Dinic算法

邻接表+dfs递归版本

```
struct edge
{
    int to, cap, rev;           //rev记录对应反向边在对应邻接表的下标
};
vector<edge> G[maxn];
int level[maxn], iter[maxn];        //level记录bfs时与s的距离，iter记录当前弧

void add_edge(int from, int to, int cap)
{
    G[from].push_back((edge){to, cap, G[to].size()});
    G[to].push_back((edge){from, 0, G[from].size()-1});
}

void bfs(int s, int t)
{
    queue<int> que;
    memset(level, -1, sizeof(level));
    level[s] = 0;
    que.push(s);
    while(!que.empty())
    {
        int v = que.front(); que.pop();
        for(int i = 0; i < G[v].size(); i++)
        {
            edge &e = G[v][i];
            if(e.cap > 0 && level[e.to] < 0)    //其可达并且第一次到达
            {
                level[e.to] = level[v] + 1;
                if(e.to == t) return;
                que.push(e.to);
            }
        }
    }
}

int dfs(int v, int t, int f)            //从v到t的最大流
{
    if(v == t) return f;
    for(int &i = iter[v]; i < G[v].size(); i++)       //弧优化
    {
        edge &e = G[v][i];
        if(e.cap > 0 && level[v] < level[e.to])       //可达并且是下一层
        {
            int d = dfs(e.to, t, min(f, e.cap));
            if(d > 0)
            {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
```

```
    }
    return 0;
}

int max_flow(int s, int t)
{
    int flow = 0, f;
    for(;;)
    {
        bfs(s, t);                  //首先bfs构造分层图
        if(level[t] < 0) return flow;          //t不可达直接退出
        memset(iter, 0, sizeof(iter));
        while((f = dfs(s, t, INF)) > 0)        //每次增加一条最短增广路
            flow += f;
    }
}
```

kuangbin优化版：1.采用链式前向星代替vector 2.手写队列与栈节约bfs与dfs时间 注意：head一开始时需要初始化成-1，链式前向星进行dfs要确定跳出的节点

```
struct edge
{
    int to, flow, cap, next;
}e[maxm];

int tol, head[maxn];                      //用链式前向星来记录
void add_edge(int u,int v, int w)
{
    e[tol].to = v; e[tol].flow = 0; e[tol].cap = w;
    e[tol].next = head[u]; head[u] = tol++;
    e[tol].to = u; e[tol].flow = w; e[tol].cap = w;
    e[tol].next = head[v]; head[v] = tol++;
}

int level[maxn], Q[maxn];               //level记录分层，Q为手写队列
void bfs(int s, int t, int n)
{
    memset(level, -1, sizeof(level[0]) * n);      //一开始初始化为-1
    int k1 = 0, k2 = 0;
    level[s] = 0;
    Q[k2++] = s;
    while(k1 < k2)
    {
        int u = Q[k1++];
        for(int i = head[u]; i != -1; i = e[i].next)
        {
            int v = e[i].to;
            if(e[i].cap > e[i].flow && level[v] == -1)   //可达并且是第一次到达
            {
                level[v] = level[u] + 1;
                if(v == t) return;                     //如果是t则bfs完成
                Q[k2++] = v;
            }
        }
    }
    return;
```

```
}

int sta[maxn], cur[maxn];              //sta记录栈（对应的边在e里的下标）， cur来复刻
head
int dfs(int s, int t, int n)
{
    int maxflow = 0, tail = 0, u = s;       //tail记录栈的长度，u来记录当前遍历到的顶
点
    for(int i = 0; i < n; i++) cur[i] = head[i];
    while(cur[s] != -1)
    {
        if(u == t)                          //如果遍历到t
        {
            int tp = INF;
            for(int i = tail - 1; i >= 0; i--) tp = min(tp, e[sta[i]].cap -
e[sta[i]].flow);  //首先算出这条增光路的贡献
            maxflow += tp;
            for(int i = tail - 1; i >= 0; i--)
            {
                e[sta[i]].flow += tp;
                e[sta[i]^1].flow -= tp;
                if(e[sta[i]].cap - e[sta[i]].flow == 0)    //如果发现某一个i已经满
流，则栈的长度变成i-1
                    tail = i;
            }
            u = e[sta[tail]^1].to;    //通过其对应边来找到u的位置
        }
        else if(cur[u] != -1 && e[cur[u]].cap > e[cur[u]].flow && level[u] <
level[e[cur[u]].to]) //这条边可以扩展的话
        {
            sta[tail++] = cur[u];
            u = e[cur[u]].to;
        }
        else                                      //顶点u现在搜索到的这条边不能扩展
        {
            while(u != s && cur[u] == -1)          //顶点u的边搜索完了
                u = e[sta[--tail]^1].to;
            cur[u] = e[cur[u]].next;               //没有搜索完，搜索下一条边
        }
    }
    return maxflow;
}

int max_flow(int s, int t, int n)
{
    int flow = 0;
    for(;;)
    {
        bfs(s, t, n);
        if(level[t] < 0) return flow;
        flow += dfs(s, t, n);
    }
}

int main()
{
    int n, m, x, y, z;
    scanf("%d %d", &n, &m);
```

```
    memset(head, -1, sizeof(head[0]) * (n + 2));
    for(int i = 1; i <= n; i++)
    {
        scanf("%d %d", &x, &y);
        add_edge(0, i, x);
        add_edge(i, n + 1, y);
    }
    for(int i = 0; i < m; i++)
    {
        scanf("%d %d %d", &x, &y, &z);
        add_edge(x, y, z);
        add_edge(y, x, z);
    }
    printf("%d\n", max_flow(0, n+1, n+2));

}
```

匈牙利算法

计算最大匹配数 还可以用Dinic算法

注意： head要初始化成-1，tol = 0,match要初始化成0(如果左边不是从0开始的话)，主函数对左边进行遍历，每次要将vis初始化

```
struct edge
{
    int to, next;
}e[maxm];                                    //maxm是边的最大值

int head[maxn], vis[maxn], match[maxn], tol;     //head存匹配方，vis和match存被匹配
方

void add_edge(int x, int y)
{
    e[tol].to = y;
    e[tol].next = head[x]; head[x] = tol++;
}

bool dfs(int x)
{
    for(int i = head[x]; ~i; i = e[i].next)
    {
        int v = e[i].to;                     //看看能否给x寻找到v的增广路径
        if(!vis[v])                          //如果v已经不能被挪走
        {
            vis[v] = 1;
            if(!match[v] || dfs(match[v]))   //如果v尚未匹配或者匹配了可以被挪走
            {
                match[v] = x;
                return true;
            }
        }
    }
    return false;
```

```
}

int main()
{
    int k, n, m, x, y, ans = 0;
    scanf("%d %d %d", &k, &m, &n);
    memset(head, 0, sizeof(head));
    for(int i = 1; i <= k; i++)
    {
        scanf("%d %d", &x, &y);
        add_edge(x, y);
    }
    for(int i = 1; i <= m; i++)
    {
        memset(vis + 1, 0, sizeof(vis[0]) * n);       //每次需要初始化vis
        if(dfs(i)) ans++;
    }
    printf("%d\n", ans);
}
```

