

Zlin的板子大全

废物Zlin的自用模版合集，代码风格一般，运行效率低下，代码实用性弱，收集完善度弱，路过大佬轻点骂

欢迎提出各种意见 本人版权意识薄弱

你看nm呢，回去训练

杂项

对拍

duipai模版

```
1  #include<iostream>
2  #include<windows.h>
3  using namespace std;
4  int main()
5  {
6      int t=1000;
7      while(t)
8      {
9          t--;
10         system("data.exe > data.txt");
11         system("a.exe < data.txt > a.txt");
12         system("b.exe < data.txt > b.txt");
13         if(system("fc a.txt b.txt")) break;
14     }
15     if(t==0) cout<<"no error"<<endl;
16     else cout<<"error"<<endl;
17
18     return 0;
19 }
20
21 // 注意编译文件的路径
22 //Mac 记得在终端编译运行, zlin's MBP默认g++-14
23 #include<iostream>
24 #include<stdlib.h> // 在 Unix 系统下包含 system 函数需要使用 <stdlib.h>
25
26 using namespace std;
27
28 int main() {
29     int t = 1000;
30     for (int i = 1; i <= t; i++) {
31         system("./data > data.txt");
32         system("./a < data.txt > a.txt");
33         system("./b < data.txt > b.txt");
34         cout << "test " << i << " :";
35         if (system("diff a.txt b.txt")) {
36             cout << "WA" << '\n';
37             break;
38         } else cout << "AC" << '\n';
39     }
40     return 0;
41 }
```

常用数据生成方式

```
1  #include <iostream>
2  #include <cstdlib> // rand(), srand()
3  #include <ctime> // time()
4  #include <set>
```

```

5  #include <vector>
6  #include <algorithm> // shuffle
7  #include <utility> // pair
8
9  // 随机打乱序列 random_shuffle(sequence.begin(), sequence.end());
10
11 int random(int n) { // 返回0~n-1之间的随机整数
12     cout << rand() % n << '\n';
13 }
14
15 void RandomArray() { // 随机生成长度为n的绝对值在1e9之内的整数序列
16     int n = random(1e5) + 1;
17     int m = 1e9;
18     for (int i = 1; i <= n; i++) {
19         cout << random(2 * m + 1) - m << '\n';
20     }
21 }
22
23 void Intervals() { // 随机生成 m个[1,n]的子区间
24     int m = 10, n = 100;
25     for (int i = 1; i <= m; i++) {
26         int l = random(n) + 1;
27         int r = random(n) + 1;
28         if (l > r) swap(l, r);
29         cout << l << " " << r << '\n';
30     }
31 }
32
33 void generateTree() { // 随机生成一棵n个点的树，用n个点n-1条边的无向图的形式输出
34     int n = 10;
35     for (int i = 2; i <= n; i++) { // 从2 ~ n之间的每个点i向 1 ~ i-1之间的点随机连一条边
36         int fa = random(i - 1) + 1;
37         int val = random(1e9) + 1;
38         cout << fa << " " << i << " " << val << '\n';
39     }
40 }
41
42 void generateGraph() { // 随机生成一张n个点m条边的无向图，图中不存在重边、自环
43     int n = 10, m = 6;
44     set<pair<int, int>> edges; // 防止重边
45     for (int i = 1; i <= n; i++) { // 先生成一棵树，保证连通
46         int fa = random(i - 1) + 1;
47         edges.insert({ fa, i + 1 });
48         edges.insert({ i + 1, fa });
49     }
50     while (edges.size() < m) { // 再生成剩余的 m-n+1 条边
51         int x = random(n) + 1;
52         int y = random(n) + 1;
53         if (x != y) {
54             edges.insert({ x, y });
55             edges.insert({ y, x });
56         }
57     }
58     // Shuffling and outputting
59     vector<pair<int, int>> Edges(edges.begin(), edges.end());
60     random_shuffle(Edges.begin(), Edges.end());

```

```

61     for (auto& edge : Edges) {
62         cout << edge.first << " " << edge.second << '\n';
63     }
64 }
65
66 // 生成一个随机字符串, 包含大小写字母、数字和问号
67 string String(int length) {
68     const string characters =
69 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789?";
70     random_device rd; // 随机设备
71     mt19937 gen(rd()); // 使用Mersenne Twister算法
72     uniform_int_distribution<> dis(0, characters.size() - 1); // 定义一个分布
73
74     string randomString;
75     for (int i = 0; i < length; ++i) {
76         randomString += characters[dis(gen)]; // 从字符集随机选择字符
77     }
78
79     return randomString;
80 }
81
82 vector<int> generate_shuffled_permutation(int n) {
83     vector<int> a(n);
84     iota(a.begin(), a.end(), 1); // 生成 [1, 2, ..., n]
85     mt19937 rng(seed);           // 可选种子, 默认使用当前时间
86     shuffle(a.begin(), a.end(), rng);
87     return a;
88 }
89
90 int main() {
91     srand(time(0));
92     /*随机生成*/
93     return 0;
94 }

```

离散化

```

1  inline vi disc(const vi& a)
2  {
3      vi v(a);
4      sort(v.begin(), v.end());
5      v.erase(unique(v.begin(), v.end()), v.end());
6      vi res(a.size());
7      for (int i = 0; i < a.size(); i++)
8          res[i] = lower_bound(v.begin(), v.end(), a[i]) - v.begin();
9      return res;
10 }

```

莫队

最优大小为 $n \cdot m^{-0.5}$

普通莫队

每次先更新右边界，避免出现l大于r的情况

```
1  const int N = 3e5;
2  int n, m, len, res = 0;
3  int w[N], cnt[N], ans[N];
4
5  struct Query {
6      int qid, l, r;
7  } q[N];
8
9  int get(int i) {
10     return i / len;
11 }
12
13 void add(int i) {
14     if (!cnt[w[i]]) ++res;
15     ++cnt[w[i]];
16 }
17
18 void del(int i) {
19     --cnt[w[i]];
20     if (!cnt[w[i]]) --res;
21 }
22
23 bool cmp(const Query &a, const Query &b) {
24     int la = get(a.l), lb = get(b.l);
25     if (la != lb) return la < lb;
26     return la & 1 ? a.r < b.r : a.r > b.r; // 奇偶区块不同方向优化
27 }
28
29 inline void Zlin() {
30     cin >> n >> m;
31     for (int i = 1; i <= n; i++) cin >> w[i]; // 读入数组
32
33     for (int i = 1; i <= m; i++) {
34         int l, r;
35         cin >> l >> r;
36         q[i] = {i, l, r}; // 记录查询
37     }
38
39     len = sqrt(n) + 1; // 以 sqrt(n) 为块的大小
40     sort(q + 1, q + m + 1, cmp); // 根据块编号和右端点排序
41
42     res = 0;
43     for (int i = 1, l = 1, r = 0; i <= m; i++) {
44         // 当前查询范围是 [q[i].l, q[i].r]
45         while (r < q[i].r) add(++r);
46         while (r > q[i].r) del(r--);
47         while (l < q[i].l) del(l++);
48         while (l > q[i].l) add(--l);
49
50         ans[q[i].qid] = res; // 记录答案
51     }
52
53     for (int i = 1; i <= m; i++)
54         cout << ans[i] << '\n'; // 输出每个查询的结果
```

修改莫队

```

1  struct Query {
2      int qid, l, r, cid;
3  } q[N];
4
5  struct Change {
6      int p, x;
7  } c[N];
8
9  int cntq = 0, cntc = 0;
10
11 int n, m, len, res = 0;
12 int w[N], cnt[N], ans[N];
13
14 int get(int i) {
15     return i / len;
16 }
17
18 void add(int i) {
19     if (!cnt[i]) ++res;
20     ++cnt[i];
21 }
22
23 void del(int i) {
24     --cnt[i];
25     if (!cnt[i]) --res;
26 }
27
28 bool cmp(const Query &a, const Query &b) {
29     int la = get(a.l), ra = get(a.r);
30     int lb = get(b.l), rb = get(b.r);
31     if (la != lb) return la < lb;
32     if (ra != rb) return ra < rb;
33     return a.cid < b.cid;
34 }
35
36 inline void Zlin() {
37     cin >> n >> m;
38     for (int i = 1; i <= n; i++) cin >> w[i];
39     for (int i = 1; i <= m; i++) {
40         char op;
41         int l, r;
42         cin >> op >> l >> r;
43         if (op == 'Q') ++cntq, q[cntq] = {cntq, l, r, cntc};
44         else c[++cntc] = {l, r};
45     }
46     len = cbrt((double)n * max(1, cntc)) + 1;
47     sort(q + 1, q + cntq + 1, cmp);
48     res = 0;
49     for (int i = 1, l = 0, r = 0, cid = 0; i <= cntq; i++) {
50         while (r < q[i].r) add(w[++r]);
51         while (r > q[i].r) del(w[r--]);

```

```

52     while (l < q[i].l) del(w[l++]);
53     while (l > q[i].l) add(w[--l]);
54     while (cid < q[i].cid) {
55         ++cid;
56         if (c[cid].p >= q[i].l && c[cid].p <= q[i].r) {
57             del(w[c[cid].p]);
58             add(c[cid].x);
59         }
60         swap(w[c[cid].p], c[cid].x);
61     }
62     while (cid > q[i].cid) {
63         if (c[cid].p >= q[i].l && c[cid].p <= q[i].r) {
64             del(w[c[cid].p]);
65             add(c[cid].x);
66         }
67         swap(w[c[cid].p], c[cid].x);
68         --cid;
69     }
70     ans[q[i].qid] = res;
71 }
72 for (int i = 1; i <= cntq; i++)
73     cout << ans[i] << '\n';
74 }

```

树上莫队

通过欧拉序,将每一个点转换为start和end两个时间戳 注意:处理链的情况,如果两个没有祖先关系,记录一点的ed,另一个点的st,同时要加上他们LCA所产生的贡献,欧拉序是不包含LCA.st

哈希

随机数生成

std::mt19937

这里使用 chrono::steady_clock::now().time_since_epoch().count() 作为随机数种子, 这样每次运行代码时种子都会不同, 从而保证生成的随机数在不同次运行间不会重复。

```

1 // 初始化随机数生成器
2 mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
3
4 // 生成一个64位随机哈希值
5 ull generateRandomHash() {
6     // 返回生成的随机哈希值
7     return rng();
8 }

```

自然溢出

hash[i]=hash[i-1]*Base+idx(s[i])

数据结构要求ull, 不能用ll

单哈希

$\text{hash}[i] = (\text{hash}[i-1] * \text{Base} + \text{idx}(s[i])) \% \text{MOD}$

数据结构无要求，可以用ll

双哈希

Base, MOD不同，进行两遍hash

异或哈希

适合无序子集，类似树同构，求子集种类，找同分异构体

防止被卡操作前先使用mt19937生成随机数异或 然后进行值域离散 之后简单前缀和相加 或者 异或操作

```
1  const ull mask = mt19937_64(time(nullptr))();
2
3  inline ull shift(ull x) {
4      x ^= mask;
5      x ^= x << 23;
6      x ^= x >> 7;
7      x ^= x << 17;
8      x ^= mask;
9      return x;
10 }
```

洛谷P5043 对于无根树找同构可以先找到重心 最多可以出现两个所以用pair存两种重心的值 然后计算重心的hash值进行比较

```
1  constexpr int N = 55;
2  const ull mask = mt19937_64(time(nullptr))();
3
4  inline ull shift(ull x) {
5      x ^= mask;
6      x ^= x << 13;
7      x ^= x >> 7;
8      x ^= x << 17;
9      x ^= mask;
10     return x;
11 }
12
13 ull ha[N];
14 pair<ull, ull> val[N];
15 vi e[N], rt;
16 int n, m, siz[N];
17
18 inline void findrt(int u, int fa) {
19     siz[u] = 1;
20     int maxn = 0;
21     for (int v: e[u]) {
22         if (v == fa) continue;
23         findrt(v, u);
24         siz[u] += siz[v];
25         maxn = max(maxn, siz[v]);
26     }
```



```

26     }
27     maxn = max(maxn, n - siz[u]);
28     if (maxn <= n / 2) rt.push_back(u);
29 }
30
31 inline void dfs(int u, int fa) {
32     siz[u] = 1;
33     ha[u] = 1;
34     for (int v: e[u]) {
35         if (v == fa) continue;
36         dfs(v, u);
37         ha[u] += shift(ha[v]);
38     }
39 }
40
41 inline void Zlin(int id) {
42     cin >> n;
43     rt.clear();
44     for (int i = 0; i <= n; i++) e[i].clear();
45     for (int i = 1, x; i <= n; i++) {
46         cin >> x;
47         if (x) {
48             e[x].push_back(i);
49             e[i].push_back(x);
50         }
51     }
52     findrt(1, 1);
53     vll res;
54     dfs(rt[0], rt[0]);
55     val[id].first = ha[rt[0]];
56     if (rt.size() >= 2) {
57         dfs(rt[1], rt[1]);
58         val[id].second = ha[rt[1]];
59     } else val[id].second = 0;
60     if (val[id].first > val[id].second) swap(val[id].first, val[id].second);
61 }

```

文件读写操作

std::ios::

数据结构

基础

bitset

进行运算操作时间复杂度为 n/w n 为bitset容器长度， w 为运行机器编码长度

```
1 // 定义两个8位的bitset, 并通过字符串初始化
2 bitset<8> b1("1010"); // b1为 00001010
3 bitset<8> b2("1100"); // b2为 00001100
4
5 // 基本操作
6 cout << "b1: " << b1 << '\n'; // 输出b1的值
7 cout << "b2: " << b2 << '\n'; // 输出b2的值
8 cout << "b1 size: " << b1.size() << '\n'; // 输出b1的位数 (大小)
9 cout << "b1 count of 1s: " << b1.count() << '\n'; // 输出b1中1的个数
10 cout << "b1 any 1s: " << b1.any() << '\n'; // 检查b1中是否存在至少一个1
11 cout << "b1 all 1s: " << b1.all() << '\n'; // 检查b1的所有位是否都是1
12 cout << "b1 none 1s: " << b1.none() << '\n'; // 检查b1的所有位是否都是0
13
14 // 位操作
15 b1.set(); // 将b1的所有位都设置为1
16 cout << "b1 after set: " << b1 << '\n';
17 b1.reset(); // 将b1的所有位都重置为0
18 cout << "b1 after reset: " << b1 << '\n';
19 b1.flip(); // 将b1的所有位取反 (0变1, 1变0)
20 cout << "b1 after flip: " << b1 << '\n';
21 b1.set(2); // 将b1的第2位 (从0开始计数) 设置为1
22 cout << "b1 after setting bit 2: " << b1 << '\n';
23 cout << "b1 test bit 2: " << b1.test(2) << '\n'; // 测试b1的第2位是否为1
24
25 // 位运算
26 cout << "b1 & b2: " << (b1 & b2) << '\n'; // b1 和 b2 的按位与操作
27 cout << "b1 | b2: " << (b1 | b2) << '\n'; // b1 和 b2 的按位或操作
28 cout << "b1 ^ b2: " << (b1 ^ b2) << '\n'; // b1 和 b2 的按位异或操作
29 cout << "~b1: " << (~b1) << '\n'; // 对b1按位取反
30 cout << "b1 << 2: " << (b1 << 2) << '\n'; // 将b1左移2位
31 cout << "b1 >> 2: " << (b1 >> 2) << '\n'; // 将b1右移2位
32
33 // 单个位访问与修改
34 cout << "b1[2]: " << b1[2] << '\n'; // 访问b1的第2位的值
35 b1[3] = 1; // 将b1的第3位 (从0开始计数) 设置为1
36 cout << "b1 after modifying bit 3: " << b1 << '\n';
37
38 // 转换操作
39 cout << "b1 to string: " << b1.to_string() << '\n'; // 将b1转换为字符串形式
40 cout << "b1 to ulong: " << b1.to_ulong() << '\n'; // 将b1转换为无符号长整数
41
42
43 struct cmp {
44     bool operator()(const tuple<int, int, int> a, const tuple<int, int, int> b) const {
45         if (get<0>(a) != get<0>(b)) return get<0>(a) < get<0>(b);
46         if (get<1>(a) != get<1>(b)) return get<1>(a) < get<1>(b);
```

```

47         return get<2>(a) < get<2>(b);
48     };
49 };
50
51 set<tuple<int, int, int>, cmp> st;

