

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 $O(N^3)$ ，预计得分25分。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      freopen("TricountUVa.in", "r", stdin);
7      freopen("TricountUVa.out", "w", stdout);
8      int n;
9      while(cin >> n && n >= 3)
10     {
11         long long cnt = 0;
12         for(int a = 1; a <= n; ++a)
13             for(int b = a + 1; b <= n; ++b)
14                 for(int c = b + 1; c <= n; ++c)
15                     if(a + b > c) ++cnt;
16         cout << cnt;
17     }
18     return 0;
19 }
```

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： $O(N)$ 。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      freopen("TricountUVa.in", "r", stdin);
7      freopen("TricountUVa.out", "w", stdout);
8      int n;
9      while(cin >> n && n >= 3)
10     {
11         long long cnt = 0;
12         for(int c = 4; c <= n; ++c)
13         {
14             if(c % 2 == 0) cnt += (c - 2) * (c - 2) / 4;
15             else cnt += (c - 3) * (c - 1) / 4;
16         }
17         cout << cnt;
18     }
19     return 0;
20 }

```

- 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： $O(N \log N)$ 。

```

1  // 区间覆盖问题
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  const int N = 15010;
6  int n, m;

```

```

7  double l, w;
8
9  struct Node
10 {
11     double l, r;
12 } p[N];
13
14 bool cmp(const Node &a, const Node &b)
15 {
16     return a.l < b.l || (a.l == b.l && a.r > b.r);
17 }
18
19 int work()
20 {
21     m = 0;
22     scanf("%d%lf%lf", &n, &l, &w);
23     for(int i = 1; i <= n; ++i)
24     {
25         double x, r; scanf("%lf%lf", &x, &r);
26         if(r < w / 2) continue;
27         p[++m].l = x - sqrt(r * r - w * w / 4);
28         p[m].r = x + sqrt(r * r - w * w / 4);
29     }
30     sort(p + 1, p + m + 1, cmp);
31     if(p[1].l > 0) return -1; // 浇灌范围无法覆盖
32     int ans = 0;
33     double s = 0; // 当前浇灌的最右端
34     int i = 1; // 当前准备选择的喷头
35     while(s < l)
36     {
37         double t = s;
38         for(; i <= m && p[i].l <= t; ++i) s = max(s, p[i].r);
39         if(s == t && s < l) return -1; // 如果不能更新并且浇灌范围无法覆盖
40         ++ans; // 新增一个喷头
41     }
42     return ans;
43 }
44
45 int main()
46 {
47     freopen("sprinkler.in", "r", stdin);
48     freopen("sprinkler.out", "w", stdout);
49     int T; scanf("%d", &T);
50     while(T--) printf("%d\n", work());
51     return 0;
52 }

```

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。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3
4  const int N = 1010;
5  const int M = 1010;
6
7  long long d[N * M];    // 所有函数值
8  int a[N], b[N], c[N];
9
10 long long f(int a, int b, int c, int x)
11 {
12     return (long long)a * x * x + b * x + c;
13 }
14
15 int main()
16 {
17     freopen("minval.in", "r", stdin);
18     freopen("minval.out", "w", stdout);
19     int n, m; cin >> n >> m;
20     for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
21     int tot = 0;        // 当前计算的函数值总数
22     for(int i = 1; i <= n; ++i)
23         for(int x = 1; x <= m; ++x)
24             d[++tot] = f(a[i], b[i], c[i], x);
25     sort(d + 1, d + tot + 1);
26     for(int i = 1; i <= m; ++i) cout << d[i] << " ";
27     return 0;
28 }
```

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： $O(M \log N)$ 。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 5e5 + 10;
5  int a[N], b[N], c[N];
6  int x[N];          // 每个函数当前的自变量值
7
8  long long f(int a, int b, int c, int x)
9  {
10     return (long long)a * x * x + b * x + c;
11 }
12
13 int main()
14 {
15     freopen("minval.in", "r", stdin);
16     freopen("minval.out", "w", stdout);
17     int n, m; cin >> n >> m;
18     for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
19     // 堆内需要存储函数值和函数编号
20     priority_queue<pair<long long, int>, vector<pair<long long, int> >,
21                   greater<pair<long long, int> > > pq;
22     for(int i = 1; i <= n; ++i) // 先求每个函数x=1时的值
23     {
24         pq.push({f(a[i], b[i], c[i], 1), i});
25         x[i] = 1;
26     }
27     for(int i = 1; i <= m; ++i)
28     {
29         pair<long long, int> t = pq.top(); pq.pop(); // 取堆顶
30         cout << t.first << " ";
31         int k = t.second; // 当前函数编号
32         pq.push({f(a[k], b[k], c[k], x[k] + 1), k}); // 计算下一个函数值
33         x[k] = x[k] + 1;
34     }
35     return 0;
36 }
```

