58         splitByValue(k, l, t[u].lch, t[u].lch);
59     }
60     pushUp(u);
61 }
62 /* 插入一个点 */
63 void insert(int x) {
64     int u, l, r;
65     t[u = ++idx] = Node{x, rand(), 1, 0, 0, 0};
66     splitByValue(x, l, r);
67     root = merge(merge(l, u), r);
68 }
69 /* 删除值为 x 的一个点 */
70 int erase(int x) {
71     int l, r, ll, rr;
72     splitByValue(x-1, l, r);
73     splitByRank(1, ll, rr, r);
74     t[ll] = Node{0, 0, 0, 0, 0};
75     root = merge(l, rr);
76     return x;
77 }
78 /* 找树 u 中值为 x 的排名 */
79 int getRankByValue(int x, int &u) {
80     int l, r, rk;
81     splitByValue(x-1, l, r, u);
82     rk = t[l].size + 1;
83     u = merge(l, r);
84     return rk;
85 }
86 /* 找树 u 中排名为 k 的值 */
87 int getValueByRank(int k, int &u) {
88     int l, r, v, ll, rr;
89     splitByRank(k-1, l, r, u);
90     splitByRank(1, ll, rr, r);
91     v = t[ll].val;
92     u = merge(l, merge(ll, rr));
93     return v;
94 }
95 /* 找到子树 u 中 x 的前驱 */
96 int getPre(int x, int &u) {
97     int l, r, pre;
98     splitByValue(x-1, l, r, u);
99     pre = getValueByRank(t[l].size, l);
100    u = merge(l, r);
101    return pre;
102 }

```

```

103 /* 找到子树 u 中 x 的后继 */
104 int getSuc(int x, int &u) {
105     int l, r, suc;
106     splitByValue(x, l, r, u);
107     suc = getValueByRank(1, r);
108     u = merge(l, r);
109     return suc;
110 }
111 /* 遍历 u */
112 void iterate(int u) {
113     if (u) {
114         pushDown(u);
115         iterate(t[u].lch);
116         printf("%d ", t[u].val);
117         iterate(t[u].rch);
118     }
119 }

```

2 动态规划

2.1 背包

2.1.1 01 背包 滚动数组 & 输出方案

```

1  for (int i = 1; i <= n; i++) for (int j = v; j >= c[i]; j--)
2      if (dp[j] < dp[j-c[i]] + w[i]) {
3          dp[j] = dp[j-c[i]] + w[i];
4          paht[i][j] = true;
5      }
6  int i = n, j = v;
7  while (i && j) {
8      if (path[i][j]) {
9          items.push(i);    // 方案
10         j -= c[i];
11     }
12     i--;
13 }

```

2.1.2 完全背包

```

1  for (int i = 1; i <= n; i++) for (int j = c[i]; j <= v; j++)
2      dp[j] = max(dp[j], dp[j-c[i]] + w[i]);

```

2.1.3 分组背包

```

1  for (int i = 0; i < n; i++) {
2      vector<int> &g = group[i];
3      for (int j = v; j > 0; j--) for (int k = 0; k < g.size(); k++)
4          if (j-c[g[k]] >= 0) dp[j] = max(dp[j], dp[j-c[g[k]]] + w[g[k]]);
5  }

```

2.2 最长公共子序列

```

1  for (int i = 1; i <= n; i++) for (int j = 1; j <= m; j++) {
2      int ans = 0;
3      if (a[i] == b[j]) ans = dp[i-1][j-1] + 1;
4      else ans = max(dp[i-1][j], dp[i][j-1]);
5      dp[i][j] = ans;
6  }

```

2.3 优化

2.3.1 决策单调性

```

1  struct Node { int l, r, p; } que[MAXN];
2  int dp[MAXN];
3  int w(int l, int r) {}; //dp[r] = dp[l] + w(l, r);
4  void DP() {
5      int tail = 0;
6      que[tail++] = Node{1, n, 0};
7      for (int i = 1; i <= n; i++) {
8          int cur = -1, L = 0, R = tail-1;
9          while (L <= R) {
10             int mid = (L+R)>>1;
11             if (que[mid].l <= i) cur = mid, L = mid + 1;
12             else R = mid - 1;
13         }
14         int p = que[cur].p, cur = -1;
15         dp[i] = dp[p] + w(p, i);
16         while (tail && dp[i] + w(i, que[tail-1].l) <= dp[que[tail-1].p] +
17             ↪ w(que[tail-1].p, que[tail-1].l))
18             cur = que[--tail].l;
19         L = que[tail-1].l, R = que[tail-1].r;
20         while (L <= R) {
21             int mid = (L+R) >> 1;
22             if (dp[i] + w(i, mid) <= dp[que[tail-1].p] + w(que[tail-1].p,
23                 ↪ mid))
24                 que[tail-1].r = mid - 1, R = mid - 1, cur = mid;
25             else L = mid + 1;
26         }
27         if (cur != -1) que[tail++] = Node{cur, n, i};
28     }
29 }

```

3 树相关

3.1 点分治

