2023暑期CSP-S/NOIP模拟赛8题解

T1 男女排队

20%的做法：

使用 2^n 进行枚举序列，最后进行判断

时间复杂度：$O(n \times 2^n)$

60%的做法：

第一种方法：

使用 dp 进行计数，考虑什么状态是我们需要考虑到的：设 $f[i][j][k]$ 表示到第 i 位为止，倒数第二位和最后一位分别是 j(0/1) 和 k(0/1) 的和法方案数。

可以得到以下 dp 转移方程：

当这一位是 0 时：

$f[i][0][0]=f[i-1][0][0]+f[i-1][1][0]$

$f[i][1][0]=f[i-1][0][1]+f[i-1][1][1]$

当这一位是1时：

$f[i][0][1]=f[i-1][0][0]$，其中$f[i-1][1][0]$不合法；

$f[i][1][1]=f[i-1][0][1]$，其中$f[i-1][1][1]$不合法；

时间复杂度：$O(4n)$

第二种方法：

设$f[i]$表示长度为i的合法方案数。

如果第n位，放0，那么前面$n-1$位，怎么放都可以。$f[n]$可以从$f[n-1]$转移过来。

如果第n位，放1，此时有可能出现非法

- 如果第$n-1$位放0，第$n-2$位只能放0，那么前面$n-3$位，怎么放都可以。$f[n]$可以从$f[n-3]$转移过来。
- 如果第$n-1$位放1，第$n-2$位只能放0，此时第$n-3$位必须放0，那么前面$n-4$位，怎么放都可以。$f[n]$可以从$f[n-4]$转移过来。

$f[n]=f[n-1]+f[n-3]+f[n-4]$

时间复杂度 $O(n)$

80%的做法：

对第二种方法，对于 $n <= 2e9$，$f[i]=f[i-1]+f[i-3]+f[i-4]$ 可以分块打表，取块的大小为 $p=1e5$ 。

首先，预处理计算 1-P 的 f[i]，并从通过两个 P 的块合并来得到所有 i%P=0 时的 f[i],f[i-1],f[i-2],f[i-3] 的值。最后从 $n/P*P+1$处开始递推。

100%的做法

对于两种方法，都可以使用矩阵乘法优化推导：

第一种方法：

不妨令 $f[i][0], f[i][1], f[i][2], f[i][3]$ 表示 $f[i][0][0], f[i][0][1], f[i][1][0], f[i][1][1]$。

那么递推式变为：

当这一位是 0 时：

$f[i][0] = f[i-1][0] + f[i-1][2]$

$f[i][2] = f[i-1][1] + f[i-1][3]$

当这一位是1时：

$f[i][1] = f[i-1][0]$

$f[i][3] = f[i-1][1]$

那么矩阵递推式为：

$$\begin{bmatrix} f[n][0] \\ f[n][1] \\ f[n][2] \\ f[n][3] \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}^{n-2} \times \begin{bmatrix} f[n-1][0] \\ f[n-1][1] \\ f[n-1][2] \\ f[n-1][3] \end{bmatrix}$$

第二种方法：

$$[f[1] \quad f[2] \quad f[3] \quad f[4]] \times \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [f[2] \quad f[3] \quad f[4] \quad f[5]]$$

$$[f[1] \quad f[2] \quad f[3] \quad f[4]] \times \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}^{n-4} = [f[n-3] \quad f[n-2] \quad f[n-1] \quad f[n]]$$

代码是第一种方法的：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
const int N = 1e7 + 10, mod = 1e4 + 7;;
LL n;
int f[2][2][2];

inline int fadd(int x, int y) {
  if(x + y >= mod) return x + y - mod;
  return x + y;
}

```

```cpp
struct Matrix {
    int num[4][4], n;

  Matrix(int x) { n = x, memset(num, 0, sizeof num); }

  Matrix(int x, int y) {
    n = x;
    memset(num, 0, sizeof num);
    for(int i = 0; i < n; ++i) num[i][i] = y;
  }

  Matrix(int x, int mtr[]) {
    n = x;
    for(int i = 0; i < n; ++i)
      for(int j = 0; j < n; ++j)
        num[i][j] = mtr[i * n + j];
  }

    Matrix operator *(const Matrix& o) const {
        Matrix ret(4);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                for (int k = 0; k < n; ++k)
                    ret.num[i][j] = fadd(ret.num[i][j], num[i][k] * o.num[k][j] % mod);
        return ret;
    }