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 5e5 + 10;
5  int a[N], b[N], c[N];
6  int x[N];          // 每个函数当前的自变量值
7
8  long long f(int a, int b, int c, int x)
9  {
10     return (long long)a * x * x + b * x + c;
11 }
12
13 struct Node
14 {
15     long long y;
16     int id;
17     bool operator<(const Node &b) const
18     {
19         return y > b.y;
20     }
21 };
22
23 int main()
24 {
25     freopen("minval.in", "r", stdin);
26     freopen("minval.out", "w", stdout);
27     int n, m; cin >> n >> m;
28     for(int i = 1; i <= n; ++i) cin >> a[i] >> b[i] >> c[i];
29     // 堆内需要存储函数值和函数编号
30     priority_queue<Node> pq;
31     for(int i = 1; i <= n; ++i) // 先求每个函数x=1时的值
32     {
33         pq.push({f(a[i], b[i], c[i], 1), i});
34         x[i] = 1;
35     }
36     for(int i = 1; i <= m; ++i)
37     {
38         Node t = pq.top(); pq.pop(); // 取堆顶
39         cout << t.y << " ";
40         int k = t.id; // 当前函数编号
41         pq.push({f(a[k], b[k], c[k], x[k] + 1), k}); // 计算下一个函数值
42         x[k] = x[k] + 1;
43     }
44     return 0;
45 }

```

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。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 2e6 + 10;
5  char s[N];
6  int n = 0;
7  map<int, int> d;    // 深度记录
8
9  map<char, int> val =
10 {
11     {'(', -1}, {'[', -2}, {'{', -3}, {'<', -4},
12     {')', 1}, {']', 2}, {'}', 3}, {'>', 4}
13 };
14
15 bool check()
16 {
17     stack<int> st;
18     for(int i = 1; i <= n; ++i)
19     {
20         if(val[s[i]] <= -1) st.push(val[s[i]], ++d[st.size()]);
21         else if(st.empty()) return false;
22         else if(st.top() + val[s[i]] != 0) return false;
23         else st.pop(), ++d[st.size()];
24     }
25     return st.empty();
26 }
27
28 int main()
29 {
30     freopen("kakko.in", "r", stdin);
31     freopen("kakko.out", "w", stdout);
32     int T; scanf("%d", &T);
33     while(T--)
34     {
35         d.clear();
36         scanf("%s", s + 1);
37         n = strlen(s + 1);
38         if(check()) printf("TRUE %d %d\n", d.rbegin()->first, d.rbegin()->second);
39         else puts("FALSE");
40     }
41     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。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 2e6 + 10;
5  char t[N];
6  struct Node
7  {
8      char c;
9      int x;
10 } s[N]; // 合并相同的任意括号
11 int n = 0;
12 map<int, int> d; // 深度记录
13
14 map<char, int> val =
15 {
16     {'(', -1}, {'[', -2}, {'{', -3}, {'<', -4},
17     {'/', -5}, {'#', -6}, {'@', -7}, {'?', -8},
18     {')', 1}, {']', 2}, {'}', 3}, {'>', 4},
19     {'\\', 5}, {'*', 6}, {'&', 7}, {'!', 8}
20 };
21
22 bool check()
23 {
24     stack<Node> st;
25     int dsum = 0; // 当前深度之和
26     for(int i = 1; i <= n; ++i)
27     {
28         if(val[s[i].c] <= -1) st.push(s[i]), dsum += s[i].x;
29         else if(val[s[i].c] >= 1 && val[s[i].c] <= 4)
30         {
31             if(st.empty()) return false;
32             if(st.top().c != '/' && val[st.top().c] + val[s[i].c] != 0) return false;
33             ++d[dsum]; // 记录一次可能的最大深度
34             Node p = st.top(); st.pop();
35             --p.x; if(p.x) st.push(p);
36             --dsum;
37         }
38         else if(s[i].c == '\\')
39         {
40             ++d[dsum]; // 记录一次可能的最大深度

