算法设计与应用基础：作业 2

18340052 何泽

Ⅰ. Number of Islands （题号200）

> 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
>
> 岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。
>
> 此外，你可以假设该网格的四条边均被水包围。
>
> ---
>
> 实例1：
>
> 输入:
> 11110
> 11010
> 11000
> 00000
>
> 输出: 1
>
> ---
>
> 实例2：
>
> 输入:
>
> 11000
> 11000
> 00100
> 00011
>
> 输出: 3
>
> 解释: 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

1. 算法思路

- 首先遍历所有元素，看元素值是否为1，也就是是否为陆地
  - 如果是 `0` ，即水，则不进行操作，继续遍历下一元素
  - 如果是 `1` ，那么首先将最终结果加 `1` ，再进入 `dfs()` 函数，上下左右遍历，并将遍历过的 `1` 赋值为 `0` ，直到无法遍历
- `dfs` 函数
  - 首先将本元素赋值为 `0`
  - 为了遍历该元素的上下左右元素，上、下、左、右在x、y方向分别的偏移值分别为 `(0.-1)` ，`(0,1)` ，`(-1,0)` ，`(1,0)` ，对应地存到两个数组中代表x、y方向的偏移
  - 如果某一方向的元素值为 `1` ，则以这个点为中心再调用 `dfs` 函数，直到都为 `0`

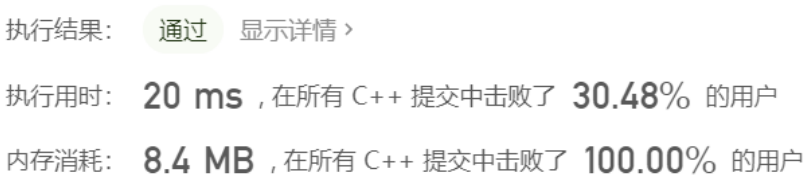
2. 复杂度分析

时间复杂度为$O(n^3)$

3. 代码

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int res = 0;
        for (int i = 0; i < grid.size(); i ++)
            for (int j = 0; j < grid[0].size(); j ++)
                if (grid[i][j] == '1') {
                    res ++;
                    dfs(grid, i, j);
                }
        return res;
    }
    void dfs(vector<vector<char>>& grid, int x, int y) {
        grid[x][y] = '0';
        int dx[4] = {0, 0, -1, 1}, dy[4] = {-1, 1, 0, 0};
        for (int i = 0; i < 4; i ++) {
            int a = x + dx[i], b = y + dy[i];
            if (a >=0 && b >= 0 && a < grid.size() && b < grid[0].size() &&
grid[a][b] == '1')
                dfs(grid, a, b);
        }
    }
};
```

4. Accept截图



II. 单词接龙 （题号127）

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：

每次转换只能改变一个字母。
转换过程中的中间单词必须是字典中的单词。

说明:

如果不存在这样的转换序列，返回 0。
所有单词具有相同的长度。
所有单词只由小写字母组成。
字典中不存在重复的单词。
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

> 示例 1:
>
> 输入:
> beginWord = "hit",
> endWord = "cog",
> wordList = ["hot","dot","dog","lot","log","cog"]
>
> 输出: 5
>
> 解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog", 返回它的长度 5
>
> ---
>
> 示例 2:
>
> 输入:
> beginWord = "hit"
> endWord = "cog"
> wordList = ["hot","dot","dog","lot","log"]
>
> 输出: 0
>
> 解释: endWord "cog" 不在字典中，所以无法进行转换。

1. 算法思路

- 大体思路是从 `beginWord` 开始转换，直到 `endWord` ，最终构成构成一个无向有环图
- 首先构建转换树，某个结点的孩子结点为一个集合，包含了所有能够进行一次转换得到的有效单词，按层从 `beginWord` 开始构建多叉树，直到遇到 `endWord` ，此时的层高+1即为最短路径
- 并且可以将图中某些重复结点的边去掉，以减少重复计算，因为只会选择先变换得到的单词，后续重复生成的相同单词的路径长度肯定会相同或更大，所以直接跳过不影响最终的结果

2. 复杂度分析

时间复杂度为$O(N^2M^2)$，其中N、M分别为词典中单词个数和每一个单词的字符数

3. 代码

