COGS 1476 数三角形

算法1：枚举法

- 分别从$1 \sim n$中枚举三角形的三条边，判断能否构成三角形。
- 设三角形的三边为a, b, c，则判定条件为**两边之和大于第三边**，即$a+b>c$且$a+c>b$且$b+c>a$。
- 枚举时为了避免重复，也为了方便判定，可以按照三边**小中大**的顺序枚举，那么判定时只需要要求两条小边之和大于最大边即可。
- 时间复杂度$\mathrm{O}(N^3)$，预计得分25分。

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    freopen("TricountUVa.in", "r", stdin);
    freopen("TricountUVa.out", "w", stdout);
    int n;
    while(cin >> n && n >= 3)
    {
        long long cnt = 0;
        for(int a = 1; a <= n; ++a)
            for(int b = a + 1; b <= n; ++b)
                for(int c = b + 1; c <= n; ++c)
                    if(a + b > c) ++cnt;
        cout << cnt;
    }
    return 0;
}
```

算法2

- 假设先枚举最长边c，为了保证能构成三角形，次大边只能在$c/2+1 \sim c-1$中选。例如$c=100$时，如果次大边选了50，那么最小边只能选$1 \sim 49$，这些都无法构成三角形，次大边只能在$51 \sim 99$中选。
- 当次大边也确定时，最小边的可选范围也就确定了，例如最大边选了c，次大边选了$c-k$，那么最小边只能选$k+1 \sim c-(k+1)$。例如，最大边和次大边选了$100, 97$，那么最小边可以选$4 \sim 96$。
- 通过观察可以推导出，如果最大边选了c的方案数为：
  - 如果c为偶数，方案数为$(c-2) \times (c-2)/4$；
  - 如果c为奇数，方案数为$(c-3) \times (c-1)/4$。

- 那么枚举最长边c，累和方案数即可。
- 时间复杂度：$\mathrm{O}(N)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    freopen("TricountUVa.in", "r", stdin);
    freopen("TricountUVa.out", "w", stdout);
    int n;
    while(cin >> n && n >= 3)
    {
        long long cnt = 0;
        for(int c = 4; c <= n; ++c)
        {
            if(c % 2 == 0) cnt += (c - 2) * (c - 2) / 4;
            else cnt += (c - 3) * (c - 1) / 4;
        }
        cout << cnt;
    }
    return 0;
}
```

- 也可以直接推导出总方案数，请读者自行研究。

C3194.喷水装置

算法

- 若想浇灌整个草坪，那么某些喷头至少要覆盖到草坪的边缘。
- 对于每个喷头，求出它能覆盖的边缘坐标范围，那么每个喷头实际上对应一个区间。
- 问题转化为：有若干个区间，选择最少的区间使得整段能被覆盖，那么这就是区间覆盖问题。
- 将区间$[l,r]$，按照l从小到大，l相同时r从大到小排序，每一次都选择一个能覆盖到最右侧的区间。
- 时间复杂度：$\mathrm{O}(N\log N)$。

```cpp
// 区间覆盖问题
#include <bits/stdc++.h>
using namespace std;

const int N = 15010;
int n, m;
```

```cpp
double l, w;

struct Node
{
    double l, r;
} p[N];

bool cmp(const Node &a, const Node &b)
{
    return a.l < b.l || (a.l == b.l && a.r > b.r);
}

int work()
{
    m = 0;
    scanf("%d%lf%lf", &n, &l, &w);
    for(int i = 1; i <= n; ++i)
    {
        double x, r; scanf("%lf%lf", &x, &r);
        if(r < w / 2) continue;
        p[++m].l = x - sqrt(r * r - w * w / 4);
        p[m].r   = x + sqrt(r * r - w * w / 4);
    }
    sort(p + 1, p + m + 1, cmp);
    if(p[1].l > 0) return -1;  // 浇灌范围无法覆盖
    int ans = 0;
    double s = 0; // 当前浇灌的最右端
    int i = 1;  // 当前准备选择的喷头
    while(s < l)
    {
        double t = s;
        for(; i <= m && p[i].l <= t; ++i) s = max(s, p[i].r);
        if(s == t && s < l) return -1; // 如果不能更新并且浇灌范围无法覆盖
        ++ans;      // 新增一个喷头
    }
    return ans;
}

int main()
{
    freopen("sprinkler.in", "r", stdin);
    freopen("sprinkler.out", "w", stdout);
    int T; scanf("%d", &T);
    while(T--) printf("%d\n", work());
    return 0;
}
```