顶点覆盖&独立集&路径覆盖

顶点覆盖： 在顶点集合中，选取一部分顶点，这些顶点能够把所有的边都覆盖了。这些点就是顶点覆盖集 最小顶点覆盖：在所有的顶点覆盖集中，顶点数最小的那个叫最小顶点集合。

独立集： 在所有的顶点中选取一些顶点，这些顶点两两之间没有连线，这些点就叫独立集 最大独立集： 在左右的独立集中，顶点数最多的那个集合

路径覆盖： 在图中找一些路径，这些路径覆盖图中所有的顶点，每个顶点都只与一条路径相关联。 最小路径覆盖： 在所有的路径覆盖中，路径个数最小的就是最小路径覆盖了。

在二分图中： 最大匹配 = 最小顶点覆盖

首先证明，在最大匹配中，每条匹配边连接的两个顶点a,b最多只有一个与非匹配点有连边。 用反证法：假设a与c，b与d这件都有边，且c，d都不是匹配点，则可以去掉连接a,b的匹配边，加上连接a,c和连接b,d的匹配边，是匹配数+1，这与最大匹配矛盾。这样，我们构造这样一个顶点集合：对于每条匹配边，选择其连接的两个点中的一个（如果两个点有与非匹配点有连边的点，则选那个点；否则随便选一个）。

这个集合中有最大匹配数个点，我们证明：这个点集能覆盖所有的边。 若一条边是匹配边，则其显然被覆盖，若一条边不是匹配边： 1）若其与某匹配顶点有连边，则该匹配顶点必在我们构造的点集中，所以该边被覆盖 2）若其连接着两个非匹配点，则可以增加这条边为匹配边，是匹配数+1，这与最大匹配矛盾，故此情况不成立

而匹配的这m条边又是没有公共交点的, 所以一个顶点覆盖至少需要m个点； 综上： 最大匹配数 = 最小顶点覆盖

POJ3041 一个矩阵，某些位置有小行星，有一种炸弹，一次可以炸掉一行或者一列，现在问题是需要最少用多少这样的炸弹。 分析：行作为左顶点，列作为右顶点，若该行该列有交点，则连线，转化为求最小顶点覆盖

```
int main()
{
```

```
    scanf("%d %d", &n, &k);
    int x, y;
    for(int i = 1; i <= k; i++)
    {
        scanf("%d %d", &x, &y);
        add_edge(x, y);
    }

    for(int i = 1; i <= n; i++)
    {
        memset(vis, 0, sizeof(vis));
        if(dfs(i)) ans++;
    }
    printf("%d\n", ans);
}
```

POJ3216 一个r×c的牧场，有些地方有泥泞，有些地方有草，可以筑成水平或者竖直的墙(长度不限)，要求盖住所有泥泞，但不能盖住草，问最少建几做墙？ 分析： 相比于之前的行列为左右点，可以一行连续的点作为一个序号，一列连续的点也成序号，这样一个泥泞可以得到行序号和列序号，如果连线，则这个问题就可以划归成最小顶点覆盖问题。

在二分图中：最大独立集 = 顶点个数 – 最小顶点覆盖（最大匹配）

设最大匹配为m, 则最小顶点覆盖为m，去掉这个覆盖后，就不存在边，剩下的肯定是独立集，而因为这m个顶点各有一个匹配，这些匹配边不相连，想要成为一个独立集至少要去掉m个顶点。

最大团：顶点都互相相连的最大顶点集合(二分图默认某一边都是互相连的) 二分图的最大团=补图的最大独立集

POJ3692 有一群男生女生，男生都相互认识，女生都相互认识，有k组关系来表示男生女生认识，现在要找最多的人出来，这些人相互认识 分析：找团比较困难，如果男生和女生之间不认识，就连线(补图)，这样将最大团转化为求最大独立集了

```
int main()
{
    int G, B, m, x, y, ans, cas = 1;
    while(~scanf("%d %d %d", &G, &B, &m) && G)
    {
        memset(head, -1, sizeof(head));        //初始化head为-1, tol为0
        tol = ans = 0;
        memset(mark, 0, sizeof(mark));
        memset(match, 0, sizeof(match));
        for(int i = 1; i <= m; i++)
        {
            scanf("%d %d", &x, &y);
            mark[x][y] = 1;
        }
        for(int i = 1; i <= G; i++)
            for(int j = 1; j <= B; j++)
                if(!mark[i][j])
                    add_edge(i, j);            //建立补图
        for(int i = 1; i <= G; i++)
        {
            memset(vis, 0, sizeof(vis));
```

```
            if(dfs(i)) ans++;
        }
        printf("case %d: %d\n", cas++, G + B - ans);
    }
}
```

在二分图中，最小路径 = 顶点个数 - 最大匹配 = 最大独立集

一个有向无环图，最大独立集的m个点互不相连，所以最短路径至少为m, 从这m个点出m条路径，如果还有点没有被覆盖，那么最大独立集就不是m了。

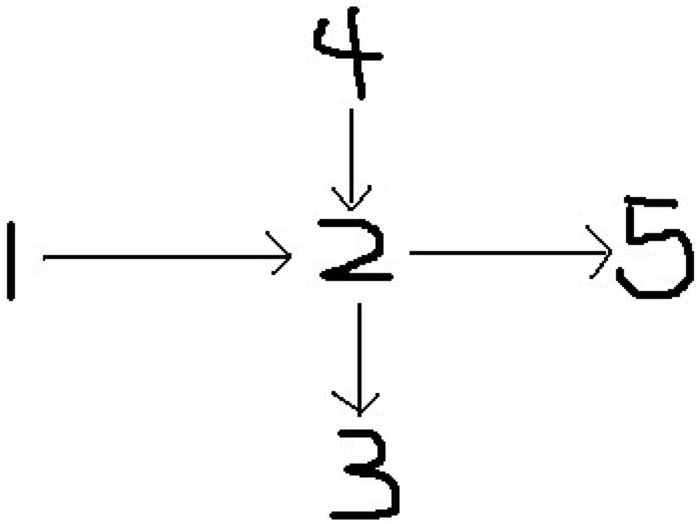
POJ2060 有很多人预订出租车，如果出租车做完一个任务能够敢到下一个任务，就不需要在调度一辆出租车了，现在请问最少需要几辆出租车。 分析： 假设有n个任务，每个任务为一个点，如果某个任务做完能到下一个任务，则两点连线，则就是求这个图的最小路径，假设一开始选的n条路径都只是单个点，如果没多一条连线，则路径数就会减1，则就是要求连线最多。如果把一个任务拆成两个点，如果任务A做完能到B，则左A到右B连线，这样求最多连线就转化为求最大匹配的问题了，答案就是n - 最大匹配

```
int main()
{
    int t, k, Hour, Minute, cnt, ans;
    scanf("%d", &t);
    while(t--)
    {
        memset(head, -1, sizeof(head));            //初始化
        memset(match, 0, sizeof(match));
        tol = cnt = ans = 0;
        scanf("%d", &k);
        for(int i = 1; i <= k; i++)
        {
            scanf("%d:%d %d %d %d %d", &Hour, &Minute, &pro[i].x1, &pro[i].y1,
&pro[i].x2, &pro[i].y2);
            pro[i].start = Hour * 60 + Minute;
        }

        for(int i  = 1; i <= k; i++)              //建立边
            for(int j = 1; j <= k; j++)
                if(check(i, j))
                    add_edge(i, j);
        for(int i = 1; i <= k; i++)            //匈牙利算法求最大匹配
        {
            memset(vis, 0, sizeof(vis));
            if(dfs(i)) ans++;
        }
        printf("%d\n", k - ans);
    }
}
```

