A. Circuit Math

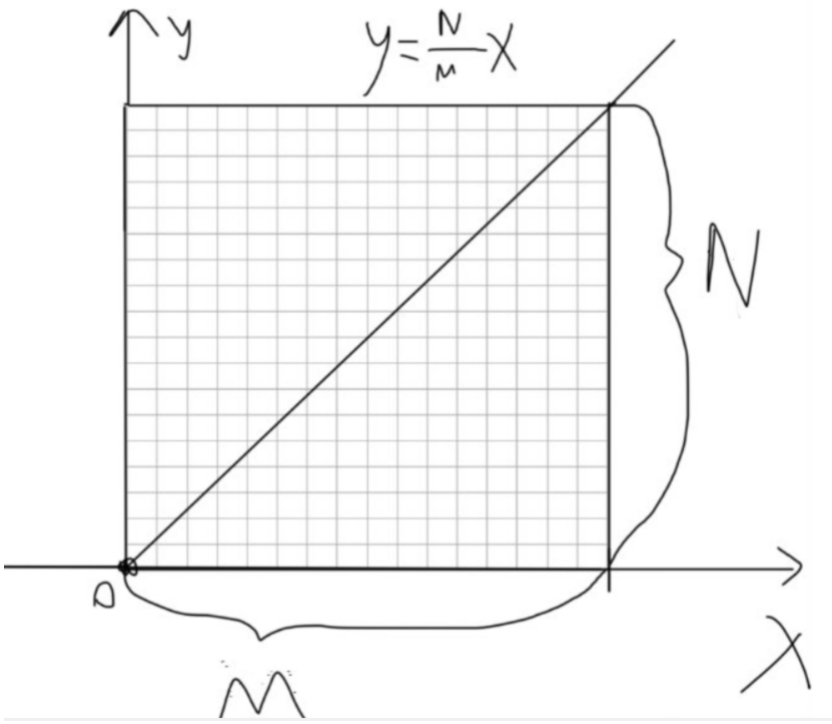
后缀表达式的问题，可以用栈解决：

- 如果遇到输入变量，将其值压入栈中
- 如果遇到运算符 $-$ ，弹出栈顶一个值，取非后再将结果压入栈中
- 如果遇到运算符 $*$ 或 $+$，弹出栈顶的两个值，进行相应运算后再将结果压入栈中
- 将所有电路描述处理完后，栈中留下的值就是该电路的输出

```
#include<bits/stdc++.h>

using namespace std;

stack<int> s;
map<char, int> m;

int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int n, ans, v1, v2;
    char value, op;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> value;
        m['A' + i] = (value == 'T') ? 1 : 0;  //根据输入对字母的值进行初始化
    }
    while (cin >> op) {
        if (isalpha(op))  //输入为字母，则其值入栈
            s.push(m[op]);
        else if (op == '+')  //输入为或运算
        {
            v1 = s.top(); //取出栈顶元素v1
            s.pop();
            v2 = s.top();  //取出栈顶元素v2
            s.pop();
            if (v1 == 1 || v2 == 1)  //对v1,v2进行或运算
                ans = 1;
            else
                ans = 0;
            s.push(ans);  //将运算结果入栈
        } else if (op == '*')  //输入为与运算
        {
            v1 = s.top();  //取出栈顶元素v1
            s.pop();
            v2 = s.top();  //取出栈顶元素v2
            s.pop();
            if (v1 == 1 && v2 == 1)  //对v1,v2进行或运算
                ans = 1;
            else
                ans = 0;
            s.push(ans);  //将运算结果入栈
        } else  //输入为非运算
        {
            v1 = s.top();  //取出栈顶元素v1
            s.pop();
            if (v1 == 1)  //对v1进行非运算，即取反
```

```
                ans = 0;
            else
                ans = 1;
            s.push(ans);  //将运算结果入栈
        }
    }
    if (s.top())  //当全部输入处理完毕后，栈中所剩的唯一元素即是表达式的值
        cout << "T" << endl;
    else
        cout << "F" << endl;
    return 0;
}
```

B. Diagonal Cut



首先画个图，对角线方程为$y=\frac{N}{M}$，设正方形左下角点坐标为$(p,\ q)$，则正方形中心$(p+\frac{1}{2},q+\frac{1}{2})$，当对角线过正方形中心时，正方形面积被分为两个相等的部分，由此可列出方程：