```

```

41         while(!st.empty() && s[i].x)
42         {
43             Node p = st.top(); st.pop();
44             int y = min(p.x, s[i].x);
45             p.x -= y, s[i].x -= y, dsum -= y;
46             if(p.x) st.push(p);
47         }
48         if(s[i].x) return false;
49     }
50 }
51 return st.empty();
52 }
53
54 int main()
55 {
56     freopen("kakko.in", "r", stdin);
57     freopen("kakko.out", "w", stdout);
58     int T; scanf("%d", &T);
59     while(T--)
60     {
61         d.clear();
62         n = 0;
63         scanf("%s", t + 1);
64         for(int i = 1; t[i] != '\0'; ++i)
65         {
66             if(val[t[i]] >= -4 && val[t[i]] <= 4) s[++n] = { t[i], 1 };
67             else if(val[t[i]] <= -5) // / # @ ?
68             {
69                 if(s[n].c != '/') s[++n] = { '/', 0 };
70                 if(t[i] == '/') s[n].x += 1;
71                 else if(t[i] == '#') s[n].x += 2;
72                 else if(t[i] == '@') s[n].x += 4;
73                 else if(t[i] == '?') s[n].x += 8;
74             }
75             else if(val[t[i]] >= 5) // \ * & !
76             {
77                 if(s[n].c != '\\') s[++n] = { '\\', 0 };
78                 if(t[i] == '\\') s[n].x += 1;
79                 else if(t[i] == '*') s[n].x += 2;
80                 else if(t[i] == '&') s[n].x += 4;
81                 else if(t[i] == '!') s[n].x += 8;
82             }
83         }
84         if(check()) printf("TRUE %d %d\n", d.rbegin()->first, d.rbegin()->second);
85         else puts("FALSE");
86     }
87
88     return 0;
89 }

```

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。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 1e5 + 10;
5  bool v[N];    // 是否为素数
6  int f[N];
7  bool vis[N];
8
9  int get(int x)
10 {
11     if(x == f[x]) return x;
12     else return f[x] = get(f[x]);
13 }
14
15 void merge(int x, int y)
16 {
17     int fx = get(x), fy = get(y);
18     if(fx != fy) f[fy] = fx;
19 }
20
21 int main()
22 {
23     freopen("setb.in", "r", stdin);
24     freopen("setb.out", "w", stdout);
25     int A, B, P; cin >> A >> B >> P;
26     for(int i = 1; i <= B; ++i) f[i] = i;
27     memset(v, 1, sizeof(v));
28     for(int i = 2; i <= B; ++i)
29     {
30         if(!v[i]) continue;
31         for(int j = 2; j <= B / i; ++j)
32         {
33             v[i * j] = 0;
34             if(i >= P) merge(i, i * j);    // 有大于P的公共质因子
35         }
36     }
37     for(int i = A; i <= B; ++i) vis[get(f[i])] = true;
38     int cnt = 0;
39     for(int i = 1; i <= B; ++i) if(vis[i]) ++cnt;
40     cout << cnt;
```

```
40 return 0;
42 }
```

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。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 110;
5  int a[N];
6  int st[N];
7
8  int main()
9  {
10     freopen("bamboo.in", "r", stdin);
11     freopen("bamboo.out", "w", stdout);
12     int T; scanf("%d", &T);
13     while(T--)
14     {
15         int n; scanf("%d", &n);
16         for(int i = 1; i <= n; ++i) scanf("%d", &a[i]);
17         int q; scanf("%d", &q);
18         while(q--)
19         {
20             int l, r; scanf("%d%d", &l, &r);
21             int top = 0;
22             for(int i = l; i <= r; ++i)
23             {
24                 int flag = 0;
25                 for(int j = top; j >= 1; --j)
26                     if(a[i] == st[j])
27                     {
28                         flag = 1;
29                         while(st[top] != a[i]) --top;
30                         --top;
31                         break;
32                     }
33                 if(!flag) st[++top] = a[i];
34             }
35             printf("%d\n", top);
36         }
37     }
```

```

37     }
38     return 0;
39 }