```

priority_queue

不标注默认大根堆，pair内容先比较第一个元素，然后比较第二个元素

```

1 //升序队列
2 priority_queue <int,vector<int>,greater<int> > q;
3 //降序队列 默认降序
4 priority_queue <int,vector<int>,less<int> >q;

```

set/multiset

set 自动排序，去重

multiset 自动排序，不去重

```

1 s.begin(); // 返回set容器的第一个元素的地址（迭代器）
2 s.end(); // 返回set容器的最后一个元素的地址（迭代器）
3 s.rbegin(); // 返回逆序迭代器，指向容器元素最后一个位置
4 s.rend(); // 返回逆序迭代器，指向容器第一个元素前面的位置
5 s.clear(); // 删除set容器中的所有的元素,返回unsigned int类型O(N)
6 s.empty(); // 判断set容器是否为空
7 s.insert(x); // 插入一个元素 O(NlogN)
8 s.size(); // 返回当前set容器中的元素个数O(1)
9 s.erase(iterator); // 删除定位器iterator指向的值
10 s.erase(first, second) ; // 删除定位器first和second之间的值
11 s.erase(key_value); // 删除键值key_value的值O(NlogN) // multiset中是删除这个值的所有元素，要仅删除一个，只能用迭代器
12 s.find(x); //查找set中的某一元素，有则返回该元素对应的迭代器，无则返回结束迭代器
13 s.lower_bound(x); // 返回大于等于x的第一个元素的迭代器
14 s.upper_bound(x); // 返回大于x的第一个元素的迭代器 把这串东西放入代码框，解释加上注释

```

vector

```

1 vector<int> v; // 定义空的 int 类型 vector
2 vector<int> v1(10); // 初始化大小为 10 的 vector，默认值为 0
3 vector<int> v2(10, 5); // 初始化大小为 10 的 vector，每个元素值为 5
4 vector<int> v3 = {1, 2, 3, 4}; // 使用初始化列表创建 vector
5 sort(v.begin(), v.end()); // 升序排序
6 sort(v.begin(), v.end(), greater<int>()); // 降序排序
7 v.size(); // 返回 vector 的元素个数
8 v.capacity(); // 返回当前 vector 的容量（可容纳的元素个数）
9 v.empty(); // 检查 vector 是否为空，返回 true 或 false
10 v.front(); // 返回第一个元素
11 v.back(); // 返回最后一个元素
12 v.push_back(5); // 在 vector 尾部添加元素 5
13 v.insert(v.begin(), 10); // 在第一个位置插入元素 10
14 v.insert(v.begin() + 2, 7); // 在第 3 个位置插入元素 7
15 v.pop_back(); // 删除 vector 尾部的元素

```

```

16 v.erase(v.begin() + 1); // 删除第 2 个元素
17 v.erase(v.begin(), v.begin() + 3); // 删除前 3 个元素
18 v.clear(); // 清空所有元素

```

stack

```

1 stk.push(x); // 将 x 压入栈
2 stk.pop(); // 移除栈顶元素
3 stk.empty(); // 判断是否为空
4 stk.size(); // 获取栈中元素的个数
5 stk1.swap(stk2); // 交换 stk1 和 stk2 的内容

```

二进制基础操作

builtin_popcount(x) 统计 x 的二进制表示中 1 的个数（适用于 int 类型） `builtin_popcount(5) == 2 (101)`

builtin_popcountll(x) 统计 long long 类型的 1 数量 `builtin_popcountll(9) == 2 (1001)`

builtin_clz(x) 计算 x 的前导零个数（适用于 int） `builtin_clz(8) == 28 (000...1000)`

`__builtin_clzll(x)` long long 版

builtin_ctz(x) 计算 x 的后缀零个数 `builtin_ctz(8) == 3 (1000)`

`__builtin_ctzll(x)` long long 版

并查集(DSU)

普通并查集

```

1 inline int find(int x) { return f[x] == x ? x : f[x] = find(f[x]); }
2
3 inline int find(int x) {
4     if (f[x] == x)
5         return x;
6     find(f[x]);
7 }

```

种类并查集

根据不同种类开n倍的空间，每个空间存一种关系

每个不同的空间表示一种对立关系，维护一种对立情况

ST表

```

1  inline void ST_prewrite() {
2      for (int i = 1; i <= n; i++)
3          f[i][0] = a[i];
4      int t = log2(n) + 1;
5      for (int j = 1; j < t; j++)
6          for (int i = 1; i <= n - (1 << j) + 1; i++)
7              f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
8  }
9
10 inline int ST_query(int l, int r) {
11     int k = log(r - l + 1) / log(2);
12     return max(f[l][k], f[r - (1 << k) + 1][k]);
13 }

```

树状数组

普通版本

```

1  #define lowbit(x) (x & (-x))
2
3  inline void add(int i, int k) {
4      for (; i <= n; i += lowbit(i))
5          t[i] += k;
6      return;
7  }
8
9  inline int ask(int l, int r) {
10     int sum = 0;
11     for (; r; r -= lowbit(r))
12         sum += t[r];
13     --l;
14     for (; l; l -= lowbit(l))
15         sum -= t[l];
16     return sum;
17 }

```

结构体版本

```

1  struct Ftree
2  {
3  private:
4      vi t;
5
6  public:
7      void init(int n)
8      {
9          t.assign(n + 1, 0);
10     }
11
12     void upd(int i, int v)
13     {
14         while (i < t.size())
15             {

```

```

16         t[i] += v;
17         i += i & -i;
18     }
19 }
20
21 int qry1(int i)
22 {
23     int s = 0;
24     while (i > 0)
25     {
26         s += t[i];
27         i -= i & -i;
28     }
29     return s;
30 }
31
32 int qry2(int l, int r)
33 {
34     return qry1(r) - qry1(l - 1);
35 }
36 } t;

```

树

字典树

```

1  struct Node {
2      unordered_map<char, Node*> nxt; // 子节点
3      int cnt = 0; // 以当前节点为结尾的单词个数
4      int pre = 0; // 以当前节点为前缀的单词个数
5  };
6
7  class Trie {
8  private:
9      Node* root;
10
11 public:
12     Trie() { root = new Node(); }
13
14     // 插入单词
15     void ins(const string& s) {
16         Node* cur = root;
17         for (char c : s) {
18             if (!cur->nxt[c]) cur->nxt[c] = new Node();
19             cur = cur->nxt[c];
20             cur->pre++;
21         }
22         cur->cnt++;
23     }
24
25     // 查询单词是否存在
26     bool qry(const string& s) {
27         Node* cur = root;
28         for (char c : s) {
29             if (!cur->nxt[c]) return false;

```

```

30         cur = cur->nxt[c];
31     }
32     return cur->cnt > 0;
33 }
34
35 // 查询前缀是否存在
36 bool pre(const string& s) {
37     Node* cur = root;
38     for (char c : s) {
39         if (!cur->nxt[c]) return false;
40         cur = cur->nxt[c];
41     }
42     return cur->pre > 0;
43 }
44
45 // 删除单词
46 bool del(const string& s) {
47     return delHelper(root, s, 0);
48 }
49
50 // 查询前缀个数
51 int cntPre(const string& s) {
52     Node* cur = root;
53     for (char c : s) {
54         if (!cur->nxt[c]) return 0;
55         cur = cur->nxt[c];
56     }
57     return cur->pre;
58 }
59
60 private:
61     // 删除单词的辅助函数
62     bool delHelper(Node* cur, const string& s, int d) {
63         if (!cur) return false;
64         if (d == s.size()) {
65             if (cur->cnt > 0) {
66                 cur->cnt--;
67                 cur->pre--;
68                 return true;
69             }
70             return false;
71         }
72
73         char c = s[d];
74         if (delHelper(cur->nxt[c], s, d + 1)) {
75             cur->pre--;
76             if (cur->nxt[c]->pre == 0) {
77                 delete cur->nxt[c];
78                 cur->nxt.erase(c);
79             }
80             return true;
81         }
82         return false;
83     }
84 };

```

线段树

无懒标记 懒得写

带懒标记

区间加减

```
1 struct Tree {
2     int l, r, val, tag;
3 } t[N << 2];
4
5 // 建树
6 void build(int i, int l, int r) {
7     if (l == r) {
8         t[i].l = l, t[i].r = r;
9         t[i].val = w[l];
10        return;
11    }
12    int mid = l + r >> 1;
13    build(i << 1, l, mid);
14    build(i << 1 | 1, mid + 1, r);
15    t[i].l = l, t[i].r = r;
16    t[i].val = t[i << 1].val + t[i << 1 | 1].val;
17 }
18
19 //更新懒标记
20 void pushdown(int i) {
21     if (!t[i].tag) return;
22     t[i].val += t[i].tag * (t[i].r - t[i].l + 1);
23     if (t[i].l != t[i].r) {
24         t[i << 1].tag += t[i].tag;
25         t[i << 1 | 1].tag += t[i].tag;
26     }
27     t[i].tag = 0;
28 }
29
30 //区间修改
31 void modify(int i, int l, int r, int z) {
32     if (t[i].l > r || t[i].r < l) return;
33     pushdown(i);
34     if (t[i].l >= l && t[i].r <= r) {
35         t[i].tag += z;
36         pushdown(i);
37         return;
38     }
39     modify(i << 1, l, r, z);
40     modify(i << 1 | 1, l, r, z);
41     t[i].val = t[i << 1].val + t[i << 1 | 1].val;
42 }
43
44 //区间查询
45 int query(int i, int l, int r) {
46     if (t[i].l > r || t[i].r < l) return 0;
47     pushdown(i);
48     if (t[i].l >= l && t[i].r <= r) return t[i].val;
```



```

49     return query(i << 1, l, r) + query(i << 1 | 1, l, r);
50 }

```

线段树维护区间最大子串价值(单点修区间查)

```

1  struct STree
2  {
3  private:
4      struct node
5      {
6          int l, r;
7          ll val, tag;
8          ll pre, suf;
9
10         friend node operator +(const node& a, const node& b)
11         {
12             node res;
13             res.l = min(a.l, b.l);
14             res.r = max(a.r, b.r);
15             res.val = a.val + b.val;
16             res.pre = max(a.pre, a.val + b.pre);
17             res.suf = max(b.suf, b.val + a.suf);
18             res.tag = max({a.tag, b.tag, a.suf + b.pre});
19             return res;
20         }
21     };
22
23     vector<node> t;
24
25     void pushup(int i)
26     {
27         t[i] = t[i << 1] + t[i << 1 | 1];
28     }
29
30     void build(int i, int l, int r)
31     {
32         t[i].l = l, t[i].r = r;
33         if (l == r)
34         {
35             t[i].val = t[i].tag = t[i].pre = t[i].suf = 0;
36             return;
37         }
38         int mid = l + r >> 1;
39         build(i << 1, l, mid);
40         build(i << 1 | 1, mid + 1, r);
41         pushup(i);
42     }
43
44 public:
45     void init(int n)
46     {
47         t.assign(n << 2, {});
48         build(1, 1, n);
49     }
50 }

```

```

51 void update(int i, int id, int val)
52 {
53     int lx = t[i].l, rx = t[i].r;
54     if (rx < id || lx > id)
55         return;
56     if (lx == id && rx == id)
57     {
58         t[i].val += val;
59         t[i].tag += val;
60         t[i].pre += val;
61         t[i].suf += val;
62         return;
63     }
64     update(i << 1, id, val);
65     update(i << 1 | 1, id, val);
66     pushup(i);
67 }
68
69 node query(int i, int l, int r)
70 {
71     int lx = t[i].l, rx = t[i].r;
72     if (lx == l && rx == r)
73         return t[i];
74     int mid = lx + rx >> 1;
75     if (mid >= r)
76         return query(i << 1, l, r);
77     if (mid + 1 <= l)
78         return query(i << 1 | 1, l, r);
79     return query(i << 1, l, mid) + query(i << 1 | 1, mid + 1, r);
80 }
81 } t;

```

线段树优化建图

分别创建两颗线段树 第一棵从上往下连val=0的边 第二棵从下往上连val=0的边

对于一个边 $s \rightarrow [l, r]$ 等价于从第二棵树的 $[s, s]$ 节点连接线第一棵树对应 $[l, r]$ 的节点, 价值为这条边的价值, 个数最大 $\log(n)$

对于一个边 $[l, r] \rightarrow [s]$ 等价于从第二棵树的 $[l, r]$ 对应节点连接线第一棵树 $[s, s]$ 节点, 价值为这条边的价值, 个数最大 $\log(n)$

```

1 struct Dij_Tree {
2     struct Node {
3         int l, r;
4     } t[N << 2];
5
6     struct edge {
7         int to;
8         ll val;
9     };
10
11     vector<edge> e[N << 4];
12
13     int dif = 5e5;
14     int idx[N], vis[N << 4];
15     ll dis[N << 4];

```

```

16
17
18 void build(int i, int l, int r) {
19     // cout << i << ' ' << l << ' ' << r << endl;
20     dis[i] = dis[i + dif] = INF;
21     vis[i] = vis[i + dif] = 0;
22     t[i].l = l, t[i].r = r;
23     if (l == r) {
24         // 底边互相连接
25         e[i].push_back({i + dif, 0});
26         e[i + dif].push_back({i, 0});
27         idx[l] = i;
28         return;
29     }
30     // 第一个对应坐标
31     e[i].push_back({i << 1, 0});
32     e[i].push_back({i << 1 | 1, 0});
33     // 第二个对应坐标
34     e[(i << 1) + dif].push_back({i + dif, 0});
35     e[(i << 1 | 1) + dif].push_back({i + dif, 0});
36
37     int mid = l + r >> 1;
38     build(i << 1, l, mid);
39     build(i << 1 | 1, mid + 1, r);
40 }
41
42 // op 1 表示从点到线段 点到线段说明是第二棵树的点s 连接 第一棵树的对应node
43 // op 0 表示从线段到点 线段到点说明是第二棵树的对应node 连接 第一棵树的点s
44 void update(int i, int l, int r, int s, int val, int op) {
45     if (t[i].r < l || t[i].l > r) return;
46     if (t[i].l >= l && t[i].r <= r) {
47         if (op) e[idx[s] + dif].push_back({i, val});
48         else e[i + dif].push_back({idx[s], val});
49         return;
50     }
51     update(i << 1, l, r, s, val, op);
52     update(i << 1 | 1, l, r, s, val, op);
53 }
54
55 void dij(int s) {
56     priority_queue<pair<ll, int>, vector<pair<ll, int> >, greater<> > pq;
57     dis[idx[s]] = 0;
58     pq.emplace(dis[idx[s]], idx[s]);
59     while (!pq.empty()) {
60         int u = pq.top().second;
61         pq.pop();
62         if (vis[u]) continue;
63         vis[u] = 1;
64         // cout << (u >= dif ? u - dif : u) << ' ' << dis[u] << endl;
65         for (auto [v, val]: e[u]) {
66             if (dis[v] > dis[u] + val) {
67                 dis[v] = dis[u] + val;
68                 pq.emplace(dis[v], v);
69             }
70         }
71     }

```

```

72     }
73
74     ll query(int i) { return dis[idx[i]]; }
75 } t;