$(p+\frac{1}{2})*\frac{N}{M}=q+\frac{1}{2}\ (0\le q<M,0\le q<N)$ 求p,q解的组数

其中N可写为$n*gcd(M,N)$，同理M可写为$m*gcd(M,N)$，那么方程可化为：

$(2p+1)*\frac{n}{m}=2q+1\ (0\le p<M,\ 0\le q<N)$

显然$2p+1$和$2q+1$为奇数，所以m,n均为奇数时才有解，解释如下：

1. 若n为奇，m为偶$(2p+1)*\frac{n}{m}$必为小数，$2q+1$为整数，无解;
2. 若n为奇，m为奇$(2p+1)*\frac{n}{m}$可能为整数，$2q+1$为整数，可能有解;（当$2p+1$整除m，$2q+1$整除n才有解,因为上式是整数乘以分数得到整数）
3. 若n为偶，m为偶$(2p+1)*\frac{n}{m}$必为小数，$2q+1$为整数，无解;
4. 若n为偶，m为奇$(2p+1)*\frac{n}{m}$若为整数，则是偶数，$2q+1$为奇数，无解;

方程再化为：

$\frac{2p+1}{m}=\frac{2q+1}{n}\ (1\le 2p+1<2M+1,\ 1\le 2q+1<2N+1)$

$(1\le 2p+1<2M+1,1\le 2q+1<2N+1)$可改写为：

$(m \le 2p+1 \le (2gcd(M,N)-1)*m,\ n \le 2q+1 \le (2gcd(M,N)-1)*n)$

因为$2p+1$和$2q+1$都为奇数，所以枚举范围内奇数且是m的倍数即可。

解如下：

p	q
$(m-1)/2$	$(n-1)/2$
$(3m-1)/2$	$(3n-1)/2$
$(5m-1)/2$	$(5n-1)/2$
.....	.....
$(2gcd(M,N)-1)*m-1)/2$	$((2gcd(M,N)-1)*n-1)/2$

一共$gcd(M,N)$个解。

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

ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

int main() {
    ll n, m;
    cin >> n >> m;
    ll g = gcd(n, m);
    n /= g, m /= g;
    if ((n & 1) && (m & 1))
        cout << g << endl;
    else
        cout << "0" << endl;
    return 0;
}
```

C. Gerrymandering

每行的第一个数表示选举票数最多的人（A或B），第二个数表示A浪费的票数，第三个数表示B浪费的票数。最后一个小数，应该将A在各个地区获得的总票数与B在各个地区的总票数的差值，将这个差值除以A和B所有的总票数。

```
#include <bits/stdc++.h>

using namespace std;

#define N 1005

int main() {
    int p, d, district;    //p:选举 ， d:地区 ， district: 地区编号
    long long a[N] = {0}, b[N] = {0}, ra, rb, sum = 0;
    cin >> p >> d;
```

```
    for (int i = 0; i < p; ++i) {
        cin >> district >> ra >> rb;
        a[district] += ra;
        b[district] += rb;
        sum = sum + ra + rb;
    }
    double rate = 1.0;
    ra = rb = 0;
    for (int i = 1; i <= d; ++i) {
        if (a[i] > b[i]) {
            cout << "A " << a[i] - (a[i] + b[i]) / 2 - 1 << " " << b[i] << "\n";
            rb += b[i];
            ra = ra + a[i] - (a[i] + b[i]) / 2 - 1;
        } else {
            cout << "B " << a[i] << " " << b[i] - (a[i] + b[i]) / 2 - 1 << "\n";
            ra += a[i];
            rb = rb + b[i] - (a[i] + b[i]) / 2 - 1;
        }
    }
    rate = rate / sum * abs(ra - rb);
    printf("%.10f\n", rate);
    return 0;
}
```

D. Missing Numbers

我们需要在输入的连续递增的n个数中找出缺失的数字，并排出来。由于补全后的数字是从1开始依次递增的数字，我们可以在 for 循环中输入数字并进行标记，如果最后一个数字与n相等输出 $good\ \ job$；如果输入最后一个数字大于 n，那么需要遍历输出未被标记的数字。

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

bool b[200];
int main()
{
    int n, x, a = 0;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        cin >> x;
        a = max(a, x);
        b[x] = 1;    //标记出现过的数字
    }
    if (a == n)
    {
        cout << "good job" << endl;
    }
    else
    {
        for (int i = 1; i <= a; i++)
            if (!b[i])
                cout << i << endl;
    }
    return 0;
}
```

E. NVWLS

总体思路：

- 使用AC**自动机**进行多模式串匹配。
- 在匹配过程中用DP递推最终答案。

具体描述：

1. 将所有单词元音去掉，得到新单词，将新单词插入AC自动机中，并保存新单词原本含有的元音个数($vowNum$)（当多个老单词去掉元音后得到的新单词一样时，保存含有元音个数最多的老单词的元音个数即可）。
2. S串：待还原串。