```

1  int sz[MAXN], vis[MAXN]; // sz[u] 为每次 dfs 计算以 u 为根的子树的大小
2  // descendant 保存第一次调用 dfs 的点 u 到包含 u 的树中的所有的点 v 及他们之间的距离
3  vector<PII> descendant;
4  /* 遍历, 同时计算 sz 和 descendant
5  /* 如果第一次调用 dis 参数为 0, dis 表示根 (第一次调用 dfs 的点) 到当前点 u 的距离
6  /* 不为 0 的话会在所有距离上加上这个第一次调用传入的参数 dist0) */
7  void dfs(int u, int f, int dis) {
8      sz[u] = 1;
9      descendant.emplace_back(u, dis);
10     for (int i = head[u]; i; i = edges[i].next) {
11         Edge &e = edges[i];
12         if (e.to != f && !vis[e.to]) {
13             dfs(e.to, u, dis + e.w);
14             sz[u] += sz[e.to];
15         }
16     }
17 }
18 /* 求包含 u 的树的重心 */
19 int center(int u) {
20     // 每一次调用 dfs 前都要清空 descendant
21     descendant.clear(); dfs(u, 0, 0);
22     int tot_sz = descendant.size();
23     for (auto des : descendant) {

```

```

24     int is_center = 1;
25     int x = des.first;
26     for (int i = head[x]; i; i = edges[i].next) {
27         Edge &e = edges[i];
28         if (vis[e.to]) continue; // e.to 已删除, 不考虑
29         // sz[x] > sz[v] ⇒ v 是 x 的儿子, sz[v] >  $\frac{n}{2}$ , v 不是重心
30         if (sz[x] > sz[e.to] && (sz[e.to] << 1) > tot_sz) {
31             is_center = 0; break;
32         }
33         // sz[x] < sz[v] ⇒ v 是 x 的父亲, tot\_sz - sz[x] 是 v 向上的大小
34         if (sz[x] < sz[e.to] && ((tot_sz - sz[x]) << 1) > tot_sz) {
35             is_center = 0; break;
36         }
37     }
38     if (is_center) {
39         // 找到重心, 需要以重心为根, 求一下树中所有点到重心的距离
40         // 这样调用完 center 后就可以保存所有点到重心的距离了
41         descendant.clear(); dfs(x, 0, 0); return x;
42     }
43 }
44 return -1;
45 }
46
47 void divide(int u) {
48     int c = center(u); vis[c] = 1; // 标记重心, 删去
49     work(); // 已经在 center 函数中处理了 " 经过重心的 '半路径' "
50     // 在 work 函数中考虑如何把两条半路径组合成一条路径
51     // 并考虑如何处理数据, 回答问题
52     for (int i = head[c]; i; i = edges[i].next) {
53         Edge &e = edges[i];
54         if (vis[e.to]) continue; // e.to 已经删除, 不考虑
55         iework(); // 考虑把 " 假的路径 " 容斥掉. 如果是路径, 那么求一次
56                 // dfs(e.to, c, e.w) 得到的 descendant 就是所有
57                 // " 假的路径 ", 这时候虽然第一次调用 dfs 的是 e.to,
58                 // 但由于加上的 e.w, 也可以认为是点到 c 的距离
59         divide(e.to); // 找到剩余的点 (所在的子树), 继续处理
60     }
61 }

```

4 图论

4.1 匈牙利二分图匹配

```

1  /* 寻找左边点 u 的匹配, 存图只需要左边点连向右边点的有向边即可 */
2  bool match(int u) {
3      for (int v : G[u]) if (!vis[v]) {
4          vis[v] = 1; // 右边点 v 访问标记
5          // 如果右边点没有被左边点匹配到, 让 u 和 v 匹配
6          // 或者被匹配的左边点 my[v] 可以找到其他右边点的匹配
7          // 那么让 my[v] 和其他右边点匹配, 让 u 和 v 匹配
8          if (!my[v] || match(my[v])) {
9              mx[u] = v, my[v] = u; // 记录匹配点
10             return true;
11         }
12     }
13     return false;
14 }
15 /* 匈牙利算法, 左边点个数 nx, 右边点个数 ny */
16 int hungarian(int nx, int ny) {
17     int cnt = 0;

```

```

18     for (int u = 1; u <= nx; u++) { // 每个左边点都尝试匹配
19         for (int v = 1; v <= ny; v++) vis[v] = 0;
20         if (match(u)) cnt++;
21     }
22     return cnt;
23 }

```

4.2 网络流

4.2.1 Dinic

```

1  int d[MAXV], cur[MAXV];
2  //bfs 找可前进边, 即  $d[v] = d[u] + 1$  且可以流的  $(u, v)$ 
3  bool bfs(int s, int t) {
4      memset(d, INTINF, sizeof(d));
5      queue<int> Q;
6      Q.push(s); d[s] = 0;
7      while (!Q.empty()) {
8          int u = Q.front(); Q.pop();
9          for (int i = head[u]; ~i; i = edges[i].next) {
10              Edge &e = edges[i];
11              //找还可以流的边
12              if (d[e.to] >= INTINF && e.cap > e.flow) {
13                  d[e.to] = d[u] + 1; Q.push(e.to);
14              }
15          }
16      }
17      return d[t] < INTINF;
18  }
19  //当前点  $u$ , flow 已经流过的流量, a 还可以继续流的流量
20  //dfs 找多条增广路
21  int dfs(int u, int a, int t) {
22      //流到终点或者流不下去了
23      //流到终点返回可流量, 在下面的  $a -= f$  时  $a=0$ , 然后结束这个点的搜索
24      if (u == t || a == 0)
25          return a;
26      int f = 0, flow = 0;
27      //当前弧优化 &i = cur 搜完了这条路, 就不会再搜这条路了, 所以从后面开始搜
28      for (int &i = cur[u]; ~i; i = edges[i].next) {
29          Edge &e = edges[i];
30          if (d[e.to] == d[u] + 1) {
31              f = dfs(e.to, min(a, e.cap - e.flow));
32              if (f > 0) {
33                  e.flow += f; flow += f;
34                  edges[i^1].flow -= f; a -= f;
35                  if (!a) break; //没有可以供其他路流的流量 (或者流完了)
36              }
37          }
38      }
39      if (a) //GAP 优化 如果后面的都流完了, 这个点还有流量剩余, 那么不用再搜这个点了
40          d[u] = -1;
41      return flow;
42  }
43  int dinic(int s, int t) {
44      int flow = 0;
45      while (bfs(s, t)) {
46          for (int i = 1; i <= v; i++) cur[i] = head[i];
47          flow += dfs(s, INTINF);
48      }
49      return flow;

```

50 }

4.2.2 费用流 EK 算法