```

李超线段树

```

1  const ll N = 1e6 + 5;
2  const ll MOD = 998244353;
3  const ll inf = 0x7fffffff;
4  const double eps = 1e-12;
5
6  struct line {
7      db k, b; //斜率和与y轴
8      int l, r;
9      int tag;
10 } t[N << 2];
11
12 //计算某条线段在某一个横坐标的纵坐标值
13 db calc(line a, int pos) { return a.k * pos + a.b; }
14
15 //求两条线段交点的横坐标
16 int cross(line a, line b) { return floor((a.b - b.b) / (b.k - a.k)); }
17
18 void build(int root, int l, int r) {
19     t[root] = {0, 0, 1, 50000, 0};
20     if (l == r) return;
21     int mid = (l + r) >> 1;
22     build(root << 1, l, mid);
23     build(root << 1 | 1, mid + 1, r);
24 }
25
26 void modify(int root, int l, int r, line k) {
27     if (k.l > r || k.r < l) return;
28     if (k.l <= l && r <= k.r) {
29         if (!t[root].tag) {
30             // 1.这个区间内没有记录有过优势线段: 直接把这个区间的势线段修改为这条线段
31             t[root] = k;
32             t[root].tag = 1;
33         } else if (calc(k, l) - calc(t[root], l) > eps && calc(k, r) - calc(t[root], r) >
34 eps) {
35             // 2.新线段完全覆盖了之前记录的线段: 优势线段为新线段, 直接赋值替换
36             t[root] = k;
37         } else if (calc(k, l) - calc(t[root], l) > eps || calc(k, r) - calc(t[root], r) >
38 eps) {
39             // 3.区间内线段有交点的情况: 判断哪根线段为优势线段, 把区间记录的值给修改一下, 然后把短的那一半
40             // 递归处理
41             int mid = (l + r) >> 1; //取出区间的中点
42             // 与中点交点更高的一条直线作为优势线段
43             if (calc(k, mid) - calc(t[root], mid) > eps) swap(t[root], k);
44             if (mid - cross(k, t[root]) > eps) {
45                 // 交点在中点的左侧, 此时老线可能比被标记的优势线段高需要修改
46                 modify(root << 1, l, mid, k);
47             } else {
48                 // 交点在中点的右侧, 同理需要修改右侧区间的优势线段

```

```

46         modify(root << 1 | 1, mid + 1, r, k);
47     }
48 }
49 return;
50 }
51 int mid = (l + r) >> 1;
52 modify(root << 1, l, mid, k);
53 modify(root << 1 | 1, mid + 1, r, k);
54 }
55
56 db query(int root, int l, int r, int x) {
57     //由于是标记永久化，查询就比较类似于标记永久化的线段树
58     //那么就要从线段树一层层递归，直到递归到某个点
59     //每个区间的最优线段的交点取 max
60     if (l == r) return calc(t[root], x);
61     else {
62         int mid = (l + r) >> 1;
63         db ans = calc(t[root], x);
64         if (x <= mid) return max(ans, query(root << 1, l, mid, x));
65         else return max(ans, query(root << 1 | 1, mid + 1, r, x));
66     }
67 }

```

扫描线

```

1  struct Line {
2      double x1, x2, y;
3      int type; // +1 表示矩形底边, -1 表示矩形顶边
4      Line(double a, double b, double c, int d) : x1(a), x2(b), y(c), type(d) {}
5  };
6
7  bool cmp(const Line &l1, const Line &l2) {
8      return l1.y < l2.y;
9  }
10
11 struct Node {
12     int cnt; // 区间被覆盖的次数
13     double len; // 当前区间的总长度
14 };
15
16 vector<double> xs; // 保存去重后的 x 坐标
17 vector<Node> seg; // 线段树
18
19 void build(int p, int l, int r) {
20     seg[p].cnt = seg[p].len = 0;
21     if (l == r) return;
22     int mid = (l + r) / 2;
23     build(p * 2, l, mid);
24     build(p * 2 + 1, mid + 1, r);
25 }
26
27 void update(int p, int l, int r, int ql, int qr, int v) {
28     if (ql > r || qr < l) return;
29     if (ql <= l && r <= qr) {
30         seg[p].cnt += v;

```

```

31     } else {
32         int mid = (l + r) / 2;
33         update(p * 2, l, mid, ql, qr, v);
34         update(p * 2 + 1, mid + 1, r, ql, qr, v);
35     }
36
37     if (seg[p].cnt > 0) {
38         seg[p].len = xs[r + 1] - xs[l]; // 完全覆盖的区间
39     } else {
40         if (l == r) {
41             seg[p].len = 0;
42         } else {
43             seg[p].len = seg[p * 2].len + seg[p * 2 + 1].len; // 合并区间
44         }
45     }
46 }

```

主席树

求静态区间K小值

```

1  struct PStree
2  {
3  private:
4      struct node
5      {
6          int l, r;
7          int val;
8          node* ls = nullptr;
9          node* rs = nullptr;
10     };
11
12     vector<node*> t;
13
14     void pushup(node& t)
15     {
16         t.val = t.ls->val + t.rs->val;
17     }
18
19     void build(node& t, int l, int r)
20     {
21         t.l = l, t.r = r;
22         if (l == r)
23         {
24             t.val = 0;
25             return;
26         }
27         t.ls = new node();
28         t.rs = new node();
29         int mid = l + r >> 1;
30         build(*t.ls, l, mid);
31         build(*t.rs, mid + 1, r);
32         pushup(t);
33     }
34

```

```

35 void update(const node& bef, node& now, int k, int val)
36 {
37     now.l = bef.l, now.r = bef.r;
38     if (now.l == k && k == now.r)
39     {
40         now.val = bef.val + val;
41         return;
42     }
43     int mid = now.l + now.r >> 1;
44     if (k <= mid)
45     {
46         now.ls = new node();
47         update(*bef.ls, *now.ls, k, val);
48         now.rs = bef.rs;
49     }
50     else
51     {
52         now.ls = bef.ls;
53         now.rs = new node();
54         update(*bef.rs, *now.rs, k, val);
55     }
56     pushup(now);
57 }
58
59 int query(const node now, int k)
60 {
61     if (now.l == now.r)
62         return now.l;
63     if (now.ls->val >= k)
64         return query(*now.ls, k);
65     return query(*now.rs, k - now.ls->val);
66 }
67
68 int query1(const node bef, const node now, int k)
69 {
70     if (now.l == now.r)
71         return now.l;
72     if (now.ls->val - bef.ls->val >= k)
73         return query1(*bef.ls, *now.ls, k);
74     return query1(*bef.rs, *now.rs, k - now.ls->val + bef.ls->val);
75 }
76
77 public:
78 void init(int n, int val)
79 {
80     t.resize(n + 1, nullptr);
81     t[0] = new node();
82     build(*t[0], 1, val);
83 }
84
85 void upd(int bef, int now, int k, int val)
86 {
87     t[now] = new node();
88     update(*t[bef], *t[now], k, val);
89 }
90

```

```

91     int qry(int now, int k)
92     {
93         return query(*t[now], k);
94     }
95
96     int qryl(int bef, int now, int k)
97     {
98         return queryl(*t[bef - 1], *t[now], k);
99     }
100 } t;

```

求区间不重复个数

```

1  const int maxn = 30010;
2  struct node {
3      int l, r;
4      int v;
5  } T[maxn * 40];
6  int cnt = 0;
7  int a[maxn], past[1000100], root[maxn];
8
9  void update(int pos, int l, int r, int &cur, int pre, int val) {
10     cur = ++cnt;
11     T[cur] = T[pre];
12     T[cur].v += val;
13     if (l == r) return;
14     int mid = l + r >> 1;
15     if (pos <= mid)
16         update(pos, l, mid, T[cur].l, T[pre].l, val);
17     else
18         update(pos, mid + 1, r, T[cur].r, T[pre].r, val);
19 }
20
21 int query(int L, int R, int l, int r, int rt) {
22     if (L == l && r == R)
23         return T[rt].v;
24     int mid = l + r >> 1;
25     if (R <= mid)
26         return query(L, R, l, mid, T[rt].l);
27     else if (L >= mid + 1)
28         return query(L, R, mid + 1, r, T[rt].r);
29     else
30         return (query(L, mid, l, mid, T[rt].l) + query(mid + 1, R, mid + 1, r, T[rt].r));
31 }
32
33 void slove() {
34     ll n;
35     cin >> n;
36     for (int i = 1; i <= n; i++) {
37         cin >> a[i];
38     }
39     for (int i = 1; i <= n; i++) {
40         if (past[a[i]]) {
41             update(past[a[i]], 1, n, root[i], root[i - 1], -1);
42             update(i, 1, n, root[i], root[i], 1);

```



```

43         } else
44             update(i, 1, n, root[i], root[i - 1], 1);
45         past[a[i]] = i;
46     }
47     ll q;
48     cin >> q;
49     while (q--) {
50         int l, r;
51         cin >> l >> r;
52         cout << query(l, r, 1, n, root[r]) << "\n";
53     }
54 }

```

左偏树/可并堆

更适合处理合并工作，合并最坏复杂度 $\log n$

平衡树(Splay)

适合用来维护有序队列

```

1  struct Node {
2      int s[2], p, val, cnt, siz; // s左右儿子,p父节点
3
4      void init(int pl, int vall) {
5          p = pl, val = vall;
6          cnt = siz = 1;
7      }
8  } t[N];
9
10 int root = 0, tot = 0;
11
12 inline void pushup(int x) // 更新点x的大小
13 {
14     t[x].siz = t[t[x].s[0]].siz + t[t[x].s[1]].siz + t[x].cnt;
15 }
16
17 inline void rotate(int x) // 旋转x
18 {
19     int y = t[x].p, z = t[y].p;
20     int k = t[y].s[1] == x;
21     t[z].s[t[z].s[1] == y] = x;
22     t[x].p = z;
23     t[y].s[k] = t[x].s[k ^ 1];
24     t[t[x].s[k ^ 1]].p = y;
25     t[x].s[k ^ 1] = y;
26     t[y].p = x;
27     pushup(x), pushup(y);
28 }
29
30 inline void splay(int x, int k) {
31     while (t[x].p != k) {
32         int y = t[x].p, z = t[y].p;
33         if (z != k)
34             (t[y].s[0] == x) ^ (t[z].s[0] == y) ? rotate(x) : rotate(y);

```

```

35     rotate(x);
36 }
37 if (k == 0)
38     root = x;
39 }
40
41 inline void find(int val) // 找到权值等于val的点并把它转为根
42 {
43     int x = root;
44     while (t[x].s[val > t[x].val] && t[x].val != val)
45         x = t[x].s[val > t[x].val];
46     splay(x, 0);
47 }
48
49 inline int get_pre(int val) // 求前驱
50 {
51     find(val);
52     int x = root;
53     if (t[x].val < val)
54         return x;
55     x = t[x].s[0];
56     while (t[x].s[1])
57         x = t[x].s[1];
58     return x;
59 }
60
61 inline int get_suc(int val) // 求后继
62 {
63     find(val);
64     int x = root;
65     if (t[x].val > val)
66         return x;
67     x = t[x].s[1];
68     while (t[x].s[0])
69         x = t[x].s[0];
70     return x;
71 }
72
73 inline void del(int val) {
74     int pre = get_pre(val);
75     int suc = get_suc(val);
76     splay(pre, 0);
77     splay(suc, pre);
78     int del = t[suc].s[0];
79     if (t[del].cnt > 1)
80         --t[del].cnt, splay(del, 0);
81     else
82         t[suc].s[0] = 0, splay(suc, 0);
83 }
84
85 // 因为预处理插入了两个无穷大和无穷小的数，所以排名不需要+1
86 inline int get_rank(int val) // 查询val的排名
87 {
88     find(val);
89     if (t[root].val < val) // 如果val没有出现，要判断根节点和val的大小关系
90         return t[t[root].s[0]].siz + t[root].cnt;

```

```

91     return t[t[root].s[0]].siz;
92 }
93
94 // 因为插入了无穷大和无穷小，所以传入k时要+1,k为实际情况的k+1
95 inline int get_val(int k) // 查询排名为k的val
96 {
97     int x = root;
98     while (1) {
99         int y = t[x].s[0];
100         if (t[x].cnt + t[y].siz < k) {
101             k -= t[x].cnt + t[y].siz;
102             x = t[x].s[1];
103         } else {
104             if (t[y].siz >= k)
105                 x = t[x].s[0];
106             else
107                 break;
108         }
109     }
110     splay(x, 0);
111     return t[x].val;
112 }
113
114 inline void insert(int val) {
115     int x = root, p = 0;
116     while (x && t[x].val != val)
117         p = x, x = t[x].s[val > t[x].val];
118     if (x)
119         ++t[x].cnt;
120     else {
121         x = ++tot;
122         t[p].s[val > t[p].val] = x;
123         t[x].init(p, val);
124     }
125     splay(x, 0);
126 }
127

```

字典树

遇到相似题目可以选择离散化，在套入字典树

```

1 void insert(string s)//建立字典树
2 {
3     int p = 0;//根结点是0
4     for(auto it : s)
5     {
6         ll j;
7         if(it >= '0' && it <= '9') j = it - '0';
8         else if(it < 'a') j = it - 'A' + 10;
9         else j = it - 'a' + 26 + 10;//A从26开始
10        // 上面三行是映射字符，将字符变为数字好处理
11        if(!ch[p][j]) ch[p][j] = ++idx;//没有找到
12        p = ch[p][j];
13        cnt[p] ++;

```

```

14     }
15
16 }
17
18 ll query(string s)//查询函数
19 {
20     int p = 0;
21     for(auto it : s)
22     {
23         ll j;
24         if(it >= '0' && it <= '9') j = it - '0';//数字
25         else if(it < 'a') j = it - 'A' + 10;//A从10开始
26         else j = it - 'a' + 26 + 10;//小写字母
27         if(!ch[p][j]) return 0;//字节节点的编号不是0，如果是0则没有这条边
28         p = ch[p][j];
29     }
30     return cnt[p];
31 }

```

树链剖分

重链剖分

快速处理一条链上的查询和修改操作

```

1  const int N = 2e5 + 5;
2
3  struct Node {
4      int dep, fa, son, siz, val, top, dfn;
5  } tn[N];
6
7  struct edge {
8      int to, nxt;
9  } e[N];
10
11 int tot = 0, head[N];
12
13 void add(int u, int v) {
14     e[++tot] = {v, head[u]};
15     head[u] = tot;
16 }
17
18
19 // 预处理，找出树的所有重儿子和重链
20 void dfs1(int u, int f) {
21     tn[u].fa = f;
22     tn[u].dep = tn[f].dep + 1;
23     tn[u].siz = 1;
24     int tmp = -1; // 临时变量，用来存储结点u的重儿子
25     for (int i = head[u], v; i; i = e[i].nxt) {
26         v = e[i].to;
27         if (v == f) continue;
28         dfs1(v, u);
29         tn[u].siz += tn[v].siz;
30         if (tn[v].siz > tmp) { // 如果结点v.siz更大，更新u的重儿子为v

```

```

31         tn[u].son = v;
32         tmp = tn[v].siz;
33     }
34 }
35 }
36
37 int tim = 0, w[N]; // w用来存储对应dfn序下的树上结点val, tim为dfn计数器
38
39 void dfs2(int u, int top) {
40     tn[u].top = top;
41     tn[u].dfn = ++tim;
42     w[tim] = tn[u].val;
43     if (!tn[u].son) return; // 如果没有重儿子, 说明为叶节点
44     dfs2(tn[u].son, top); // 向下传递重链, 重链的top一样
45     for (int i = head[u], v; i; i = e[i].nxt) {
46         v = e[i].to;
47         if (v == tn[u].fa || v == tn[u].son) continue;
48         dfs2(v, v); // 轻链的top为他自身
49     }
50 }
51
52 // 详细见线段树-带懒标记区间修改
53 void modify(int i, int l, int r, int z) {
54     // 线段树区间修改, 省略, 线段树每一位对应的是dfn
55 }
56
57 int query(int i, int l, int r) {
58     // 线段树区间查询
59 }
60
61 // 修改结点u和他的子树, 因为dfn连续, 所以映射在线段树上是区间修改
62 void change_1(int u, int z) {
63     modify(1, tn[u].dfn, tn[u].dfn + tn[u].siz - 1, z);
64 }
65
66 // 同理
67 int query_1(int u) {
68     return query(1, tn[u].dfn, tn[u].dfn + tn[u].siz - 1);
69 }
70
71 // 修改一条链上的所有值, 重链上的dfn都是连续的
72 void change_2(int x, int y, int z) {
73     while (tn[x].top != tn[y].top) { // 如果他两不在同一条重链上, 找出top深度大的, 往上翻
74         if (tn[tn[x].top].dep < tn[tn[y].top].dep) swap(x, y);
75         modify(1, tn[tn[x].top].dfn, tn[x].dfn, z);
76         x = tn[tn[x].top].fa;
77     }
78     if (tn[x].dep > tn[y].dep) swap(x, y);
79     modify(1, tn[x].dfn, tn[y].dfn, z);
80 }
81
82 // 查询一条链上的所有值, 同理
83 int query_2(int x, int y) {
84     int res = 0;
85     while (tn[x].top != tn[y].top) { // 如果他两不在同一条重链上, 找出top深度大的, 往上翻
86         if (tn[tn[x].top].dep < tn[tn[y].top].dep) swap(x, y);

