1、
算法思路：

将二维数组视作一个无向图，两个 1 相邻视作两个相连的点。然后对这个数组进行一次遍历，采用深度优先搜索的方法，将遍历过的点设置为 0，那么最后所有的点都会被设置为 0，其中作为深度优先搜索的起始点有多少个就有多少个岛屿

复杂度分析：

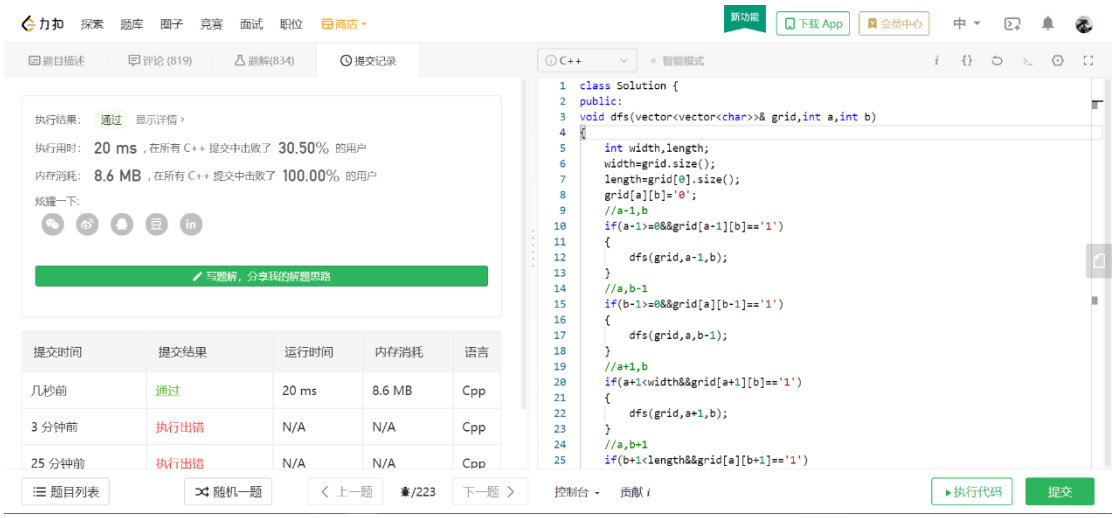
将这个数组遍历一次，时间复杂度为 O(n*m)，空间复杂度为 O(1)，其中，n*m 为数组的大小

代码：

```
class Solution {
public:
void dfs(vector<vector<char>>& grid,int a,int b)
{
    int width,length;
    width=grid.size();
    length=grid[0].size();
    grid[a][b]='0';
    //a-1,b
    if(a-1>=0&&grid[a-1][b]=='1')
    {
        dfs(grid,a-1,b);
    }
    //a,b-1
    if(b-1>=0&&grid[a][b-1]=='1')
    {
        dfs(grid,a,b-1);
    }
    //a+1,b
    if(a+1<width&&grid[a+1][b]=='1')
    {
        dfs(grid,a+1,b);
    }
    //a,b+1
    if(b+1<length&&grid[a][b+1]=='1')
    {
        dfs(grid,a,b+1);
    }
}
    int numIslands(vector<vector<char>>& grid) {
        int island_number=0;
        int width,length;
        width=grid.size();
```

```
        if(width==0)
        return 0;
        length=grid[0].size();
        for(int i=0;i<width;i++)
        {
            for(int j=0;j<length;j++)
            {
                if(grid[i][j]=='1')
                {
                    island_number++;
                    dfs(grid,i,j);
                }
            }
        }
        return island_number;
    }
};
```

截图：



2、

算法思路：

我们将在数组中的所有单词视作一个点，然后如果两个单词之间只有一个字母的差别的话，就将其视作两点之间存在一条边。那么我们便可以把问题简化成要求的两点之间最短路径的值是多少。

