2023暑期CSP-S/NOIP模拟赛9 题解

T1 镜像字符串

设 $S[l,r]$ 表示字符串 S 第 l~r 位形成的子串 （若 l > r 表示 $S[r,l]$ 的反串）。

对于 20% 的做法：

枚举所有的前缀 $S[1,i]$ ，按照题意模拟整个过程，以 i 为对称轴，得到新的传 $S'[1,2i-1]$ ，并与串 $S[1,2i-1]$ 对所有位置进行一一对比。再对 $S'[1,2i-1]$ 递归的进行模拟，直到能够覆盖整个串 S。

时间复杂度：$O(|S|^3)$

对于 40% 的做法：

在对比的过程使用哈希来替代直接一一对比。

时间复杂度：$O(|S|^2)$

对于 100% 的做法：

考虑每个满足条件的初始串 $S[1,i]$，如果满足两种情况，我们称 i 是好的：

- $2i-1 \geq n$ ，对于这种情况，初始串翻转一次就可以满足条件；
- $2i-1 < n$ ，对于这种情况，初始串需要需要多次翻转

对于第二种情况，其本质是一个迭代的过程。假设他翻转了 t 次，每次被翻转的串是 $S[1,p_i]$ ，那么在最后一次翻转时，串 $S[1,p_t]]$ 满足第一种情况。而 $S[1,p_{t-1}]$ 翻转得到了 $S[1,p_t]$ ，也就是当 $S[1,p_{t-1}]$ 翻转后得到串对应 p_t 是好的时候，$S[1,p_{t-1}]$ 对应的 p_{t-1} 也是好的。

因此我们可以从后往前来考虑每个前缀，如果他翻转后仍然是好的，那么这个位置也是好的（有边界情况）。

具体的实现：

- 对于 $2i-1 \geq n$ ，只需要满足 $S[i,n] = S[i,2i-n]$
- 对于 $2i-1 < n$ ，满足 $2i-1$ 是好的，还需要满足 $S[i,2i-1] = S[i,1]$

使用字符串哈希判断即可。

时间复杂度：$O(n)$

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1000006;

struct Hash {
    int n;
    int base, P;
    int bin[N], hash[N];
    Hash() {}
    void prepare(int _base, int _P, char *s) {
        base = _base;
```

```
        P = _P;
        n = strlen(s + 1);
        bin[0] = 1;
        hash[0] = 0;

        for (int i = 1; i <= n; ++i) {
            hash[i] = (1LL * hash[i - 1] * base % P + s[i]) % P;
            bin[i] = 1LL * bin[i - 1] * base % P;
        }
    }
    int get_hash(int l, int r) {
        if (l > r) return get_hash(n + 1 - l, n + 1 - r);
        return (hash[r] - 1LL * hash[l - 1] * bin[r - l + 1] % P + P) % P;
    }
} h, r;

int n;
char st[N];
bool ye[N];

const int _base = 233;
const int _P = 998244353;

void work() {
    cin >> (st + 1);
    n = strlen(st + 1);
    h.prepare(_base, _P, st);
    for (int i = n >> 1; i; --i) swap(st[i], st[n - i + 1]);
    r.prepare(_base, _P, st);
    for (int i = 1; i <= n; ++i) ye[i] = 0;
    ye[n] = 1;
    for (int i = n - 1; i >= 1; --i) {
        if (i * 2 - 1 >= n) {
            if (h.get_hash(i, n) == r.get_hash(i, 2 * i - n)) ye[i] = 1;
        } else {
            if (ye[2 * i - 1] && h.get_hash(i, 2 * i - 1) == r.get_hash(i, 1)) ye[i] = 1;
        }
    }
    for (int i = 1; i < n; ++i) if (ye[i]) cout << i << " ";
    cout << n << '\n';
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int T;
    cin >> T;
    while (T--) work();
```