    Matrix operator ^(LL power) {
        Matrix tmp = *this;
        Matrix ret = Matrix(4, 1);
        while (power) {
            if (power & 1)ret = ret * tmp;
            tmp = tmp * tmp;
            power >>= 1;
        }
        return ret;
    }

  void Out() {
    for(int i = 0; i < n; ++i) {
      for(int j = 0; j < n; ++j) cout << num[i][j] << " ";
      cout << '\n';
    }
  }
};


int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
  int mat1[16] {
```

```
    1, 0, 0, 0,
    1, 0, 0, 0,
    1, 0, 0, 0,
    1, 0, 0, 0
  };
  Matrix f2(4, mat1);
  int mat2[16] {
    1, 0, 1, 0,
    1, 0, 0, 0,
    0, 1, 0, 1,
    0, 1, 0, 0
  };
  Matrix A(4, mat2);
  Matrix fn = (A ^ (n - 2)) * f2;
  int ans = 0;
  for(int i = 0; i < 4; ++i) ans = fadd(ans, fn.num[i][0]);
  cout << ans << '\n';
  return 0;
}
```

T2 树上最多不相交路径

20%的做法（subtask 1+2）：

使用 2^m 枚举路径是否被选择，对选择了的路径，可以采用搜索+栈的方式给对应路径标记出来。

时间复杂度：$O(n*2^m)$

额外20%的做法 (subtask 3)：

使用 2^m 枚举路径是否被选择，对选择了的路径，使用二进制位 0/1 表示这些点是否已经经过。

对于经过了哪些点，可以搜索进行预处理并使用 bitset 进行记录。

时间复杂度：$O(2^m * \frac{n}{32})$

额外20%的做法 (subtask 5)：

对一条链的情况，可以考虑贪心，按照从下往上的顺序或者从上往下的顺序进行贪心。

每个路径有一个深度小的端点，有一个深度大的端点，路径相当于一个区间，使得路径之间两个端点的深度之间不相交。这是一个经典的区间问题。

按照深度大的端点排序进行贪心即可。

时间复杂度：$O(nlog_2n)$

100%的做法：

考虑对于树的情况，能否从链的情况扩展过来，路径 (u,v) 能够影响到的点是从 u 和 v 开始一直到他们的 lca(u,v)，所有经过这些点的路径在选择了这条路径后，都会被舍弃。

从 lca(u,v) 的角度来考虑，两条路径相交，一定是某条路径的 lca 在令一条路径上。

选择一条路径之后，它的lca对应的子树中的点，是不能在通过lca走向子树外的点。

从贪心的角度需要将影响的范围尽量变小，可以按照从下往上的顺序考虑，按照lca的深度来选取。

具体实现：选取一条路径之后，把它lca内的所有节点都标记掉，表示不可以在使用。一个点如果已经被标记了，就不用在继续往下标记了。这样每个点，只会被标记一次。

时间复杂度：$O(nlog_2n)$

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e5 + 10;
int n, m, f[maxn][20], d[maxn];
bool mk[maxn];
vector<int> v[maxn];
struct Path {
    int x, y, lca, d;
    bool operator <(Path b)const {
        return d > b.d;
    }
} t[maxn];

void dfs(int x, int p) {
    f[x][0] = p;
    d[x] = d[p] + 1;
    for (int i = 0; i < v[x].size(); i++)
    if (v[x][i] != p) dfs(v[x][i], x);
}

void fill(int x, int p) {
    if (mk[x]) return;
    mk[x] = 1;
    for (int i = 0; i < v[x].size(); i++)
        if (v[x][i] != p) fill(v[x][i], x);
}

int LCA(int x, int y) {
    if (d[x] < d[y]) swap(x, y);
    for (int i = 18; ~i; i--)
        if ((1 << i) & (d[x] - d[y])) x = f[x][i];
    if (x == y) return x;
    for (int i = 18; ~i; i--)
        if (f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
    cin >> n >> m;
    for (int i = 1, x, y; i < n; i++) {
    cin >> x >> y;
        v[x].push_back(y);
        v[y].push_back(x);
    }
    for (int i = 1; i <= m; i++) cin >> t[i].x >> t[i].y;
```

```cpp
    dfs(1, 0);
    for (int i = 1; i <= 18; i++)
        for (int j = 1; j <= n; j++)
            f[j][i] = f[f[j][i - 1]][i - 1];
    for (int i = 1; i <= m; i++) {
        t[i].lca = LCA(t[i].x, t[i].y);
        t[i].d = d[t[i].lca];
    }
    sort(t + 1, t + m + 1);
    int ans = 0;
    for (int i = 1; i <= m; i++) {
        if (mk[t[i].x] || mk[t[i].y]) continue;
        ans++;
        fill(t[i].lca, f[t[i].lca][0]);
    }
    cout << ans << '\n';
}
```