复杂度分析：

时间复杂度：O(n)，空间复杂度：O(n)，n 为数组的长度（也就是单词的个数）。

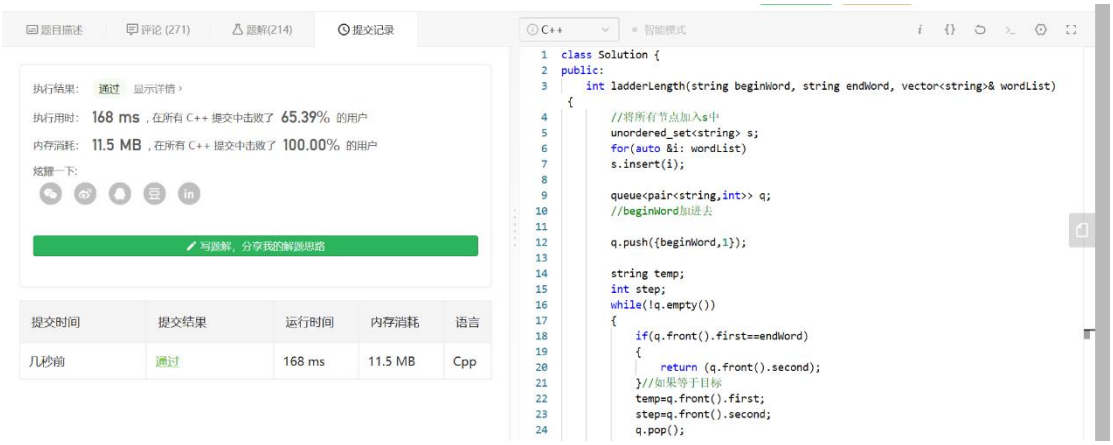
代码：

```
class Solution {
public:
```

```
    int ladderLength(string beginWord, string endWord, vector<string>&
wordList)     {
        //将所有节点加入 s 中
        unordered_set<string> s;
        for(auto &i: wordList)
        s.insert(i);

        queue<pair<string,int>> q;
        //beginWord 加进去

        q.push({beginWord,1});

        string temp;
        int step;
        while(!q.empty())
        {
            if(q.front().first==endWord)
            {
                return (q.front().second);
            }//如果等于目标
            temp=q.front().first;
            step=q.front().second;
            q.pop();

            char ch;
            for(int i=0;i<temp.length();i++)
            {
                ch=temp[i];
                for(char c='a';c<='z';c++)
                {
                    if(ch==c)
                    continue;
                    temp[i]=c;
                    if(s.find(temp)!=s.end())
                    {
                        q.push({temp,step+1});
                        s.erase(temp);
                    }
                    temp[i]=ch;
                }
            }
        }
        return 0;
    }
```

```
};
```