```

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 $f[i][j - 1] + 1 \leq n$ 。
- 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 $f[i][j] \leq r$ ，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： $O(TQ \log N)$ ；得分：100分。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int N = 15010;
5  int a[N];
6  int p[15]; // 后面最近出现的位置
7  int f[N][25]; // 消除2^j次能消除到哪
8
9  int main()
10 {
11     freopen("bamboo.in", "r", stdin);
12     freopen("bamboo.out", "w", stdout);
13     int T; scanf("%d", &T);
14     while(T--)
15     {
16         int n; scanf("%d", &n);
17         for(int i = 1; i <= n; ++i) scanf("%d", &a[i]);
18         for(int j = 0; j <= 20; ++j)
19             for(int i = 1; i <= n; ++i)
20                 f[i][j] = n + 1;
21         for(int i = 1; i <= 13; ++i) p[i] = n + 1;
22         for(int i = n; i >= 1; --i) f[i][0] = p[a[i]], p[a[i]] = i;
23         for(int j = 1; j <= 20; ++j)
24             for(int i = 1; i <= n; ++i)
25                 if(f[i][j - 1] + 1 <= n)
26                     f[i][j] = f[f[i][j - 1] + 1][j - 1];

```

```

27     int q; scanf("%d", &q);
28     while(q--)
29     {
30         int l, r; scanf("%d%d", &l, &r);
31         int ans = 0;
32         while(l <= r)
33         {
34             while(l <= r && f[l][0] > r) ++l, ++ans; // 消除一次越界
35             if(l > r) break;
36             for(int j = 20; j >= 0; --j)
37                 if(f[l][j] <= r) { l = f[l][j] + 1; break;}
38         }
39         printf("%d\n", ans);
40     }
41 }
42 return 0;
43 }

```

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。

```

1 // 50分 O(N^2)
2 #include <cstdio>
3 #include <cstring>
4 #include <algorithm>
5 #include <vector>
6 using namespace std;
7
8 const int N = 1010;
9 int n;
10 int ans = 0;
11 vector<int> g[N];
12
13 // 遍历从x点出发的所有路径
14 void dfs(int x, int fa, int val)
15 {
16     for(int j = 0; j < g[x].size(); ++j)
17     {
18         int y = g[x][j];
19         if(y != fa)

```

```

20     {
21         int t = g[y].size();
22         ans = max(ans, val - 1 + t - 1);
23         dfs(y, x, val - 1 + t - 1);
24     }
25 }
26 }
27
28 int main(void)
29 {
30     freopen("drill.in", "r", stdin);
31     freopen("drill.out", "w", stdout);
32     scanf("%d", &n);
33     for(int i = 1; i <= n - 1; ++i)
34     {
35         int x, y;
36         scanf("%d%d", &x, &y);
37         g[x].push_back(y);
38         g[y].push_back(x);
39     }
40     for(int i = 1; i <= n; ++i) dfs(i, 0, g[i].size());
41     printf("%d\n", ans);
42     return 0;
43 }

```

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： $O(N)$ 。

```

1 // 100分 O(N)
2 #include <cstdio>
3 #include <cstring>
4 #include <algorithm>
5 #include <vector>
  using namespace std;

```

```

0
8  const int N = 2e5 + 10;
9  int n;
10 int ans = 0, d[N];
11 vector<int> g[N];
12
13 void dp(int x, int fa)
14 {
15     for(int j = 0; j < g[x].size(); ++j)
16     {
17         int y = g[x][j];
18         if(y != fa) dp(y, x);
19     }
20     int s = g[x].size() - (fa != 0);    // 孩子个数
21     d[x] = s;
22     int t = 0; // 孩子中当前d值最大
23     for(int j = 0; j < g[x].size(); ++j)
24     {
25         int y = g[x][j];
26         if(y != fa)
27         {
28             d[x] = max(d[x], d[y] + s - 1);
29             ans = max(ans, d[y] + s - 1 + (fa != 0));    // 和一个孩子组成路径
30             ans = max(ans, t + d[y] + s - 2 + (fa != 0));    // 和两个孩子组成路径
31             t = max(t, d[y]);
32         }
33     }
34 }
35
36 int main(void)
37 {
38     freopen("drill.in", "r", stdin);
39     freopen("drill.out", "w", stdout);
40     scanf("%d", &n);
41     for(int i = 1; i <= n - 1; ++i)
42     {
43         int x, y;
44         scanf("%d%d", &x, &y);
45         g[x].push_back(y), g[y].push_back(x);
46     }
47     dp(1, 0);
48     printf("%d\n", ans);
49     return 0;
50 }

```

