2023暑期CSP-S/NOIP模拟赛 5 题解

T1 香槟塔

n层容器，每层的容量为 a_i，第 i 层满了就会流进第 i+1层，总共有q次操作包括倒入第 i 层 v 的体积和询问第 k 层当前香槟的体积。

30%-50%的做法：

模拟倒入操作，对香槟的流动枚举第 i 层到最后一层。

时间复杂度：$O(nq)$

100%的做法：

观察到算法的瓶颈在于每次香槟的流动，思考有没有什么特殊的性质能够让我们快速的处理？

可以发现，当一层容器的容量满了以后，这一层只会进行香槟的传递，并不会再容纳香槟。也就是说，当一个容器满了以后，这一层香槟的体积也不会再发生变化。

通过这个性质，我们可以使用一个链表来维护仍然可以容纳香槟的层，或者使用并查集来跳过已经满了的层。

时间复杂度 $O(\sum a_i)$

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e5 + 10;
int n, m, opt;
int fa[maxn], lim[maxn], p[maxn];

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> lim[i];
    cin >> m;
    for (int i = 1; i <= n + 1; ++i) fa[i] = i;
    for (; m; --m) {
        cin >> opt;
        int k, x;
        if (opt == 1) {
            cin >> k >> x;
            for (k = find(k); k <= n && x > 0; k = find(k)) {
                if (lim[k] - p[k] <= x) {
                    x -= lim[k] - p[k];
                    p[k] = lim[k];
                    fa[k] = k + 1;
```

```
                } else {
                    p[k] += x;
                    x = 0;
                }
            }
        } else {
            cin >> k;
            cout << p[k] << '\n';
        }
    }
}
```

T2 距离

求n个点两两之间距离平方的和。

40%做法：

暴力枚举每一个点对。

时间复杂度：$O(n^2)$

80%-100%的做法：

对计算的式子展开有：

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}(x_i-x_j)^2+(y_i-y_j)^2$$

$$=\sum_{i=1}^{n}\sum_{j=i+1}^{n}x_i^2+x_j^2+y_i^2+y_j^2-2x_ix_j-2y_iy_j$$

对每个点 (x_i,y_i)，考虑他对答案的贡献，他会与其他 n-1 个点都进行这样的计算，i 的贡献= i 单独产生的贡献+ i 与其他点一起产生的贡献。

1) i 单独产生的贡献：x_i^2,y_i^2 会有 n-1 次的贡献；

2) i 与其他点一起产生的贡献： $2x_ix_j,2y_iy_j$ 也会有 n-1 次贡献，但是在 i 计算贡献的时候会被算到，在 j 计算贡献的时候也会被算到，也就是会被算两次。可以考虑只计算 i < j 的贡献，也可以考虑重复计算后再除以2。

当然，直接使用数学方法对式子分析，也可以上述考虑贡献的结果。

这里有一点需要注意的是，需要简单计算一下最后答案的大小：

对于 80% 的范围，答案的大小为 $1e5*1e5*1e4*1e4=1e18$，需要开 longlong；

而对于 100% 的范围，答案大小为 $1e5*1e5*1e5*1e5=1e20$，已经超过了 longlong 的范围，可以写个高精度或者使用 __int128 来进行代替。

注意，__int128 不支持 IO 操作，需要用一位一位的进行输出和读入。

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

```cpp
typedef long long LL;
const int maxn = 1e5 + 10;
int n;
LL x[maxn], y[maxn], sumx[maxn], sumy[maxn];