POJ 2594 一个有向无环图中， 有若干条连接的路线， 问最少放多少个机器人，可以将整个图上的点都

走过(点可以重复) 分析：



如果用最短路径算答案时3，但是实际是2，这时候需要先用floyd跑一遍传递闭包，让1和3,1和5,4和3有连线，这样在进行最短路径

```
void floyd()
{
    for(int k = 1; k <= n; k++)                 //K为中间节点
        for(int i = 1; i <= n; i++)
            if(d[i][k])                     //如果i和k可达
            for(int j = 1; j <= n; j++)
                if(d[k][j]) d[i][j] = 1;     //如果k和j可达
}

bool dfs(int x)
{
    for(int i = 1; i <= n; i++)
    {
        if(!vis[i] && d[x][i])             //如果x可达i并且i没有被访问
        {
            vis[i] = 1;
            if(!match[i] || dfs(match[i]))
            {
                match[i] = x;
                return true;
            }
        }
    }
    return false;
}

int main()
{
    int x, y, ans;
    while(~scanf("%d %d", &n, &m) && n)
    {
        memset(match, 0, sizeof(match));
        memset(d, 0, sizeof(d));
        ans = 0;
        for(int i = 1; i <= m; i++)
        {
            scanf("%d %d", &x, &y);
```

```
            d[x][y] = 1;
        }
        floyd();                            //先跑传递闭包
        for(int i = 1; i <= n; i++)         //再跑匈牙利算法
        {
            memset(vis, 0, sizeof(vis));
            if(dfs(i)) ans++;
        }
        printf("%d\n", n - ans);
    }
}
```

KM算法

计算最大匹配值

若是求最小匹配，只需要将边权置为相反数，要记住m <= n

```
int val[maxn][maxn], vis_A[maxn], vis_B[maxn], match[maxn];     //vis来记录已被匹配的，match记录B方匹配到了A方的哪个
int ex_A[maxn], ex_B[maxn], slack[maxn], m, n;                    //记录A方和B方的期望，A方有m个，B方有n个
//slack 任意一个参与匹配A方能换到任意一个这轮没有被选择过的B方所需要降低的最小值
//这是二分图的最优匹配(首先是A集合的完备匹配，然后保证权值最大)
//所以一定保证 m <= n，否则会陷入死循环，若是A集合点多的话可以把B集合补充到和A一样多，设置-INF的边
bool dfs(int x)
{
    vis_A[x] = 1;
    for(int i = 1; i <= n; i++)
    {
        if(!vis_B[i])                           //每一轮匹配，B方每一个点只匹配一次
        {
            int gap = ex_A[x] + ex_B[i] - val[x][i];
            if(gap == 0)                        //如果符合要求
            {
                vis_B[i] = 1;
                if(!match[i] || dfs(match[i]))  //如果v尚未匹配或者匹配了可以被挪走
                {
                    match[i] = x;
                    return true;
                }
            }
            else slack[i] = min(slack[i], gap);
        }
    }
    return false;
}

int km()
{
    memset(match, 0, sizeof(match));          //match为0表示还没有匹配
    fill(ex_B + 1, ex_B + 1 + n, 0);          //B方一开始期望初始化为0
    for(int i = 1; i <= m; i++)               //A方期望取最大值
    {
```

```
        ex_A[i] = val[i][1];
        for(int j = 2; j <= n; j++)
            ex_A[i] = max(ex_A[i], val[i][j]);
    }
    for(int i = 1; i <= m; i++)              //尝试解决A方的每一个节点
    {
        memset(slack + 1, INF, sizeof(slack[0]) * n);
        for(;;)
        {
            memset(vis_A + 1, 0, sizeof(vis_A[0]) * m);     //记录AB双方有无被匹配过
            memset(vis_B + 1, 0, sizeof(vis_B[0]) * n);
            if(dfs(i))  break;
            int d = INF;
            for(int j = 1; j <= n; j++)    if(!vis_B[j]) d = min(d, slack[j]);
            //if(d == INF)  break;                        //找不到完全匹配
            for(int j = 1; j <= m; j++) if(vis_A[j]) ex_A[j] -= d;
            for(int j = 1; j <= n; j++)
            {
                if(vis_B[j]) ex_B[j] += d;
                else slack[j] -= d;
            }
        }
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
    {
        if(match[i])               // 可以加 && val[match[i]][i] > -INF 去除一些匹配
            ans += val[match[i]][i];
    }
    return ans;
}
```

费用流

每次寻找花费最少的增广路，由于增广路中有的路径长度可能为负数，所以不能直接用Dijkstra，引入了势的概念。

要初始化V

需要注意： h要清零，G要清零，V要弄清从0开始还是1开始(中间有一个初始化需要注意，底下默认从1开始)

```
typedef pair<int, int> P;             //first存距离，second存顶点编号

struct edge
{
    int to, cap, cost, rev;
};

vector<edge> G[maxn];             //用邻接表存图，maxn表示最大顶点数
int V, E, h[maxn], dist[maxn];    //h来存每个点的势，初始无负环为0，dist存距离
int prevv[maxn], preve[maxn];     //分别表示最短路中的前驱节点和对应的边
```

```
void add_edge(int from, int to, int cap, int cost)  //向图中增加一条从from到to容量为cap费用为cost的边
{
    G[from].push_back((edge){to, cap, cost, G[to].size()});
    G[to].push_back((edge){from, 0, -cost, G[from].size() - 1});
}

int min_cost_flow(int s, int t, int f)                     //如果不存在值，返回-1
{
    int res = 0;
    //memset(h, 0, sizeof(h));                             //按情况初始化
    while(f > 0)
    {
        priority_queue<P, vector<P>, greater<P> > que;
        memset(dist, INF, sizeof(dist));                   //Dijkstra求最短路径
        dist[s] = 0;
        que.push(P(0, s));
        while(!que.empty())
        {
            P p = que.top(); que.pop();
            int v = p.second;
            if(dist[v] < p.first) continue;               //该已经被更新或者被访问过
            for(unsigned int i = 0; i < G[v].size(); i++) //更新
            {
                edge &e = G[v][i];
                if(e.cap > 0 && dist[e.to] > dist[v] + e.cost + h[v] - h[e.to])
                {
                    dist[e.to] = dist[v] + e.cost + h[v] - h[e.to];
                    prevv[e.to] = v;
                    preve[e.to] = i;
                    que.push(P(dist[e.to], e.to));
                }
            }
        }
        if(dist[t] == INF) return -1;        //不能再增广
        for(int v = 1; v <= V; v++) h[v] += dist[v];   //默认顶点从1开始，对势进行更改

        int d = f;
        for(int v = t; v != s; v = prevv[v])
            d = min(d, G[prevv[v]][preve[v]].cap);          //求最大增广量
        f -= d;
        res += d * h[t];
        for(int v = t; v != s; v = prevv[v])                //对图进行更新
        {
            edge &e = G[prevv[v]][preve[v]];
            e.cap -= d;
            G[v][e.rev].cap += d;
        }
    }
    return res;
}
```