```
1   class Solution {
2   public:
3       int ladderLength(string beginWord, string endWord, vector<string>& wordList)
    {
4           int word_size = wordList[0].length();
5           int list_size = wordList.size();
6           vector<set<string>> levels(list_size + 1, set<string>());
7           vector<int> flags(list_size, 0);
8           levels[0].insert(beginWord);
9           for (int i = 0; i < levels.size(); i++) {
10              for (auto it = levels[i].begin(); it != levels[i].end(); it++) {
11                  for (int k= 0; k < list_size; k++) {
12                      if (flags[k] == 1)
13                          continue;
14                      int l = 0, cnt = 0;
15                      while (l < word_size && cnt <= 1) {
16                          if ((*it)[l] != wordList[k][l])
17                              cnt++;
18                          l++;
19                      }
```

```
                    if (cnt == 1) {
                        flags[k] = 1;
                        levels[i+1].insert(wordList[k]);
                        if (wordList[k] == endWord)
                            return i + 2;
                    }
                }
            }
        }
        return 0;
    }
};
```

4. Accept截图

执行结果： 通过 显示详情 >

执行用时： 968 ms ，在所有 C++ 提交中击败了 26.25% 的用户

内存消耗： 11.5 MB ，在所有 C++ 提交中击败了 100.00% 的用户

Ⅲ. 访问所有节点的最短路径 （题号847）

给出 graph 为有 N 个节点（编号为 0, 1, 2, ..., N-1）的无向连通图。

graph.length = N，且只有节点 i 和 j 连通时，j != i 在列表 graph[i] 中恰好出现一次。

返回能够访问所有节点的最短路径的长度。你可以在任一节点开始和停止，也可以多次重访节点，并且可以重用边。

示例 1：

输入：[[1,2,3],[0],[0],[0]]
输出：4
解释：一个可能的路径为 [1,0,2,0,3]

示例 2：

输入：[[1],[0,2,4],[1,3,4],[2],[1,2]]
输出：4
解释：一个可能的路径为 [0,1,4,2,3]

提示：

1. $1 <= graph.length <= 12$
2. $0 <= graph[i].length < graph.length$

1. 算法思路

- 首先定义一个类，用来标记节点的状态，即是否被访问
- 然后从所有点同时出发进行BFS，按层次遍历

2. 复杂度分析

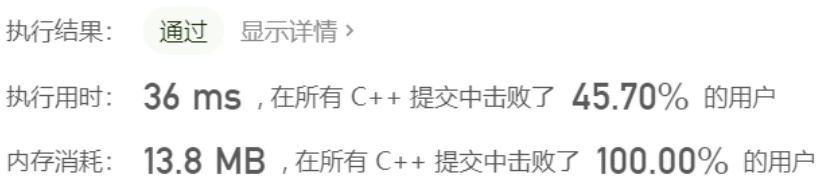
时间复杂度为$O(N \times 2^N)$

3. 代码

```cpp
struct State {
    int cover;
    int i;
    State(int _cover, int _i) : cover(_cover), i(_i) {}
};

class Solution {
public:
    int shortestPathLength(vector<vector<int>>& adj) {
        const int N = adj.size();
        const int END = (1 << N) - 1;
        vector<State> currs;
        vector<vector<int>> dist(1 << N, vector<int>(N, 0));
        for (int i = 0; i < N; ++i) {
            currs.push_back(State(1 << i, i));
            dist[1 << i][i] = 0;
        }
        for (int level = 0; !currs.empty(); ++level) {
            vector<State> nexts;
            for (auto curr : currs) {
                for (int j : adj[curr.i]) {
                    int v = curr.cover | (1 << j);
                    if (v == END) return level + 1;
                    if (!dist[v][j]) {
                        dist[v][j] = 1;
                        nexts.push_back(State(v, j));
                    }
                }
            }
            currs = nexts;
        }
        return 0;
    }
};
```

4. Accept截图



IV. 跳跃游戏 （题号55）

> 给定一个非负整数数组，你最初位于数组的第一个位置。
>
> 数组中的每个元素代表你在该位置可以跳跃的最大长度。

> 判断你是否能够到达最后一个位置。
>
> 示例 1:
>
> 输入: [2,3,1,1,4]
> 输出: true
> 解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。
>
> 示例 2:
>
> 输入: [3,2,1,0,4]
> 输出: false
> 解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

1. 算法思路

- 从后往前遍历，定义一个变量为到达最后一点目前遍历元素所需要的最小步长
- 在遍历过程中，如果该元素值小于当前最小步长，那么步长加一
- 如果该元素值大于当前最小步长，那么步长置为1

2. 复杂度分析

时间复杂度$O(N)$

3. 代码