void print(__int128 x) {
    int a[100], tot = 0;
    while(x > 0) a[++tot] = x % 10, x /= 10;
    for(int i = tot; i; --i) cout << a[i];
    cout << '\n';
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n;
    for(int i = 1; i <= n; ++i) {
        cin >> x[i] >> y[i];
        sumx[i] = sumx[i - 1] + x[i];
        sumy[i] = sumy[i - 1] + y[i];
    }
    __int128 ans = 0;
    for(int i = 1; i <= n; ++i) {
        ans += (n - 1) * (x[i] * x[i] + y[i] * y[i]);
        ans -= 2 * x[i] * (sumx[n] - sumx[i]);
        ans -= 2 * y[i] * (sumy[n] - sumy[i]);
    }
    print(ans);
    return 0;
}
```

T3 毕业旅行

必须经过[1,k]的城市至少一次，设计一个从长度为d的序列，代表d天经过的城市有哪些（相邻两天的城市必须要有直接的连边），求有多少种不同的d序列

20%的做法：

直接枚 n! 枚举所有可能的方案，检查是否满足边的条件和 k 的限制

时间复杂度 $O(n! * k)$

额外10%的做法：

对于 k = 0，没有必须要经过的城市，考虑直接使用 dp 来计算。

令 $f[i][j]$ 表示，在第 i 天经过城市 j 时的方案数。考虑转移方程：

$f[i+1][y] += f[i][x], \exists (x,y) \in edge$

考虑到天数 d 会很大，n 的数值只有20，并且转移方程的形式与具体哪一天无关，可以考虑使用矩阵乘法来加速计算过程。

$$\begin{pmatrix} f[i+1][1] \\ f[i+1][2] \\ \cdots \\ f[i+1][n] \end{pmatrix} = \begin{pmatrix} a[1][1] & \cdots & a[1][n] \\ \cdots & & \\ a[n][1] & \cdots & a[n][n] \end{pmatrix} \begin{pmatrix} f[i][1] \\ f[i][2] \\ \cdots \\ f[i][n] \end{pmatrix}$$

其中 a 矩阵为 n 个点的邻接矩阵。

时间复杂度：$O(n^3 * log_2 d)$

额外20%的做法：

对于 k!=0 的情况，考虑 k<=5，我们可以使用状态压缩来记录已经经过了哪些点。

令 $f[i][j][k]$ 表示，在第 i 天经过城市 j 时，必须经过的城市已经经过的状态为 k （二进制位 0/1 表示）的方案数。

$f[i+1][y][y <= k\ ?\ k|(1 << y)\ :\ k] += f[i][x][k], \exists (x,y) \in edge$

对于 d 比较小的情况，可以直接枚举天数来实现整个dp过程。

时间复杂度：$O(d * n * m * 2^k)$

100%的做法：

对于 d 比较大的情况，第三个子任务的直接枚举 d 的过程不能使用，考虑如何计算?

如果想用第二个子任务的矩阵进行优化，那么对于必须要经过的城市有没有经过这个状态的记录不能添加进去，那应该如何处理呢?

观察到 k<=7 整体比较小，既然我们不能记录必须要经过的城市有没有经过，那么可以换个角度考虑，直接强制不经过某个必须要经过的城市，使用容斥原理来进行计算。

容斥的式子为：$ans = f_{强制关闭0个重要城市} - f_{强制关闭1个重要城市} + f_{强制关闭2个重要城市} - f_{强制关闭3个重要城市} + \cdots\cdots$

对于强制不能经过的城市是哪些，可以采用 2^k 来进行枚举，强制不经过某个城市只需在构建邻接矩阵时无视与这个城市相连的边就行了。

```
#include <bits/stdc++.h>
#define N 22
using namespace std;
const int mod = 1e9 + 9;
int n, m, s, d;
int X[170], Y[170], a[N][N], ans[N][N], fans, cnt;