C2334.最小函数值

算法1

- 每个函数的函数值是递增的，但是它们彼此之间的函数值大小则无确定关系。
- 而只求最小的m个函数值，那么每个函数最多只需要求前m个函数值。
- 将求出的$n \times m$个函数值从小到大排序，取前m个即可。
- 时间复杂度：O($MN \log(MN)$)，空间复杂度：O(MN)，注意数据范围。

```cpp
#include<bits/stdc++.h>
using namespace std;

const int N = 1010;
const int M = 1010;

long long d[N * M];   // 所有函数值
int a[N], b[N], c[N];

long long f(int a, int b, int c, int x)
{
    return (long long)a * x * x + b * x + c;
}

int main()
{
    freopen("minval.in", "r", stdin);
    freopen("minval.out", "w", stdout);
    int n, m; cin >> n >> m;
    for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
    int tot = 0;      // 当前计算的函数值总数
    for(int i = 1; i <= n; ++i)
        for(int x = 1; x <= m; ++x)
            d[++tot] = f(a[i], b[i], c[i], x);
    sort(d + 1, d + tot + 1);
    for(int i = 1; i <= m; ++i) cout << d[i] << " ";
    return 0;
}
```

算法2

- 首先，考虑最小的函数值，必定是所有函数$x = 1$时的函数值的最小值， 设这个函数为f_k。
- 其次，第二小的函数值必定除了f_k之外的所有函数$x = 1$时的函数值和函数f_k当$x = 2$时的函数值的最小值，其他函数值依次类推。
- 那么每次只需要保留每个函数的一个函数值(最小的)即可，每次在这些函数值中找最小的就是答案。
- 将函数值放在最小堆中，每次取堆顶，每次计算的新的函数值放入堆中即可。
- 但是只放函数值是不够的，因为每一次还要知道最小的函数值是哪个函数，所以堆内需要同时存储函数值和函数编号。
- 时间复杂度：O($M \log N$)。

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 5e5 + 10;
int a[N], b[N], c[N];
int x[N];       // 每个函数当前的自变量值

long long f(int a, int b, int c, int x)
{
    return (long long)a * x * x + b * x + c;
}

int main()
{
    freopen("minval.in", "r", stdin);
    freopen("minval.out", "w", stdout);
    int n, m; cin >> n >> m;
    for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
    // 堆内需要存储函数值和函数编号
    priority_queue<pair<long long, int>, vector<pair<long long, int> >,
                   greater<pair<long long, int> > > pq;
    for(int i = 1; i <= n; ++i) // 先求每个函数x=1时的值
    {
        pq.push({f(a[i], b[i], c[i], 1), i});
        x[i] = 1;
    }
    for(int i = 1; i <= m; ++i)
    {
        pair<long long, int> t = pq.top(); pq.pop();  // 取堆顶
        cout << t.first << " ";
        int k = t.second;   // 当前函数编号
        pq.push({f(a[k], b[k], c[k], x[k] + 1), k}); // 计算下一个函数值
        x[k] = x[k] + 1;
    }
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 5e5 + 10;
int a[N], b[N], c[N];
int x[N];       // 每个函数当前的自变量值

long long f(int a, int b, int c, int x)
{
    return (long long)a * x * x + b * x + c;
}

struct Node
{
    long long y;
    int id;
    bool operator<(const Node &b) const
    {
        return y > b.y;
    }
};