```
    return 0;
}
```

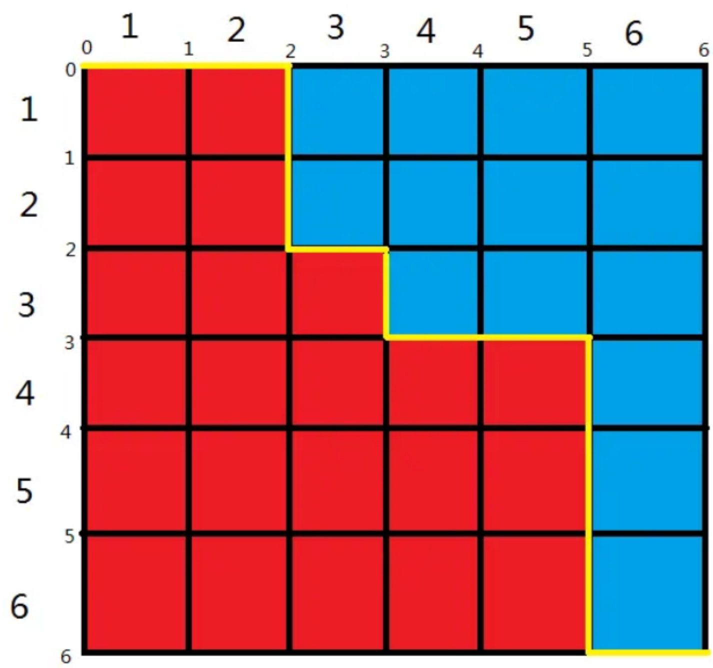
T2 田园生活

红格代表被豌豆洒水器喷洒到的方格，蓝格代表被玉米洒水器喷洒到的方格。

对于 20% 的数据：

定义：(a,b) 指网格线的交点，[a,b] 指网格。

仔细想想，可以发现红蓝双方的占据的格子是被一条从点(0,0)到(n,n)的分割线分开的。如图中的黄色线所示。



然后考虑哪些地方需要洒水器？可以发现，在分割线的拐角处一定要放置一个洒水器（否则拐角处无法被覆盖），其他的地方可以填也可以不填。而对于这个分割线，假设有 k 个拐角（一定要放的位置），方阵的没有被占据的总面积为 S，那么这种分割线总共有 2^{S-k} 种方案。

用 2^{2n} 去枚举分割线的走势，然后去计算有多少个拐角。

时间复杂度：$O(n*2^{2n})$

对于 40% 的数据：

考虑使用 dp 来计算这个方案数，设 $dp[i][j][k][0/1]$ 表示到达点 (i,j)，一共有 k 个拐角，从上一个点到当前点延伸方向为横向(0)/纵向(1)的分割线总共有多少条。

不产生拐角：

- $dp[i][j][k][0] -> dp[i][j+1][k][0]$
- $dp[i][j][k][1] -> dp[i+1][j][k][1]$

拐角转移，需要注意生成拐角的位置不能被禁止放洒水器：

- $dp[i][j][k][0] -> dp[i+1][j][k+1][1]$
- $dp[i][j][k][1] -> dp[i][j+1][k+1][1]$

最后的答案 $dp[n][n][k][0/1]$ 的每个 k 统计答案。

时间复杂度: $O(n^3)$

对于 100% 的做法：

考虑采用直接计算的方式，将中间的 k 维优化掉。

设 $dp[i][j][0/1]$ 表示到达点 (i,j) ，已经将前 i 行灌溉好了，从上一个点到当前点延伸方向为横向(0)/纵向(1)时候的总方案数。最后的答案就是：$dp[n][n][0]+dp[n][n][1]$ 。

对于 $dp[i][j][0]$ ，是从同一行转移过来，不会增加当前总的面积 S：

- 从 $dp[i][j-1][0]$ 转移过来，不会产生新的拐点，$dp[i][j][0]=dp[i][j-1][0]$
- 从 $dp[i][j-1][1]$ 转移过来，会产生一个新的拐点，需要新放置一个灌溉器，原来的方案数 2^{S-k} 因为多放置一个灌溉器原来的方案数要变成原来的 $\frac{1}{2}$ ，所以 $dp[i][j][0]=\frac{dp[i][j-1][1]}{2}$

综合起来就是：$dp[i][j][0]=dp[i][j-1][0]+\frac{dp[i][j-1][1]}{2}$

而对于 $dp[i][j][1]$ ，是从上一行转移过来，会增加当前总的面积 $S=S+sum_{i-1}$，其中 sum_i 表示第 i 行的能放置的格子数：

- 从 $dp[i-1][j][1]$ 转移过来，不会产生新的拐点，但增加了 S ，$dp[i][j][1]=dp[i-1][j][1]*2^{sum_i}$
- 从 $dp[i][j-1][0]$ 转移过来，会增加当前总的面积 S，同时也会会产生一个新的拐点，需要新放置一个灌溉器，原来的方案数 2^{S-k} 变成 $2^{S-k-1+sum_i}$ ，所以 $dp[i][j][1]=dp[i-1][j][0]*2^{sum_i-1}$

综合起来就是：$dp[i][j][1]=dp[i-1][j][1]*2^{sum_i}+dp[i-1][j][0]*2^{sum_i-1}$ 。

同样的，拐角转移，需要注意生成拐角的位置不能被禁止放洒水器。

最后整体的时间复杂度：$O(n^2)$