void multi(int a[N][N], int b[N][N]) {
    int c[N][N];
    memset(c, 0, sizeof(c));
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            for (int k = 1; k <= n; ++k)
                c[i][k] = (1ll * a[i][j] * b[j][k] + c[i][k]) % mod;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            a[i][j] = c[i][j];
}
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n >> m >> s >> d;
    for (int i = 1; i <= m; ++i) cin >> X[i] >> Y[i];
    for (int i = 0; i < 1 << s; ++i) {
        memset(a, 0, sizeof(a));
        memset(ans, 0, sizeof(ans));
        for (int j = 1; j <= m; ++j)
            if ((X[j] > s || i >> X[j] - 1 & 1) && (Y[j] > s || i >> Y[j] - 1 &
1)) {
                a[X[j]][Y[j]]++;
                a[Y[j]][X[j]]++;
            }
        cnt = 0;
        for (int j = 1; j <= n; ++j) if ((i >> j - 1 & 1) || j > s) ans[1][j] =
1, cnt++;
        int tmp = d - 1;
        while (tmp) {
            if (tmp & 1) multi(ans, a);
            multi(a, a);
            tmp /= 2;
        }
        for (int j = 1; j <= n; ++j) {
            if (n - cnt & 1) fans = (fans - ans[1][j]) % mod;
            else fans = (fans + ans[1][j]) % mod;
        }
    }
    cout << (fans + mod) % mod << '\n';
    return 0;
}
```

T4 潜入计划

在 n 个点 m 条边的图，每个点有个高度 h_i ，一个人能从点 s 的高度为 x 的位置到点 t （花费 T 的时间）当且仅当 $x-T>0$ 且 $x-T<=h_t$ ，且同一个点变换高度 x 到高度 y 花费的时间为 abs(x-y)，求一个从 1 号点高度为 X 到 n 号点最高位置 h_n 的最短时间，

20%的做法：

令 $d[i][j]$ 表示在 i 号楼，第 j 层时花费最小的时间是多少。

使用 $d[i][j]$ 代替原来的 $d[i]$ 正常跑最短路，以 Dijkstra 为例，可以点信息 $<i,d>$ 可以替换 $<i,j,d>$，其他的一一对应替换即可。更新的时候对这个楼的所有的楼层高度都更新。

时间复杂度 $O((n+m)*H*log_2(nH))$。

100%的做法：

本题的瓶颈在于每栋楼的楼层数量太大，所要保存的状态太多，能不能减少状态？

考虑最终答案的构成，一定是由很多飞行+一些上升+一些下降构成。

由于在任何一个点上升或者下降代价是一样的，所以：

对于上升操作来说，只要保证前面飞行合法就不需要上升。当且仅当我飞不过去了才上升。

对于下降操作来说，只要我不会越过目标点就不需要下降。当且仅当我会越过目标点才下降。

不存在一种更优解使得这种解通过提前上升或者下降来使得时间花费缩短。因为假设存在一种“更优解”，可以通过尽量延后上升操作而构造出一组满足上面两个原则的同样优的解。

最后的某条合法最短路径一定是长这样：先一直下降高度，直到高度为零，然后每次都上升到需要的高度后再飞。

这样就不会出现明明能够飞、但是高度不够而导致花费不必要的代价的情况。这样我们在跑最短路的同时记录一下当前所在点的高度，然后讨论一下每次飞的时候边的权重就可以了。这样实际的状态的数量只有正常最短路的状态数量。

可以在输入时直接删掉不合法的边。

时间复杂度 $O((n+m)log_2(n))$

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
const int N = 1e5 + 5, M = 3e5 + 5;
const LL inf = 1e18;
int n, m, x, h[N], h2[N];
LL dis[N];
int head[N], ver[M * 2], nxt[M * 2], edge[M * 2], l;
priority_queue<pair<int, int> > q;

inline void insert(int x, int y, int z) {
    l++;
    ver[l] = y;
    edge[l] = z; //hight
    nxt[l] = head[x];
    head[x] = l;
}

inline void Dijkstra(int now) {
    for(int i = 1; i <= n; ++i) dis[i] = inf;
    q.push(make_pair(0, 1));
    dis[1] = 0;
    h2[1] = now; //hight
    while (!q.empty()) {
        int x = q.top().second, d = -q.top().first, h1 = h2[x];
        q.pop();
        if (dis[x] != d) continue;
        for (int i = head[x]; i; i = nxt[i]) {
            int y = ver[i], tmp, w = -1;
            if (h1 == 0 && edge[i] <= h[x])
                w = 2 * edge[i], tmp = 0; // climb as needed
            else if (h1 - edge[i] > h[y])
                w = h1 - h[y], tmp = h[y]; // too high
            else if (h1 - edge[i] < 0 && edge[i] <= h[x])
```

```
                w = 2 * edge[i] - h1, tmp = 0; // climb as needed
            else if (h1 - edge[i] >= 0 && h1 - edge[i] <= h[y])
                w = edge[i], tmp = h1 - edge[i]; // too high
            if (w != -1 && dis[y] > dis[x] + w) {
                dis[y] = dis[x] + w;
                h2[y] = tmp;
                q.push(make_pair(-dis[y], y));
            }
        }
    }
    dis[n] += abs(h[n] - h2[n]);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n >> m >> x;
    for (int i = 1; i <= n; i++) cin >> h[i];
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        if (w <= h[u]) insert(u, v, w);
        if (w <= h[v]) insert(v, u, w);
    }
    Dijkstra(x);
    if (dis[n] >= inf) cout << "-1" << '\n';
    else cout << dis[n] << '\n';
    return 0;
}
```