```

```

87         res += query(1, tn[tn[x].top].dfn, tn[x].dfn);
88         x = tn[tn[x].top].fa;
89     }
90     if (tn[x].dep > tn[y].dep) swap(x, y);
91     res += query(1, tn[x].dfn, tn[y].dfn);
92     return res;
93 }

```

长链剖分

1 |

实链剖分

1 |

分块

普通分块

```

1  inline void init()
2  {
3      int len = sqrt(n), tot = (n - 1) / len + 1;
4      for (int i = 1; i <= tot; i++)
5          l[i] = r[i - 1] + 1, r[i] = i * len;
6      r[tot] = n;
7      for (int i = 1; i <= tot; i++)
8          for (int j = l[i]; j <= r[i]; j++)
9              belong[j] = i;
10 }

```

时间分块

数学

基础

cbrt() 返回立方根

Ceil 向上取整

floor 向下取整

$$2ab = (a+b)^2 - a^2 - b^2$$

$$2ab + 2ac + 2bd = (a+b+c)^2 - a^2 - b^2 - c^2$$

多元同理

大数不能直接用sqrt，要自己用二分查找求值

叉积

AB*AC小于零说明AB能顺时针旋转到AC，大于零说明逆时针

pi = 3.14159265358979323846

cout保留几位小数

cout << fixed << setprecision(12) << ans << '\n';

裴蜀定理

对于任意整数 a, m 不全为0

$$a \cdot x + m \cdot y = \gcd(a, m) \quad (1)$$

极角排序

o表示原点

```
1 bool cmp(node a, node b) {  
2     if (cross(o, a, b) == 0) return a.x < b.x;  
3     return cross(o, a, b) > 0;  
4 }
```

扩展欧几里得公式

$$\begin{aligned} \gcd(a, b) &= \gcd(b, a \bmod b) \\ ax + by = \gcd(a, b) &\Rightarrow a \bmod b = a - k \cdot b \quad (k = \lfloor a/b \rfloor) \end{aligned} \quad (2)$$

递归到更小的子问题后，可以逐步构造出 x 和 y 。同时可以求一个值mod另一个值的逆元

```

1 // 扩展欧几里得算法, 返回 gcd(a, b), 并且计算出 x 和 y
2 // 使得 ax + by = gcd(a, b)
3 int exgcd(int a, int b, int &x, int &y) {
4     if (!b) {
5         x = 1, y = 0;
6         return a;
7     }
8     int x1, y1, gcd = exgcd(b, a % b, x1, y1);
9     x = y1, y = x1 - a / b * y1;
10    return gcd;
11 }

```

位运算

前缀和异或：从0~x连续异或的结论

$$f(x) = \begin{cases} x, & \text{if } x \bmod 4 = 0 \\ 1, & \text{if } x \bmod 4 = 1 \\ x + 1, & \text{if } x \bmod 4 = 2 \\ 0, & \text{if } x \bmod 4 = 3 \end{cases} \quad (3)$$

面积计算

三角形计算：

海伦公式：

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad (s = \frac{a+b+c}{2}) \quad (4)$$

素数

```

1 vi primes;
2 vector<bool> is_p;
3
4 inline void init(int n) {
5     is_p.assign(n + 1, true);
6     is_p[0] = is_p[1] = false;
7     for (int i = 2; i <= n; i++) {
8         if (is_p[i]) primes.push_back(i);
9         for (int j: primes) {
10             if (i * j > n) break;
11             is_p[i * j] = false;
12             if (i % j == 0) break;
13         }
14     }
15 }

```

FFT

快速计算多项式乘法/大数乘法

主体

```

1 const double PI = acos(-1.0);

```



```

2
3 struct Cp {
4     double r, i;
5
6     Cp(double _r = 0.0, double _i = 0.0) : r(_r), i(_i) {}
7
8     Cp operator+(const Cp &o) const {
9         return Cp(r + o.r, i + o.i);
10    }
11
12    Cp operator-(const Cp &o) const {
13        return Cp(r - o.r, i - o.i);
14    }
15
16    Cp operator*(const Cp &o) const {
17        return Cp(r * o.r - i * o.i, r * o.i + i * o.r);
18    }
19 };
20
21 // 进行 FFT 或 IFFT, d == 1 表示 FFT, d == -1 表示 IFFT
22 void fft(vector<Cp> &a, int n, int d) {
23     for (int p = 1, q = 0; p < n - 1; p++) {
24         for (int k = n >> 1; (q ^ k) < k; k >>= 1);
25         if (p < q) swap(a[p], a[q]);
26     }
27     for (int m = 2; m <= n; m <= 1) {
28         Cp wm(cos(2 * PI / m), sin(d * 2 * PI / m));
29         for (int p = 0; p < n; p += m) {
30             Cp w(1, 0);
31             for (int j = 0; j < m / 2; j++) {
32                 Cp u = a[p + j];
33                 Cp t = w * a[p + j + m / 2];
34                 a[p + j] = u + t;
35                 a[p + j + m / 2] = u - t;
36                 w = w * wm;
37             }
38         }
39     }
40     if (d == -1) {
41         for (int p = 0; p < n; p++) {
42             a[p].r /= n;
43             a[p].i /= n;
44         }
45     }
46 }

```

大数乘法

```

1 // 大数乘法主函数
2 vector<int> multiply(const vector<int>& A, const vector<int>& B) {
3     int n = 1;
4     while (n < A.size() + B.size()) n <= 1; // 找到大于等于 A.size() + B.size() 的最小 2 的幂
5     vector<Cp> a(n), b(n);
6
7     for (int p = 0; p < A.size(); p++) a[p] = Cp(A[p], 0);

```

```

8     for (int p = 0; p < B.size(); p++) b[p] = Cp(B[p], 0);
9
10    fft(a, n, 1);
11    fft(b, n, 1);
12
13    for (int p = 0; p < n; p++) a[p] = a[p] * b[p]; // 点乘
14    fft(a, n, -1);
15
16    vector<int> res(n);
17    for (int p = 0; p < n; p++) res[p] = int(a[p].r + 0.5); // 四舍五入取整
18    for (int p = 0; p < n - 1; p++) {
19        res[p + 1] += res[p] / 10; // 处理进位
20        res[p] %= 10;
21    }
22    while (res.size() > 1 && res.back() == 0) res.pop_back(); // 去掉前导0
23    return res;
24 }

```

多项式乘法

```

1 // 多项式乘法
2 vector<int> multiply(const vector<int> &A, const vector<int> &B) {
3     int n = 1;
4     while (n < A.size() + B.size()) n <= 1; // 取大于等于 A.size() + B.size() 的最小2的幂
5     vector<Cp> a(n), b(n);
6
7     for (int p = 0; p < A.size(); p++) a[p] = Cp(A[p], 0);
8     for (int p = 0; p < B.size(); p++) b[p] = Cp(B[p], 0);
9
10    // 进行 FFT 变换
11    fft(a, n, 1);
12    fft(b, n, 1);
13
14    // 点乘: 每个位置上的系数相乘
15    for (int p = 0; p < n; p++) a[p] = a[p] * b[p];
16
17    // 逆 FFT 变换
18    fft(a, n, -1);
19
20    // 提取结果并处理进位
21    vector<int> res(n);
22    for (int p = 0; p < n; p++)
23        res[p] = round(a[p].r);
24
25    return res;
26 }

```

NTT

主体

受模数的限制，数也比较大，但精度不易缺失

```

1 const int MOD = 998244353; // 质数模数 p

```

```

2  const int G = 3;           // 原根 g
3
4  // 快速幂计算 a^b % mod
5  int mod_pow(int a, int b, int mod) {
6      int res = 1;
7      while (b > 0) {
8          if (b % 2 == 1) res = 1LL * res * a % mod;
9          a = 1LL * a * a % mod;
10         b /= 2;
11     }
12     return res;
13 }
14
15 // NTT 核心函数
16 void ntt(vector<int> &a, int n, int inv) {
17     // 二进制反转置换
18     for (int i = 1, j = 0; i < n; i++) {
19         int bit = n >> 1;
20         while (j >= bit) {
21             j -= bit;
22             bit >>= 1;
23         }
24         j += bit;
25         if (i < j) swap(a[i], a[j]);
26     }
27
28     // 进行 NTT
29     for (int len = 2; len <= n; len <= 1) {
30         int wlen = inv == 1 ? mod_pow(G, (MOD - 1) / len, MOD) : mod_pow(mod_pow(G, (MOD - 1) / len, MOD), MOD - 2, MOD);
31         for (int i = 0; i < n; i += len) {
32             int w = 1;
33             for (int j = 0; j < len / 2; j++) {
34                 int u = a[i + j];
35                 int v = 1LL * a[i + j + len / 2] * w % MOD;
36                 a[i + j] = (u + v) % MOD;
37                 a[i + j + len / 2] = (u - v + MOD) % MOD;
38                 w = 1LL * w * wlen % MOD;
39             }
40         }
41     }
42
43     // 如果是逆变换, 需要除以 n (即乘以 n 的逆元)
44     if (inv == -1) {
45         int n_inv = mod_pow(n, MOD - 2, MOD);
46         for (int &x : a) x = 1LL * x * n_inv % MOD;
47     }
48 }

```

多项式求逆

```

1  // 多项式乘法
2  vector<int> poly_mult(const vector<int> &a, const vector<int> &b) {
3      int n = 1;
4      while (n < a.size() + b.size()) n <= 1;

```

```

5
6     vector<int> A(a.begin(), a.end()), B(b.begin(), b.end());
7     A.resize(n);
8     B.resize(n);
9
10    ntt(A, false);
11    ntt(B, false);
12
13    for (int i = 0; i < n; i++)
14        A[i] = (1LL * A[i] * B[i]) % MOD;
15
16    ntt(A, true);
17
18    return A;
19 }
20
21 // 多项式求逆
22 vector<int> poly_inv(const vector<int> &a) {
23     int n = a.size();
24     vector<int> res(1, pow_mod(a[0], MOD - 2)); // 初始逆多项式为 a[0] 的逆元
25
26     for (int len = 1; len < n; len *= 2) {
27         vector<int> temp(res.begin(), res.end());
28         temp.resize(2 * len);
29         vector<int> mult = poly_mult(temp, a);
30         for (int i = 0; i < len; i++) {
31             res.push_back((2LL * res[i] - mult[i] + MOD) % MOD); // 更新逆多项式
32         }
33     }
34     return res;
35 }

```

如何求原根

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  // 快速幂
6  long long mpow(long long b, long long e, long long m) {
7      long long r = 1;
8      while (e) {
9          if (e & 1) r = (r * b) % m;
10         b = (b * b) % m;
11         e >>= 1;
12     }
13     return r;
14 }
15
16 // 查找原根
17 long long g_r(long long p) {
18     long long p1 = p - 1;
19     vector<long long> f;
20
21     // 找到 p-1 的质因数

```

```

22     for (long long i = 2; i * i <= p1; i++) {
23         if (p1 % i == 0) {
24             f.push_back(i);
25             while (p1 % i == 0) p1 /= i;
26         }
27     }
28     if (p1 > 1) f.push_back(p1);
29
30     // 寻找原根
31     for (long long g = 2; g < p; g++) {
32         bool is_r = true;
33         for (long long q : f) {
34             if (mpow(g, (p - 1) / q, p) == 1) {
35                 is_r = false;
36                 break;
37             }
38         }
39         if (is_r) return g;
40     }
41     return -1; // 如果没有找到
42 }
43
44 int main() {
45     long long p = 7; // 可以替换为任意素数
46     cout << "Primitive root of " << p << " is: " << g_r(p) << endl;
47     return 0;
48 }

```

计算几何

高斯面积计算公式

$$A = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) + (x_n y_1 - x_1 y_n) \right| \quad (5)$$

计算向量夹角

计算坐标系中两个线段之间的夹角

$$\cos \theta = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{|\mathbf{v}_1| |\mathbf{v}_2|}$$

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = v_{1x} \cdot v_{2x} + v_{1y} \cdot v_{2y} + v_{1z} \cdot v_{2z}$$

$$|\mathbf{v}_1| = \sqrt{v_{1x}^2 + v_{1y}^2 + v_{1z}^2}$$

$$|\mathbf{v}_2| = \sqrt{v_{2x}^2 + v_{2y}^2 + v_{2z}^2} \quad (6)$$

叉积

如果叉积为正 ($\mathbf{A} \times \mathbf{B} > 0$) : 表示向量 \mathbf{B} 在向量 \mathbf{A} 的逆时针方向。

如果叉积为负 ($\mathbf{A} \times \mathbf{B} < 0$) : 表示向量 \mathbf{B} 在向量 \mathbf{A} 的顺时针方向。

如果叉积为零 ($\mathbf{A} \times \mathbf{B} = 0$) : 表示向量 \mathbf{A} 和 \mathbf{B} 是共线的 (即它们在同一直线上)。

$$\mathbf{A} \times \mathbf{B} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} = (A_y B_z - A_z B_y)\mathbf{i} - (A_x B_z - A_z B_x)\mathbf{j} + (A_x B_y - A_y B_x)\mathbf{k} \quad (\text{三维坐标系}) \quad (7)$$

$$\mathbf{A} \times \mathbf{B} = A_x B_y - A_y B_x \quad (\text{二维坐标系})$$

构建凸包

```
1  struct P {
2      int x, y;
3
4      // 比较函数, 先按 x 排序, 若 x 相同则按 y 排序
5      bool operator<(const P &p) const {
6          return x < p.x || (x == p.x && y < p.y);
7      }
8  };
9
10 // 计算向量 cross product (AB × AC), 用于判断点的相对位置
11 int cross(const P &a, const P &b, const P &c) {
12     return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
13 }
14
15 // 求凸包
16 vector<P> convexHull(vector<P> &pts) {
17     int n = pts.size();
18     if (n < 3) return pts; // 点数小于3无法构成凸包
19
20     // 先对点集进行排序
21     sort(pts.begin(), pts.end());
22
23     vector<P> h;
24
25     // 构建下半凸包
26     for (int i = 0; i < n; ++i) {
27         while (h.size() >= 2 && cross(h[h.size() - 2], h.back(), pts[i]) <= 0) {
28             h.pop_back(); // 移除不满足凸包性质的点
29         }
30         h.push_back(pts[i]);
31     }
32
33     // 构建上半凸包
34     int t = h.size() + 1; // 记录下半部分点的个数
35     for (int i = n - 1; i >= 0; --i) {
36         while (h.size() >= t && cross(h[h.size() - 2], h.back(), pts[i]) <= 0) {
37             h.pop_back(); // 移除不满足凸包性质的点
38         }
39         h.push_back(pts[i]);
40     }
41
42     h.pop_back(); // 移除最后一个点, 因为它在上下两部分中都出现了
43     return h;
```

旋转卡壳

旋转卡壳，求凸包的直径，可以处理三点共线

```

1  struct P {
2      double x, y;
3  };
4
5  // 计算两点之间的欧几里得距离
6  double dist(const P &p1, const P &p2) {
7      return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
8  }
9
10 // 计算向量叉积
11 double cross(const P &o, const P &a, const P &b) {
12     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
13 }
14
15 // 使用旋转卡壳算法求凸包的直径（最远点对距离）
16 double rotCalipers(const vector<P> &h) {
17     int n = h.size();
18     if (n == 1) return 0.0;
19     if (n == 2) return dist(h[0], h[1]);
20
21     int k = 1;
22     double maxD = 0.0;
23     for (int i = 0; i < n; ++i) {
24         while (abs(cross(h[i], h[(i + 1) % n], h[(k + 1) % n])) > abs(cross(h[i], h[(i + 1) % n], h[k]))) {
25             k = (k + 1) % n;
26         }
27         maxD = max(maxD, dist(h[i], h[k]));
28         maxD = max(maxD, dist(h[(i + 1) % n], h[k]));
29     }
30     return maxD;
31 }

```

线性基

可以插入也可以删除

异或线性基

最后求出来k个答案，要注意0的情况，如果k!=n+1说明存在0

模版

```

1  struct LinearBasis {
2      ll basis[62];
3      bool zero;
4
5      LinearBasis() {

```