```

1  int d[MAXV], a[MAXV], inq[MAXV], p[MAXV];
2  bool spfa(int s, int t, int &flow, int &cost) {
3      memset(d, INTINF, sizeof(d)); memset(a, 0, sizeof(a));
4      queue<int> Q;
5      Q.push(s), inq[s] = 1;
6      d[s] = 0, a[s] = INTINF, p[s] = -1;
7      while (!Q.empty()) {
8          int u = Q.front();
9          Q.pop(), inq[u] = 0;
10         for (int i = head[u]; ~i; i = edges[i].next) {
11             Edge &e = edges[i];
12             if (d[e.to] > d[u] + e.cost && e.cap > e.flow) {
13                 d[e.to] = d[u] + e.cost;
14                 a[e.to] = min(a[u], e.cap - e.flow);
15                 p[e.to] = i;
16                 if (!inq[e.to]) Q.push(e.to), inq[e.to] = 1;
17             }
18         }
19     }
20     if (d[t] >= INTINF) return false;
21     flow += a[t];
22     cost += d[t] * a[t];
23     for (int i = p[t]; ~i; i = p[edges[i^1].to])
24         edges[i].flow += a[t], edges[i^1].flow -= a[t];
25     return true;
26 }
27 void mcmf(int s, int t, int &flow, int &cost) {
28     while (spfa(s, t, flow, cost));
29 }

```

5 数学

5.1 取模函数

```

1  int pls(LL a, LL b) {
2      a += a < 0 ? P : 0, b += b < 0 ? P : 0;
3      return a + b >= P ? a + b - P : a + b;
4  }
5  int mult(LL a, LL b) {
6      a += a < 0 ? P : 0, b += b < 0 ? P : 0;
7      return a * b >= P ? a * b % P : a * b;
8  }

```

5.2 快速幂

```

1  /* power(x) 为 x 在模 P 意义下的逆元, P 必须为质数 */
2  int power(int a, int b = P-2) {
3      int res = 1;
4      while (b) {
5          if (b&1) res = mult(res, a);
6          a = mult(a, a);
7          b >>= 1;
8      }
9      return res;
10 }

```


5.3 多项式

5.3.1 拉格朗日插值

$$L(x) = \sum_{i=0}^{n-1} y_i \ell_i(x)$$

$$\ell_i(x) = \prod_{j=0, j \neq i}^{n-1} \frac{(x - x_j)}{(x_i - x_j)} = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_{n-1})}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_{n-1})}$$

$$L(x) = \sum_{i=0}^{n-1} y_i \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}$$

```

1  /* 求 n 个点值确定的 n 次数多项式拉格朗日插值在 m 处的函数值 */
2  /* 点 (x, y) 从下标 0 开始 */
3  int Lagrange(int *x, int *y, int n, int m) {
4      int L = 0;
5      for (int i = 0; i < n; i++) {
6          int p = 1, q = 1;
7          for (int j = 0; j < n; j++) if (j != i)
8              p = mult(p, m - x[j]), q = mult(q, x[i] - x[j]);
9          L = pls(L, mult(y[i], mult(p, power(q))));
10     }
11     return L;
12 }

```

5.3.2 横坐标连续拉格朗日插值

$$pre_i(\xi) = \prod_{j=0}^i (\xi - x_j), suf_i(\xi) = \prod_{j=i}^{n-1} (\xi - x_j)$$

$$\ell_i(x) = \frac{pre_{i-1}(x) \cdot suf_{i+1}(x)}{(-1)^{n-1-i} (x_i - x_0)! (x_{n-1} - x_i)!}$$

```

1  // finv[x] 表示 (x!)^{-1}, inv_neg_1 表示 (-1)^{-1}
2  // l[i] 是第 i 项拉格朗日基本多项式
3  int pre[MAXA], suf[MAXA], finv[MAXN], inv_neg_1, l[MAXN];
4  /* 预处理阶乘逆元, 注意要处理到最高次数 */
5  void init(int n) {
6      int facn = 1;
7      for (int i = 2; i <= n; i++) facn = mult(facn, i);
8      finv[n] = power(facn);
9      for (int i = n-1; ~i; i--) finv[i] = mult(i+1, finv[i+1]);
10     inv_neg_1 = power(-1);
11 }
12 /* 求 n 个点值确定的 n 次数多项式拉格朗日插值在 x 处的函数值 */
13 /* 其中点的横坐标为 s, s+1, s+2..., s+n-1, (i, y) 从下标 s 开始 */
14 int Lagrange(int s, int *y, int n, int x) {
15     pre[0] = x - s;
16     for (int i = 1; i < n; i++) pre[i] = mult(pre[i-1], x - i - s);
17     suf[n] = 1;
18     for (int i = n-1; ~i; i--) suf[i] = mult(suf[i+1], x - i - s);
19     l[0] = mult(suf[1], finv[n-1]);
20     if ((n-1)&1) l[0] = mult(l[0], inv_neg_1);
21     for (int i = 1; i < n; i++) {
22         l[i] = mult(mult(pre[i-1], suf[i+1]), mult(finv[i], finv[n-1-i]));
23         if ((n-1-i)&1) l[i] = mult(l[i], inv_neg_1);
24     }
25     int L = 0;

```