POJ3680

给定N个带权的开区间，i号区间覆盖(ai, bi)， 权重为wi，现在要在里面选一些区间，要求任意点都不被超过K个区间覆盖，目标是最大化总的权重。

分析： 首先将N个区间的2N个点离散化，我设源点S = 0， 汇点T = 2 * n + 1, 在1到2N之间都连上流量为k，费用为零的边，若是某两个点之间有区间，则在这两个点之间连上流量为1，费用为-wi的边，这样求得流量为k的费用流为题意所求。
由于一开始存在负边权，所以若ai和bi之间存在负边权，则由S向bi连流量为1，费用为0的边，bi和ai连流量为1，费用为wi的边，而ai与T连流量为1，费用为0的边，我们求k + n的流量，这样相当于把原来的负边权给填满，就可以用Dikstra优化的费用流了。

```
int main()
{
    int t, n, k;
    scanf("%d", &t);
    while(t--)
    {
        convert.clear();
        scanf("%d %d", &n, &k);
        for(int i = 1; i <= n; i++)
        {
            scanf("%d %d %d", &a[i], &b[i], &w[i]);
            convert.push_back(a[i]);  convert.push_back(b[i]);
        }
        sort(convert.begin(), convert.end());           //进行离散化
        convert.erase(unique(convert.begin(), convert.end()), convert.end());

        int m = convert.size(), s = 0, t = m + 1, res = 0;
        V = m + 2;
        for(int i = 0; i < V; i++) G[i].clear();
        add_edge(s, 1, k, 0);  add_edge(m, t, k, 0);
        for(int i = 1; i <= m - 1; i++)
            add_edge(i, i + 1, k, 0);
        for(int i = 1; i <= n; i++)         //构建反向边
        {
            int u = find(convert.begin(), convert.end(), a[i]) - convert.begin()
+ 1;
            int v = find(convert.begin(), convert.end(), b[i]) - convert.begin()
+ 1;
            add_edge(s, v, 1, 0);
            add_edge(v, u, 1, w[i]);
            add_edge(u, t, 1, 0);
            res -= w[i];
        }
        res += min_cost_flow(s, t, k + n);
        printf("%d\n", -res);
    }
}
```

树型dp & 二分图权值匹配

题目大意是给定两棵形状一样的树，每个点都有不同的值，问第一棵树最少改变几个结点的值可以和第二棵完全相同?

我一开始想的是递归去做，首先判断某两个点的子树是否相同，然后对其相同的儿子分别去试试(比如儿子$1, 2, 3$形状相同，那么可以有$3!$ 种试法)，这样搞是阶乘复杂度，炸内存了。。

其实这个匹配的过程可以用二分图来完成，降到多项式复杂度。首先 $dp[x][y]$ 表示第一二棵树分别以 x, y 为结点的子树要相同的最少花费，若是形状都不一样就让值为 INF，这个过程我们可以先进行树型dp，求出 $dp[a][b]$ (a 和 b 分别为 x, y 的儿子)，然后进行二分图的权值匹配，得到 $dp[x][y]$，这个思想比较巧妙。

然后顺便复习了一下 km 算法的板子，才知道 km 算法是完备匹配...由于km 算法的数组 val 是全

局变量，要首先将所有的 $dp[a][b]$ 算完，再去更新 km 算法的数组，**切记。**

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 510;
const int INF = 1e6;
const ll mod = 998244353;

int val[maxn][maxn], vis_A[maxn], vis_B[maxn], match[maxn];     //vis来记录已被匹配的，match记录B方匹配到了A方的哪个
int ex_A[maxn], ex_B[maxn], slack[maxn], m, n;                      //记录A方和B方的期望, A方有m个，B方有n个
//slack 任意一个参与匹配A方能换到任意一个这轮没有被选择过的B方所需要降低的最小值
//这是二分图的最优匹配(首先是A集合的完备匹配，然后保证权值最大)
//所以一定保证 m <= n, 否则会陷入死循环，若是A集合点多的话可以把B集合补充到和A一样多，设置-INF的边
bool dfs(int x)
{
    vis_A[x] = 1;
    for(int i = 1; i <= n; i++)
    {
        if(!vis_B[i])                           //每一轮匹配，B方每一个点只匹配一次
        {
            int gap = ex_A[x] + ex_B[i] - val[x][i];
            if(gap == 0)                        //如果符合要求
            {
                vis_B[i] = 1;
                if(!match[i] || dfs(match[i]))   //如果v尚未匹配或者匹配了可以被挪走
                {
                    match[i] = x;
                    return true;
                }
            }
            else slack[i] = min(slack[i], gap);
        }
    }
    return false;
}

int km()
{
    memset(match, 0, sizeof(match));            //match为0表示还没有匹配
    fill(ex_B + 1, ex_B + 1 + n, 0);             //B方一开始期望初始化为0
    for(int i = 1; i <= m; i++)                 //A方期望取最大值
    {
        ex_A[i] = val[i][1];
        for(int j = 2; j <= n; j++)
            ex_A[i] = max(ex_A[i], val[i][j]);
    }
    for(int i = 1; i <= m; i++)               //尝试解决A方的每一个节点
    {
```