```

6      memset(basis, 0, sizeof(basis));
7      zero = false;
8  }
9
10     bool insert(ll x) {
11         for (int i = 60; ~i; i--) {
12             if (x >> i & 1) {
13                 if (!basis[i]) {
14                     basis[i] = x;
15                     return true;
16                 }
17                 x ^= basis[i];
18             }
19         }
20         zero = true;
21         return false;
22     }
23
24     // 查询异或最大值
25     ll query_max() {
26         ll res = 0;
27         for (int i = 60; ~i; i--) if ((res ^ basis[i]) > res) res ^= basis[i];
28         return res;
29     }
30
31     // 查询异或最小非 0 值
32     ll query_min() {
33         for (int i = 0; i <= 60; i++) if (basis[i]) return basis[i];
34         return 0; // 如果都是 0
35     }
36 } linear;

```

高斯消元法

```

1  inline void gauss() {
2      for (int i = 63; ~i && k <= n; i--) {
3          for (int j = k; j <= n; j++)
4              if (a[j] & (1ll << i)) {
5                  swap(a[j], a[k]);
6                  break;
7              }
8          if (!(a[k] & (1ll << i))) continue;
9          for (int j = 1; j <= n; j++)
10             if (j != k && (a[j] & (1ll << i))) a[j] ^= a[k];
11             ++k;
12     }
13 }

```

区间线性基

更新当前位置永远保证是最右一位

```

1  inline void insert(int x, int id) {
2      int t = id;
3      for (int i = 30; ~i; i--) {

```



```

4     p[id][i] = p[id - 1][i];
5     pos[id][i] = pos[id - 1][i];
6 }
7 for (int i = 30; ~i; i--) {
8     if (!(x & (1 << i))) continue;
9     if (!p[id][i]) {
10        p[id][i] = x;
11        pos[id][i] = t;
12        return;
13    } else if (pos[id][i] < t) {
14        swap(p[id][i], x);
15        swap(pos[id][i], t);
16    }
17    x ^= p[id][i];
18 }
19 }
20 // 最大值高位可能与地位冲突，要比较是否更大
21 int query_max(int l, int r)
22 {
23     int ans = 0;
24     for(int i = 30; ~i; i--)
25         if(pos[r][i] >= l && (ans ^ p[r][i]) > ans)
26             ans ^= p[r][i];
27     return ans;
28 }
29
30 int query_min(int l, int r)
31 {
32     for(int i = 0; i <= 30; i++)
33         if(pos[r][i] >= l && p[r][i])
34             return p[r][i];
35     return 0;
36 }

```

高精度

模版

```

1 struct BigInt
2 {
3     vi now; // 按位存储 低位在前 高位在后
4     bool tag = false; // 判断是否是负数
5
6     void init(string s)
7     {
8         int l = 0, r = s.size() - 1;
9         if (s[0] == '-')
10            tag = true, l = 1;
11        while (r >= l)
12            now.push_back(s[r--] - '0');
13        trim(now);
14    }
15
16    // 清除前导零
17    void trim(vi& a)

```

```

18     {
19         while (a.back() == 0)
20             a.pop_back();
21         if (a.empty())
22             a.push_back(0), tag = false;
23     }
24
25     // 比较绝对值大小
26     bool checkabs(const BigInt& a, const BigInt& b)
27     {
28         if (a.now.size() != b.now.size())
29             return a.now.size() > b.now.size();
30         for (int i = a.now.size() - 1; i >= 0; i--)
31             if (a.now[i] != b.now[i])
32                 return a.now[i] > b.now[i];
33         return true;
34     }
35
36     // 加法
37     BigInt add(const BigInt& a, const BigInt& b)
38     {
39         BigInt res;
40         res.tag = a.tag;
41         int now = 0;
42         for (int i = 0; i < max(a.now.size(), b.now.size()) || now; i++)
43         {
44             int sum = now;
45             if (i < a.now.size())
46                 sum += a.now[i];
47             if (i < b.now.size())
48                 sum += b.now[i];
49             res.now.push_back(sum % 10);
50             now = sum / 10;
51         }
52         trim(res.now);
53         return res;
54     }
55
56     // 减法
57     BigInt sub(const BigInt& a, const BigInt& b)
58     {
59         BigInt res;
60
61     }
62 };

```

加法

存储数据

lenc = max(lena, lenb), 字符串读取输入, 翻转存入数组

```

1  int a[N], b[N], c[N];
2  int lena, lenb, lenc;

```

相加操作

```

1 vector<int> add(vector<int>& A, vector<int>& B)
2 {
3     //如果B更大, 因为下面代码都是以第一个形参作为for结束条件, 所以要让大的是第一个形参
4     if (A.size() < B.size()) return add(B, A);
5
6     vector<int> C;
7     int t = 0; //用来判断是否进位
8     //注意这里是逆序的数从前往后加的
9     for (int i = 0; i < A.size(); ++i)
10    { //for循环是以大的数来作为循环结束条件的
11        t += A[i]; //for循环以A为结束条件, 这里不用格外判断
12        if (i < B.size()) t += B[i]; //因没以B为结束条件, 故这里要格外判断是否可以加
13        C.push_back(t % 10); //加出来的数要的是余数
14        t /= 10; //判断是否有进位
15    }
16    if (t) C.push_back(t); //有可能最后加完还有进位
17
18    return C;
19 }

```

减法

a - b

```

1 vector<int> sub(vector<int>& A, vector<int>& B)
2 { //利用cmp函数比较, 使大的数一定是A, 与for循环代码相符
3
4     vector<int> C;
5     int t = 0; //判断借位
6     for (int i = 0; i < A.size(); ++i)
7     {
8         t = A[i] - t; //每次都会减掉借位
9         if (i < B.size()) t -= B[i];
10        //关于(t+10)%10 (t是减出来的数)
11        //t若为正数(但<=9)其=t%10+10%10=t
12        //t若为负数, 正好可以借位+10然后取余数即可
13        C.push_back((t + 10) % 10);
14        if (t >= 0) t = 0;
15        else t = 1; //<0肯定有借位了
16    }
17    //因为两个数相减会导致有多余的0出现, 故去除前导0
18    //size()>1是因为可能真的相减出现0, 这种0不算前导0
19    while (C.size() > 1 && C.back() == 0) C.pop_back();
20
21    return C;
22 }

```

乘法

从低位到高位, 先累加乘积, 然后进位, 存余

```

1 vector<int> mul(vector<int>& A, int b)
2 {
3     vector<int> C;

```

```

4   for (int i = 0, t = 0; i < A.size() || t; ++i)
5   {
6       if (i < A.size()) t += A[i] * b; //加上t是因为上一次可能有乘出来的进位
7       C.push_back(t % 10);
8       t /= 10; //计算进位
9   }
10  //当b是0时, 会出现前导0
11  while (C.size() > 1 && C.back() == 0) C.pop_back();
12
13  return C;
14  }

```

除法

从高位到低位

大数a除以小数b, r保存余数

```

1  vector<int> div(vector<int>& A, int b, int& r)
2  {
3      vector<int> C;
4      for (int i = A.size() - 1; i >= 0; --i)
5      {
6          r = r * 10 + A[i];
7          C.push_back(r / b);
8          r %= b; //计算余数
9      }
10     //逆置:因为我们是正常求, 但最后是倒着读的, 且便于去除前导0
11     reverse(C.begin(), C.end());
12     while (C.size() > 1 && C.back() == 0) C.pop_back();
13
14     return C;
15 }

```

比较大小

```

1  //比较哪个数大, 注意这里的数是从倒序存的, 故后面的才是高位
2  bool cmp(vector<int>& A, vector<int>& B)
3  {
4      if (A.size() != B.size()) return A.size() > B.size();
5      for (int i = A.size() - 1; i >= 0; --i)
6          if (A[i] != B[i])
7              return A[i] > B[i]; //不等从高位开始比
8
9      return true; //相等
10 }

```

快速幂

快速求 a^n 的值

```

1 // 快速幂函数: 计算 a^b % mo
2 ll qpow(ll a, ll b) {
3     ll res = 1;
4     while (b) {
5         if (b & 1) res = res * a % mo;
6         a = a * a % mo;
7         b >>= 1;
8     }
9     return res;
10 }

```

GCD

利用更减相损术和builtin内置函数，二进制运算速度更快

```

1 int qGCD(int a, int b)
2 {
3     int az = __builtin_ctz(a), bz = __builtin_ctz(b); // 如果数据ll, 用函数ctzll
4     int z = min(az, bz), dif;
5     b >>= bz;
6     while (a)
7     {
8         a >>= az;
9         dif = abs(b - a);
10        az = __builtin_ctz(dif);
11        if (a < b)
12            b = a;
13        a = dif;
14    }
15    return b << z;
16 }

```

$gcd(x, y) = 1, x + y = n$ 求x, y对数, 欧拉函数

```

1 int phi(int n) {
2     int res = n;
3     for (int i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             res -= res / i;
6             while (n % i == 0) n /= i;
7         }
8     }
9     if (n > 1) res -= res / n;
10    return res;
11 }

```

EXGCD

求 $ax + by == gcd(a, b)$

```

1  ll exgcd(ll a, ll b, ll &x, ll &y) {
2      if (!b) {
3          x = 1, y = 0;
4          return a;
5      }
6      ll x1, y1;
7      ll g = exgcd(b, a % b, x1, y1);
8      x = y1;
9      y = x1 - a / b * y1;
10     return g;
11 }

```

$$A \cdot x \equiv B \pmod{M} \quad (8)$$

$$g = \gcd(A, M) \quad (9)$$

$$\frac{A}{g} \cdot x \equiv \frac{B}{g} \pmod{\frac{M}{g}} \quad (10)$$

$$\text{最小循环节 } t = \frac{M}{g} \quad (11)$$

逆元

费马定理

给定两个数a,p, p为质数, a^{p-2} 为a模p的乘法逆元

```

1  // 快速幂函数: 计算 a^b % mo
2  ll qpow(ll a, ll b) {
3      ll res = 1;
4      while (b) {
5          if (b & 1) res = res * a % mo;
6          a = a * a % mo;
7          b >>= 1;
8      }
9      return res;
10 }
11
12 // 费马小定理求逆元: a 的逆元 % mo
13 ll inv(ll a) {
14     return qpow(a, mo - 2);
15 }

```

递推逆元

递推逆元: 如果你需要在区间 $[1, n]$ 内计算逆元, 可以使用递推的方式

设 $inv[1] = 1$ 。对于 i 大于2小于n的区间

$$inv[i] = (mod - (mod/i) \times inv[mod\%i]) \pmod{mod} \quad (12)$$

预处理法,乘法逆元

适用范围：n, m在1e5以内，且取模的数mod为素数时
利用快速幂求逆元

```
1 inline void init() // 预处理, fac[]表示阶乘, inf[]表示阶乘的逆元
2 {
3     fac[0] = inf[0] = 1;
4     for (int i = 1; i <= N; i++) {
5         fac[i] = fac[i - 1] * i % mod;
6         inf[i] = inf[i - 1] * quick_pow(i, mod - 2) % mod;
7     }
8 }
```

组合数

基础

$$\sum_{i=0}^n \binom{n}{i} \cdot i = n \cdot 2^{n-1} \quad (13)$$

$$\sum_{i=0}^n \binom{n}{i} \cdot i^2 = n(n+1) \cdot 2^{n-2} \quad (14)$$

$$\sum_{i=0}^n \binom{n-i}{k} \cdot i = C(n+1, k+2) \quad (15)$$

直接定义公式法

组合数公式 $C(n, k)$ 的定义为：

$$C(n, k) = \frac{n!}{k!(n-k)!} \quad (16)$$

其中 $(n!)$ 表示 (n) 的阶乘。这个方法可以用递推或循环计算阶乘，然后利用公式求出组合数。

递推公式法 (Pascal's Triangle)

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad (17)$$

```
1 const int N = 1000; // 定义最大 N 值 适合N<5e3的情况
2 long long C[N+1][N+1];
3
4 void init_comb() {
5     for (int i = 0; i <= N; ++i) {
6         C[i][0] = C[i][i] = 1; // 边界条件
7         for (int j = 1; j < i; ++j) {
8             C[i][j] = C[i-1][j-1] + C[i-1][j]; // 递推公式
9         }
10    }
11 }
```

逆元法 (费马小定理)

对于模 (p) (质数) 的组合数计算，利用费马小定理可以高效求组合数。公式为：

$$C(n, k) = \frac{n!}{k!(n-k)!} \mod p \quad (18)$$

利用费马小定理求逆元：

$$a^{-1} \equiv a^{p-2} \mod p \quad (19)$$

这样可以通过预处理阶乘和逆元，快速求出组合数。

```

1  const int N = 100000; // 定义最大 N 值
2  const ll mo = 1e9 + 7;
3  ll fact[N + 1], inv[N + 1];
4
5  // 快速幂求 a^b % mo
6  ll qpw(ll a, ll b) {
7      long long res = 1;
8      while (b) {
9          if (b % 2 == 1) res = res * a % mo;
10         a = a * a % mo;
11         b /= 2;
12     }
13     return res;
14 }
15
16 // 预处理阶乘和逆元
17 void init_fact() {
18     fact[0] = inv[0] = 1;
19     for (int i = 1; i <= N; ++i) {
20         fact[i] = fact[i - 1] * i % mo;
21     }
22     inv[N] = qpw(fact[N], mo - 2); // 利用费马小定理求逆元
23     for (int i = N - 1; i >= 1; --i) {
24         inv[i] = inv[i + 1] * (i + 1) % mo;
25     }
26 }
27
28 // 快速求组合数
29 ll comb(int n, int k) {
30     if (k > n || k < 0) return 0;
31     return fact[n] * inv[k] % mo * inv[n - k] % mo;
32 }

```

逐项计算法

$$[C(n, k) = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}] \quad (20)$$

```

1  ll comb(int n, int k) { // 避免溢出
2      if (k > n) return 0;
3      long long res = 1;
4      for (int i = 1; i <= k; ++i) {
5          res = res * (n - i + 1) / i;
6      }
7      return res;
8  }

```


Lucas 定理

对于较大的 (n) 和 (k) ，在模 (p) 的情况下，可以使用 Lucas 定理计算组合数。当 (n) 和 (k) 非常大，但 (p) 是质数时，Lucas

定理是一种有效的求解方法。

将 (n) 和 (k) 分解成模 (p) 的系数来递归计算组合数：

$$[C(n, k) \bmod p = C(n \bmod p, k \bmod p) \times C(n/p, k/p) \bmod p] \quad (21)$$

```
1 ll C(ll x, ll y) { //暴力算组合数
2     if (x > y) return 0;
3     return ((jc[y] * inv[x]) % p * inv[y - x]) % p;
4 }
5
6 ll work(ll n, ll m) { //Lucas 定理
7     if (!n) return 1;
8     return (work(n / p, m / p) * C(n % p, m % p)) % p;
9 }
```

图论

基础

奇数完全图的欧拉路径等于他的所有边，欧拉路径要求图中奇数度的定点不超过二

圆方树

对于每一块个点双,建一个方点连接这个点双的所有圆点

```
1 void tarjan(int u) {
2     dfn[u] = low[u] = ++tot;
3     stk[++top] = u;
4     for (int v: e[u]) {
5         if (!dfn[v]) {
6             tarjan(v);
7             low[u] = min(low[u], low[v]);
8             if (low[v] >= dfn[u]) {
9                 int nx = ++Tree::n, vx;
10                do {
11                    vx = stk[top--];
12                    Tree::add(nx, vx);
13                } while (vx != v);
14                Tree::add(nx, u);
15            }
16        } else low[u] = min(low[u], dfn[v]);
17    }
18 }
```

哈密顿路径

状压DP

mask按位存经过的点

例如: mask=5 换成二进制位 101 说明节点2和0已经经过。 mask=10 换成二进制 1010 说明节点3和1已经经过

```
1 // 全局变量定义
2 int n, dp[1 << N][N]; // n: 节点数量, dp: 动态规划表
3 vi e[N]; // 邻接表
4
5 // 初始化函数
6 void init() {
7     // 将 dp 表初始化为 -inf, 表示未访问的状态
8     for (int i = 0; i < (1 << n); i++)
9         memset(dp[i], -inf, sizeof(dp[i])); // 每个状态都初始化为 -inf
10    for (int i = 0; i < n; i++)
11        e[i].clear(); // 清空邻接表
12 }
13
14 // 哈密顿路径计算函数
15 int hmd() {
16     // 从每个节点作为起点初始化
17     for (int i = 0; i < n; i++)
```