```

26     for (int i = 0; i < n; i++) L = pls(L, mult(y[i], l[i]));
27     return L;
28 }

```

5.3.3 重心拉格朗日插值 (增删点)

$$w_i = \frac{1}{\prod_{j \neq i} (x_i - x_j)}$$

$$L(x) = \frac{\sum_{i=0}^{n-1} \frac{w_i}{x - x_i} y_i}{\sum_{i=0}^{n-1} \frac{w_i}{x - x_i}}$$

```

1  /* 向点集 S(X, Y) 中加入点 (x, y), W 为重心权 */
2  void add(int x, int y, VI &X, VI &Y, VI &W) {
3      int n = X.size();
4      X.push_back(x), Y.push_back(y), W.push_back(1);
5      for (int i = 0; i < n; i++) W[i] = mult(W[i], power(X[i] - x));
6      for (int i = 0; i < n; i++) W[n] = mult(W[n], x - X[i]);
7      W[n] = power(W[n]); // 维护重心权
8  }
9  /* 计算点集 S(X, Y) 表示的多项式在 x 处的函数值, W 为重心权 */
10 int Lagrange(VI &X, VI &Y, VI &W, int x) {
11     int L = 0, g = 0, n = X.size();
12     for (int i = 0; i < n; i++) {
13         // 如果是已知点, 则直接求, 否则  $\frac{1}{x-x_i}$  没有意义
14         if (x == X[i]) return Y[i];
15         int p = mult(W[i], power(x - X[i]));
16         L = pls(L, mult(p, Y[i])), g = pls(g, p);
17     }
18     return mult(L, power(g));
19 }

```

6 计算几何

6.1 误差修正 (eps 和 sgn 函数)

```

1  const double eps = 1e-8;
2  int sgn(double x) {
3      if (fabs(x) < eps) return 0;
4      return x > 0 ? 1 : -1;
5  }

```

6.2 向量和点

```

1  struct Vec {
2      double x, y;
3      Vec operator + (const Vec &B) const { return Vec{x + B.x, y + B.y}; }
4      Vec operator - (const Vec &B) const { return Vec{x - B.x, y - B.y}; }
5      Vec operator * (const double k) const { return Vec{x * k, y * k}; }
6      Vec operator / (const double k) const { return Vec{x / k, y / k}; }
7      double operator * (const Vec &B) const { return x * B.x + y * B.y; }
8      double operator ^ (const Vec &B) const { return x * B.y - y * B.x; }
9      bool operator == (const Vec &B) const {
10         return !sgn(x - B.x) && !sgn(y - B.y);
11     }
12     bool operator < (const Vec &V) const {
13         return x == V.x ? y < V.y : x < V.x;
14     }
15     double length() {

```

```

16     Vec A = *this;
17     return sqrt(A * A);
18 }
19 /* 化为长度为 r 的向量 */
20 Vec trunc(double r){
21     double l = length();
22     if(!sgn(l)) return (*this);
23     r /= l;
24     return Vec{x * r, y * r};
25 }
26 /* 逆时针旋转 90 度 */
27 Vec rotleft(){ return Vec{-y, x}; }
28 /* 顺时针旋转 90 度 */
29 Vec rotright(){ return Vec{y, -x}; }
30 /* 绕着点 P 逆时针旋转 angle */
31 Vec rotate(Vec p, double angle) {
32     Vec v = (*this) - p;
33     double c = cos(angle), s = sin(angle);
34     return Vec{p.x + v.x * c - v.y * s, p.y + v.x * s + v.y * c};
35 }
36 };
37 typedef Vec Point;    // 点的记录方式和向量相同
38 /* 求两向量的夹角 (无向, 且夹角在 [0, PI] 之间, 弧度制) */
39 double angle(Vec &A, Vec &B) {
40     return acos((A * B) / A.length() / B.length());
41 }
42 /* 两点距离 */
43 double pointDistance(Point A, Point B) { return (A-B).length(); }
44 /* 三角形面积 */
45 double triangle_area(Point A, Point B, Point C) {
46     Vec v = B - A, u = C - A;
47     return fabs(v ^ u) / 2;
48 }

```

6.3 直线和线段

```

1 struct Line {
2     Point P; Vec v; // 点向式, 注意两点需要转化再存储
3     /* 获取直线上某一点 (线段起点) */
4     Point point(double t = 0) { return P + (v * t); }
5     /* 获取线段终点 */
6     Point endpoint() { return point(1); }
7     /* 点相对直线的位置, 1 为右边, -1 为左边, 0 在直线上 */
8     int pointRelative(Point Q) { return sgn((Q - P) ^ v); }
9     /* 点到直线的有向距离, 正为右边, 负为左边, 0 在直线上 */
10    double pointDistance(Point Q) { return ((Q - P) ^ v) / v.length(); }
11    /* 点在直线上的投影 */
12    Point pointProjection(Point Q) {
13        return point(((Q - P) * v) / v.length() / v.length());
14    }
15    /* 点是否在线段上 (不包括端点) 如需包括端点, 改 < 为 <= */
16    bool isContainPoint(Point Q) {
17        return !pointRelative(Q) && sgn((Q-point()) * (Q-endpoint())) < 0;
18    }
19    /* 线段长度 */
20    double length() { return v.length(); }
21    /* 求点 P 关于直线的对称点 */
22    Point pointSymmetry(Point P){
23        Point Q = pointProjection(P);
24        return Point{2 * Q.x - P.x, 2 * Q.y - P.y};

```