int main()
{
    freopen("minval.in", "r", stdin);
    freopen("minval.out", "w", stdout);
    int n, m; cin >> n >> m;
    for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
    // 堆内需要存储函数值和函数编号
    priority_queue<Node> pq;
    for(int i = 1; i <= n; ++i) // 先求每个函数x=1时的值
    {
        pq.push({f(a[i], b[i], c[i], 1), i});
        x[i] = 1;
    }
    for(int i = 1; i <= m; ++i)
    {
        Node t = pq.top(); pq.pop();  // 取堆顶
        cout << t.y << " ";
        int k = t.id;   // 当前函数编号
        pq.push({f(a[k], b[k], c[k], x[k] + 1), k}); // 计算下一个函数值
        x[k] = x[k] + 1;
    }
    return 0;
}
```

C1214.括号匹配

子任务1

- 对于40%的数据，没有通配符，那么直接按照括号匹配算法即可。
- 括号的层数为左括号进栈后栈的大小，可以用`STL map`来维护括号层数数值出现的次数。

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 2e6 + 10;
char s[N];
int n = 0;
map<int, int> d;   // 深度记录

map<char, int> val =
{
    {'(', -1}, {'[', -2}, {'{', -3}, {'<', -4},
    {')', 1}, {']', 2}, {'}', 3}, {'>', 4}
};

bool check()
{
    stack<int> st;
    for(int i = 1; i <= n; ++i)
    {
        if(val[s[i]] <= -1) st.push(val[s[i]]), ++d[st.size()];
        else if(st.empty()) return false;
        else if(st.top() + val[s[i]] != 0) return false;
        else st.pop(), ++d[st.size()];
    }
    return st.empty();
}

int main()
{
    freopen("kakko.in", "r", stdin);
    freopen("kakko.out", "w", stdout);
    int T; scanf("%d", &T);
    while(T--)
    {
        d.clear();
        scanf("%s", s + 1);
        n = strlen(s + 1);
        if(check()) printf("TRUE %d %d\n", d.rbegin()->first, d.rbegin()->second);
        else puts("FALSE");
    }
    return 0;
```

```
}
```

算法

- 对于通配符，将左括号都转化为`/`及其数量，右括号都转化为`\`及其数量。
- 于是括号匹配变为包含普通括号、`/`、`\`的问题，不过`/`和`\`可能一次包含多个。
- 所以栈内除了要保存字符，还要保存字符出现的次数，括号的层数为站内所有次数之和。

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 2e6 + 10;
char t[N];
struct Node
{
    char c;
    int x;
} s[N]; // 合并相同的任意括号
int n = 0;
map<int, int> d;   // 深度记录

map<char, int> val =
{
    {'(', -1}, {'[', -2}, {'{', -3}, {'<', -4},
    {'/', -5}, {'#', -6}, {'@', -7}, {'?', -8},
    {')', 1}, {']', 2}, {'}', 3}, {'>', 4},
    {'\\', 5}, {'*', 6}, {'&', 7}, {'!', 8}
};

bool check()
{
    stack<Node> st;
    int dsum = 0;   // 当前深度之和
    for(int i = 1; i <= n; ++i)
    {
        if(val[s[i].c] <= -1) st.push(s[i]), dsum += s[i].x;
        else if(val[s[i].c] >= 1 && val[s[i].c] <= 4)
        {
            if(st.empty()) return false;
            if(st.top().c != '/' && val[st.top().c] + val[s[i].c] != 0) return false;
            ++d[dsum]; // 记录一次可能的最大深度
            Node p = st.top(); st.pop();
            --p.x; if(p.x) st.push(p);
            --dsum;
        }
        else if(s[i].c == '\\')
        {
            ++d[dsum]; // 记录一次可能的最大深度
```

```
            while(!st.empty() && s[i].x)
            {
                Node p = st.top(); st.pop();
                int y = min(p.x, s[i].x);
                p.x -= y, s[i].x -= y, dsum -= y;
                if(p.x) st.push(p);
            }
            if(s[i].x) return false;
        }
    }
    return st.empty();
}

int main()
{
    freopen("kakko.in", "r", stdin);
    freopen("kakko.out", "w", stdout);
    int T; scanf("%d", &T);
    while(T--)
    {
        d.clear();
        n = 0;
        scanf("%s", t + 1);
        for(int i = 1; t[i] != '\0'; ++i)
        {
            if(val[t[i]] >= -4 && val[t[i]] <= 4) s[++n] = { t[i], 1 };
            else if(val[t[i]] <= -5) // / # @ ?
            {
                if(s[n].c != '/') s[++n] = { '/', 0 };
                if(t[i] == '/') s[n].x += 1;
                else if(t[i] == '#') s[n].x += 2;
                else if(t[i] == '@') s[n].x += 4;
                else if(t[i] == '?') s[n].x += 8;
            }
            else if(val[t[i]] >= 5) // \ * & !
            {
                if(s[n].c != '\\') s[++n] = { '\\', 0 };
                if(t[i] == '\\') s[n].x += 1;
                else if(t[i] == '*') s[n].x += 2;
                else if(t[i] == '&') s[n].x += 4;
                else if(t[i] == '!') s[n].x += 8;
            }
        }
        if(check()) printf("TRUE %d %d\n", d.rbegin()->first, d.rbegin()->second);
        else puts("FALSE");
    }