```
#include <bits/stdc++.h>
#define N 2010
#define mod 1000000007
#define half 500000004
using namespace std;

inline void add(int &a, int b) {
    a = (0ll + a + b) % mod;
}

int mul(int a, int b) {
    return (1ll * a * b) % mod;
}

int n, dp[N][N][2], sum[N], poww[N * N];
char ch[N][N];

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
```

```
    poww[0] = 1;
    for (int i = 1; i <= n * n; i++) poww[i] = mul(2, poww[i - 1]);
    for (int i = 1; i <= n; i++) {
        cin >> (ch[i] + 1);
        for (int j = 1; j <= n; j++) sum[i] += (ch[i][j] == '.');
    }
    dp[0][0][0] = dp[0][0][1] = 1;
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            if (!i && !j) continue;
            if (j > 0) {
                add(dp[i][j][0], dp[i][j - 1][0]);
                if (ch[i][j] == '.')
                    add(dp[i][j][0], mul(dp[i][j - 1][1], half));
            }
            if (i > 0) {
                add(dp[i][j][1], mul(dp[i - 1][j][1], poww[sum[i]]));
                if (ch[i][j] == '.')
                    add(dp[i][j][1], mul(dp[i - 1][j][0], poww[sum[i] - 1]));
            }
        }
    }
    cout <<  (0ll + dp[n][n][0] + dp[n][n][1]) % mod << '\n';
    return 0;
}
```

T3 台风防范

对于 20% 的做法：

枚举每次被删除的边，按照路径长短从小到大将每条备用边替换被删除的边，使用并查集判断，直到有一条边满足条件。

时间复杂度：$O(nm)$

对于额外 20% 的做法：

对于一条链的情况，可以发现每一条备用边能够覆盖一段范围，这段范围的某一条边被删除，就可以使用这条备用来替代，所以只需要对每条边找到覆盖它的最小的备用是哪一条即可。

可以使用线段树等数据结构维护最小值信息。

时间复杂度：$O(mlog_2n)$

对于 100% 的做法：

~~从链转为树的情况，可以直接使用树链剖分来维护。~~

我们知道对于树中的每条边，能够替换他的是能覆盖他的距离最小的备用路径，可不可以考虑对每条树边，维护能覆盖到它的所有的边集？

对每个备用路径(u,v)，在 u 处打一个标记，在 v 处打一个标记，对每个点维护一个set，用来存储每条树边能够覆盖到它的备用边集。从下往上遍历整棵树，在第一次遇到标记的时候将加入备用边set中，表示一条路径的开始，第二次遇到的时候从set中删掉，表示一条路径到这里就结束了。对于一个节点的多个儿子节点，采用启发式合并将儿子信息合并至当前节点。

使用启发式合并遍历一个节点 u，我们按以下的步骤进行（轻重儿子的处理参考轻重链剖分）：

1. 先遍历 u 的轻（非重）儿子，并计算答案，但 **不保留遍历后它对 set 的影响**；
2. 遍历它的重儿子，**保留它对 set 的影响**；（也就是重儿子的信息不需要重复计算）
3. 再次遍历 u 的轻儿子的子树结点，加入这些结点的贡献，以得到 u 的答案。

对每条边，遍历到的时候取set中的最小值作为答案。