$dp[i]!=-inf$：$S_{1.....i}$ 可以由某些单词构成，并且这些单词的元音个数和为$dp[i]$；$dp[0]=0$。

用S串去AC自动机中匹配，如果当前匹配成功的字符(第i个字符)是某个单词(假设为第x个单词)的结尾时，则令

$dp[i]=max(dp[i-len[x]]+vowNum[x],dp[i])$。

$dp[lenS]$即为可构成S串的所有单词组合中包含的最大元音数的组合。打印答案只需要在$dp[i]$被更新时记录下单词的下标即可。

3. 需要注意的是：如果对于每个字符S_i都通过fail指针寻找以S_i结尾且成功匹配的单词，那么对于特定数据，算法将会超时。因此，需要在构建$fail$指针同时，构建一个$last$指针，指向以S_i结尾且成功匹配的单词来优化上诉过程。

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1e5 + 5, maxm = maxn * 3;

int vowNum[maxn], len[maxn];
char newWord[maxn];
string wd[maxn];


int trie[maxm][26], idx[maxm], fail[maxm];
int tot = 0, last[maxm];

void insert(char *nw, int id, int lenw) {
    int p = 0;
    for (int i = 0; i < lenw; i++) {
        int c = nw[i] - 'A';
        if (!trie[p][c])
            trie[p][c] = ++tot;
        p = trie[p][c];
    }
    if (vowNum[id] >= vowNum[idx[p]])
        idx[p] = id;
}

void getNewWord(string word, int id) {
    int lenw = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word[i] == 'A' || word[i] == 'E' || word[i] == 'I' || word[i] == 'O'
|| word[i] == 'U')
            vowNum[id]++;
```

```
        else
            newWord[lenw++] = word[i];
    }
    newWord[lenw] = '\0';

    len[id] = lenw;
    insert(newWord, id, lenw);
    //cout<<newWord<<" nw"<<endl;
}

int Q[maxm];

void build() {
    int t = 0, h = 1;
    for (int i = 0; i < 26; i++) {
        if (trie[0][i])
            Q[++t] = trie[0][i];
    }
    while (h <= t) {
        //cout<<1<<endl;
        int u = Q[h++];
        for (int i = 0; i < 26; i++) {
            if (trie[u][i]) {
                fail[trie[u][i]] = trie[fail[u]][i];
                Q[++t] = trie[u][i];
            } else
                trie[u][i] = trie[fail[u]][i];
        }
        last[u] = idx[fail[u]] ? fail[u] : last[fail[u]];
    }
}

char str[maxm];
int dp[maxm];
int pre[maxm], ans[maxm];

void sovle() {
    int p = 0;
    memset(dp, -1, sizeof(dp));
    dp[0] = 0;
    int lens = strlen(str + 1);
    for (int i = 1; i <= lens; i++) {
        int c = str[i] - 'A';
        p = trie[p][c];
        for (int j = p; j; j = last[j]) {
            if (idx[j]) {
                int x = dp[i - len[idx[j]]];
                if (x != -1 && dp[i] < x + vowNum[idx[j]]) {
                    pre[i] = i - len[idx[j]], ans[i] = idx[j];
                    dp[i] = x + vowNum[idx[j]];
                }
            }
        }
    }
    vector<int> v;
    for (int i = lens; i; i = pre[i])
        v.push_back(ans[i]);
    for (int i = v.size() - 1; i > 0; i--)
```