    return 0;
}
```

C0487.整数合并

算法

- 埃氏筛的过程中每个数都被它的所有质因子筛一次。
- 在埃氏筛的过程中，当用大于等于P质数i进行筛选时，合并所有i与其倍数所在集合。
- 集合合并可以使用并查集。

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 10;
bool v[N];   // 是否为素数
int f[N];
bool vis[N];

int get(int x)
{
    if(x == f[x]) return x;
    else return f[x] = get(f[x]);
}

void merge(int x, int y)
{
    int fx = get(x), fy = get(y);
    if(fx != fy) f[fy] = fx;
}

int main()
{
    freopen("setb.in", "r", stdin);
    freopen("setb.out", "w", stdout);
    int A, B, P; cin >> A >> B >> P;
    for(int i = 1; i <= B; ++i) f[i] = i;
    memset(v, 1, sizeof(v));
    for(int i = 2; i <= B; ++i)
    {
        if(!v[i]) continue;
        for(int j = 2; j <= B / i; ++j)
        {
            v[i * j] = 0;
            if(i >= P) merge(i, i * j);    // 有大于P的公共质因子
        }
    }
    for(int i = A; i <= B; ++i) vis[get(f[i])] = true;
    int cnt = 0;
    for(int i = 1; i <= B; ++i) if(vis[i]) ++cnt;
    cout << cnt;
```

```
    return 0;
}
```

COGS 4187 接竹竿

算法1

- 模拟
- 对于每个询问，利用栈模拟接竹竿的过程。
- 时间复杂度：O(TQN)；得分：30分。

```
#include <bits/stdc++.h>
using namespace std;

const int N = 110;
int a[N];
int st[N];

int main()
{
    freopen("bamboo.in", "r", stdin);
    freopen("bamboo.out", "w", stdout);
    int T; scanf("%d", &T);
    while(T--)
    {
        int n; scanf("%d", &n);
        for(int i = 1; i <= n; ++i) scanf("%d", &a[i]);
        int q; scanf("%d", &q);
        while(q--)
        {
            int l, r; scanf("%d%d", &l, &r);
            int top = 0;
            for(int i = l; i <= r; ++i)
            {
                int flag = 0;
                for(int j = top; j >= 1; --j)
                    if(a[i] == st[j])
                    {
                        flag = 1;
                        while(st[top] != a[i]) --top;
                        --top;
                        break;
                    }
                if(!flag) st[++top] = a[i];
            }
            printf("%d\n", top);
        }
```

```
    }
    return 0;
}
```

算法2

- 利用模拟来接竹竿速度慢的原因是每次都只消除一串，可以利用倍增法快速消除。
- 定义$f[i][j]$表示从$a[i]$开始消除2^j次能消除的位置。
- 显然$f[i][0] = a[i]$后面第一个与$a[i]$相同的牌出现的位置，如果没有，则令其为$n+1$。
- 显然$f[i][j] = f[f[i][j-1]+1][j-1]$，要求$f[i][j-1]+1 \le n$。
- 那么对于询问$[l,r]$，令$i=l$，表示当前尝试消除的牌。
- 首先看$f[i][0]$，如果$f[i][0] > r$，说明消除一次会越界，那么就令$i+1$，当前牌无法消除，记录答案$+1$。一直执行上述判定，直到牌全部用完或者当前牌可以消除。
- 如果当前牌可以消除，那么从大到小枚举消除的次数2^j，如果$f[i][j] \le r$，那么就令$i = f[i][j]+1$。
- 重复以上过程直到能消除的牌全部消除完。
- 时间复杂度：$\mathrm{O}(TQ\log N)$；得分：100分。

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 15010;
int a[N];
int p[15];  // 后面最近出现的位置
int f[N][25]; // 消除2^j次能消除到哪

int main()
{
    freopen("bamboo.in", "r", stdin);
    freopen("bamboo.out", "w", stdout);
    int T; scanf("%d", &T);
    while(T--)
    {
        int n; scanf("%d", &n);
        for(int i = 1; i <= n; ++i) scanf("%d", &a[i]);
        for(int j = 0; j <= 20; ++j)
            for(int i = 1; i <= n; ++i)
                f[i][j] = n + 1;
        for(int i = 1; i <= 13; ++i) p[i] = n + 1;
        for(int i = n; i >= 1; --i) f[i][0] = p[a[i]], p[a[i]] = i;
        for(int j = 1; j <= 20; ++j)
            for(int i = 1; i <= n; ++i)
                if(f[i][j - 1] + 1 <= n)
                    f[i][j] = f[f[i][j - 1] + 1][j - 1];
```