```
1   ![4](C:\Users\03031\iCloudDrive\大二下\算法设计与应用基础\作业\作业2\4.PNG)class
    Solution {
2   public:
3       bool canJump(vector<int>& nums) {
4           int n=nums.size();
5           if(n==1)
6               return true;
7           int minstep=1;
8           for(int i=n-2;i>0;i--){
9               if(nums[i]<minstep)
10                  minstep++;
11              else
12                  minstep=1;
13          }
14          if(nums[0]<minstep)
15              return false;
16          return true;
17      }
18  };
```

4. Accept截图

执行结果： 通过 显示详情 >

执行用时： 20 ms ，在所有 C++ 提交中击败了 41.30% 的用户

内存消耗： 12.9 MB ，在所有 C++ 提交中击败了 8.70% 的用户

V. 加油站 （题号134）

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

如果题目有解，该答案即为唯一答案。
输入数组均为非空数组，且长度相同。
输入数组中的元素均为非负数。

示例 1:

输入:
gas = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出: 3

解释:
从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。

示例 2:

输入:
gas = [2,3,4]
cost = [3,4,3]

输出: -1

解释:
你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论怎样，你都不可能绕环路行驶一周。

1. 算法思路

- 首先想到的解法就是暴力算，遍历每一个点，考虑从第n个点出发能否回到该点，只需要判断当前油量减去到下一加油站的油耗是否大于等于0即可
- 但是暴力算法做了很多重复的计算，效率很低，于是考虑改进
- 如果 i 能到达的最远的点是 j，那么 i + 1 到 j 之间的节点都不可能绕一圈了，证明如下：
  - 假设 i + 1 的节点能绕一圈，那么就意味着从 i + 1 开始一定能到达 j + 1
  - 又因为从 i 能到达 i + 1，所以从 i 也能到达 j + 1。
  - 但事实上，i 最远到达 j，产生矛盾。所以 i + 1 的节点一定不能绕一圈。同理，其他的也是一样的证明。
- 所以下一次的 i 我们不需要从 i + 1 开始考虑，直接从 j + 1 开始考虑即可。

2. 代码

- 暴力法：

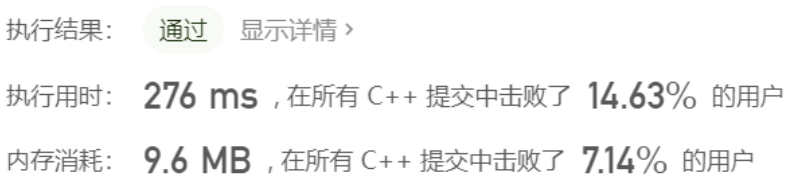
```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        for (int i = 0; i < n; i++) {
            int j = i;
            int remain = gas[i];
            while (remain - cost[j] >= 0) {
                remain = remain - cost[j] + gas[(j + 1) % n];
                j = (j + 1) % n;
                if (j == i)
                    return i;
            }
        }
        return -1;
    }
};
```

- 优化后：

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        for (int i = 0; i < n; i++) {
            int j = i;
            int remain = gas[i];
            while (remain - cost[j] >= 0) {
                remain = remain - cost[j] + gas[(j + 1) % n];
                j = (j + 1) % n;
                if (j == i)
                    return i;
            }
            if (j < i)
                return -1;
            i = j;
        }
        return -1;
    }
};
```

3. Accept截图

- 暴力法:



- 优化后