1、

算法思路：

使用回溯算法，在保证插入数量不超过n和插入）的前提是（的数量大于）这两个条件下递归调用就可以了。

复杂度分析：

时间复杂度：O()

空间复杂度：O(n)

代码：

class Solution {

public:

    vector<string> generateParenthesis(int n) {

        vector<string> res;

        int lc = 0, rc = 0;

        dfs(res, "", n, lc, rc);

        return res;

    }

    void dfs(vector<string>& res, string path, int n, int lc, int rc) {

        if (rc > lc || lc > n || rc > n) return;

        if (lc == rc && lc == n) {

            res.push\_back(path);

            return;

        }

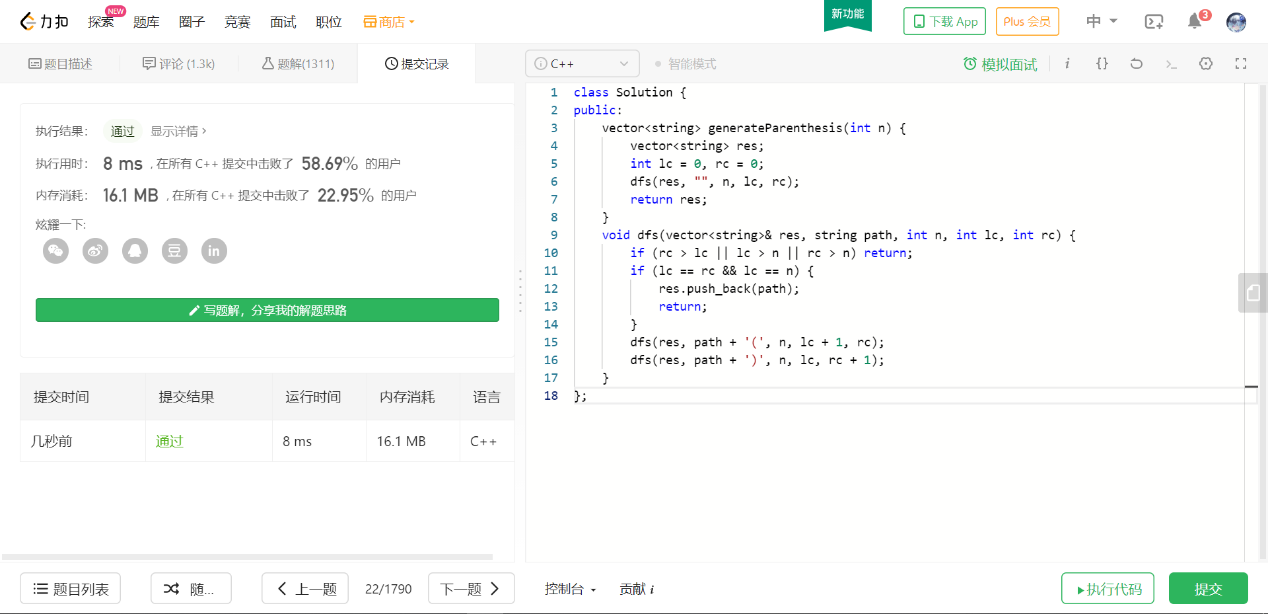
        dfs(res, path + '(', n, lc + 1, rc);

        dfs(res, path + ')', n, lc, rc + 1);

    }

};

截图：



2、

算法思路：

同样采用回溯算法，每次对于一个数都有选和不选两种选择，递归调用的两个if条件是终止条件，当target==0时将答案存入数组中，最后返回ans即可。

复杂度分析：

时间复杂度：O(S)，S=所有可行解的长度之和

空间复杂度：O(target)

代码：

class Solution {

public:

    void dfs(vector<int> &candidates,vector<vector<int> > &ans,int idx,int target,vector<int> &temp)//传入candidate数组、答案、选取到哪个位置、目标值d还剩余多少没选,当前已经选取的数组