```

18     dp[1 << i][i] = 1; // 每个节点的状态设置为可访问，路径长度为1
19
20     // 遍历所有状态和节点
21     for (int mask = 1; mask < (1 << n); mask++) {
22         for (int u = 0; u < n; u++) {
23             if (dp[mask][u] == -inf) continue; // 如果状态不可达，跳过
24
25             // 遍历与节点 u 相邻的所有节点 v
26             for (int v: e[u]) {
27                 if (mask & (1 << v)) continue; // 如果 v 已访问，跳过
28                 int newMask = mask | (1 << v); // 更新状态，标记节点 v 为已访问
29                 dp[newMask][v] = max(dp[newMask][v], dp[mask][u] + 1); // 更新经过节点 v 的最大
路径长度
30             }
31         }
32     }
33
34     int res = 1; // 至少会有一个节点
35     // 查找经过所有节点的最大路径
36     for (int i = 0; i < n; i++)
37         res = max(res, dp[(1 << n) - 1][i]); // 更新最终结果
38     return res; // 返回最大路径长度
39 }

```

欧拉路径

无向图

存在欧拉通路的充要条件

非零度顶点是连通的

恰有 2 个奇度顶点

存在欧拉回路的充要条件

非零度顶点是连通的

顶点的度数都是偶数

有向图

存在欧拉通路的充要条件

非零度顶点是弱连通的

至多一个顶点的出度与入度之差为 1

至多一个顶点的入度与出度之差为 1

其他顶点的入度和出度相等

存在欧拉回路的充要条件

非零度顶点是强连通的

每个顶点的入度和出度相等

```

1 void add(int u, int v) {
2     e[u].push_back(v);
3     out[u]++, in[v]++;
4 }

```

```

5
6 bool check_eul() {
7     int l = 0, r = 0;
8     for (int i = 1; i <= n; i++) {
9         if (out[i] - in[i] == 1) l++;
10        else if (in[i] - out[i] == 1) r++;
11        else if (in[i] != out[i]) return false;
12    }
13    return (l == 0 && r == 0) || (l == 1 && r == 1);
14 }
15
16 void findEul(int start) {
17     stack<int> stk;
18     stk.push(start);
19     while (!stk.empty()) {
20         int u = stk.top();
21         if (!e[u].empty()) {
22             int v = e[u].back();
23             e[u].pop_back(); // 移除已访问的边
24             stk.push(v);
25         } else {
26             eul.push_back(u);
27             stk.pop();
28         }
29     }
30 }

```

Hierholzer 算法

贪心想，每次走到走不下去为止，那个点就为欧拉路径的顶点，把他放入ans中，走过一条边要把那条边删除

```

1 void dfs(int u) {
2     while (!e[u].empty()) {
3         int v = e[u].back();
4         e[u].pop_back();
5         if (!vis[u][v]) continue;
6         vis[u][v] = vis[v][u] = 0;
7         dfs(v);
8     }
9     ans.push_back(u);
10 }

```

找环思路

无向图找环

DFS

```

1 bool Dfs(int u) {
2     vis[u] = 1;
3     for (int i = head[u]; i; i = e[i].nxt) {
4         int v = e[i].to;
5         if (!vis[v]) {
6             f[v] = u;

```

```

7         if (Dfs(v)) return true;
8     } else if (v != f[u]) {
9         cc.push_back(v);
10        for (int x = u; x != v; x = f[x])
11            cc.push_back(x);
12        return true;
13    }
14 }
15 return false;
16 }

```

DSU

判断奇数环和偶数环

二分图染色法

有向如找环

计算每个点的链长,同时换上点长度为环的大小

```

1 inline void dfs(int x, int y) {
2     int xx = x + dx[mp[x][y]], yy = y + dy[mp[x][y]];
3     if (xx > n || xx < 1 || yy > m || yy < 1) return;
4     stk[++top] = {x, y};
5     instk[x][y] = vis[x][y] = 1;
6     if (instk[xx][yy]) {
7         cc.push_back({xx, yy});
8         pii now = {x, y};
9         do {
10            cc.push_back(now);
11            now = fa[now.first][now.second];
12        } while (now != make_pair(xx, yy));
13        for (auto [xs, ys]: cc) val[xs][ys] = cc.size();
14        cc.clear();
15    } else {
16        fa[xx][yy] = {x, y};
17        dfs(xx, yy);
18        if (val[x][y] == 1) val[x][y] = val[xx][yy] + 1;
19    }
20    instk[x][y] = 0;
21 }

```

最短路算法

Dijkstra

主体, 优先队列为小根堆

时间复杂度: $n\log^m$

```

1 inline void dijkstra() {
2     dis[s] = 0;

```

```

3     q.push({0, s});
4     while (!q.empty()) {
5         int x = q.top().pos;
6         q.pop();
7         if (vis[x])
8             continue;
9         vis[x] = 1;
10        for (int i = head[x]; i; i = a[i].next) {
11            int y = a[i].to;
12            if (dis[y] > dis[x] + a[i].w) {
13                dis[y] = dis[x] + a[i].w;
14                if (!vis[y])
15                    q.push({dis[y], y});
16            }
17        }
18    }
19 }

```

Bellman-Ford

堆优化版——SPFA

时间复杂度:最好 $O(m)$, 最坏 $O(nm)$, 菊花图的情况

```

1  inline void spfa() {
2      dis[s] = 0;
3      vis[s] = 1;
4      queue<int> q;
5      q.push(s);
6      int x;
7      while (!q.empty()) {
8          x = q.front();
9          q.pop();
10         vis[x] = 0;
11         for (int i = head[x]; i; i = e[i].next) {
12             int y = e[i].to, d = e[i].dis;
13             if (dis[y] > dis[x] + d) {
14                 dis[y] = dis[x] + d;
15                 if (!vis[y])
16                     vis[y] = 1, q.push(y);
17             }
18         }
19     }
20 }

```

Johnson

Johnson优化, 能有处理负权值和有负环的情况

时间复杂度: $n^2 \log m$

预处理:先给每条边新添加一条0边

```

1  for (int i = 1; i <= n; i++)
2      add_edge(0, i, 0);

```

然后利用SPFA来判断负环，同时创建h数组（等同于势能，处理负权值）

```
1  inline bool spfa(int s)
2  {
3      for (int i = 1; i <= n; i++)
4          h[i] = 63, vis[i] = 0;
5      h[s] = 0;
6      vis[s] = 1;
7      queue<int> q;
8      q.push(s);
9      int x;
10     while (!q.empty())
11     {
12         x = q.front();
13         q.pop();
14         vis[x] = 0;
15         for (int i = head[x]; i; i = e[i].next)
16         {
17             int y = e[i].to, d = e[i].w;
18             if (h[y] > h[x] + d)
19             {
20                 h[y] = h[x] + d;
21                 if (!vis[y])
22                 {
23                     ++num[y]; // 说明经过当前点，次数加1
24                     if (num[y] > n) // n为自己设定的上限，如果循环次数超过n，说明存在负环，直接
返回
25                         return false;
26                     vis[y] = 1, q.push(y);
27                 }
28             }
29         }
30     }
31     return true;
32 }
```

SPFA预处理完成后利用h数组更新权重

```
1  for (int j = 1; j <= n; j++)
2      for (int i = head[j]; i; i = e[i].next)
3          e[i].w = e[i].w + h[j] - h[e[i].to];
```

更新完成后可以保证权值全为正数，随后根据题意运行Dijkstra，最终输出答案要注意减去h数组差值

LCA/最近公共祖先

倍增

```
1  struct edge {
2      int to, nxt;
3  } e[N];
4
5  int tot = 0, head[N];
6
```

```

7 void add(int u, int v) {
8     e[++tot] = {v, head[u]};
9     head[u] = tot;
10 }
11
12 int dep[N], fa[N][22];
13
14 inline void dfs(int u, int f) {
15     dep[u] = dep[f] + 1;
16     fa[u][0] = f;
17     for (int i = 1; i <= 19; i++)
18         fa[u][i] = fa[fa[u][i - 1]][i - 1];
19     for (int i = head[u], c; i; i = e[i].nxt)
20         if (e[i].to != f)
21             dfs(e[i].to, u);
22 }
23
24 inline int lca(int u, int v) {
25     if (dep[u] < dep[v])
26         swap(u, v);
27     for (int i = 19; i >= 0; i--)
28         if (dep[fa[u][i]] >= dep[v])
29             u = fa[u][i]; // 让u, v处于同一层
30     if (u == v)
31         return u;
32     for (int i = 19; i >= 0; i--)
33         if (fa[u][i] != fa[v][i])
34             u = fa[u][i], v = fa[v][i]; // 返回祖先的下一层
35     return fa[u][0];
36 }

```

Tarjan算法

Tarjan缩点/有向图

```

1 vi e[N], E[N];
2 int in[N], out[N];
3
4 int dfn[N], low[N], cnt;
5 int stk[N], instk[N], top;
6 int scc[N], siz[N], num;
7
8 ll val[N], f2[N], ans2;
9 int n, m, a[N], f1[N], ans1;
10
11 void tarjan(int x) {
12     dfn[x] = low[x] = ++cnt;
13     stk[++top] = x, instk[x] = 1;
14     for (int y: E[x]) {
15         if (!dfn[y]) {
16             tarjan(y);
17             low[x] = min(low[x], low[y]);
18         } else if (instk[y]) {
19             low[x] = min(low[x], dfn[y]);
20         }
21     }
22 }

```



```

21     }
22     if (low[x] == dfn[x]) {
23         int y;
24         ++num;
25         do {
26             y = stk[top--];
27             instk[y] = 0;
28             scc[y] = num;
29             val[num] += a[y];
30             siz[num]++;
31         } while (x != y);
32     }
33 }
34
35 void build_new() {
36     for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i);
37     for (int x = 1; x <= n; x++)
38         for (int y: E[x])
39             if (scc[x] != scc[y])
40                 e[scc[x]].push_back(scc[y]);
41     for (int i = 1; i <= num; i++) {
42         sort(e[i].begin(), e[i].end());
43         e[i].erase(unique(e[i].begin(), e[i].end()), e[i].end());
44     }
45     for (int x = 1; x <= num; x++)
46         for (int y: e[x])
47             ++out[x], ++in[y];
48 }

```

Tarjan缩点/无向图

普通版

```

1  constexpr int N = 2e5 + 10;
2  vi e[N], tree[N];
3  int cnt = 0, comp_count = 0;
4  int dfn[N], low[N], fa[N], compID[N], vis[N], siz[N];
5  set<pii> bridges;
6
7  inline void tarjan(int u) {
8      vis[u] = true;
9      dfn[u] = low[u] = ++cnt;
10     for (int v: e[u]) {
11         if (!vis[v]) {
12             fa[v] = u;
13             tarjan(v);
14             low[u] = min(low[u], low[v]);
15             if (low[v] > dfn[u])
16                 bridges.insert({min(u, v), max(u, v)});
17         } else if (v != fa[u]) low[u] = min(low[u], dfn[v]);
18     }
19 }
20
21 inline void dfs(int u, int ID) {
22     compID[u] = ID;

```

```

23     for (int v: e[u])
24         if (compID[v] == -1 && bridges.find({min(u, v), max(u, v)}) == bridges.end())
25             dfs(v, ID);
26 }
27
28 inline void build(int n) {
29     fill(compID + 1, compID + n + 1, -1);
30
31     for (int i = 1; i <= n; i++)
32         if (compID[i] == -1)
33             dfs(i, ++comp_count);
34     for (auto it: bridges) {
35         int u = compID[it.first], v = compID[it.second];
36         tree[u].push_back(v);
37         tree[v].push_back(u);
38     }
39     for (int i = 1; i <= n; i++) siz[compID[i]]++;
40 }

```

类标准

```

1  class Brige
2  {
3  private:
4      int n;
5      vector<bool> vis;
6      vector<vi> e, tree;
7      set<pii> bridges;
8      vi parent, dfn, low, compID;
9      int tot = 0, comp_ID = 0;
10
11     inline void dfs(int u, int ID)
12     {
13         compID[u] = ID;
14         for (int v : e[u])
15             if (!compID[v] && bridges.find({min(u, v), max(u, v)}) != bridges.end())
16                 dfs(v, ID);
17     }
18
19     inline void tarjan(int u)
20     {
21         vis[u] = true;
22         dfn[u] = low[u] = ++tot;
23         for (int v : e[u])
24         {
25             if (!vis[v])
26             {
27                 parent[v] = u;
28                 tarjan(v);
29                 low[u] = min(low[u], low[v]);
30                 if (low[v] > dfn[u])
31                     bridges.insert({min(u, v), max(u, v)});
32             }
33             else if (v != parent[u]) low[u] = min(low[u], dfn[v]);
34         }

```

```

35     }
36
37 public:
38     Brige(vector<vi> ee, int nn): e(ee), n(nn)
39     {
40         vis.assign(n + 1, false);
41         parent.assign(n + 1, 0);
42         dfn.assign(n + 1, 0);
43         low.assign(n + 1, 0);
44         compID.assign(n + 1, -1);
45     }
46
47     inline void build(int n)
48     {
49         compID.assign(n + 1, -1);
50
51         for (int i = 1; i <= n; i++)
52             if (compID[i] == -1)
53                 dfs(i, ++comp_ID);
54         for (auto it : bridges)
55         {
56             int u = compID[it.first], v = compID[it.second];
57             tree[u].push_back(v);
58             tree[v].push_back(u);
59         }
60     }
61
62     inline void get(vector<vi>& ee, vi& compId)
63     {
64         ee = tree;
65         compId = compID;
66     }
67 };

```

标准tarjan

时间戳 dfn[x] 节点x第一次被访问的顺序

追溯值 low[x] 从x节点出发，能到的最早的时间戳

```

1  vi e[N];
2  int dfn[N], low[N], tot;
3  int stk[N], instk[N], top;
4  int scc[N], cnt;
5
6  inline void tarjan(int x) {
7  // 入x点, 盖时间戳, 入栈
8      dfn[x] = low[x] = ++tot;
9      stk[++top] = x, instk[x] = 1;
10     for (int y: e[x]) {
11         // 未访问
12         if (!dfn[y]) {
13             tarjan(y);
14             low[x] = min(low[x], low[y]);
15         } else if (instk[y])
16             low[x] = min(low[x], dfn[y]);

```

```

17     }
18     // x为强连通图的根,输出分量图
19     if (dfn[x] == low[x]) {
20         int y;
21         ++cnt;
22         do {
23             y = stk[top--];
24             instk[y] = 0;
25             scc[y] = cnt;
26         } while (y != x);
27     }
28 }

```

点双连通分量

基础性质:

- 1、除了一种比较特殊的点双，其他的点双都满足：任意两点间都存在至少两条点不重复路径。
- 2、图中任意一个割点都在至少两个点双中。
- 3、任意一个不是割点的点都只存在于一个点双中。

注意点：要在tarjan基础上加特判起点没有祖先的情况

```

1 inline void tarjan(int u, int fa) {
2     int child = 0;
3     dfn[u] = low[u] = ++tot;
4     for (int v: e[u]) {
5         if (!dfn[v]) {
6             ++child;
7             tarjan(v, u);
8             low[u] = min(low[u], low[v]);
9             if (fa != -1 && low[v] >= dfn[u]) cut[u] = 1;
10        } else if (v != fa) low[u] = min(low[u], dfn[v]);
11    }
12    if (fa == -1 && child >= 2) cut[u] = 1;
13 }

```

边双连通分量(DCC)

基础性质:

- 1、割边不属于任意边双，而其它非割边的边都属于且仅属于一个边双。
- 2、对于一个边双中的任意两个点，它们之间都有至少两条边不重复的路径。

```

1 int dfn[N], low[N], cnt, tot;
2 struct edge {
3     int u, v;
4 };
5 vector<edge> e;
6 vi h[N];
7 struct bridge {
8     int x, y;
9 } bri[N];
10

```