T3 XB的生日

20%的做法：

对于 $n \le 5, T \le 10$ 的部分，可以直接搜索，找出所有合法方案。

额外20%的做法（T<=500）：

对于 $T \le 500$ 的部分，可以直接进行 dp ，按照时间段进行划分，并记录当前已有的原料。

设 $f[i][j][k]$ 代表到时间 i 截止，到了 j 号点，拥有 k 的原料的方案数（k为二进制表示）。

按照时间进行枚举，对于每个点 u ，考虑所有由他出发的边 (u,v,w)：

- 如果经过商店 $f[i][v][k|w]$ += $f[i-2][u][k]$ 。
- 如果不经过商店 $f[i][v][k] += f[i-1][u][k]$。

时间复杂度 $O(16nmT)$

额外20%的做法（n<=5）：

对于 $n \le 5$ 但 T 很大的部分，考虑到状态数较小的时候，如果能够固定下转移的方程，可以使用矩阵乘法来优化计算。

可以构造一个大矩阵，每个点有 16 种状态，表示达到当前点已经包含哪些字符。

由于进入商店需要花费 2 的时间，对每个状态可以再新建一个节点，经过这个节点代表从某个状态出发将会进入一个商店。也即某个状态经过一条边时去商店，先去新建的节点，再去边的终点（把长度为 2 的路径变成两条长度为 1 的）。如果不去就直接连向边的终点。

由于只需要在 T 时间内回到点 1，所有要在矩阵中记录一个前缀和。(具体的前缀和处理参考下面的方法)

这样矩阵的大小将会是 $c = 5 \times 16 \times 2 + 1 = 161$ ，时间复杂度将会是 $O(c^3 * log_2 T)$。

额外20%的做法（BJMP）：

对于每条边都包含"BMJP"的部分，可以考虑所有的方案减去完全不去商店的方案。

对于每个点 u，考虑所有由他出发的边 (u,v)，$f[i][j]$ 表示时间 i 的时候，在点 j 的方案数。

所有方案的计算：类似上一部分的分裂点构造矩阵（把长度为2的路径变成两条长度为1的路径)，使用矩阵加速计算。矩阵大小为 $c = 25 \times 2 + 1 = 51$ 。

- 经过商店 $f[i][v]$ += $f[i-1][u+n]$ 。
- 不经过商店 $f[i][v] += f[i-1][u]$。

由于只需要在 T 时间内回到点 1，所有要在矩阵中记录一个前缀和。

其中对于 t 时刻的矩阵，$s_{i,1}$ 记录的是 1-t 时刻所有从 i 到 1 就结束的方案数（t 的前缀和）。

$$\begin{bmatrix} 0 & 0 & \dots & 0 \\ s_{1,1} & f[i][1] & \dots & f[i][2n] \\ 0 & 0 & \dots & 0 \\ \dots & & & \\ 0 & 0 & \dots & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & f[i-1][1] & \dots & f[i-1][2n] \\ 0 & 0 & \dots & 0 \\ \dots & & & \\ 0 & 0 & \dots & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ s_{1,1} & a_{1,1} & 0 & \dots & a_{1,2n} \\ s_{2,1} & a_{2,1} & 0 & \dots & a_{2,2n} \\ \dots & & & & \\ s_{2n,1} & a_{2n,1} & 0 & \dots & a_{2n,2n} \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & \dots & 0 & \\ s_{1,1} & f_{T,1} & f_{T,2} & \dots & f_{T,2n} \\ 0 & 0 & \dots & 0 & \\ \dots & & & & \\ 0 & 0 & \dots & 0 & \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 & \\ 0 & f[0][1] & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \\ \dots & & & & \\ 0 & 0 & \dots & 0 & \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ s_{1,1} & a_{1,1} & 0 & \dots & a_{1,2n} \\ s_{2,1} & a_{2,1} & 0 & \dots & a_{2,2n} \\ \dots & & & & \\ s_{2n,1} & a_{2n,1} & 0 & \dots & a_{2n,2n} \end{bmatrix}^T$$