```

25     }
26 };
27 typedef Line Segment; // 线段也用点向式存, 注意两点转化
28 /* 直线是否相交 */
29 bool isLineIntersection(Line l1, Line l2) { return sgn(l1.v ^ l2.v); }
30 /* 直线是否平行 */
31 bool isLineParallel(Line l1, Line l2) {
32     return !isLineIntersection(l1, l2);
33 }
34 /* 直线是否重合 */
35 bool isLineCoincident(Line l1, Line l2) {
36     return isLineParallel(l1, l2) && !l1.pointRelative(l2.point()) &&
37         ↪ !l1.pointRelative(l2.endpoint());
38 }
39 /* 求两直线的交点, 调用前需保证相交 */
40 Point getLineIntersection(Line l1, Line l2) {
41     Vec u = l1.point() - l2.point();
42     double t = (l2.v ^ u) / (l1.v ^ l2.v);
43     return l1.point(t);
44 }
45 /* 线段是否与直线相交 (不包括端点) 包括端点该 < 为 <= */
46 bool isSegmentIntersectLine(Segment s, Line l) {
47     return l.pointRelative(s.point()) * l.pointRelative(s.endpoint()) < 0;
48 }
49 /* 线段是否相交 (是否包括端点取决于上一个函数) */
50 bool isSegmentIntersecting(Segment s1, Segment s2) {
51     return isSegmentIntersectLine(s1, s2) && isSegmentIntersectLine(s2,
52         ↪ s1);
53 }

```

6.4 多边形

```

1 struct Polygon {
2     int n; // 数据大可能需要不使用结构体
3     Point points[MAXN]; // 逆时针排序
4     Line lines[MAXN]; // 边
5     void init(int _n, Point _ps[]) {
6         n = _n;
7         for (int i = 1; i <= n; i++)
8             points[i] = _ps[i];
9         points[0] = _ps[n];
10    }
11    /* 获取边 */
12    void getLines() {
13        for (int i = 0; i < n; i++)
14            lines[i+1] = Line{points[i], points[i+1] - points[i]};
15    }
16    /* 以点 p0 为基点, 进行极角排序的比较函数 */
17    struct cmp{
18        Point p;
19        cmp(const Point &p0) { p = p0; }
20        bool operator() (const Point &aa, const Point &bb) {
21            Point a = aa, b = bb;
22            int d = sgn((a - p) ^ (b - p));
23            if (d == 0)
24                return sgn(pointDistance(p, a) - pointDistance(p, b)) < 0;
25            return d > 0;
26        }
27    };
28    /* 极角排序 */

```

```

29 void norm() {
30     Point mi = points[1];
31     for (int i = 2; i <= n; i++) mi = min(mi, points[i]);
32     sort(points + 1, points + 1 + n, cmp(mi));
33 }
34 /* 求多边形有向面积 */
35 double area() {
36     double res = 0;
37     for (int i = 0; i < n; i++)
38         res += points[i] ^ points[i + 1];
39     return res / 2.0;
40 }
41 /* 求多边形周长 */
42 double circumference() {
43     double sum = 0;
44     for(int i = 0; i < n; i++)
45         sum += pointDistance(points[i], points[i + 1]);
46     return sum;
47 }
48 /* Graham 求凸包; 注意如果有影响, 要特判下所有点共点, 或者共线的特殊情况 */
49 void graham(Polygon &convex, bool is_norm) {
50     if (!is_norm) norm();
51     int &top = convex.n;
52     top = 0;
53     if (n == 1) {
54         top = 1, convex.points[1] = points[1];
55     }
56     else if (n == 2) {
57         top = 2, convex.points[1] = points[1], convex.points[2] =
            ↪ points[2];
58         if (convex.points[1] == convex.points[2]) top--;
59     }
60     else {
61         top = 2, convex.points[1] = points[1], convex.points[2] =
            ↪ points[2];
62         for (int i = 3; i <= n; i++){
63             while (top > 1 && sgn((convex.points[top] - convex.points[top
            ↪ - 1]) ^ (points[i] - convex.points[top - 1])) <= 0)
64                 top--;
65             convex.points[++top] = points[i];
66         }
67         if(convex.n == 2 && (convex.points[1] == convex.points[2]))
            ↪ convex.n--;//特判
68     }
69     convex.points[0] = convex.points[convex.n];
70 }
71 /* 判断多边形是不是凸的 */
72 bool isConvex(){
73     bool s[2] = {0, 0};
74     for (int i = 0; i < n; i++) {
75         int j = (i + 1) % n;
76         int k = (j + 1) % n;
77         s[sgn((points[j] - points[i]) ^ (points[k] - points[i])) + 1]
            ↪ = true;
78         if(s[0] && s[2]) return false;
79     }
80     return true;
81 }
82 /* 点与多边形位置; 3 点上, 2 边上, 1 内部, 0 外部 */
83 int pointRelation(Point q){

```

```

84     for(int i = 1; i <= n; i++) if(points[i] == q) return 3;
85     getLines();
86     for(int i = 1; i <= n; i++)
87         if(lines[i].isContainPoint(q)) return 2;
88     int cnt = 0;
89     for(int i = 0; i < n; i++){
90         int j = i + 1;
91         int k = sgn((q - points[j]) ^ (points[i] - points[j]));
92         int u = sgn(points[i].y - q.y), v = sgn(points[j].y - q.y);
93         if (k > 0 && u < 0 && v >= 0) cnt++;
94         if (k < 0 && v < 0 && u >= 0) cnt--;
95     }
96     return cnt != 0;
97 }
98 };

```

6.5 圆