    {

        if(idx==candidates.size())

        {

            return;

        }

        if(target==0)

        {

            ans.emplace\_back(temp);

            return;

        }

        else{

            //不选择

            dfs(candidates,ans,idx+1,target,temp);

            //选择

            if(target-candidates[idx]>=0)

        {

            temp.emplace\_back(candidates[idx]);

            dfs(candidates,ans,idx,target-candidates[idx],temp);

            temp.pop\_back();

        }

        }

    }

    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

        vector<vector<int> > ans;

        vector<int> temp;

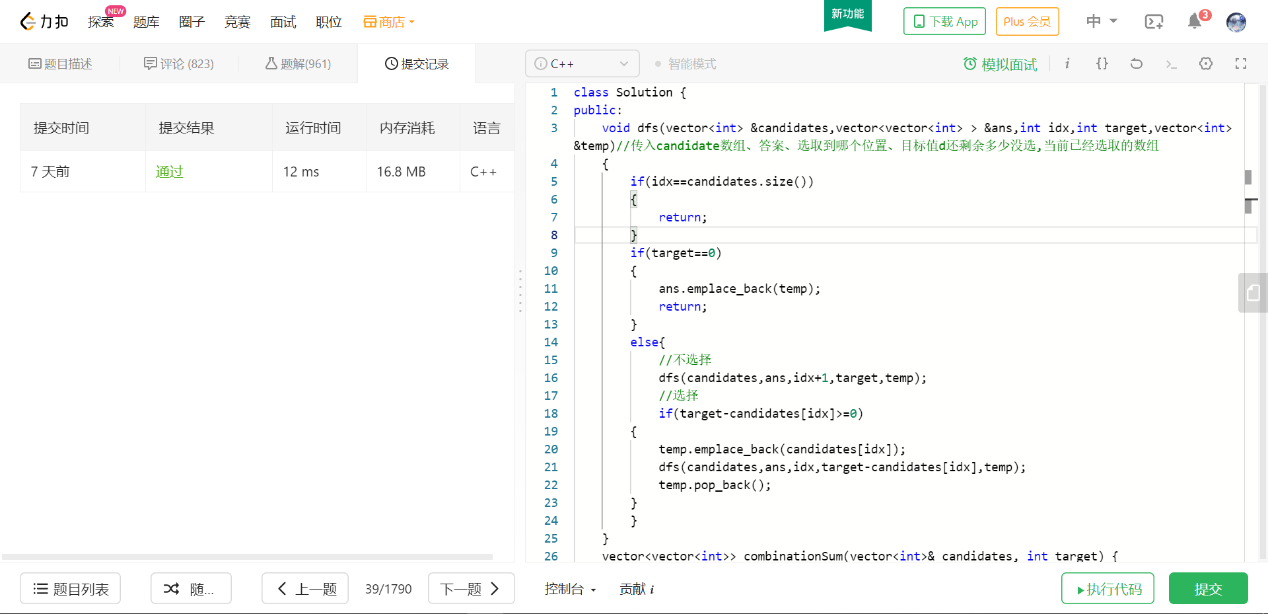
        dfs(candidates,ans,0,target,temp);

        return ans;

    }

};

截图：



3、

算法思路：

每次以双指针为左右边界（也就是「数组」的左右边界）计算出的容量中的最大值即可。

复杂度分析：

时间复杂度：O(n)，n为数组长度

空间复杂度：O(1)

代码：

class Solution {

public:

   int maxArea(vector<int>& height) {

    int res = 0;

    int i = 0;

    int j = height.size() - 1;

    while (i < j) {

        int area = (j - i) \* min(height[i], height[j]);

        res = max(res, area);

        if (height[i] < height[j]) {

            i++;

        } else {

            j--;