```
#include<bits/stdc++.h>
#define mk make_pair
using namespace std;
const int maxn = 5e4 + 10, mod = 1e9 + 7;;
int n, m;
vector<int> G[maxn];
set<pair<int, int> > p; //w, id
map<pair<int, int>, int> id; //(x, y) -> id
vector<pair<int, int> > flag[maxn]; // w, id
int ans[maxn], Size[maxn], Son[maxn];

// 预处理重儿子信息
void dfs1(int u, int f) {
  Size[u] = 1;
  Son[u] = -1;
  for(auto v: G[u]) {
    if(f == v) continue;
    dfs1(v, u);
    Size[u] += Size[v];
    if(Son[u] == -1 || Size[Son[u]] < Size[v]) Son[u] = v;
  }
}

void update(int u, int f, int op){
  for(auto v: G[u]) {
    if(v == f) continue;
    update(v, u, op);
  }
  if(op == 1) { // 添加信息
    for(auto x: flag[u]) {
      if(p.find(x) != p.end()) p.erase(x);
      else p.insert(x);
    }
  }
  else {
    for(auto x: flag[u]) {
      if(p.find(x) != p.end()) p.erase(x);
```

```
    }
  }
}

// 保留重儿子的信息，轻儿子的信息计算后清除掉
void dfs(int u, int f, bool keep) {
  // 递归计算轻儿子的答案，计算后清除掉
  for(auto v: G[u]) {
    if(f == v || v == Son[u]) continue;
    dfs(v, u, 0);
  }
  // 递归计算重儿子信息并保留下来
  if(Son[u] != -1) dfs(Son[u], u, 1);
  // 添加轻儿子的信息
  for(auto v: G[u]) {
    if(v == f || v == Son[u]) continue;
    update(v, u, 1);
  }
  // 添加当前点的信息
  for(auto x: flag[u]) {
    if(p.find(x) != p.end()) p.erase(x);
    else p.insert(x);
  }
  // 计算当前点信息
  int x = u, y = f;
  if(x > y) swap(x, y);
  if(id.count(mk(x, y))) {
    int pid = id[mk(x, y)];
    if(!p.empty()) ans[pid] = (p.begin())->first;
  }
  if(!keep) update(u, f, 0);
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> m;
  for(int i = 1; i < n; ++i) {
    int x, y;
    cin >> x >> y;
    G[x].push_back(y);
    G[y].push_back(x);
    if(x > y) swap(x, y);
    id[mk(x, y)] = i;
  }
  for(int i = 1; i <= m; ++i) {
    int x, y, z;
    cin >> x >> y >> z;
    flag[x].push_back(mk(z, i));
```

```
    flag[y].push_back(mk(z, i));
  }
  for(int i = 1; i < n; ++i) ans[i] = -1;
  dfs1(1, 0);
  dfs(1, 0, 0);
  for(int i = 1; i < n; ++i) cout << ans[i] << '\n';
  cerr << 1.0 * clock() / CLOCKS_PER_SEC << endl;
  return 0;
}
```

另外的 100% 的做法：

按照边权大小从小到大使用备用边去覆盖原来的树边，使用并查集去维护已经被覆盖的边，由于按照边权大小进行添加，重复添加不会使得答案变优。