```
        cout << wd[v[i]] << ' ';
    cout << wd[v[0]] << '\n';
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> wd[i];
        getNewWord(wd[i], i);
    }
    build();
    cin >> str + 1;
    sovle();
    return 0;
}
```

F. Prospecting

本题中，最坏和最好两种情况可以被分为两个独立求解的问题。

对于最坏情况 $worst\ case$：

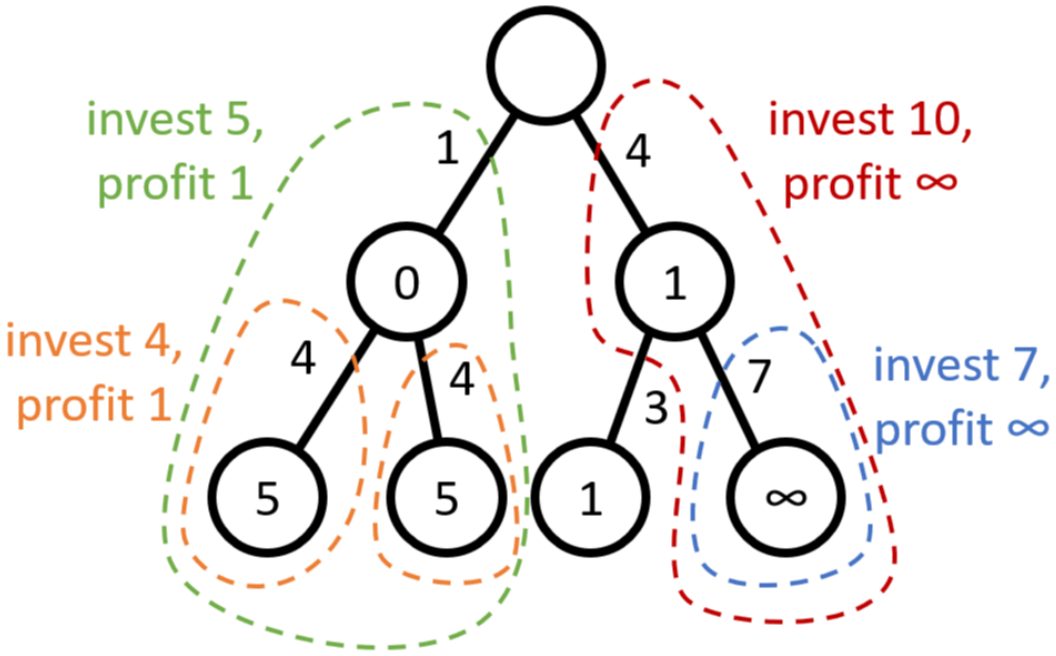
当然是在不达 $motherlode$ 的路径上尽量想办法多花钱，最后没法多花钱后直接挖到 $motherlode$ 即可。我们设在i点处可以产生的最大花费为 $worst_i$，那么对于以 i 为根的子树，无非两种花钱方法（两种 $worst_i$ 取值情况）：

1. 不探索子树，花费 $y_i - 1$（尽可能而不挖通通道）
2. 探索子树，花费 $y_i - x_i + \sum\limits_{child\ j} worst_j$（需要挖通当前通道，并且必须取得子节点上的利益）

$worst_i$ 显然是取上述两种情况的最大值。题目要求整棵树的 $worst$，是个典型的递归求解问题。

对于最好情况 $best\ case$：

先看这样一幅图：



$invest$ 是指从该子树挖掘中获利的最少投资。

$profit$ 是指挖掘该子树后可以获得的利润。

图中的虚线标示出了所有的可获利子树（$profit > 0$）。

那么怎么计算上图中的 $invest$ 和 $profit$ 呢（用来判断子树是否可获利）？对于节点 i 为根的子树，计算 $profit$ 的方法如下：

1. 只挖该点，利润为$x_i - y_i$
2. 继续探索子节点，利润为$x_i - y_i + \sum_{child\ j} \max(0, profit_j)$（取 max 是因为有的点 $profit$ 为负值，相当于不挖掘这种亏损点）

在计算 $profit$ 的同时，按照定义我们可以计算出 $invest$：

1. 对于叶子结点i，如果$x_i > y_i$，说明挖掘该点能获利，那么挖掘该点的 $invest$ 即为y_i。
2. 对于非叶子结点i，如果$x_i > y_i$，则invest为y_i，否则需要进一步挖掘才能获利，$invest$ 为 $y_i - x_i + min(invest_j)$，其中$j$为$i$的孩子，且以其为根的子树可获利。

上述过程显然也是一个典型的递归求解问题。

显然，我们只需要考虑挖掘可获利子树（谁愿意亏钱呢）。在这个基础上，我们贪心地挖掘 $invest$ 最少的子树，原因如下：

1. 挖完后，手上的钱会变多
2. 如果这棵子树包含 $motherlode$，那么显然这就是最优解。
3. 如果这棵树不包含 $motherlode$，那么显然挖掘这棵子树不会使结果更差（持有钱变多了，而且不存在更少的$invest$ 直通 $motherlode$）。

开始时，能挖的点显然只有根节点（入口）的孩子。按照题意，已挖掘部分畅通无阻，所以其实我们每步都是挑一个与我们已挖掘部分直接相连的未挖掘的点（可以理解为待挖点集）去挖，挖完后，被挖的点的孩子也将加入待挖点集中。我们需要动态维护这一待挖点集，并按照 $invest$ 从小到大的顺序模拟挖掘。如果使用优先队列维护挖掘序列，则计算的复杂度为$O(nlogn)$。

```
#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pii;

const int oo = 0x3f3f3f3f;

int n;
int motherlode;
vector<int> p, x, y, worst, profit, invest;
vector<vector<int> > G;
priority_queue<pii, vector<pii>, greater<> > pq;

int worst_case(int cur) {
    int res = 0;
    for (int u:G[cur])
        res += worst_case(u);
    return max(y[cur] - 1, y[cur] - x[cur] + res);
}

int get_profit(int cur) {
    int childInvest = oo;
    profit[cur] = x[cur] - y[cur];
    for (int u:G[cur]) {
        profit[cur] += max(0, get_profit(u));
        if (profit[u] > 0) childInvest = min(childInvest, invest[u]);
    }
```