        }

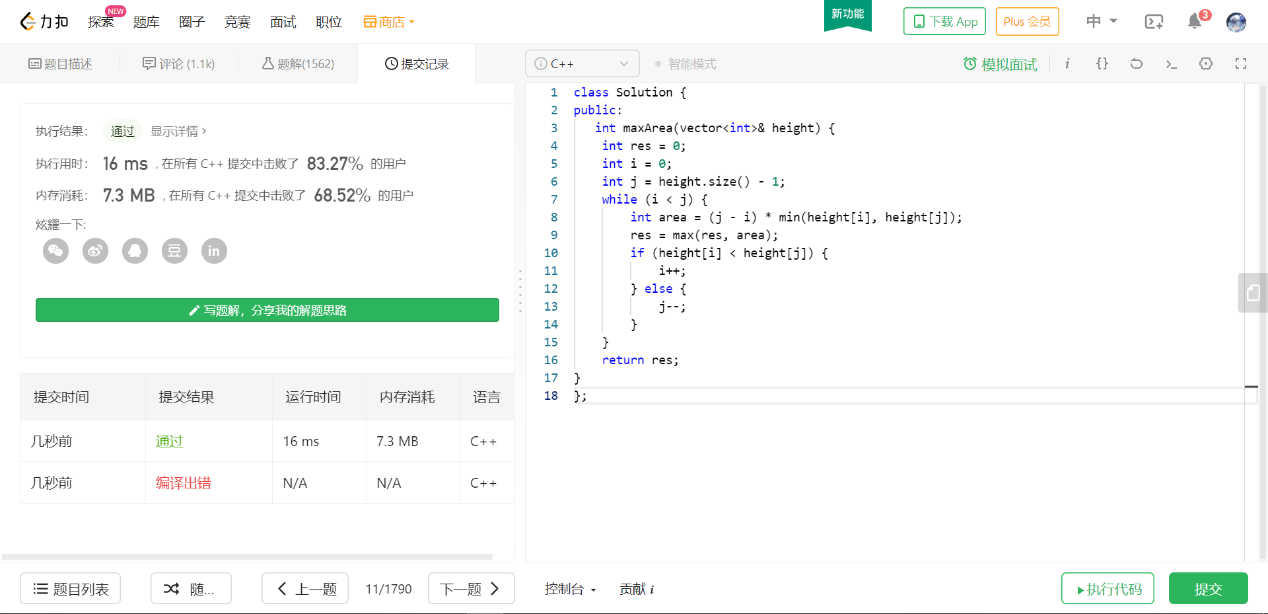
    }

    return res;

}

};

截图：



4、

算法思路：

差分约束的思想，使用前缀和转化推导不等式进行解决。

复杂度分析：

时间复杂度：O(n2)

空间复杂度：O(n)

代码：

#include <cstdio>

#include <cstring>

#include <queue>

using namespace std;

const int N = 25, M = 100;

struct E {

    int v, w, next;

} e[M];

int t, n, len, k, p[N], d[N], h[N], num[N], cnt[N];

bool vis[N];

void add(int u, int v, int w) {

    e[len].v = v;

    e[len].w = w;

    e[len].next = h[u];

    h[u] = len++;

}

bool spfa(int s24) {

    memset(h, 0, sizeof(h)), len = 1;

    memset(d, -0x3f, sizeof(d));//最小值求最长路

    memset(cnt, 0, sizeof(cnt));

    memset(vis, false, sizeof(vis));

    d[0] = 0; //从0可以遍历所有的边

    //根据枚举的s24进行建图

    for (int i = 1; i <= 24; i++) {

        add(i - 1, i, 0), add(i, i - 1, -num[i]);

        if (i >= 8) {

            add(i - 8, i, p[i]);

        } else {

            add(i + 16, i, -s24 + p[i]);

        }

    }

    //由于我们还需要设置 s24为定值 所以创建 s24<=k s24 >= k 这样s24就是定值了(k是枚举的s24的值)

    add(0, 24, s24), add(24, 0, -s24);

    //spfa看是否有解

    queue<int> q;

    q.push(0);