截图：



3、

算法思路：

对每个点作为起始点进行 BFS，终点是所有的点都被遍历过了，然后在这里面求最短路问题，并且采用位压缩的方式存储节点是否被遍历的状态。

复杂度分析：

时间复杂度：$O(2^N*N)$。

空间复杂度：$O(2^N*N)$。

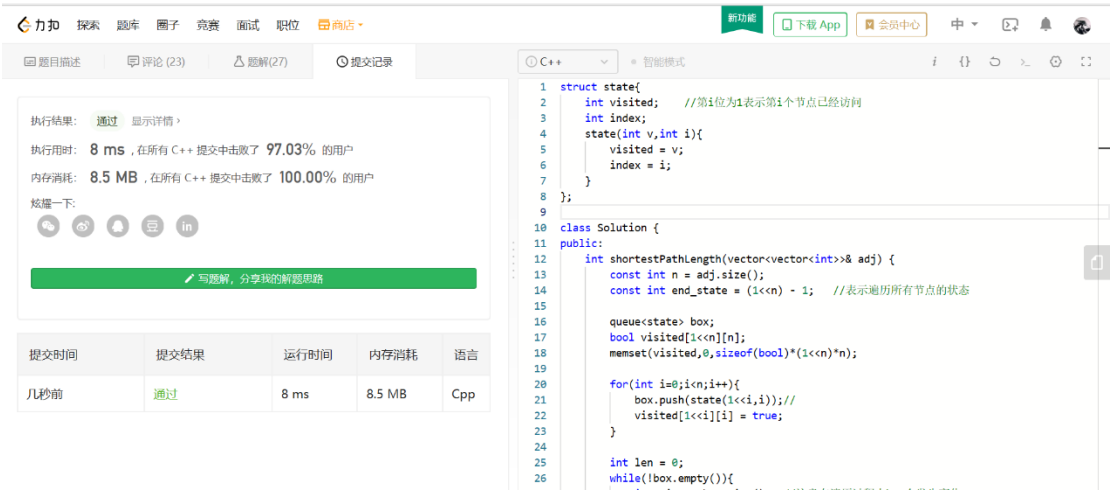
N 是节点个数。

代码：

```
struct state{
    int visited;    //第 i 位为 1 表示第 i 个节点已经访问
    int index;
    state(int v,int i){
        visited = v;
        index = i;
    }
};

class Solution {
public:
    int shortestPathLength(vector<vector<int>>& adj) {
        const int n = adj.size();
        const int end_state = (1<<n) - 1;   //表示遍历所有节点的状态

        queue<state> box;
        bool visited[1<<n][n];
        memset(visited,0,sizeof(bool)*(1<<n)*n);
```

```
        for(int i=0;i<n;i++){
            box.push(state(1<<i,i));//
            visited[1<<i][i] = true;
        }

        int len = 0;
        while(!box.empty()){
            int size = box.size();  //注意在遍历过程中 box 会发生变化
            for(int i=0;i<size;i++){
                state tmp = box.front();
                box.pop();

                for(int j=0;j<adj[tmp.index].size();j++){//BFS 的下一层
                            int next = adj[tmp.index][j];
                    int s = tmp.visited | (1<<next);    //表示访问节点
next，更新 state
                    if(s==end_state){
                        return len+1;
                    }
                    if(!visited[s][next]){  //未访问过
                        visited[s][next] = true;
                        box.push(state(s,next));
                    }
                }
            }
            len++;
        }
        return 0;
    }
};
```

截图:



4、

算法思路：用一个 k 来代表当前可以达到的最远距离，每次 i++时，更新 k 的值，如果遍历完整个数组 i>k 均不成立那么就可以达到数组末尾，否则则无法到达。

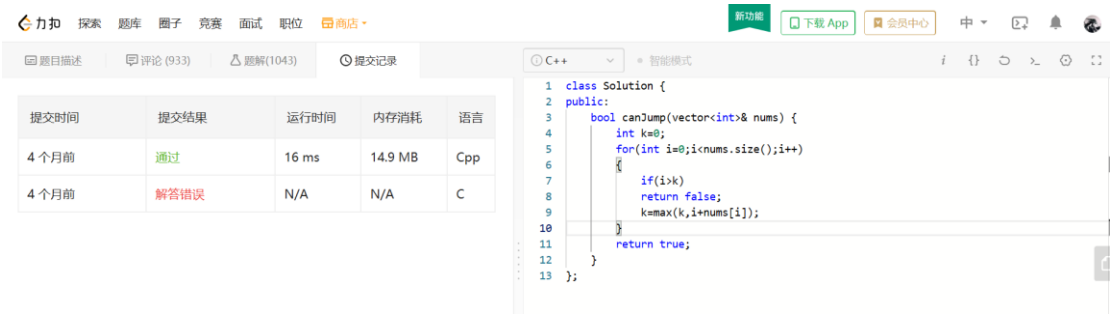
复杂度分析：

时间复杂度：O(n)，空间复杂度：O(1)，n 为数组大小。

代码：

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int k=0;
        for(int i=0;i<nums.size();i++)
        {
            if(i>k)
            return false;
            k=max(k,i+nums[i]);
        }
        return true;
    }
};
```

截图：



5、
算法思路：

采用最笨的方法，每次从一个起点开始计算，看下能否在符合规则的情况下跑完全程，如果可以就退出，不然就继续遍历，直到遍历完所有加油站为止。

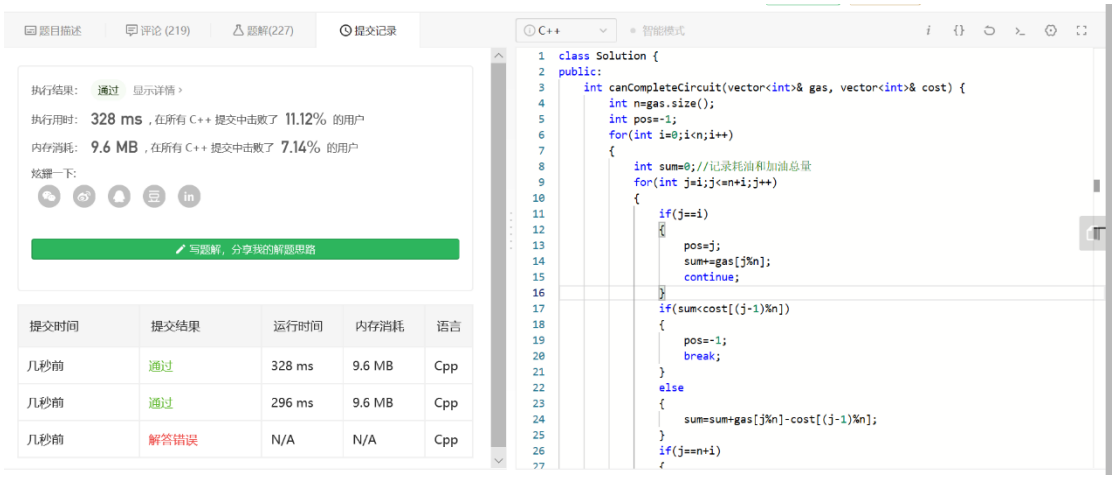
复杂度分析：

时间复杂度：$O(n^2)$，空间复杂度：O(1)。n 为 gas/cost 数组的大小。

代码：

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n=gas.size();
        int pos=-1;
        for(int i=0;i<n;i++)
        {
            int sum=0;//记录耗油和加油总量
            for(int j=i;j<=n+i;j++)
            {
                if(j==i)
                {
                    pos=j;
                    sum+=gas[j%n];
                    continue;
                }
                if(sum<cost[(j-1)%n])
                {
                    pos=-1;
                    break;
                }
                else
                {
                    sum=sum+gas[j%n]-cost[(j-1)%n];
                }
                if(j==n+i)
                {
                    goto A;
                }
            }
        }
        A:
        return pos;
    }
};
```

截图：


题目描述
评论 (219)
题解(227)
提交记录
C++
智能模式
执行结果： 通过 显示详情 ›
执行用时： 328 ms ，在所有 C++ 提交中击败了 11.12% 的用户
内存消耗： 9.6 MB ，在所有 C++ 提交中击败了 7.14% 的用户
炫耀一下：
写题解，分享我的解题思路
提交时间 提交结果 运行时间 内存消耗 语言
几秒前 通过 328 ms 9.6 MB Cpp
几秒前 通过 296 ms 9.6 MB Cpp
几秒前 解答错误 N/A N/A Cpp
class Solution {
public:
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
int n=gas.size();
int pos=-1;
for(int i=0;i<n;i++)
{
int sum=0;//记录耗油和加油总量
for(int j=i;j<n+i;j++)
{
if(j==i)
{
pos=j;
sum+=gas[j%n];
continue;
}
if(sum<cost[(j-1)%n])
{
pos=-1;
break;
}
else
{
sum=sum+gas[j%n]-cost[(j-1)%n];
}
if(j==n+i)