```
        memset(slack + 1, INF, sizeof(slack[0]) * n);
        for(;;)
        {
            memset(vis_A + 1, 0, sizeof(vis_A[0]) * m);     //记录AB双方有无被匹配
过
            memset(vis_B + 1, 0, sizeof(vis_B[0]) * n);
            if(dfs(i))  break;
            int d = INF;
            for(int j = 1; j <= n; j++)    if(!vis_B[j]) d = min(d, slack[j]);
            //if(d == INF)  break;                        //找不到完全匹配
            for(int j = 1; j <= m; j++) if(vis_A[j]) ex_A[j] -= d;
            for(int j = 1; j <= n; j++)
            {
                if(vis_B[j]) ex_B[j] += d;
                else slack[j] -= d;
            }
        }
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
    {
        if(match[i])              // 可以加 && val[match[i]][i] > -INF 去除一些匹配
            ans += val[match[i]][i];
    }
    return ans;
}

vector<int> G1[maxn], G2[maxn];
int dp[maxn][maxn], rt1, rt2;

void solve(int x, int y)
{
    if(G1[x].size() != G2[y].size())
    {
        dp[x][y] = INF;
        return;
    }

    if(x != y)  dp[x][y]++;
    if(G1[x].size() == 0)  return;

    for(unsigned int i = 0; i < G1[x].size(); i++)
        for(unsigned int j = 0; j < G2[y].size(); j++)
            solve(G1[x][i], G2[y][j]);                //切记先让子树完成之后再更新
val

    for(unsigned int i = 0; i < G1[x].size(); i++)
        for(unsigned int j = 0; j < G2[y].size(); j++)
            val[i+1][j+1] = -dp[G1[x][i]][G2[y][j]];       //因为是求最小匹配，所以
置为负数，不存在的边置为-INF
    m = n = G1[x].size();
    int tmp = -km();
    if(tmp >= INF)  dp[x][y] = INF;
    else dp[x][y] += tmp;
}

int main()
{
```

```
    scanf("%d", &n);
    for(int i = 1; i <= n; i++)
    {
        int tmp;
        scanf("%d", &tmp);
        if(tmp == 0)  rt1 = i;
        else  G1[tmp].pb(i);
    }
    for(int i = 1; i <= n; i++)
    {
        int tmp;
        scanf("%d", &tmp);
        if(tmp == 0)  rt2 = i;
        else  G2[tmp].pb(i);
    }
    solve(rt1, rt2);
    printf("%d\n", dp[rt1][rt2]);
}
```

无向图求最小割 Stoer-Wagner算法

注意：对于图中任意两点s和t, 它们要么属于最小割的两个不同集中, 要么属于同一个集。如果是后者, 那么合并s和t后并不影响最小割. 基于这么个思想, 如果每次能求出图中某两点之间的最小割, 然后更新答案后合并它们再继续求最小割, 就得到最终答案了.

算法分析： 1. min=INF，首先任意固定一个顶点P

2.从点P用"类似"prim的算法扩展出"最大生成树"，记录最后扩展的顶点和最后扩展的边

3.计算最后扩展到的顶点的切割值（即与此顶点相连的所有边权和），若比min小更新min

4.合并最后扩展的那条边的两个端点为一个顶点

5.转到2，合并N-1次后结束

注意：1.Prim算法更新的d[i]，表示的是现在树上的点到i点的边权和，每次用Prim算法扩展到最后加进去的两个点为s, t, 则s-t割的最小值即为最后扩展的边的值。

每次d要初始化成成0(中间有边则++)， 底下模板点从0开始，若不联通，则最小割为0

```
int dis[maxn], v[maxn], d[maxn][maxn], vis[maxn];

int Stoer_Wagner(int n)
{
    int res = INF;
    for(int i = 0; i < n; i++)          //初始化第i个节点就是i
        v[i] = i;
    while(n > 1)                       //每一次循环一次Prim,少掉一个节点
    {
        int maxp = 1, pre = 0;
        for(int i = 1; i < n; i++)          //首先找到可以扩展的最大的点，记为maxp
        {
            dis[v[i]] = d[v[0]][v[i]];
            if(dis[v[i]] > dis[v[maxp]])
                maxp = i;
        }
        memset(vis, 0, sizeof(vis));
```

```
        vis[v[0]] = 1;
        for(int i = 1; i < n; i++)
        {
            if(i == n - 1)                                      //如果扩展到最后一个点
            {
                res = min(res, dis[v[maxp]]);                     //更新最小割
                for(int j = 0; j < n; j++)
                {
                    d[v[pre]][v[j]] += d[v[j]][v[maxp]];        //将v[maxp]的边加到v[[pre]上
                    d[v[j]][v[pre]] = d[v[pre]][v[j]];
                }
                v[maxp] = v[--n];           //此时所有的v[maxp]已经合并到了v[pre],需要去除v[maxn]，便将v[n-1]移到v[maxn]的位置
                break;
            }
            vis[v[maxp]] = 1;                 //扩展最大的点
            pre = maxp;
            maxp = -1;
            for(int j = 1; j < n; j++)        //更新dis的值，并找出下一个可以扩展的最大的点
            {
                if(!vis[v[j]])
                {
                    dis[v[j]] += d[v[pre]][v[j]];
                    if(maxp == -1 || dis[v[maxp]] < dis[v[j]])
                        maxp = j;
                }
            }
        }
    }
    return res;
}
```

带花树 一般图匹配

给定一张图，有 n 个点，m 条边，然后给每个点一个值d[i]，问是否可以通过选择一些边，让每个点的度等于这些值。

若 u, v 相连，那么把 u 拆成 d[u] 个点， v 拆成 d[v]个点，之间的边拆成两个点 u1, v1, u1v1相连，并且 u1和所有u拆的点相连，v1和所有v拆成的点相连。

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

using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<double, double> P;
const int maxn = 502;
const int INF = 0x3f3f3f3f;
const double eps = 1e-11;
const ll mod = 1e9 + 7;

int n, m;
```

```cpp
vector<int> G[maxn];                              //存边

//带花树模板，n为匹配的边，从1开始计数
int mark[maxn], match[maxn], pre[maxn], fa[maxn];
int lca_clk, lca_mk[maxn];
pair<int, int> ce[maxn];

void connect(int u, int v) {
    match[u] = v;
    match[v] = u;
}
int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

void flip(int s, int u) {
    if (s == u) return;
    if (mark[u] == 2) {
        int v1 = ce[u].first, v2 = ce[u].second;
        flip(match[u], v1);
        flip(s, v2);
        connect(v1, v2);
    } else {
        flip(s, pre[match[u]]);
        connect(pre[match[u]], match[u]);
    }
}

int get_lca(int u, int v) {
    lca_clk++;
    for (u = find(u), v = find(v); ; u = find(pre[u]), v = find(pre[v])) {
        if (u && lca_mk[u] == lca_clk) return u;
        lca_mk[u] = lca_clk;
        if (v && lca_mk[v] == lca_clk) return v;
        lca_mk[v] = lca_clk;
    }
}

void access(int u, int p, const pair<int, int>& c, vector<int>& q) {
    for (u = find(u); u != p; u = find(pre[u])) {
        if (mark[u] == 2) {
            ce[u] = c;
            q.push_back(u);
        }
        fa[find(u)] = find(p);
    }
}

bool aug(int s) {
    fill(mark, mark + n + 1, 0);
    fill(pre, pre + n + 1, 0);
    iota(fa, fa + n + 1, 0);
    vector<int> q = {s};
    mark[s] = 1;
    for (int t = 0; t < (int) q.size(); ++t) {
        // q size can be changed
        int u = q[t];
        for (int &v: G[u]) {
            if (find(v) == find(u)) continue;
            if (!mark[v] && !match[v]) {
```

```
                flip(s, u);
                connect(u, v);
                return true;
            } else if (!mark[v]) {
                int w = match[v];
                mark[v] = 2; mark[w] = 1;
                pre[w] = v; pre[v] = u;
                q.push_back(w);
            } else if (mark[find(v)] == 1) {
                int p = get_lca(u, v);
                access(u, p, {u, v}, q);
                access(v, p, {v, u}, q);
            }
        }
    }
    return false;
}

int solve()                    //匹配n个顶点，从1开始计数,返回匹配个数，match记录结果
{
    fill(match + 1, match + n + 1, 0);
    lca_clk = 0;
    int ans = 0;
    for (int i = 1; i <= n; i++) if(!match[i]) ans += aug(i);
    return ans;
}

void init(int x)
{
    for (int i = 1;i <= x;i++)
        G[i].clear(), lca_mk[i] = 0;
}

int x[102], y[102], deg[52], d[52];

int main()
{
    while(~scanf("%d %d", &n, &m))
    {
        for(int i = 1; i <= n; i++)
            scanf("%d", &d[i]);
         fill(deg + 1, deg + 1 + n, 0);
        for(int i = 1; i <= m; i++)
        {
            scanf("%d %d", &x[i], &y[i]);
            deg[x[i]]++;  deg[y[i]]++;
        }

        int flag = 1;
        for(int i = 1; i <= n; i++)
        {
            if(deg[i] < d[i])
            {
                flag = -1;
                break;
            }
        }
        if(flag < 0)
```

```
        {
            puts("No");
            continue;
        }

        int num = 0;
        vector<int> tmp[52];
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= d[i]; j++)
                tmp[i].push_back(++num);
        n = num + 2 * m;
        init(n);
        for(int i = 1; i <= m; i++)
        {
            num++;
            G[num].push_back(num + 1);
            G[num+1].push_back(num);
            for(auto &u: tmp[x[i]])
                G[u].push_back(num), G[num].push_back(u);
            num++;
            for(auto &u: tmp[y[i]])
                G[u].push_back(num), G[num].push_back(u);
        }

        if(n % 2 == 0 && solve() == n / 2)  puts("Yes");
        else puts("No");
    }
}
```