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        vis[u] = false;

        for (int j = h[u]; j; j = e[j].next) {

            int v = e[j].v;

            int w = d[u] + e[j].w;

            if (w > d[v]) {

                d[v] = w;

                cnt[v] = cnt[u] + 1;

                if (cnt[v] >= 25) return true;

                if (!vis[v]) q.push(v), vis[v] = true;

            }

        }

    }

    return false;

}

int main() {

    scanf("%d", &t);

    while (t--) {

        memset(num, 0, sizeof(num));//num[i] 代表i时刻的人数

        for (int i = 1; i <= 24; i++) scanf("%d", &p[i]);

        scanf("%d", &n);

        for (int i = 1; i <= n; i++) {

            scanf("%d", &k);

            num[++k]++;

        }

        //枚举一下s24的值

        bool ok = false;

        for (int i = 1; i <= n; i++) {

            if (!spfa(i)) {

                //找到一组结果

                printf("%d\n", d[24]); //输出一下一共需要的最少人数

                ok = true;

                break;

            }

        }

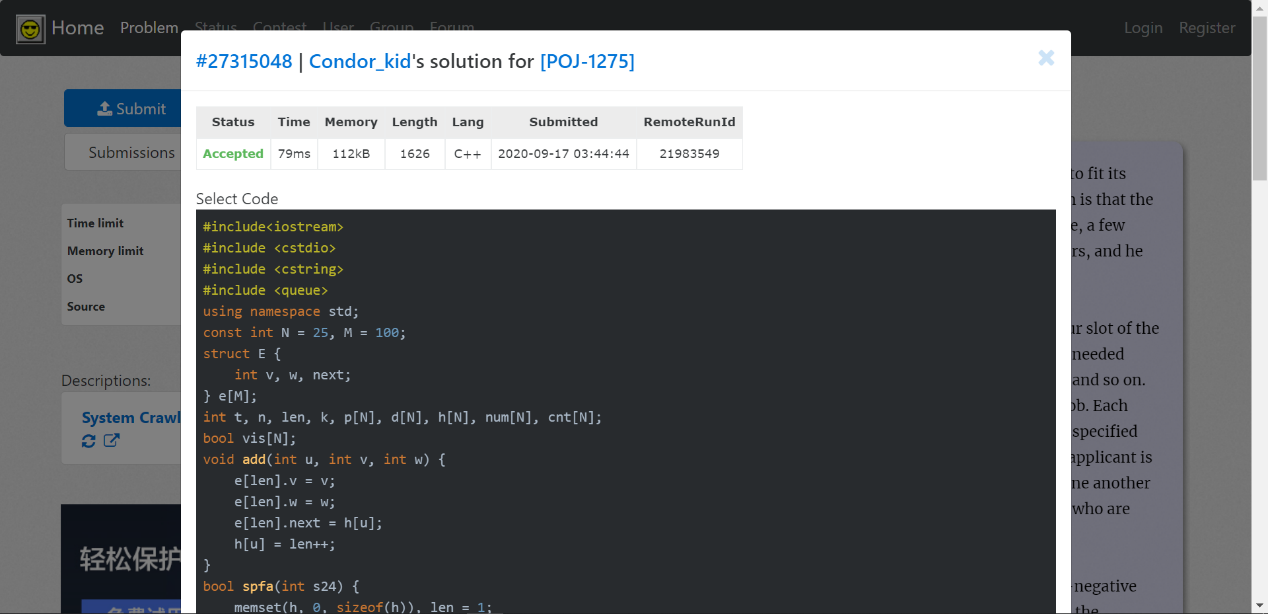
        if (!ok) printf("No Solution\n");

    }

    return 0;

}

截图：



5、

NP问题：能在多项式时间内验证得出一个正确解的问题。

P问题是NP问题的子集。

先证明它至少是一个NP问题，再证明其中一个已知的NP-完全性问题能约化到它。