```
    invest[cur] = y[cur] < x[cur] ? y[cur] : max(y[cur], y[cur] - x[cur] +
childInvest);
    return profit[cur];
}

int best_case(int root) {
    int cur;
    int money = 0, bestCost = 0;
    get_profit(root);
    pq.emplace(0, 0);
    do {
        cur = pq.top().second;
        pq.pop();
        bestCost = max(bestCost, y[cur] - money);
        money += x[cur] - y[cur];
        for (int u:G[cur])
            if (profit[u] > 0) pq.emplace(invest[u], u);
    } while (cur != motherlode);
    return bestCost;
}

void init() {
    cin >> n;
    p.resize(n);
    x.resize(n);
    y.resize(n);
    worst.resize(n);
    profit.resize(n);
    invest.resize(n);
    G.resize(n);
    for (int i = 1; i < n; i++) {
        cin >> p[i] >> x[i] >> y[i];
        G[p[i]].push_back(i);
        if (x[i] == -1) motherlode = i, x[i] = oo;
    }
    x[0] = 0, y[0] = 0;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);
    init();
    cout << worst_case(0) + 1 << ' ' << best_case(0) << endl;
    return 0;
}
```

G. Research Productivity Index

设提交了n篇，P_i为通过i篇的概率，$f(x)=x^{x/n}$，那么题目要求计算的期望$\varepsilon=\sum_{i=1}^{n} f(i)*P_i$，因此我们只需要求出通过$i$篇的最大概率，当选出$i$个时贪心的想肯定是选择通过概率最大的前$i$篇论文，因此只需要在计算概率之前对所有论文按通过率降序，这样求解时一定是最大概率。

注意到题目的数据范围很小，设$d[i][j]$为提交i篇论文且通过j篇的概率，状态转移方程为：

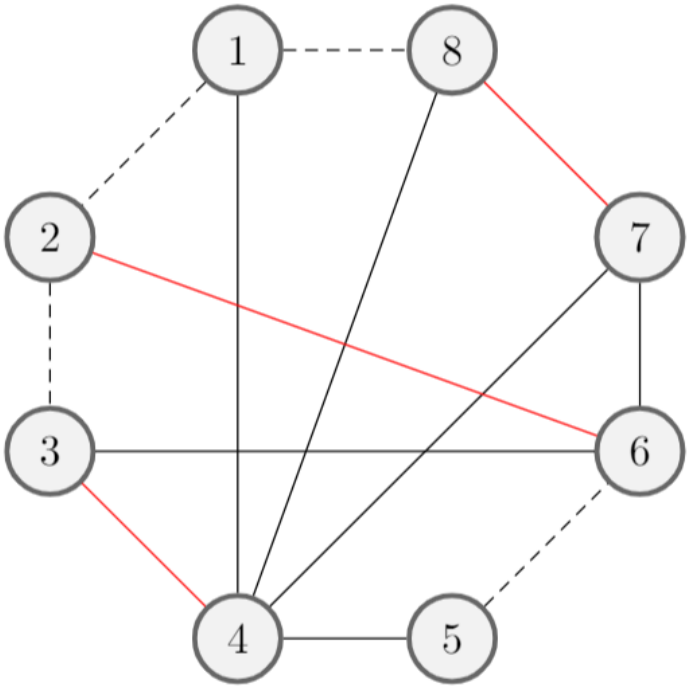
$$d[i][j]=d[i-1][j]*(1.0-p[i])+d[i-1][j-1]*p[i]$$

特别地，对于$d[i][0], d[i][i]$我们需要单独计算，详见代码。求出$d[i][j]$后分别计算提交$i(1 \leq i \leq n)$时的期望然后取max即可，时间复杂度$O(n^2)$。