SPFA求负权+01分数规划

洛谷P1768 给定一个有向图，每条边有v和c，是否存在一个回路，使得∑vi / ∑ci最大

分析：若是存在∑vi / ∑ci > k, 那么最终答案一定比k要大，可化简为 $k\sum c_i - \sum v_i < 0$ *即等价于把边权化为* $k\sum c - \sum v$,问是否存在一个负环，若是存在负环，则让l = mid, 否则让r = mid, 一开始让l = 0，若是这个图一开始就没有环，则一直找不到负环，最后l仍为0，输出-1
spfa求负环用的是dfs的方法可以判断单点出发寻找负环，若是图一开始不连通的话，处理比较麻烦，所以将0作为超级源点，这样对答案不影响，同时只需从0出发即可；

寻找负环的思路是，存在负环时，最短路会一直在上面绕，我们猜测一条路径经过两次同一点即可判为存在负环。

一开始vis要清0，dis要初始化成INF（这样第一次到达时才会被更新）

若是要找正环，则将跑最短路换成跑最长路，并且dis初始化成-INF（这样第一次到达才会被更新）

```
struct edge
{
    int to, v, c;
};

vector<edge> G[maxn];
int vis[maxn], n, m;
double dis[maxn], l ,r, mid;
```

```
bool spfa(int s)
{
    vis[s] = 1;
    for(unsigned int i = 0; i < G[s].size(); i++)
    {
        edge e = G[s][i];
        double x = mid * e.c - e.v;
        if(dis[e.to] > dis[s] + x)          //如果这个点可以扩展
        {
            if(vis[e.to]) return true;          //如果在最短路中，则存在负环
            dis[e.to] = dis[s] + x;
            if(spfa(e.to)) return true;
        }
    }
    vis[s] = 0;                             //回溯
    return false;
}

int main()
{
    scanf("%d %d", &n, &m);
    int x, y, v, c;
    for(int i = 1; i <= m; i++)
    {
        scanf("%d %d %d %d", &x, &y, &v, &c);
        G[x].push_back((edge){y, v, c});
    }
    for(int i = 1; i <= n; i++)
        G[0].push_back((edge){i, 0, 0});

    l = 0, r = 201;
    while(r - l > eps)                      //二分判断
    {
        memset(dis, INF, sizeof(dis));
        memset(vis, 0, sizeof(vis));
        dis[0] = 0;
        mid = (l + r) / 2;
        if(spfa(0)) l = mid;
        else r = mid;
    }
    if(l == 0) printf("-1\n");
    else printf("%.1f\n", l);

}
```

Dijkstra求最短路径(有向无向非负边)

d数组记录顶点s到其他顶点的最短路径，vector用来表示邻接表，edge用来记录边(记录to与cost)

用邻接表记录边：

```
scanf("%d %d %d", &x, &y, &z);
G[x].push_back(edge(y, z));
G[y].push_back(edge(x, z));             //若是无向边需要双向记录
```

定义结构体和参数：

```
const int INF = 0x3f3f3f3f;
typedef pair<int, int> P;

struct edge
{
    int to, cost;
    edge(int k1, int k2): to{k1}, cost{k2} {}
};
int d[maxn], V, E;
vector<edge> G[maxn];
priority_queue< P, vector<P>, greater<P> > que;     //第一维记录cost,第二维记录to
```

执行函数：

```
void Dijkstra(int s)
{
    memset(d, INF, sizeof(d));
    d[s] = 0;
    que.push(P(0, s));

    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int v = p.second;
        if(d[v] < p.first) continue;            //去掉已经被访问过的节点和被更新过的
边长
        for(unsigned int i = 0; i < G[v].size(); i++)   //更新边长
        {
            edge e = G[v][i];
            if(d[e.to] > d[v] + e.cost)
            {
                d[e.to] = d[v] + e.cost;
                que.push(P(d[e.to], e.to));
            }
        }
    }
}
```

若是要记录最短路径条数，或者要记录一条路径时：

```
int d[maxn], pre[maxn], state[maxn], V, E;        //pre记录一条路径，state记录条数
vector<edge> G[maxn];
priority_queue< P, vector<P>, greater<P> > que;
void Dijkstra(int s)
{
    memset(d, INF, sizeof(d));
    d[s] = 0;
    state[s] = 1;
    que.push(P(0, s));

    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int v = p.second;
```

```
        if(d[v] < p.first) continue;
        for(unsigned int i = 0; i < G[v].size(); i++)
        {
            edge e = G[v][i];
            if(d[e.to] == d[v] + e.cost)         //等于情况要相加路径
                state[e.to] += state[v];
            else if(d[e.to] > d[v] + e.cost)     //小于情况
            {
                state[e.to] = state[v];
                pre[e.to] = v;                  //更新前驱结点
                d[e.to] = d[v] + e.cost;
                que.push(P(d[e.to], e.to));
            }
        }
    }
}
```