```

1  const double PI = 3.14159265358;
2  struct Circle {
3      Point O; double r; //圆心和半径
4      /* 面积 */
5      double area() { return PI * r * r; }
6      /* 周长 */
7      double circumFERENCE() { return 2 * PI * r; }
8      /* 点和圆的关系; 0 圆上, 1 圆内, -1 圆外 */
9      int pointRelation(Point P){
10         double dst = pointDistance(O, P);
11         if (sgn(dst - r) < 0) return 1;
12         else if (!sgn(dst - r)) return 0;
13         return -1;
14     }
15     /* 直线和圆的关系; 返回交点个数 */
16     int lineRelation(Line l){
17         double dst = fabs(l.pointDistance(O));
18         if (sgn(dst - r) < 0) return 2;
19         else if (!sgn(dst - r)) return 1;
20         return 0;
21     }
22     /* 两圆的关系; 5 相离, 4 外切, 3 相交, 2 内切, 1 内含 */
23     int circleRelation(Circle v) {
24         double d = pointDistance(O, v.O);
25         if (sgn(d - r - v.r) > 0) return 5;
26         if (sgn(d - r - v.r) == 0) return 4;
27         double l = fabs(r - v.r);
28         if (sgn(d - r - v.r) < 0 && sgn(d - l) > 0) return 3;
29         if (sgn(d - l) == 0) return 2;
30         return 1;
31     }
32     /* 求两个圆的交点; 返回交点个数 */
33     int getCircleIntersection(Circle v, Point &p1, Point &p2) {
34         int rel = circleRelation(v);
35         if (rel == 1 || rel == 5) return 0;
36         double d = pointDistance(O, v.O);
37         double l = (d * d + r * r - v.r * v.r) / (2 * d);
38         double h = sqrt(r * r - l * l);
39         Point tmp = O + (v.O - O).trunc(l);
40         p1 = tmp + ((v.O - O).rotleft().trunc(h));
41         p2 = tmp + ((v.O - O).rotright().trunc(h));
42         if(rel == 2 || rel == 4) return 1;

```

```

43     return 2;
44 }
45 /* 求直线和圆的交点; 返回交点个数 */
46 int getLineIntersection(Line l, Point &p1, Point &p2){
47     if(!(*this).lineRelation(l)) return 0;
48     Point A = l.pointProjection(0);
49     double d = l.pointDistance(0);
50     d = sqrt(r * r - d * d);
51     if (!sgn(d)) {
52         p1 = p2 = A;
53         return 1;
54     }
55     p1 = A + l.v.trunc(d), p2 = A - l.v.trunc(d);
56     return 2;
57 }
58 /* 求过某点的切线; 返回切线条数 */
59 int getTangentLine(Point q, Line &u, Line &v) {
60     int x = pointRelation(q);
61     if(x == 2) return 0;
62     if(x == 1) {
63         v = u = Line{q, q - O.rotleft()};
64         return 1;
65     }
66     double d = O.distance(q), l = r * r / d, h = sqrt(r * r - l * l);
67     u = Line{q, O + ((q-O).trunc(l) + (q-O).rotleft().trunc(h)) - q};
68     v = Line{q, O + ((q-O).trunc(l) + (q-O).rotright().trunc(h)) - q};
69     return 2;
70 }
71 /* 求三角形 OAB 与圆相交的面积 */
72 double triangIntersectingArea(Point A, Point B) {
73     if(!sgn((O - A) ^ (O - B))) return 0.0;
74     Point q[5];
75     int len = 0;
76     q[len++] = A;
77     Line l(A, B - A);
78     Point p1,p2;
79     if (getLineIntersection(l, q[1], q[2]) == 2) {
80         if(sgn((A - q[1]) * (B - q[1])) < 0) q[len++] = q[1];
81         if(sgn((A - q[2]) * (B - q[2])) < 0) q[len++] = q[2];
82     }
83     q[len++] = B;
84     if (len == 4 && sgn((q[0] - q[1]) * (q[2] - q[1])) > 0)
85         swap(q[1], q[2]);
86     double res = 0;
87     for (int i = 0; i < len - 1; i++) {
88         if (!pointRelation(q[i]) || !pointRelation(q[i + 1])) {
89             double arg = angle(q[i] - O, q[i + 1] - O);
90             res += r * r * arg / 2.0;
91         }
92         else res += fabs((q[i] - O) ^ (q[i + 1] - O)) / 2.0;
93     }
94     return res;
95 }
96 };
97 /* 三角形外接圆 */
98 Circle getOuterCircle(Point A, Point B, Point C) {
99     Line u = Line((A + B) / 2, ((A + B) / 2) + ((B - A).rotleft()));
100    Line v = Line((B + C) / 2, ((B + C) / 2) + ((C - B).rotleft()));
101    Point O = get_line_intersect(u, v);
102    double r = pointDistance(A, O);

```