```
#include <bits/stdc++.h>

using namespace std;

int n, a[105];
double p[105], d[105][105];
//d[i][j]代表提交了i篇正确j篇的概率

int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    sort(a + 1, a + 1 + n, greater<int>());
    for (int i = 1; i <= n; i++) p[i] = 1.0 * a[i] / 100;
    d[0][0] = 1.0;
    for (int i = 1; i <= n; i++) {
        d[i][0] = d[i - 1][0] * (1.0 - p[i]);
        for (int j = 1; j < i; j++)
            d[i][j] = d[i - 1][j] * (1.0 - p[i]) + d[i - 1][j - 1] * p[i];
        d[i][i] = d[i - 1][i - 1] * p[i];
    }
    double ans = 0;
    for (int i = 1; i <= n; i++) {
        double res = 0;
        for (int j = 1; j <= i; j++)
            res += pow(j, 1.0 * j / i) * d[i][j];
        ans = max(ans, res);
    }
    printf("%.10lf\n", ans);
    return 0;
}
```

H. Running Routus

本题看似复杂，实则不难，比较裸的区间DP。



我们可以这样设计状态：$dp_{i,j}$ $(i \le j)$表示区间 $[i,j]$ 内合法连接方案中最多的连接数，$v_{i,j}$ 则表示从 i 到 j 是否存在路径，存在为 1，不存在则为 0。

考虑选中一条路径情况下的最优解。如上图，当选中结点 2 向结点 6 的跑道，剩余的跑道就只能在 $[3,4,5]$，$[7,8,1]$ 两个区间内选择，我们可以选择$(3,4),(7,8)$两条跑道以达到在该选择下的最优解。

如此分析，状态转移方程为：$dp_{i,j} = max(dp_{i,j}\ ,\ dp_{i+1,k-1} + v_{i,k} + dp_{k+1,j})\ (i \le k \le j)$

该方案的时间复杂度为 $O(n^3)$。

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

int n;
int v[505][505];
int dp[505][505];

int main() {
    ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
    memset(dp, 0, sizeof(dp));
    cin >> n;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) cin >> v[i][j];
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n - i; ++j)
            for (int k = j; k <= j + i; ++k)
                dp[j][j + i] = max(dp[j][j + i], dp[j + 1][k - 1] + v[j][k] +
dp[k + 1][j + i]);
    cout << dp[1][n];
    return 0;
}
```

I. Slow Leak

先 $floyd$ 跑一遍，找到每一对点之间的最短距离 。然后建立这样一个图：把所有的加油站和起点1和终点n，放在一起 ，如果一对点的距离超过d，那么视为无法通过，最后在新图上跑一遍 $floyd$ 或者 $Dijkstra$。

```
#include <bits/stdc++.h>

using namespace std;
#define ll long long
const int N = 550;
const ll inf = 0x3f3f3f3f3f;  //注意这里有5个3f，只有四个会过不了
ll dp[N][N];

int main() {
    ios::sync_with_stdio(false);
    cout.tie(NULL);

    ll n, m, t, d;
    ll sta[N];
    cin >> n >> m >> t >> d;
    memset(dp, inf, sizeof(dp));

    for (int i = 1; i <= t; i++) cin >> sta[i];
    sta[t + 1] = 1, sta[t + 2] = n;

    while (m--) {
        ll a, b, c;
        cin >> a >> b >> c;
        dp[a][b] = c;
        dp[b][a] = c;
    }
    //第一遍floyd是为了知道每一对点之间的最短距离，如果最短距离仍然大于d那么一定不能跑
    for (ll k = 1; k <= n; k++) {
        for (ll i = 1; i <= n; i++) {
            for (ll j = 1; j <= n; j++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
    //处理距离大于d的数据
    for (ll i = 1; i <= t + 2; i++) {
        for (ll j = 1; j <= t + 2; j++) {
            ll di = sta[i], dj = sta[j];
            if (dp[di][dj] > d) dp[di][dj] = inf;
        }
    }
    //最短路算法
    for (ll k = 1; k <= t + 2; k++) {
        for (ll i = 1; i <= t + 2; i++) {
            for (ll j = 1; j <= t + 2; j++) {
                ll di = sta[i], dj = sta[j], dk = sta[k];
                dp[di][dj] = min(dp[di][dj], dp[di][dk] + dp[dk][dj]);
            }
        }
    }
    //如果dp[1][n]==inf,那么说明路径没有被缩短过，输出stuck
```

```
    if (dp[1][n] == inf) cout << "stuck";
    else cout << dp[1][n];
    return 0;
}
```

J. Stop Counting!

设前缀平均值最大为$dp[i]$， a为后缀的和，z为后缀的数量：

由$\frac{dp[i]*i+a}{i+z} > dp[i]$

得到：$dp[i-1]*z < a$

得到：$dp[i-1] < \frac{a}{z}$

由此可以知道后缀的平均值必须大于前缀时才可以让全部的平均值变大，但后缀的平均值大于前缀的最大平均值时，就不需要前缀了，因为后缀本身就比前缀要大。

若找到前缀的最大赔率，全都大于后缀的最大赔率，则不管后缀加多少都只会使赔率降低；后缀的最大赔率同理，注意：如果前缀后缀最大平均值小于0，则题中要找的最大平均值为0。