Floyd求最短路径(任意两点，可以处理负边)

```
void floyd()               //d[u][v]表示(u,v)的权值，不存在时设为INF，不过d[i][i] = 0
{
    for(int k = 1; k <= V; k++)
        for(int i = 1; i <= V; i++)
            for(int j = 1; j <= V; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

最大权闭合子图

闭合图：在一个图中，我们选取一些点构成集合，若集合中任意点连接的任意出弧，所指向的终点也在V中，则这个集合以及所有这些边构成闭合图。

首先有一个有向连通图，每个点带有一个权值，此时，构建一个超级源点s，一个超级汇点t，所有的点按权值的正负连接到s和t上，若原来有边，则变成INF，转换成一个边权值有向图

①该带边权有向图的关于s-t最小割，是简单割

简单割：割集中所有的边，都与s或t相连接。 显然的，因为不与s,t相连的边，权值都是INF，最小割不可能割在INF的边上

②该图中的每一个简单割产生的两个子图，我们记含有点s的是图S，含有点t的是图T，则图S是闭合图

简单割内不包含边权为INF的边，即不含有连通两个图的边（除了连接在s,t点上的边之外）；图S中（不包含S）没有边与图T连通

③最小割产生的图S和图T，图S为最大权闭合子图；

因为割集中所有的边，不是连接在s上，就是连接在t上；　我们记割集中，所有连接在s上的边的权值和为x1，所有连接在t上的边的权值和为x2，而割集中所有边权值和为X=x1+x2；记图S中所有点的权值和为W，记其中正权值之和为w1，负权值之和为 - w2，故W = w1 - w2；　而 W + X = w1 - w2 + x1 + x2，由于x2 = w2（因为图S中所有负权值的点，必然连接到t点，而图S必然要与t分割开；故割集中，“连接在t点上的边权值和”就是“图S中所有负权值点的权值之和，取负”），因而W + X = w1 + x1；而显然的，w1 + x1是整个图中所有正权值之和，记为SUM；故W = SUM - X，即 “图S中所有点的权值和” = “整个图中所有正权值之和” - “割集中所有边权值和”，然后，因为SUM为定值，只要我们取最小割，则“图S中所有点的权值和”就是最大的，即此时图S为图S为最大权闭合子图。

解题思路：

①先记录整个图中，所有正点权值的和；②建立对应流网络，求最大流，最大流在数值上等于最小割，故我们得到了流网络的s-t最小割；　③"所有正点权值的和"减去"s-t最小割"，即得最大权闭合子图的权值和。

另一个理解角度

一开始我们想把所有的正权值都选上，即s满流流到各个正顶点，若是之后连有负权值，则会流到t，但是流到t的流量不会超过s流到这个正顶点的流量，留了多少就是损失了多少，最多就没有获益，而所求的最大流即为损失量，若是s到某个顶点满流(即无法到达)，则说明这个顶点无用，会全部损失，即将这条边割掉，也符合最小割的定义了。

POJ2987 n个点的有向图有m条边，求最大权闭合子图(要求子图的点数量尽量少) 分析：用最大权闭合子图求完后，从s点开始深搜，如果碰上满流的边则认为是割边，这样获得的子图点数量是最少的

```
void dfs2(int s)
{
    vis[s] = 1;
    cnt++;
    for(int i = 0; i < G[s].size(); i++)
        if(!vis[G[s][i].to] && G[s][i].cap > 0) dfs2(G[s][i].to);  //如果是可达
}

int main()
{
    int n, m, s, t, tmp, x, y;
    scanf("%d %d", &n, &m);
    s = 0, t = n + 1;
    for(int i = 1; i <= n; i++)
    {
        scanf("%d", &tmp);
        if(tmp > 0)
        {
            add_edge(s, i, tmp);     //正边连S
            sum += tmp;              //记录所有正边和
        }
        else add_edge(i, t, -tmp);   //负边连t
    }
    for(int i = 1; i <= m; i++)
    {
        scanf("%d %d", &x, &y);
        add_edge(x, y, INF);         //原边成INF
    }
    ll ans = max_flow(s, t);
    dfs2(s);                        //搜索得到S子图
    printf("%d %lld\n", cnt - 1, sum - ans);
}
```

kosaraju算法

给定一个有向图，问有几个结点，图中的任意结点都可以到达？$(1 \le n \le 1e4)$

分析

若是知道了图中的强连通分量，**将一个强连通分量缩成一个点的话，那么原图可以变成是一个DAG**(有向无环图)。若是只有一个结点出度为0，那么其它结点都可以到达这个点；若是有大于一个，那么就不存在任何结点都可以到达的点了。

所以我们可以用 kosaraju 算法求出强连通分量，进行缩点，然后求出出度为0的缩点判断答案。

kosaraju算法的步骤是：①正向建图和反向建图；②遍历正(反)向图，记录后序(最先遍历到的结点在后面)；③根据后序遍历反(正)向图，每次dfs都可以遍历到一个强连通分量。

kosaraju算法的核心是先通过一次 dfs 得到合适的遍历顺序，然后第二次 dfs 得到强连通分量，觉得知乎上一个很好的讲解：link (用食物链关系来解释这个算法过程)

代码

```
#include <vector>
#include <stdio.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e4 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 1e9 + 7;

int n, m, vis[maxn], st[maxn], mark[maxn], num[maxn], d[maxn], tol, tag;
vector<int> G1[maxn], G2[maxn];

void dfs1(int x)
{
    vis[x] = 1;
    for(unsigned int i = 0; i < G1[x].size(); i++)
    {
        int y = G1[x][i];
        if(vis[y])  continue;
        dfs1(y);
    }
     st[++tol] = x;        //最先遍历到的结点在栈顶
}

void dfs2(int x)
{
    vis[x] = 1;
    for(unsigned int i = 0; i < G2[x].size(); i++)
    {
        int y = G2[x][i];
        if(vis[y])  continue;
        dfs2(y);
    }
    mark[x] = tag;              //同一个dfs遍历到的结点属于同一个强连通分量
    num[tag]++;
}

void kosaraju()
{
    for(int i = 1; i <= n; i++)
        if(!vis[i])
            dfs1(i);
```