```
#include <bits/stdc++.h>
const int maxn = 5e4 + 7;
using namespace std;
int cnt = 0, tot = 0;
int Size[maxn], id[maxn], Son[maxn], tp[maxn], dep[maxn], fa[maxn], f[maxn];
int head[maxn << 1], mins[maxn];

struct node {
    int num, next1;
} ed[maxn << 1];

int getfa(int x) {
    return fa[x] == x ? x : fa[x] = getfa(fa[x]);
}

void add(int x, int y) {
    cnt++;
    ed[cnt].num = y;
    ed[cnt].next1 = head[x];
    head[x] = cnt;
}

void dfs0(int x, int fa1) {
    Size[x] = 1;
    int x1 = x, maxs = 0;
    for (int i = head[x]; i; i = ed[i].next1) {
        int v = ed[i].num;
        if (v != fa1) {
            f[v] = x;
            dep[v] = dep[x] + 1;
            dfs0(v, x);
            Size[x] += Size[v];
            if (Size[v] > maxs) {
                maxs = Size[v];
```

```
                x1 = v;
            }
        }
    }
    Son[x] = x1;
}

void dfs1(int x, int fa1) {
    int x1 = Son[x];
    id[x] = ++tot;
    if (x == x1) return;
    tp[x1] = tp[x];
    dfs1(x1, x);
    for (int i = head[x]; i; i = ed[i].next1) {
        int v = ed[i].num;
        if (v != x1 && v != fa1) {
            tp[v] = v;
            dfs1(v, x);
        }
    }
}

int lca(int x, int y) {
    while (tp[x] != tp[y]) {
        if (dep[tp[x]] < dep[tp[y]]) swap(x, y);
        x = f[tp[x]];
    }
    if (dep[x] > dep[y]) swap(x, y);
    return x;
}

struct node1 {
    int x1, y1, w1;
} d[maxn];

bool cmp(node1 x, node1 y) {
    return x.w1 < y.w1;
}

void dfs(int x, int y, int d) {
    while (dep[y] > dep[x]) {
        int y1 = getfa(y);
        if (dep[y1] <= dep[x]) break;
        if (y1 == y) mins[y] = d;
        fa[y] = fa[f[y]];
        y = getfa(y);
    }
}

```