故只需遍历得到前缀平均值与后缀平均值的最大值即可。

```
#include<bits/stdc++.h>

using namespace std;
#define ll long long

double A[1000005];
double pre[1000005];
double suf[1000005];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> A[i];
        pre[i] = pre[i - 1] + A[i];//得到前缀和
    }
    for (int i = n; i >= 1; i--) {
        suf[i] = suf[i + 1] + A[i];//得到后缀和
    }
    double max1 = 0;//初始化max1，当前缀和与后缀和的平均值都小于0时取0

    for (int i = 1; i <= n; i++) {
        max1 = max(max1, pre[i] / i);//求前缀最大平均值
        max1 = max(max1, suf[i] / (n - i + 1));//求后缀最大平均值
    }
    printf("%.9lf\n", max1);//输出为小数点后9位
}
```

K. Summer Trip

这题看着完全不能暴力，因为暴力需要三个循环。但是其实每一个循环都跑不满，知道开头的元素，找结尾的元素，若要满足开头元素和结尾元素只有一个的话，则这段序列的种类数最多为26。

```
#include<bits/stdc++.h>

using namespace std;
#define ll long long

int main() {
    string s;
    cin >> s;
    ll sum = 0;
    for (int i = 0; i < s.length() - 1; i++)//枚举从s[i]为起始元素开始的每一个的情况
    {
        if (s[i] != s[i + 1])//当s[i+1]与s[i]相同时就直接跳到下一个。
        {
            int indc = 0;
            for (int j = i + 1; j < s.length(); j++)//向后枚举所有情况，看适不适合做末尾元素
            {
                if (indc >= 26)//小优化，当到达第27种元素的时候必然不能做末尾元素
                    break;
                if (s[j] == s[i])//当有与起始元素相同的元素时退出，
                {
                    break;
                }
                bool is = false;//标记s[j]是否在前面存在
                if (s[j] != s[j - 1])//当遇到连续的末尾元素时退出，不进入第三层循环
                {
                    for (int k = i + 1; k < j; k++)//找从i+1到j-1是否存在与s[j]相同的元素
                    {
                        if (s[k] == s[j])//存在与s[j]相同的元素就退出循环。
                        {
                            is = true;//存在
                            break;
                        }
                    }
                    if (!is)//不存在就满足好的序列，总数+1，且种类数+1
                    {
                        sum++;
                        indc++;
                    }
                }
            }
        }
    }
    cout << sum << endl;
    return 0;
}
```