```
    fill(vis + 1, vis + 1 + n, 0);
    for(int i = tol; i >= 1; i--)
        if(!vis[st[i]])
        {
            tag++;
            dfs2(st[i]);
        }

    for(int i = 1; i <= n; i++)
    {
        for(unsigned int j = 0; j < G1[i].size(); j++)
        {
            int y = G1[i][j];
            if(mark[i] == mark[y])  continue;
            d[mark[i]]++;                    //标记连通分量对应点的出度
        }
    }

    int cnt = 0, ans;
    for(int i = 1; i <= tag; i++)
    {
        if(!d[i])
        {
            if(cnt++ == 1)
            {
                printf("0\n");
                return;
            }
            ans = num[i];
        }
    }
    printf("%d\n", ans);
}

int main()
{
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= m; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        G1[x].pb(y);  G2[y].pb(x);    //正向建图和反向建图
    }
    kosaraju();
}
```

建图思想

给定一个长为 $n(n \leq 1e5)$ 的只含数字1 - 8的字符串，每出现一个逆序对 (a, b) (其中b< a) 就会有 $P_{a,b}$ 的 cost，比如字符串 85511 的 cost 为 $2 \times P_{8,5} + 2 \times P_{8,1} + 4 \times P_{5,1}$。

此外还有一个变换操作，可以花费 $C_{a,b}$ 将所有的 a 换成 b, b 换成 a，如花费 $C_{8,5}$(或者 $C_{5,8}$) 可以将字符串 85511 变成 58811。我们可以进行任意次的变换操作，最后要计算逆序对的花费，求总共最

小的花费。

输入为长度n，数字字符串，以及8×8 的 P 矩阵和 C 矩阵，其中 P 矩阵是下三角矩阵(因为正序对的花费自然为0)， C 矩阵是对称矩阵。

分析

其实要求逆序对的花费，就是求**各种数对的花费**，因为正序对的花费都是 0。

由于变换操作是**相同的全部数字一起变**的，即一开始数字一样的位置，无论经过多少次变换，最后还是一样的；一开始不一样的数字，最后也肯定不一样的。那么字符串就可以最多分成 8 组， 第 i 组最开始是表示数字 i， 经过若干次变换之后，这一组可能会变成其他任何数字。

要计算的花费分为变换的花费和变换后数对的花费。

数对花费：若用 $dp[i][j]$ 表示 第 i 组和第 j 组能组成的 $(i\text{，}j)$数对的个数，则没有变换的数对花费就是 $\sum_{i=1}^{8}\sum_{j=1}^{8}dp[i][j]\times P[i][j]$, 若记第 i 组数最后变成 $mark[i]$， 则花费就是 $\sum_{i=1}^{8}\sum_{j=1}^{8}dp[i][j]\times P[mark[i]][mark[j]]$。

可以通过线性时间计算 $dp[i][j]$：

```
for(int i = 0; s[i]; i++)                    //计算组对的个数
    {
        int tmp = s[i] - '0';
        for(int j = 1; j <= 8; j++)
            dp[j][tmp] += num[j];
        num[tmp]++;
    }
```

变换花费：一开始各个组对应的序列就是 12345678， 此时的变换花费为0， 而最多有$8!$ 这么多序列可以变换，那该怎么计算到每个序列的最少花费呢?

一开始我用的是递归的方式，即每个序列可以通过变换两组的数字变成另一个序列，但是这样的话每一个序列可以有 C_8^2 种变换，那么$8!$ 种序列的开销太大了。比如从 12345678，变成 87654321，有很多种方式，通过递归找到最短的变换方式并不是好的选择。

由此引入了**建图的思想**，将这个问题考虑成一张图，序列为点，其中相邻点(可以通过一次操作变换得到)之间的边即为变换的花费，那么我们要求的即是最初序列到其他序列的最小花费，即为**单源最短路问题**了。

如果用Dijkstra 方法计算，边数为 $\frac{8!\times C_8^2}{2}$， 点数即为8!，而怎么将一个序列映射到点的标号呢？这里可以用生成排列的序号，如 12345678 是0, 12345687是1, 87654321是$8!-1$：

```
int GetIndex(int* a)                    //获取一个排列的序号，从0开始
{
    int res = 0;
    for(int i = 1; i < 8; i++)
    {
        int cnt = 0;
        for(int j = i + 1; j <= 8; j++)
            if(a[i] > a[j])
                cnt++;
        res += cnt * fac[8 - i];        //fac[i]表示阶乘
    }
    return res;
}
```

综上，只需要预先计算序列之间的变换花费 $(O(\frac{8!\times C_8^2}{2}lg8!))$, 预处理组对数$(O(8n))$， 对每个序列计算数对花费$(O(8!\times 64))$。

代码

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

using namespace std;
typedef unsigned long long ll;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const double eps = 1e-8;

int fac[10], n, dp[9][9], num[9];
ll P[9][9], C[9][9];
char s[maxn];

int GetIndex(int* a)                    //获取一个排列的序号，从0开始
{
    int res = 0;
    for(int i = 1; i < 8; i++)
    {
        int cnt = 0;
        for(int j = i + 1; j <= 8; j++)
            if(a[i] > a[j])
                cnt++;
        res += cnt * fac[8 - i];
    }
    return res;
}

struct node
{
    int a[9];
    ll w;
    bool operator<(const node& m) const         //使得优先队列可以小顶堆
    {
        return w > m.w;
    }
};
ll d[41000];
priority_queue<node> que;

void Dijkstra()
{
    memset(d, INF, sizeof(d));
    d[0] = 0;                    //初始位置为序列12345678
    node u;
    u.w = 0;
    for(int i = 1; i <= 8; i++)
        u.a[i] = i;
    que.push(u);

    while(!que.empty())
    {
        node p = que.top(); que.pop();
        int k1 = GetIndex(p.a);
        if(d[k1] < p.w) continue;            //去掉已经被访问过的节点和被更新过的边长

        node tmp = p;
```

```
        for(int i = 1; i < 8; i++)                  //遍历与这个序列相邻的序列，进
行更新
            for(int j = i + 1; j <= 8; j++)
            {
                swap(tmp.a[i], tmp.a[j]);
                int k2 = GetIndex(tmp.a);
                if(d[k2] > d[k1] + C[i][j])
                {
                    d[k2] = d[k1] + C[i][j];
                    tmp.w = d[k2];
                    que.push(tmp);
                }
                swap(tmp.a[i], tmp.a[j]);
            }
    }
}

int main()
{
    fac[1] = 1;                         //计算阶乘
    for(int i = 2; i <= 8; i++)
        fac[i] = fac[i-1] * i;

    scanf("%d %s", &n, s);
    for(int i = 1; i <= 8; i++)
        for(int j = 1; j <= 8; j++)
            scanf("%lld", &P[i][j]);
    for(int i = 1; i <= 8; i++)
        for(int j = 1; j <= 8; j++)
            scanf("%lld", &C[i][j]);
    Dijkstra();                         //计算序列之间转换的花费

    for(int i = 0; s[i]; i++)               //计算组对的个数
    {
        int tmp = s[i] - '0';
        for(int j = 1; j <= 8; j++)
            dp[j][tmp] += num[j];
        num[tmp]++;
    }

    int mark[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8};          //对每一个序列计算答案
    ll ans = LLONG_MAX;
    do
    {
        ll res = d[GetIndex(mark)];
        for(int i = 1; i <= 8; i++)
            for(int j = 1; j <= 8; j++)
                res += dp[i][j] * P[mark[i]][mark[j]];
        ans = min(res, ans);
    }while(next_permutation(mark + 1, mark + 8 + 1));

    printf("%lld\n", ans);
}
```

虽然题目给的n 是 1e5， 不过要开到 1e6，奇奇怪怪

收获

① 建图的思想，长姿势了，瞬间将一个复杂的递归转成一个带log的线性做法。
②复习了离散数学中学的生成排列，以及C++ next_permutation的用法