```cpp
int x[maxn], y[maxn];
int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
    int n, m, x2, y2, w2;
  cin >> n >> m;
    for (int i = 1; i <= n; i++) fa[i] = i;
    for (int i = 1; i < n; i++) {
    cin >> x[i] >> y[i];
        add(x[i], y[i]);
        add(y[i], x[i]);
    }
    dfs0(1, 0);
    dfs1(1, 0);
    for (int i = 1; i <= m; i++) {
    cin >> x2 >> y2 >> w2;
        d[i].x1 = x2;
        d[i].y1 = y2;
        d[i].w1 = w2;
    }
    sort(d + 1, d + 1 + m, cmp);
    for (int i = 1; i <= m; i++) {
        int x3 = d[i].x1, y3 = d[i].y1, w2 = d[i].w1;
        int p = lca(d[i].x1, d[i].y1);
        dfs(p, x3, w2);
        dfs(p, y3, w2);
    }
    for (int i = 1; i < n; i++) {
        if (dep[x[i]] < dep[y[i]]) swap(x[i], y[i]);
        if (mins[x[i]]) cout << mins[x[i]] << '\n';
        else cout << "-1" << '\n';
    }
    return 0;
}
```

T4 体操表演

对于额外 10% 的做法：

所有 $f(k)$ 相等，每一个位置得到的奖励相同，期望的奖励就是 $f(k)$ ，直接输出 $f(k)$ 就可以了。

对于 20%~40% 的做法：

假设 $E(x)$ 表示 Alice 在 x 所能得到的最大期望奖励，$E_0=0, E_{n+1}=0$，那么有：

$$E_i=\max\left(\frac{E_{i-1}+E_{i+1}}{2}, f_i\right) \quad (1 \leqslant i \leqslant n)$$

观察这个期望式子我们可以得知，在一些位置移动的期望收益比当前位置停止的收益低，即在题目所述的条件下移动，一旦到达这些点，最优的选择就是待在这个点（也即等式的 max 取值为 f_i ）。

考虑所有满足 $E_i = f_i$ 的位置的集合 $S = \{x|E(x) = f_x\}$，对于集合中两个相邻的元素 p, q $(p < q)$，所有夹在他们中间的点 $x \in (p, q)$ 满足的式子为：$E_x = \frac{E_{x-1}+E_{x+1}}{2}$，也即 $E(x) - E(x-1) = E(x+1) - E(x)$，也就是一个等差数列。而对于这些位置 $x \in (p, q)$，他们最优的选择一定是移动到 p 或者 q，并且停留在那里。也即对于这些位置，最终的 $E(x) = prob(x- > p) * E(p) + prob(x- > q) * E(q)$。

现在我们得到了一个优秀的做法，就是对于每个点，找到其前后的第一个停止点，得到其移动的期望收益然后与自己的停止收益取最大值。

一个结论：在长度为 L 的数轴上的位置 x 处，每次进行左右移动（左右概率都为0.5），若到达 0 或 L 即停止，则到达0停止的概率为 $\frac{L-x}{L}$，到达 L 停止的概率为 $\frac{x}{L}$。

证明：设 f_i 表示从 i 开始到 L 停止的概率，则 $f_i = \frac{f_{i-1}+f_{i+1}}{2}$，这个是一个等差数列，其中 $f(0) = 0, f(1) = 1$，可以得到 $f_i = \frac{x}{L}$。对应的到 0 停止的概率为 $\frac{L-x}{L}$。

对于所有点 x，$E(x) = f_p * \frac{q-x}{q-p} + f_q * \frac{x-p}{q-p}$。

对每个点 x 直接枚举两个点（满足位置关系）作为停止点（因为真正的停止点的答案一定优于非停止点），计算最优的答案。

时间复杂度：$O(n^2)$

对于 100% 的做法：

现在的问题是，如何求出所有的停止点？

分析 E(x) 的形式，其实 (x, E(x)) 是经过 $(p, f_p), (q, f_q)$ 两点的直线上的一个点。

要使得 E(x) 最大，不难发现 (x,E(x)) 一定在点集 (i, f_i) 形成的凸包上。

时间复杂度：$O(n)$

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
const int N = 101000, bs = 1e5;
int n, tp;

struct vec {
    int x; LL y;

    vec() {}

    vec(const int &X, const LL &Y): x(X), y(Y) {}

    friend inline vec operator - (const vec &A, const vec &B) {
        return vec(A.x - B.x, A.y - B.y);
    }

    friend inline LL operator * (const vec &A, const vec &B) {
        return A.x * B.y - A.y * B.x;
    }
} p, st[N];

```

```
void push(vec p) {
    while (tp && (p - st[tp]) * (st[tp] - st[tp - 1]) <= 0) --tp;
    st[++tp] = p;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
    vec p;
  push(vec(0, 0));
    for (int i = 1; i <= n; ++i) {
    p.x = i, cin >> p.y, p.y *= bs, push(p);
  }
    push(vec(n + 1, 0));
    for (int i = 1, j = 0; i <= n; ++i) {
        while (j < tp && st[j].x < i) ++j;
        if (st[j].x == i) cout << st[j].y << '\n';
        else
            cout<< ((st[j].x - i)*st[j - 1].y + (i - st[j - 1].x)*st[j].y) / (st[j].x - st[j - 1].x) << '\n';
    }
    return 0;
}
```