```

103     return Circle{0, r};
104 }
105 /* 三角形内切圆 */
106 Circle getInnerCircle(Point A, Point B, Point C) {
107     double m = atan2(B.y - A.y, B.x - A.x);
108     double n = atan2(C.y - A.y, C.x - A.x);
109     Line u = Line{A, Point{cos((n + m) / 2), sin((n + m) / 2)}};
110     m = atan2(A.y - B.y, A.x - B.x), n = atan2(C.y - B.y, c.x - B.x);
111     Line v = Line{B, Point{cos((n+m)/2),sin((n+m)/2)}};
112     Point O = get_line_intersect(u, v);
113     Line AB = Line{A, B - A};
114     r = fabs(AB.pointDistance(O));
115     return Circle{O, r};
116 }
117 /* 两圆相交面积 */
118 double circleIntersectingArea(Circle c1, Circle c2){
119     int rel = c1.circleRelation(c2);
120     if(rel >= 4) return 0.0;
121     if(rel <= 2) return min(c1.area(),c2.area());
122     double d = pointDistance(c1.O, c2.O);
123     double hf = (c1.r + c2.r + d) / 2.0;
124     double ss = 2 * sqrt(hf * (hf - c1.r) * (hf - c2.r) * (hf - d));
125     double a1 = acos((c1.r*c1.r + d*d - c2.r*c2.r) / (2.0 * c1.r * d));
126     a1 *= c1.r * c1.r;
127     double a2 = acos((c2.r*c2.r + d*d - c1.r*c1.r) / (2.0 * c2.r * d));
128     a2 *= c2.r * c2.r;
129     return a1 + a2 - ss;
130 }

```

7 杂项

7.1 最长单调子序列

```

1  int len = 1;
2  b[0] = a[0];
3  for (int i = 1; i < n; i++) {
4      if (a[i] >= b[len-1]) b[len++] = a[i];
5      else {
6          int p = lower_bound(b, b + len, a[i]);
7          b[p] = a[i];
8      }
9  }

```

7.2 莫队

7.2.1 普通莫队

```

1  int n, q, id[MAXN], block_size, a[MAXN];
2  struct Query {
3      int idx, l, r;
4      bool operator < (const Query &Q) const {
5          // 排序规则: l 在同一块, 按 r 递增排序; 不在同一块按块编号递增排序
6          return id[l] == id[Q.l] ? r < Q.r : l < Q.l;
7      }
8  } query[MAXN];
9  /* 增加值, 更新维护信息 */
10 void add(int c) {}
11 /* 删除值, 更新维护信息 */
12 void del(int c) {}
13 /* 通过当前维护的信息计算当前询问的答案 */

```



```

14 void getAns(int idx) {}
15 /* 初始化, 计算块大小和块编号, 并离线询问排序 */
16 void init() {
17     block_size = sqrt(n);
18     for (int i = 1; i <= n; i++)
19         id[i] = (i-1) / block_size + 1;
20     sort(query + 1, query + q + 1);
21 }
22 void solve() {
23     init();
24     int l = query[1].l, r = l-1;
25     for (int i = 1; i <= q; i++) {
26         while (l > query[i].l) add(a[--l]);
27         while (r < query[i].r) add(a[++r]);
28         while (l < query[i].l) del(a[l++]);
29         while (r > query[i].r) del(a[r--]);
30         getAns(query[i].idx);
31     }
32 }

```

7.2.2 带修莫队

```

1 // q 是询问个数, T 是修改总时间 (个数), qc 是询问和修改的总数 (读入)
2 int T = 0, n, q = 0, qc, block_size, id[MAXN], a[MAXN];
3 struct Query {
4     int idx, t, l, r;
5     bool operator < (const Query &Q) const {
6         // 排序规则: l 在同一块, r 在同一块, 按 t 递增排序;
7         // r 不在同一块按 r 所在块编号递增排序;
8         // l 不在同一块按 l 所在块编号递增排序
9         return id[l] == id[Q.l] ? id[r] == id[Q.r] ? t < Q.t :
10             id[r] < id[Q.r] : id[l] < id[Q.l];
11     }
12 } query[MAXQ];
13 struct Modify { int pos, a; } modify[MAXQ];
14 /* 增加值, 更新维护信息 */
15 void add(int c) {}
16 /* 删除值, 更新维护信息 */
17 void del(int c) {}
18 /* 通过当前维护的信息计算当前询问的答案 */
19 void getAns(int idx) {}
20 /* 应用时间为 t 的修改, 如果修改的点在当前维护的区间, 则需要统计维护 */
21 void change(int l, int r, int t) {
22     int &pos = modify[t].pos, &c = modify[t].a;
23     if (l <= pos && pos <= r)
24         del(a[pos]), add(c);
25     swap(c, a[pos]);
26 }
27 /* 初始化, 计算块大小和块编号, 并离线询问排序 */
28 void init() {
29     block_size = pow(n, 2./3.);
30     for (int i = 1; i <= n; i++)
31         id[i] = (i-1) / block_size + 1;
32     sort(query+1, query+q+1);
33 }
34 void solve() {
35     init();
36     int l = query[1].l, r = l-1, t = 0;
37     for (int i = 1; i <= q; i++) {
38         while (l > query[i].l) add(a[--l]);

```

```

39     while (r < query[i].r) add(a[++r]);
40     while (l < query[i].l) del(a[l++]);
41     while (r > query[i].r) del(a[r--]);
42     while (t < query[i].t) change(l, r, ++t);
43     while (t > query[i].t) change(l, r, t--);
44     getAns(query[i].idx);
45 }
46 }

```

7.2.3 回滚莫队