完全不经过商店：所有边都当做长度为1的路径。

- 不经过商店 $f[i][v] += f[i-1][u]$。

矩阵大小为 $c = 25 + 1 = 26$。

100%的做法：

考虑到总共的字符种类数比较少，参考上一部分的做法，可以考虑容斥来解决，考虑只经过特定的字符的方案。

所有的方案数先减去有 1 个字符一定不走的方案，再加上 2 种字符一定不走的情况，再减去有 3 个字符一定不走的方案，再加上 4 种字符一定不走的情况。

使用上一个部分分裂点构造矩阵的方法，需要注意的是某种字符不经过的时候，所有相关的边都不考虑即可构造出相应的矩阵。矩阵大小为 $c = 25 \times 2 + 1 = 51$ 。

时间复杂度：$O(2^4 c^3 log_2 T)$

```
#include <bits/stdc++.h>
using namespace std;
const int M = 55, N = 505, P = 5557;
int n, m, T, EA[N], EB[N], EC[N], mp[N];
int A[M][M], B[M][M];

void mult(int A[M][M], int B[M][M], int C[M][M]) {
    int T[M][M];
```

```cpp
    for (int i = 0; i <= 2 * n; i++)
        for (int j = 0; j <= 2 * n; j++) {
            T[i][j] = 0;
            for (int k = 0; k <= 2 * n; k++) T[i][j] += A[i][k] * B[k][j];
        }
    for (int i = 0; i <= 2 * n; i++)
        for (int j = 0; j <= 2 * n; j++)
            C[i][j] = T[i][j] % P;
}

int solve(int has) { // B * A ^ n
    memset(A, 0, sizeof(A));
    memset(B, 0, sizeof(B));
    A[0][0] = 1;
    for (int i = 1; i <= n; i++) A[i][i + n] = 1; // 添加经过商店的点 i+n
    for (int i = 1; i <= m; i++) {
        int a = EA[i], b = EB[i];
        A[a][b]++; // 不经过商店的
        if ((EC[i] | has) == has) A[a + n][b]++; //经过商店的 a -> a+n -> b
    }
    for (int i = 1; i <= 2 * n; i++) A[i][0] = A[i][1]; // 初始 s_i0 = a_i1
    B[1][1] = 1;
    for (int i = 0; (1 << i) <= T; i++) {
        if (T & (1 << i)) mult(B, A, B);
        mult(A, A, A);
    }
    return B[1][0] % P;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> m;
    mp['B'] = 1, mp['J'] = 2, mp['M'] = 4, mp['P'] = 8;
    for (int i = 1; i <= m; i++) {
        char str[9];
    cin >> EA[i] >> EB[i] >> str;
        for (int j = 0; j < strlen(str); j++) EC[i] |= mp[str[j]];
    }
  cin >> T;
    int ans = 0;
    for (int i = 0; i < 16; i++) {
        int f = 1;
        for (int j = 0; j < 4; j++) if ((i >> j) & 1) f = -f;
        ans = (ans + solve(i) * f + P) % P;
    }
  cout << ans << '\n';
    return 0;
}
```

T4 组队比赛

20%的做法：

使用搜索+剪枝来实现枚举每个组包含哪些人。

注意到将 n 个人分到不同的 k 个集合中是第二类斯特林数，在 n=15 的情况下，最大的结果是 5e8.

可以直接写搜索或者打表记录下答案。

直接搜索实现方法：按照第二类斯特林数的递推方法，dfs(i,j)表示现在考虑了 i 个人，分成了 j 个集合，然后考虑将 i+1 放在前 j 个集合还是新加的第 j+1 个集合。可以使用数组来记录具体集合划分的情况。

时间复杂度: $O(S(n,k))$

40%-60%的做法：

一些前提

对于一个组内的游戏时间，若加入一个新的人，则有三种情况:
①:组游戏时间不变(新人的空闲时间完整包含原组的游戏时间)
②:组游戏时间减短(新人的空闲时间部分包含原组的游戏时间)
③:组游戏时间变为 0 ，即非法(新人的空闲时间完全不与原组的游戏时间重合)

我们将所有人保存在数组 p 中($p_i.l$ 表示第 i 个人的起始游戏时间， $p_i.r$ 表示第 i 个人的终止游戏时间)。
我们根据一个人的游戏时间是否包含他人的游戏时间分为两类 $p1与p2(p1表示包含他人,p2表示不包括)$，那么显然， $p1$ 中的人的分组只需要从两种情况中考虑:
①:单人一组(组内游戏时间即他自己的游戏时间)
②:与他所包含的人一组(组内游戏时间只会被他完整包含，即他加入时不会对组游戏时间造成影响)
若我们知道 $p2$ 中的人分为 i 组时最优解，则此时 $p1$ 中选出游戏时间最长的 $k-i$ 人单独分组，剩下的 $p1$ 中的人与其包含的人同组，便是此时的最优解。