```

11 inline void add(int x, int y) {
12     e.push_back({x, y});
13     h[x].push_back(e.size() - 1);
14 }
15
16 inline void tarjan(int x, int in_edg) {
17     dfn[x] = low[x] = ++tot;
18     for (int i = 0; i < h[x].size(); i++) {
19         int j = h[x][i], y = e[j].v;
20         if (!dfn[y]) {
21             tarjan(y, j);
22             low[x] = min(low[x], low[y]);
23             if (low[y] > dfn[x]) //如果low值大于dfn值, 说明只能从x到y为割边
24                 bri[++cnt] = {x, y};
25         } else if (j != (in_edg ^ 1)) //判断是否为反边
26             low[x] = min(low[x], dfn[y]);
27     }
28 }

```

DCC - SCC

```

1 struct Tree {
2     struct edge {
3         int v, id;
4     };
5
6     vector<edge> e[N], g[N];
7     int eid = -1;
8     bool bridges[N];
9     int dfn[N], low[N], tick;
10
11     void adde(int u, int v) {
12         ++eid;
13         e[u].push_back({v, eid});
14         e[v].push_back({u, eid});
15     }
16
17     void tarjan(int u, int pid) {
18         dfn[u] = low[u] = ++tick;
19         for (auto [v, id]: e[u]) {
20             if (!dfn[v]) {
21                 tarjan(v, id);
22                 low[u] = min(low[u], low[v]);
23                 if (low[v] > dfn[u]) bridges[id] = true;
24             } else if (id != pid) low[u] = min(low[u], dfn[v]);
25         }
26     }
27
28     int siz[N], compID[N], cnum;
29
30     void dfs(int u, int cid) {
31         ++siz[cid];
32         compID[u] = cid;
33         for (auto [v, eid]: e[u]) {
34             if (compID[v] == -1 && !bridges[eid]) {

```

```

35         dfs(v, cid);
36     }
37 }
38 }
39
40 vi tree[N];
41
42 void build(int n) {
43     fill(dfn + 1, dfn + 1 + n, 0);
44     fill(low + 1, low + 1 + n, 0);
45     tick = 0;
46     for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i, -1);
47     fill(siz + 1, siz + 1 + n, 0);
48     fill(compID + 1, compID + 1 + n, -1);
49     cnum = 0;
50     for (int i = 1; i <= n; i++) if (compID[i] == -1) dfs(i, ++cnum);
51     for (int u = 1; u <= n; u++) {
52         for (auto [v, id]: e[u]) {
53             if (compID[u] != compID[v]) tree[compID[u]].push_back(compID[v]);
54             else g[u].push_back({v, id});
55         }
56     }
57 }
58
59 bool vis_c[N];
60 ll vis_g[N * 2];
61
62 void dfs1(int u) {
63     for (auto [v, id]: g[u]) {
64         if (vis_g[id]) continue;
65         vis_g[id] = 1ll * u * N + v;
66         dfs1(v);
67     }
68 }
69
70 void dcc_scc(int n) {
71     for (int i = 1; i <= n; i++) if (!vis_c[compID[i]]) dfs1(i), vis_c[compID[i]] =
true;
72 }
73
74 int fa[N];
75
76 void dfs2(int u, int f) {
77     fa[u] = f;
78     for (int v: tree[u]) {
79         if (v == f) continue;
80         dfs2(v, u);
81     }
82 }
83
84 int find_s() {
85     int id = 0;
86     for (int i = 1; i <= cnum; i++) if (siz[id] < siz[i]) id = i;
87     dfs2(id, 0);
88     return siz[id];
89 }

```

```

90
91     bool check(int u, int v, int tid) {
92         if (fa[compID[u]] == compID[v]) return true;
93         int ux = vis_g[tid] / N, vx = vis_g[tid] % N;
94         if (ux == u && vx == v) return true;
95         return false;
96     }
97 } t;

```

拓扑排序

有向无环图（DAG），可以判断有向图中是否有环

DFS算法

通过c数组来存放颜色，表示不同的状态

```

1  vector<int> e[N], tp; // e[N]类似邻接表，存放有向边，tp存拓扑排序
2  int c[N];             // 存放颜色，0表示为经过，1表示已经被经过，-1表示正在被经过
3  bool dfs(int u)
4  {
5      c[u] = -1;
6      for (auto v : e[u])
7      {
8          if (~c[v]) // 说明有环存在
9              return 0;
10         if (!c[v] && !dfs(v)) // 递归，说明v下面有环
11             return 0;
12     }
13     c[u] = 1;
14     tp.push_back(u);
15     return 1;
16 }
17 bool toposort()
18 {
19     memset(c, 0, sizeof c);
20     for (int i = 1; i <= n; i++) // 遍历每个点，如果颜色没被标记，进行搜索
21         if (!c[i] && !dfs(i))
22             return 0;
23     reverse(tp.begin(), tp.end());
24     return 1;
25 }

```

卡恩算法（Kahn）

通过队列来维护入度为0的集合

```

1  vector<int> e[N], tp; // e[N]类似邻接表，存放有向边，tp存拓扑排序
2  int din[N];           // 存放每个点的入度
3  bool toposort()
4  {

```

```

5  queue<int> q;
6  for (int i = 1; i <= n; i++)
7      if (!din[i])
8          q.push(i);
9  while (!q.empty())
10 {
11     int u = q.front();
12     q.pop();
13     tp.push_back(u);
14     for (auto v : e[u])
15         if (--din[v] == 0)
16             q.push(v);
17 }
18 return tp.size() == n;
19 }

```

最小生成树

稀疏图一般选择 prim

稠密图一般选择 Kruskal

Prim

```

1  struct edge
2  {
3      int v, w;
4  };
5  vector<edge> e[N];
6  int d[N], vis[N];
7  priority_queue<pair<int, int>> q;
8
9  bool prim(int s)
10 {
11     for (int i = 0; i <= n; i++)
12         d[i] = inf;
13     d[s] = 0;
14     q.push({0, s});
15     while (!q.empty())
16     {
17         int u = q.top().second;
18         q.pop();
19         if (vis[u])
20             continue;
21         vis[u] = 1;
22         ans += d[u];
23         ++cnt;
24         for (auto ed : e[u])
25         {
26             int v = ed.v, w = ed.w;
27             if (d[v] > w)
28             {
29                 d[v] = w;
30                 q.push({-d[v], v});
31             }
32         }
33     }
34 }

```



```

33     }
34     return cnt == n;
35 }

```

Kruskal

```

1  struct Edge {
2      int u, v; // 边的两个端点
3      int w;    // 边的权重
4
5      // 重载小于运算符以便于排序
6      bool operator<(const Edge &other) const {
7          return w < other.w;
8      }
9  };
10
11  int fa[N], siz[N];
12
13  // 并查集查找, 路径压缩
14  int find(int u) { return fa[u] == u ? u : fa[u] = find(fa[u]); }
15
16  // 并查集合并
17  void merge(int x, int y) {
18      int fx = find(x);
19      int fy = find(y);
20      if (fx != fy) {
21          // 按秩合并
22          if (siz[fx] < siz[fy])
23              fa[fx] = fy;
24          else if (siz[fx] > siz[fy])
25              fa[fy] = fx;
26          else {
27              fa[fy] = fx;
28              siz[fx]++;
29          }
30      }
31  }
32
33  // Kruskal 算法
34  int kruskal(int n, vector<Edge> &edges) {
35      // 初始化并查集
36      for (int i = 1; i <= n; i++) {
37          fa[i] = i;
38          siz[i] = 0;
39      }
40      sort(edges.begin(), edges.end()); // 按权重升序排序
41      int tot = 0; // 最小生成树的权重
42      for (const auto &e: edges) {
43          int u = e.u;
44          int v = e.v;
45          if (find(u) != find(v)) {
46              tot += e.w; // 加入边的权重
47              merge(u, v); // 合并两个集合
48          }
49      }

```

```

50     return tot; // 返回最小生成树的总权重
51 }

```

二分图

最大匹配问题

匈牙利算法

时间复杂度 $O(nm)$

```

1  vector<int> g[N]; // 邻接表
2  int mt[N];       // 存储匹配
3  bool vis[N];     // 访问标记
4
5  // 深度优先搜索 (DFS) 查找增广路径
6  bool dfs(int u) {
7      for (int v : adj[u]) {
8          if (!vis[v]) {
9              vis[v] = true;
10             // 如果 v 没有匹配或 v 的匹配点可以找到其他匹配
11             if (mt[v] == -1 || dfs(mt[v])) {
12                 mt[v] = u;
13                 return true;
14             }
15         }
16     }
17     return false;
18 }
19
20 // 匈牙利算法求二分图最大匹配
21 int hungarian(int n) {
22     memset(mt, -1, sizeof mt); // 初始化匹配数组, -1 表示没有匹配
23     int res = 0; // 匹配的数量
24     for (int u = 0; u < n; ++u) {
25         memset(vis, false, sizeof(vis)); // 每次查找增广路径时重置访问标记
26         if (dfs(u)) { // 如果找到增广路径, 匹配数加1
27             ++res;
28         }
29     }
30     return res;
31 }

```

Hopcroft-Karp算法

时间复杂度 $O(n^{0.5}m)$

```

1  int n, m; // n: 左侧顶点数, m: 右侧顶点数
2  vi mtl(N), mtr(N), dis(N); // mtl, mtr: 左侧和右侧的匹配情况 dis: 记录距离 (用于 BFS)
3  vector<vi> g(N); // 存储二分图的邻接表
4
5  bool bfs() {
6      queue<int> q;
7      for (int u = 1; u <= n; u++) {

```

```

8         if (mtl[u] == -1) { //初始化起点, 如果没有被匹配过, 距离为零, 放入队列
9             dis[u] = 0;
10            q.push(u);
11        } else { //如果有, 则赋值为inf
12            dis[u] = inf;
13        }
14    }
15    bool check = false;
16    while (!q.empty()) {
17        int u = q.front();
18        q.pop();
19        for (int v: g[u]) {
20            int vv = mtr[v]; //表示右边能到达的点的匹配点
21            if (vv == -1) { //如果为-1, 说明这个右边的点没有被匹配, 能直接使用
22                check = true;
23            } else if (dis[vv] == inf) { //如果不为-1, 说明他和vv匹配, 把vv放到队列中, 同时更新
24                dis[vv] = dis[u] + 1;
25                q.push(vv);
26            }
27        }
28    }
29    return check;
30 }
31
32 bool dfs(int u) {
33     for (int v: g[u]) {
34         int vv = mtr[v];
35         if (vv == -1 || (dis[vv] == dis[u] + 1 && dfs(vv))) {
36             mtl[u] = v;
37             mtr[v] = u;
38             return true;
39         }
40     }
41     dis[u] = inf; //重置距离
42     return false;
43 }
44
45 int HK() {
46     for (int i = 1; i <= n; i++) mtl[i] = -1;
47     for (int i = 1; i <= m; i++) mtr[i] = -1;
48     int mt = 0;
49     while (bfs()) // 分阶段寻找增广路径
50         for (int u = 1; u <= n; u++)
51             if (mtl[u] == -1 && dfs(u)) // 如果没被匹配过同时找到增广路径, 匹配数加1
52                 ++mt;
53     return mt;
54 }

```

网络流

最大流

V是节点数 E是边数

EK算法

时间复杂度 $O(VE^2)$

```
1 struct EK {
2     struct Edge {
3         int from, to, cap, flow;
4
5         Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {
6         }
7     };
8
9     int n, m; // n: 点数, m: 边数
10    vector<Edge> edges; // edges: 所有边的集合
11    vector<int> G[MAXN]; // G: 点 x -> x 的所有边在 edges 中的下标
12    int a[MAXN], p[MAXN];
13    // a: 点 x -> BFS 过程中最近接近点 x 的边给它的最大流
14    // p: 点 x -> BFS 过程中最近接近点 x 的边
15
16    void init(int n) {
17        for (int i = 0; i < n; i++) G[i].clear();
18        edges.clear();
19    }
20
21    void AddEdge(int from, int to, int cap) {
22        edges.push_back(Edge(from, to, cap, 0));
23        edges.push_back(Edge(to, from, 0, 0));
24        m = edges.size();
25        G[from].push_back(m - 2);
26        G[to].push_back(m - 1);
27    }
28
29    int Maxflow(int s, int t) {
30        int flow = 0;
31        for (;;) {
32            memset(a, 0, sizeof(a));
33            queue<int> Q;
34            Q.push(s);
35            a[s] = INF;
36            while (!Q.empty()) {
37                int x = Q.front();
38                Q.pop();
39                for (int i = 0; i < G[x].size(); i++) {
40                    // 遍历以 x 作为起点的边
41                    Edge &e = edges[G[x][i]];
42                    if (!a[e.to] && e.cap > e.flow) {
43                        p[e.to] = G[x][i]; // G[x][i] 是最近接近点 e.to 的边
44                        a[e.to] = min(a[x], e.cap - e.flow); // 最近接近点 e.to 的边赋给它的流
45                        Q.push(e.to);
46                    }
47                }
48                if (a[t]) break; // 如果汇点接受到了流, 就退出 BFS
49            }
50            if (!a[t]) break; // 如果汇点没有接受到流, 说明源点和汇点不在同一个连通分量上
51            for (int u = t; u != s; u = edges[p[u]].from) {
52                // 通过 u 追寻 BFS 过程中 s -> t 的路径
```

```

53         edges[p[u]].flow += a[t]; // 增加路径上边的 flow 值
54         edges[p[u] ^ 1].flow -= a[t]; // 减小反向路径的 flow 值
55     }
56     flow += a[t];
57 }
58 return flow;
59 }
60 } ek;

```

Dinic算法

在普通情况下，DINIC算法时间复杂度为 $O(V^2E)$

在二分图中，DINIC算法时间复杂度为 $O(V^{0.5}E)$

```

1  struct MF {
2      struct edge {
3          int v, nxt, cap, flow;
4      } e[N];
5
6      int fir[N], cnt = 0;
7
8      int n, S, T;
9      ll maxflow = 0;
10     int dep[N], cur[N];
11
12     void init() {
13         memset(fir, -1, sizeof fir);
14         cnt = 0;
15     }
16
17     void addedge(int u, int v, int w) {
18         e[cnt] = {v, fir[u], w, 0};
19         fir[u] = cnt++;
20         e[cnt] = {u, fir[v], 0, 0};
21         fir[v] = cnt++;
22     }
23
24     bool bfs() {
25         queue<int> q;
26         memset(dep, 0, sizeof(int) * (n + 1));
27
28         dep[S] = 1;
29         q.push(S);
30         while (q.size()) {
31             int u = q.front();
32             q.pop();
33             for (int i = fir[u]; ~i; i = e[i].nxt) {
34                 int v = e[i].v;
35                 if (!dep[v] && e[i].cap > e[i].flow) {
36                     dep[v] = dep[u] + 1;
37                     q.push(v);
38                 }
39             }
40         }
41         return dep[T];

```

```

42     }
43
44     int dfs(int u, int flow) {
45         if (u == T || !flow) return flow;
46
47         int ret = 0;
48         for (int &i = cur[u]; ~i; i = e[i].nxt) {
49             int v = e[i].v, d;
50             if (dep[v] == dep[u] + 1 && (d = dfs(v, min(flow - ret, e[i].cap - e[i].flow))))
51             {
52                 ret += d;
53                 e[i].flow += d;
54                 e[i ^ 1].flow -= d;
55                 if (ret == flow) return ret;
56             }
57         }
58         return ret;
59     }
60
61     void dinic() {
62         while (bfs()) {
63             memcpy(cur, fir, sizeof(int) * (n + 1));
64             maxflow += dfs(S, INF);
65         }
66     } mf;

```

费用流

EK算法改編

```

1  struct EK {
2      struct Edge {
3          int from, to, cap, flow, cost;
4
5          Edge(int u, int v, int c, int f, int co) : from(u), to(v), cap(c), flow(f), cost(co)
6      {
7          }
8      };
9
10     int n, m; // n: 点数, m: 边数
11     vector<Edge> edges; // edges: 所有边的集合
12     vector<int> G[MAXN]; // G: 点 x -> x 的所有边在 edges 中的下标
13     int dis[MAXN], a[MAXN], p[MAXN]; // dis: 计算最短路径的数组
14     bool inQueue[MAXN]; // 记录是否在队列中
15
16     void init(int n) {
17         for (int i = 0; i < n; i++) G[i].clear();
18         edges.clear();
19     }
20
21     // 添加边 (from -> to, capacity, cost)
22     void AddEdge(int from, int to, int cap, int cost) {
23         edges.push_back(Edge(from, to, cap, 0, cost));
24         edges.push_back(Edge(to, from, 0, 0, -cost)); // 反向边, 费用取反

```