```

1 // 求区间重要度, 重要度: 某数出现的次数乘以他本身, 这个值的最大值
2 int n, q, block_size, id[MAXN], ID = 0, a[MAXN], cnt[MAXN], tmpcnt[MAXN];
3 LL num[MAXN], ans[MAXN];
4 unordered_map<int, int> mp; // 原题数据需要离散化
5 /* 获得块左端点 */
6 int getBlockR(int block_id) { return min(block_id * block_size, n); }
7 /* 获得块右端点 */
8 int getBlockL(int block_id) { return getBlockR(block_id-1) + 1; }
9 struct Query {
10     int idx, l, r;
11     bool operator < (const Query &Q) const {
12         // 排序规则: l 在同一块, 按 r 递增排序; 不在同一块按块编号递增排序
13         return id[l] == id[Q.l] ? r < Q.r : id[l] < id[Q.l];
14     }
15 } query[MAXN];
16 /* 初始化, 计算块大小和块编号, 并离线询问排序 */
17 void init() {
18     block_size = sqrt(n);
19     for (int i = 1; i <= n; i++)
20         id[i] = (i-1) / block_size + 1;
21     sort(query+1, query+1+q);
22 }
23 void solve() {
24     init();
25     int l = 1, r = l-1; LL mx = -1;
26     for (int i = 1; i <= q; i++) {
27         // l 和上一个不在同一块中, 回滚永久信息
28         if (id[query[i].l] != id[query[i-1].l]) {
29             mx = -1;
30             while (r >= l) --cnt[a[r--]];
31             l = getBlockL(id[query[i].l]+1);
32             r = l-1;
33         }
34         // 维护永久信息
35         while (r < query[i].r) {
36             cnt[a[++r]]++;
37             mx = max(mx, cnt[a[r]] * num[a[r]]);
38         }
39         // 暴力维护临时信息
40         LL tmpmx = -1;
41         for (int j = query[i].l; j <= min(query[i].r,
42             ↪ getBlockR(id[query[i].l])); j++)
43             // 注意统计临时信息要合并永久信息的部分
44             tmpmx = max(tmpmx, (++tmpcnt[a[j]] + cnt[a[j]]) * num[a[j]]);
45         // 回滚临时信息
46         for (int j = query[i].l; j <= min(query[i].r,
47             ↪ getBlockR(id[query[i].l])); j++)
48             tmpcnt[a[j]]--;
49         ans[query[i].idx] = max(tmpmx, mx); // 更新答案

```

```

48     }
49 }

```

7.3 图论分块

```

1  //  $n$  个点  $m$  条边的简单图, 修改点权, 询问邻居点权和
2  int n, m, deg[MAXN], S, a[MAXN], sum[MAXN];
3  VI G[MAXN], E[MAXN];    //  $G$  是原图,  $E$  是指向重点的有向边
4  /* 根据度数确定轻重点, 向重点连的边存在  $E$  里 */
5  void build() {
6      S = sqrt(2 * m);    // 度数大于  $\sqrt{2m}$  的为重点
7      for (int i = 1; i <= n; i++)
8          for (int v : G[i]) if (deg[v] > S)
9              E[i].push_back(v);
10 }
11 /* 询问点  $u$  的邻居的点权和 */
12 int query(int u) {
13     if (deg[u] > S) return sum[u]; // 如果是重点, 返回维护的邻居和
14     int res = 0;
15     for (int v : G[u]) res += a[v]; // 如果是轻点, 直接暴力
16     return res;
17 }
18 /* 修改点  $u$  的权值 */
19 void update(int u, int d) {
20     a[u] += d;
21     for (int v : E[u]) sum[v] += d; // 维护重点的邻居和
22 }

```

8 附录

8.1 Vim 配置

新建文件 ~/.vimrc

```

1 set nu                " 显示行号
2 set ai                " 自动缩进
3 set cindent           "C 语言自动缩进
4 set ts=4              "tab 宽度为 4
5 set sw=4              " 缩进宽度为 4
6
7 imap {}<CR> {}<CR><SPACE><CR>}<UP><END><BACKSPACE>    " 映射, 方便写 (个人习惯)
8
9 " 设置编译运行的命令快捷键
10 map <F5> :call Compile()<CR>
11 map <F6> :call Run()<CR>
12 map <F7> :call CompileRun()<CR>
13
14 func! Compile()
15     silent w
16     silent !clear
17     exe "!g++ % -o %< -std=c++11 -Wall -DNGCS -g"
18 endfunc
19
20 func! Run()
21     silent !clear
22     exe "!time ./%<"
23 endfunc
24
25 func! CompileRun()
26     exe Compile()
27     exe Run()
28 endfunc

```

8.2 Vim 录制宏

录制: 在 normal 模式下按 q, 然后再按 a-z 中的某个字符 (小写, 代表覆盖. 大写代表追加). 在 normal 模式下按 q 结束录制.

输入: 在 normal 模式下按下 @, 再按下 a-z 中的某个字符, 输入对应字符保存的宏.

8.3 GDB 命令

功能	命令
设置断点	b 行号/函数名
当条件满足时断点	b 行号/函数名 if 条件
打印变量/表达式的值	p 变量名/表达式
持续显示变量/表达式值	disp 变量名/表达式
单步运行	n
单步进入	s
继续 (下一个断点)	c
退出	q
(重新) 运行	r

8.4 对拍

8.4.1 Bash 脚本

```

1 while true; do
2     ./X-data > ./X.in
3     ./X < ./X.in > ./X.out
4     ./X-bf < ./X.in > ./X-bf.out

```

```

5      if diff ./X.out ./X-bf.out; then
6          printf "AC\n"
7      else
8          printf "WA\n"
9          cat ./X.in
10         exit 0
11     fi
12 done

```

8.4.2 cmd 批处理脚本

```

1 @echo off
2 :loop
3     X-data.exe > X.in
4     X.exe < X.in > X.out
5     X-bf.exe < X.in > X-bf.out
6     fc X.out X-bf.out
7     if not errorlevel 1 goto loop
8     pause
9 goto loop

```

8.5 数学公式

8.5.1 几何

缺球体积	$V = \pi H^2 \left(R - \frac{H}{3} \right)$
两圆相交弓型高度/	$h_1 = r_1 - \frac{r_1^2 - r_2^2 + d^2}{2d}$
两球相交缺球高度	$h_2 = r_2 - \frac{r_2^2 - r_1^2 + d^2}{2d}$