L. Traveling Merchant

对于正向和反向从星期一到星期天建立14颗线段树，维护3个信息，最大值Max，最小值Min，右边减去左边的最大值 $_Max$，根据查询的l,r得出应该查询哪个方向星期几的线段树，由于只记录了每个子树的答案，若区间在不同的子树上就要根据左右子树的信息得到当前的答案，建树复杂度$14*nlog(n)$，单次查询复杂度$log(n)$。

```
#include <bits/stdc++.h>
#pragma GCC optiMinze(2)
using namespace std;
#define iny long long
const iny maxn = 1e5 + 10;
struct node
{
    int Max;
    int Min;
    int _Max;
    node(){
        Max=Min=_Max=0;
    }
};
node tree[15][maxn << 2];
int v[15][maxn], d[maxn], n, q, s;

inline void push_up(iny p)
{
    tree[s][p]._Max = max(tree[s][2 * p]._Max, max(tree[s][2 * p + 1]._Max,
tree[s][2 * p + 1].Max - tree[s][2 * p].Min));
    tree[s][p].Max = max(tree[s][2 * p].Max, tree[s][2 * p + 1].Max);
    tree[s][p].Min = min(tree[s][2 * p].Min, tree[s][2 * p + 1].Min);
}

void build(iny p, iny l, iny r)
{
    if (l == r)
    {
        tree[s][p]._Max = 0;
        tree[s][p].Max = v[s][l];
        tree[s][p].Min = v[s][l];
        return;
    }
    iny Mind = l + r >> 1;
    build(p << 1, l, Mind);
    build((p << 1) + 1, Mind + 1, r);
    push_up(p);
}

node query(iny l, iny r, iny p, iny pl, iny pr)
{
    if (l <= pl && r >= pr)
    {
        return tree[s][p];
    }
    iny Mind = pl + pr >> 1;
    node ans1, ans2, ans3;
    if (l <= Mind)
    {
```

```
        ans2 = query(l, r, p << 1, pl, Mind);
        ans1 = ans2;
    }
    if (r > Mind)
    {
        ans3 = query(l, r, (p << 1) + 1, Mind + 1, pr);
        if (l <= Mind)
        {
            ans1._Max = max(ans3.Max - ans2.Min, max(ans3._Max, ans2._Max));
            ans1.Min = min(ans2.Min, ans3.Min);
            ans1.Max = max(ans2.Max, ans3.Max);
        }
        else
            ans1 = ans3;
    }
    return ans1;
}

int main()
{
    scanf("%d",&n);
    for (iny i = 1; i <= n; i++)
    {
        scanf("%d%d", &v[0][i], &d[i]);
    }
    for (iny i = 0; i <= 6; i++)
    {
        iny nw = i;
        for (iny j = 1; j <= n; j++)
        {
            if (nw == 0 || nw == 6)
            {
                v[i + 1][j] = v[0][j];
            }
            if (nw == 1 || nw == 5)
            {
                v[i + 1][j] = v[0][j] + d[j];
            }
            if (nw == 2 || nw == 4)
            {
                v[i + 1][j] = v[0][j] + 2 * d[j];
            }
            if (nw == 3)
            {
                v[i + 1][j] = v[0][j] + 3 * d[j];
            }
            nw++;
            nw %= 7;
        }
    }
    for (s = 1; s <= 7; s++)
        build(1, 1, n);
    for (iny i = 1; i <= n / 2; i++)
    {
        swap(v[0][i], v[0][n - i + 1]);
        swap(d[i], d[n - i + 1]);
    }
    for (iny i = 0; i <= 6; i++)
```

```
    {
        iny nw = i;
        for (iny j = 1; j <= n; j++)
        {
            if (nw == 0 || nw == 6)
            {
                v[i + 8][j] = v[0][j];
            }
            if (nw == 1 || nw == 5)
            {
                v[i + 8][j] = v[0][j] + d[j];
            }
            if (nw == 2 || nw == 4)
            {
                v[i + 8][j] = v[0][j] + 2 * d[j];
            }
            if (nw == 3)
            {
                v[i + 8][j] = v[0][j] + 3 * d[j];
            }
            nw++;
            nw %= 7;
        }
    }
    for (s = 8; s <= 14; s++)
        build(1, 1, n);
    scanf("%d", &q);
    while (q--)
    {
        iny l, r;
        cin >> l >> r;
        if (l <= r)
        {
            s = (700000 + 2 - l) % 7;
            if (s == 0)
                s = 7;
        }
        else
        {
            s = (700000 + l - n + 1) % 7;
            if (s == 0)
                s = 7;
            s += 7;
            l = n - l + 1;
            r = n - r + 1;
        }
        node ans = query(l, r, 1, 1, n);
        printf("%d\n", ans._Max);
    }
    return 0;
}
```

M. Zipline

因为绳索的总长度f是一定的,所以人滑行到不同位置x的时候距离地面的高度l是不同的.

$(0 \le x \le w, r \le l \le min(g,h))$

设绳索总长度为$f(x,l) = \sqrt{(g-l)^2 + x^2} + \sqrt{(h-l)^2 + (w-x)^2}$

可知在x一定的情况下,l越大f越小;l越小f越大。

所以最小长度时$l = min(g,h)$,最大长度时$l = r$.

最小长度时x的位置为高度较小的端点处(根据三角形两边之和大于第三条边)。

此时$f = \sqrt[2]{w^2 + (h-g)^2}$。

最大长度时x的位置即x取何值时$f(x) = \sqrt[2]{(g-r)^2 + x^2} + \sqrt[2]{(h-r)^2 + (w-x)^2}$最大

求出$f(x)$导数，使得导数为0的点即为最大值点。

因为$f(x)$导数为单调函数，所以可以采用二分法找到导数值为0的点。

然后带回原方程即可得解。

```
#include<bits/stdc++.h>

using namespace std;
typedef long long ll;

int main() {
    int n;
    cin >> n;
    while (n--) {
        ll w, g, h, r; // 注意数据范围,平方会爆int
        cin >> w >> g >> h >> r;
        // 求长度的最小值：l = min(g,h)，且人处于高度较低处
        double mi = sqrt((w * w) + (h - g) * (h - g));

        // 求长度的最大值: l = r,二分找出导数为0的点
        g = g - r;
        h = h - r;
        double L = 0, R = w;
        while (R - L > 1e-8) // 设置精度一般比题目要求多两位
        {
            double x = (R + L) / 2;
            double f = x / sqrt(g * g + x * x) - (w - x) / sqrt(h * h + (w - x)
* (w - x));
            if (f < 0) L = x;
            else R = x;
        }
        // 带回原方程求解
        double ma = sqrt(g * g + L * L) + sqrt(h * h + (w - L) * (w - L));
        printf("%.8lf %.8lf\n", mi, ma);
    }
}
```