```

24     m = edges.size();
25     G[from].push_back(m - 2);
26     G[to].push_back(m - 1);
27 }
28
29 // 使用 SPFA (Shortest Path Faster Algorithm) 找增广路径
30 bool SPFA(int s, int t) {
31     fill(dis, dis + n, INF);
32     memset(inQueue, 0, sizeof(inQueue));
33     queue<int> Q;
34     Q.push(s);
35     dis[s] = 0;
36     a[s] = INF;
37     inQueue[s] = true;
38     while (!Q.empty()) {
39         int u = Q.front();
40         Q.pop();
41         inQueue[u] = false;
42         for (int i: G[u]) {
43             Edge &e = edges[i];
44             if (e.cap > e.flow && dis[e.to] > dis[u] + e.cost) {
45                 dis[e.to] = dis[u] + e.cost;
46                 a[e.to] = min(a[u], e.cap - e.flow);
47                 p[e.to] = i;
48                 if (!inQueue[e.to]) {
49                     Q.push(e.to);
50                     inQueue[e.to] = true;
51                 }
52             }
53         }
54     }
55     return dis[t] != INF;
56 }
57
58 // 最大费用流计算
59 int MaxFlow(int s, int t) {
60     int flow = 0, cost = 0;
61     while (SPFA(s, t)) {
62         int f = a[t]; // 增广流量
63         flow += f;
64         cost += f * dis[t]; // 计算费用
65         // 更新流量和反向流量
66         for (int u = t; u != s; u = edges[p[u]].from) {
67             edges[p[u]].flow += f;
68             edges[p[u] ^ 1].flow -= f; // 反向边流量减去
69         }
70     }
71     return cost; // 返回最大费用
72 }
73 };

```

Dinic算法改编

```

1 struct MCMF {
2     static const int N = 5e3 + 5, M = 1e5 + 5;
3     int tot = 1, lnk[N], cur[N], ter[M], nxt[M], cap[M], cost[M];

```

```

4  ll dis[N], ret;
5  bool vis[N];
6
7  void init(int n) {
8      tot = 1;
9      memset(lnk, 0, sizeof(int) * (n + 2));
10     ret = 0;
11 }
12
13 void add(int u, int v, int w, int c) {
14     ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, cap[tot] = w, cost[tot] = c;
15 }
16
17 void addedge(int u, int v, int w, int c) { add(u, v, w, c), add(v, u, 0, -c); }
18
19 bool spfa(int s, int t) {
20     memset(dis, 0x3f, sizeof(dis));
21     memcpy(cur, lnk, sizeof(lnk));
22     queue<int> q;
23     q.push(s), dis[s] = 0, vis[s] = true;
24     while (!q.empty()) {
25         int u = q.front();
26         q.pop(), vis[u] = false;
27         for (int i = lnk[u]; i; i = nxt[i]) {
28             int v = ter[i];
29             if (cap[i] && dis[v] > dis[u] + cost[i]) {
30                 dis[v] = dis[u] + cost[i];
31                 if (!vis[v]) q.push(v), vis[v] = true;
32             }
33         }
34     }
35     return dis[t] != dis[N - 1];
36 }
37
38 int dfs(int u, int t, int flow) {
39     if (u == t) return flow;
40     vis[u] = true;
41     int ans = 0;
42     for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
43         int v = ter[i];
44         if (!vis[v] && cap[i] && dis[v] == dis[u] + cost[i]) {
45             int x = dfs(v, t, min(cap[i], flow - ans));
46             if (x) ret += x * cost[i], cap[i] -= x, cap[i ^ 1] += x, ans += x;
47         }
48     }
49     vis[u] = false;
50     return ans;
51 }
52
53 int dinic(int s, int t) {
54     int ans = 0;
55     while (spfa(s, t)) {
56         int x;
57         while ((x = dfs(s, t, INF)) ans += x;
58     }
59     return ans;

```



```
60     }
61 } mcmf;
```

上下界费用流

无源汇上下界

```
1  struct MCMF {
2      static const int MAXN = 10010;
3      static const int MAXE = 50010;
4      static const int INF = 0x3f3f3f3f;
5
6      struct Edge {
7          int v, nxt, cap, flow, cost;
8      } e[MAXE];
9
10     int fir[MAXN], cur[MAXN], cnt = 0;
11     int S, T, superS, superT;
12     int dis[MAXN], pre[MAXN];
13     bool vis[MAXN];
14     int demand[MAXN]; // 点的需求
15     long long maxflow = 0, mincost = 0;
16
17     void init(int n) {
18         memset(fir, -1, sizeof(int) * (n + 5));
19         memset(demand, 0, sizeof(int) * (n + 5));
20         cnt = 0;
21         maxflow = mincost = 0;
22     }
23
24     void addRawEdge(int u, int v, int cap, int cost) {
25         e[cnt] = {v, fir[u], cap, 0, cost};
26         fir[u] = cnt++;
27         e[cnt] = {u, fir[v], 0, 0, -cost};
28         fir[v] = cnt++;
29     }
30
31     // 添加带上下界的边: 下界 l, 上界 r, 花费 cost
32     void addEdge(int u, int v, int l, int r, int cost) {
33         // 维护真实花费
34         mincost += 1LL * l * cost;
35
36         // 加边容量 r - l
37         addRawEdge(u, v, r - l, cost);
38
39         // 调整点需求
40         demand[u] -= l;
41         demand[v] += l;
42     }
43
44     bool spfa() {
45         memset(dis, INF, sizeof dis);
46         memset(vis, 0, sizeof vis);
47         memcpy(cur, fir, sizeof fir);
48         std::queue<int> q;
```

```

49     dis[S] = 0;
50     vis[S] = true;
51     q.push(S);
52
53     while (!q.empty()) {
54         int u = q.front(); q.pop();
55         vis[u] = false;
56         for (int i = fir[u]; ~i; i = e[i].nxt) {
57             int v = e[i].v;
58             if (e[i].cap > e[i].flow && dis[v] > dis[u] + e[i].cost) {
59                 dis[v] = dis[u] + e[i].cost;
60                 pre[v] = i;
61                 if (!vis[v]) {
62                     vis[v] = true;
63                     q.push(v);
64                 }
65             }
66         }
67     }
68     return dis[T] != INF;
69 }
70
71 int dfs(int u, int flow) {
72     if (u == T || flow == 0) return flow;
73     vis[u] = true;
74     int ret = 0;
75     for (int &i = cur[u]; ~i && ret < flow; i = e[i].nxt) {
76         int v = e[i].v;
77         if (!vis[v] && dis[v] == dis[u] + e[i].cost && e[i].cap > e[i].flow) {
78             int d = dfs(v, std::min(flow - ret, e[i].cap - e[i].flow));
79             if (d) {
80                 e[i].flow += d;
81                 e[i ^ 1].flow -= d;
82                 ret += d;
83                 mincost += 1LL * d * e[i].cost;
84             }
85         }
86     }
87     vis[u] = false;
88     return ret;
89 }
90
91 bool dinic() {
92     bool has_flow = false;
93     while (spfa()) {
94         int flow;
95         memset(vis, 0, sizeof vis);
96         while ((flow = dfs(S, INF))) {
97             maxflow += flow;
98             has_flow = true;
99         }
100     }
101     return has_flow;
102 }
103
104 // 建立超级源汇

```

```

105     bool buildSuperGraph(int n) {
106         superS = n + 1, superT = n + 2;
107         S = superS, T = superT;
108         for (int i = 1; i <= n; ++i) {
109             if (demand[i] > 0) {
110                 addRawEdge(superS, i, demand[i], 0);
111             } else if (demand[i] < 0) {
112                 addRawEdge(i, superT, -demand[i], 0);
113             }
114         }
115
116         // 跑最大流判断是否所有需求都满足
117         int total_demand = 0;
118         for (int i = 1; i <= n; ++i) {
119             if (demand[i] > 0) total_demand += demand[i];
120         }
121         dinic();
122
123         return maxflow == total_demand;
124     }
125 };

```

字符串

AC自动机

```
1 // 回跳边:父节点回跳所指节点的儿子
2 // 转移边:当前节点回跳边所指节点的儿子
3 struct AC_auto
4 {
5     private:
6         struct node
7         {
8             int val = 0;
9             node* nex = nullptr;
10            node* next[26] = {nullptr};
11        };
12
13        static const int N = 1e6 + 5;
14        node pool[N];
15        int tot = 0;
16
17        node* alloc()
18        {
19            pool[tot] = node();
20            return &pool[tot++];
21        }
22
23        node* root;
24
25    public:
26        void init()
27        {
28            tot = 0;
29            root = alloc();
30        }
31
32        void ins(const string& s, int val)
33        {
34            node* now = root;
35            for (char it : s)
36            {
37                if (now->next[it - 'a'] == nullptr)
38                    now->next[it - 'a'] = alloc();
39                now = now->next[it - 'a'];
40            }
41            now->val += val;
42        }
43
44        void build()
45        {
46            queue<node*> q;
47            root->nex = root;
48            for (int i = 0; i < 26; i++)
49            {
50                if (root->next[i] != nullptr)
```

```

51         q.push(root->next[i]);
52     else
53         root->next[i] = root;
54     root->next[i]->nex = root;
55 }
56 while (!q.empty())
57 {
58     node* now = q.front();
59     q.pop();
60     for (int i = 0; i < 26; i++)
61     {
62         if (now->next[i] != nullptr)
63         {
64             now->next[i]->nex = now->nex->next[i];
65             q.push(now->next[i]);
66         }
67         else now->next[i] = now->nex->next[i];
68     }
69 }
70 }
71
72 int qry(const string& s)
73 {
74     int res = 0;
75     node* now = root;
76     for (char it : s)
77     {
78         now = now->next[it - 'a'];
79         for (node* cal = now; cal != root && ~cal->val; cal = cal->nex)
80             res += cal->val, cal->val = -1;
81     }
82     return res;
83 }
84 } t;

```

后缀自动机

```

1  /*
2   * 基础信息:
3   * 合法性: 子节点的最短串的最长后缀=父节点的最长串
4   * 节点的子串长度: 最长子串=len[i] 最短子串=len[i]-len[fa[i]]
5   * 节点的子串数量=len[i]-len[fa[i]]
6   * 子串的出现次数=cnt[i]
7   */
8  struct SAM
9  {
10 private:
11     int tot = 1, np = 1;
12     vi fa, len, cnt;
13     vector<unordered_map<char, int>> ch;
14
15     void extend(char c)
16     {
17         int p = np;
18         np = ++tot;

```

```

19     len[np] = len[p] + 1;
20     cnt[np] = 1;
21     // 从链接边向前遍历,更新旧点的链接边
22     while (p && !ch[p][c])
23     {
24         ch[p][c] = np;
25         p = fa[p];
26     }
27     // 更新链接边
28     if (!p)
29         fa[np] = 1; // 指向根节点,是新字符,直接创建
30     else
31     {
32         int q = ch[p][c];
33         if (len[q] == len[p] + 1)
34             fa[np] = q; // 相邻,合法,直接加边
35         else
36         {
37             // 不相邻,不合法,要求新建一个链接点,然后把p点之前的所有点的路径更新
38             int nq = ++tot;
39             len[nq] = len[p] + 1;
40             fa[nq] = fa[q];
41             fa[q] = nq;
42             fa[np] = nq;
43             while (p && ch[p][c])
44             {
45                 ch[p][c] = nq;
46                 p = fa[p];
47             }
48             // 将原先的转移边复制到nq上
49             ch[nq] = ch[q];
50         }
51     }
52 }
53
54 public:
55     // n不该是题目给的n,要足够大保证链接点够,建议为 n*2 大小
56     void init(string s)
57     {
58         int n = s.length() * 2;
59         tot = np = 1;
60         fa.assign(n + 1, 0);
61         len.assign(n + 1, 0);
62         cnt.assign(n + 1, 0);
63         // ch 存放链接边信息
64         ch.assign(n + 1, unordered_map<char, int>());
65
66         for (auto it : s)
67             extend(it);
68     }
69
70     ll get_count()
71     {
72         ll res = 0;
73         for (int i = 1; i <= tot; i++)
74             res += len[i] - len[fa[i]];

```

```

75         return res;
76     }
77 } sam;

```

Manacher 判断回文串

```

1  inline string get_new(const string& s)
2  {
3      string res = "#";
4      for (auto it : s)
5      {
6          res += it;
7          res += '#';
8      }
9      return res;
10 }
11
12 // res代表以i为中心的回文串的长度
13 inline vi work(const string& x)
14 {
15     string s = get_new(x);
16     int n = s.length();
17     vi res(n);
18     // 当前最长回文中间点所在位置
19     int c = 0;
20     for (int i = 0; i < n; i++)
21     {
22         int l = 2 * c - i;
23         if (i < c + res[c]) res[i] = min(res[l], c + res[c] - i);
24         while (i - res[i] - 1 >= 0 && i + res[i] + 1 < n && s[i - res[i] - 1] == s[i +
res[i] + 1]) ++res[i];
25         if (i + res[i] > c + res[c]) c = i;
26     }
27     return res;
28 }
29
30 // 判断是否为回文串, l r为未修改前字符串的坐标
31 bool ch(int l, int r, vi& p)
32 {
33     int ll = l * 2 + 1, rr = r * 2 + 1;
34     int mid = ll + rr >> 1;
35     return mid + p[mid] - 1 >= rr;
36 }

```

矩阵乘法求字符串匹配

```

1  vector<vll> mul(vector<vll> A, vector<vll> B) {
2      int n = A.size();
3      vector<vll> C(n, vll(n, 0));
4      for (int i = 0; i < n; ++i)
5          for (int j = 0; j < n; ++j)
6              for (int k = 0; k < n; ++k)
7                  C[i][j] = (C[i][j] + A[i][k] * B[k][j] % mo) % mo;
8      return C;

```

```

9  }
10
11  vector<vll> build(string S, char c) {
12      int n = S.size();
13      vector<vll> F(n + 1, vll(n + 1, 0));
14      for (int i = 0; i <= n; ++i) {
15          if (i < n && S[i] == c)
16              F[i][i + 1] = 1;
17          F[i][i] = 1;
18      }
19      return F;
20  }
21
22  ll cal(string S, string T) {
23      int n = S.size();
24      vector<vll> res(n + 1, vll(n + 1, 0));
25      for (int i = 0; i <= n; ++i) res[i][i] = 1;
26      for (char c: T) {
27          vector<vll> Fc = build(S, c);
28          res = mul(res, Fc);
29      }
30      return res[0][n];
31  }

```

KMP

蓝书P82

```

1  void getNext(string s, int len) {
2      next[0] = 0;
3      int k = 0; //k = next[0]
4      for (int i = 1; i < len; i++) {
5          while (k > 0 && s[i] != s[k]) k = next[k - 1]; //k = next[k-1]
6          if (s[i] == s[k]) k++;
7          next[i] = k; //next[j+1] = k+1 | next[j+1] = 0
8      }
9  }
10
11  //返回模式串T中字符串s第一次出现的位置下标，找不到则返回-1
12  int kmp(string T, string S) {
13      int len_T = T.length();
14      int len_S = S.length();
15      for (int i = 0, j = 0; i < len_T; i++) {
16          while (j > 0 && T[i] != S[j]) j = next[j - 1];
17          if (T[i] == S[j]) j++;
18          if (j == len_S) return i - len_S + 1;
19      }
20      return -1;
21  }
22
23  //返回模式串T中字符串s出现的次数，找不到则返回0
24  int kmp(string T, string S) {
25      int sum = 0;
26      int len_T = T.length();
27      int len_S = S.length();

```



```
28     for (int i = 0, j = 0; i < len_T; i++) {
29         while (j > 0 && T[i] != S[j]) j = next[j - 1];
30         if (T[i] == S[j]) j++;
31         if (j == len_S) {
32             sum++;
33             j = next[j - 1];
34         }
35     }
36     return sum;
37 }
```