那么，我们将问题转化为求 $p2$ 中的人的分组情况。
对于 $p2$ 中的人我们按照起始游戏时间按从小到大进行排序(则此时每个人的终止游戏时间也是从小到大，因为若存在 p_i, p_j 满足 $p_i.l <= p_j.l \&\& p_i.r >= p_j.r$ ，则 $p_i包含p_j$ ，不会被分到 $p2$ 中)，显然在分组时，最优分组的组内成员一定是连续的(若不连续则定然同时有两组或以上不连续，此时交换成员可以得到最优解)。

具体做法：

首先我们对每个人的起始游戏时间按从小到大进行排序，然后分类 $p1与p2$ ($n1$ 和 $n2$ 表示数组大小)。
我们用 sum_i 存储 $p1$ 中游戏时间前 i 大的人的游戏时间总和。
我们定义 $dp_{i,j}$ 表示 $p2$ 中的前 i 个人分成 j 组时的游戏时间总和的最大值。
那么当总人数增加时，我们考虑最后一组包含哪些人，然后进行状态转移。
对于 $dp_{i,j}$ ，我们枚举最后一组的人数 k (上文已说明最优解的单组成员应是连续的)，然后从 $dp_{i-k,j-1}$ 转移，加上该组的游戏时间(注意本组游戏时间须合法)，取最大值。
状态转移方程如下:

$$dp_{i,j} = \max_{k=1}^{i-k\geq j-1} dp_{i-k,j-1} + p2_{i-k+1}.r - p2_i.l(p2_{i-k+1}.r > p2_i.l)$$

对于最终答案的处理，我们枚举 $p2$ 中分 i 组的情况的最优解加上 $p1$ 中游戏时间前 $k-i$ 长的人的游戏时间总和的最大值，若 $p2$ 中分任意组都不合法，则无解。

$$ans = \max_{i=0}^{\min(k,n1)} sum_i + dp_{n2,k-i}$$

最后的时间复杂度：$O(n^3)$，但实际是不到 n^3。

100%的做法：

对于上面的转移方程：

$$\mathrm{dp_{i,j}} = \max_{\mathrm{k=1}}^{\mathrm{i-k\geq j-1}} \mathrm{dp_{i-k,j-1}} + \mathrm{p2_{i-k+1}.r} - \mathrm{p2_i.l}(\mathrm{p2_{i-k+1}.r} > \mathrm{p2_i.l})$$

对于 $dp_{i-k,j-1} + p2_{i-k+1}.r$ 的值，对于 i 从小到大枚举的过程中，可以对 k 维护维护一个最大值的单调队列（递减），将 k 这一维优化掉。

时间复杂度：$O(n^2)$

```
#include <bits/stdc++.h>
using namespace std;
const int SIZE = 8005;
int n, m, K;
struct Node {
    int L, R;
    bool operator < (const Node& x1) const {
        if (L == x1.L) return R > x1.R;
        return L < x1.L;
    }
} a[SIZE], b[SIZE];
int dp[SIZE][SIZE];
int Q[SIZE], hed, tal;
int len[SIZE], N;

int main () {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> K;
  for (int i = 1; i <= n; ++i) cin >> a[i].L >> a[i].R;
    sort(a + 1, a + n + 1);
    int mi = 2e9;
  for (int i = n; i; --i) {
        if (mi <= a[i].R) len[++N] = a[i].R - a[i].L;
        else mi = a[i].R, b[++m] = a[i];
    }
    sort(len + 1, len + N + 1, greater<int>());
  memset(dp, -1, sizeof dp);
    dp[0][0] = 0;
    reverse(b + 1, b + m + 1);
  for (int i = 1; i <= K; ++i) {
        hed = 1, tal = 0;
    for (int j = 1; j <= m; ++j) {
            if (~dp[i - 1][j - 1]) {
                while (hed <= tal && dp[i - 1][Q[tal]] + b[Q[tal] + 1].R <= dp[i - 1][j - 1] + b[j].R) --tal;
                Q[++tal] = j - 1;
            }
            while (hed <= tal && b[Q[hed] + 1].R <= b[j].L) ++hed;
            if (hed <= tal) dp[i][j] = dp[i - 1][Q[hed]] + b[Q[hed] + 1].R - b[j].L;
```

```
        }
    }
    int ans = 0, sum = 0;
  for (int i = 0; i <= min(K, N); ++i) {
        sum += len[i];
        if (~dp[K - i][m]) ans = max(ans, sum + dp[K - i][m]);
    }
  cout << ans << '\n';
}
```