```
        int q; scanf("%d", &q);
        while(q--)
        {
            int l, r; scanf("%d%d", &l, &r);
            int ans = 0;
            while(l <= r)
            {
                while(l <= r && f[l][0] > r) ++l, ++ans;  // 消除一次越界
                if(l > r) break;
                for(int j = 20; j >= 0; --j)
                    if(f[l][j] <= r) { l = f[l][j] + 1; break;}
            }
            printf("%d\n", ans);
        }
    }
    return 0;
}
```

COGS 3651 消防演练

算法1：枚举

- 枚举任意两个顶点x, y，求出两者之间的路径，求出相应封锁的道路，取最大值。
- 以顶点x为根，进行`dfs`，可以求出到任意顶点的路径，求出最大值即可。
- 时间复杂度：O(N^2)。
- 预计得分：50分。

```
// 50分 O(N^2)
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;

const int N = 1010;
int n;
int ans = 0;
vector<int> g[N];

// 遍历从x点出发的所有路径
void dfs(int x, int fa, int val)
{
    for(int j = 0; j < g[x].size(); ++j)
    {
        int y = g[x][j];
        if(y != fa)
```

```
        {
            int t = g[y].size();
            ans = max(ans, val - 1 + t - 1);
            dfs(y, x, val - 1 + t - 1);
        }
    }
}

int main(void)
{
    freopen("drill.in", "r", stdin);
    freopen("drill.out", "w", stdout);
    scanf("%d", &n);
    for(int i = 1; i <= n - 1; ++i)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        g[x].push_back(y);
        g[y].push_back(x);
    }
    for(int i = 1; i <= n; ++i) dfs(i, 0, g[i].size());
    printf("%d\n", ans);
    return 0;
}
```

算法2：树形DP

- 定义$d[x]$表示从顶点x**出发**向x的子树中走的路径的封锁道路数，这里路径可以只包含一个点。
- 那么对于任意顶点x为根的子树，可以有两种路径：
  - 顶点x和它的一个孩子组合成的路径；
  - 顶点x和它的两个孩子组合成的路径。
- 如果顶点x和它的**一个**孩子组合成的路径，那么对于它的孩子$y \in Son(x)$，则有$d[x] = max(d[x], d[y] + s - 1)$，其中$s = |Son(x)|$，而答案$ans = \max(ans, d[y] + s - 1 + has_fa)$，其中$has_fa$表示$x$是否有父结点。
- 如果顶点x和它的**两个**孩子组合成的路径，那么对于它的孩子$y_i, y_j \in Son(x)$，显然有$ans = \max(ans, d[y_i] + d[y_j] + s - 2 + has_fa)$，其中$has_fa$表示$x$是否有父结点。
- 时间复杂度：$\mathrm{O}(N)$。

```
// 100分 O(N)
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;
```

```cpp

const int N = 2e5 + 10;
int n;
int ans = 0, d[N];
vector<int> g[N];

void dp(int x, int fa)
{
    for(int j = 0; j < g[x].size(); ++j)
    {
        int y = g[x][j];
        if(y != fa) dp(y, x);
    }
    int s = g[x].size() - (fa != 0);    // 孩子个数
    d[x] = s;
    int t = 0;// 孩子中当前d值最大
    for(int j = 0; j < g[x].size(); ++j)
    {
        int y = g[x][j];
        if(y != fa)
        {
            d[x] = max(d[x], d[y] + s - 1);
            ans = max(ans, d[y] + s - 1 + (fa != 0));       // 和一个孩子组成路径
            ans = max(ans, t + d[y] + s - 2 + (fa != 0));   // 和两个孩子组成路径
            t = max(t, d[y]);
        }
    }
}

int main(void)
{
    freopen("drill.in", "r", stdin);
    freopen("drill.out", "w", stdout);
    scanf("%d", &n);
    for(int i = 1; i <= n - 1; ++i)
    {
        int x, y;
        scanf("%d%d", &x, &y);
        g[x].push_back(y), g[y].push_back(x);
    }
    dp(1, 0);
    printf("%d\n", ans);
    return 0;
}
```