算法设计与应用 作业2

姓名：TRY

学号：

专业：计算机科学与技术

1.【Leetcode 200】岛屿数量

（1）算法思路

- 这道题考察的是DFS\BFS算法，本质处理都一样，都是计算图中的子图数目。下面展示DFS算法和DFS算法的优化。
- 首先，可以将二维网络看成无向图，相邻（水平或竖直）是1的一对数可以看成有边相连。
- 然后，扫描图。如果该位置是`'1'`，则以该位置开始进行DFS。**关键：在DFS过程中，每个遍历到的点都被重新改为`'0'`。**
- 每次DFS，都要递归进行4个方向的DFS**（在边界范围内）**。
- 最终的岛屿数就是我们进行的DFS的次数。

（2）复杂度分析

- **时间复杂度：O(MN)**，每个点都要被遍历2/3/4遍，其中M和N分别为行数和列数。
- **空间复杂度：O(MN)**，在最坏情况下，整个图都是由陆地组成的，DFS的深度达到MN。

（3）代码

- **法一：一般的DFS**

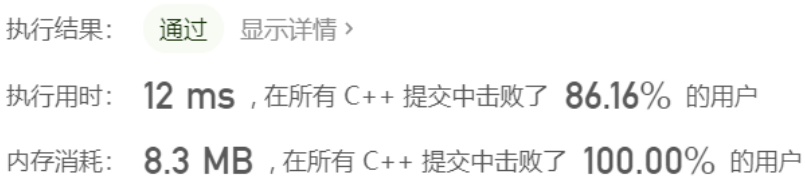
```
void dfs(vector<vector<char>>& grid, int row, int col)
{
    int m = grid.size(), n = grid[0].size();
    if (grid[row][col] == '1')
    {
        grid[row][col] = '0';
        if (col != n - 1)
            dfs(grid, row, col + 1);
        if (row != m - 1)
            dfs(grid, row + 1, col);
        if (col != 0)
            dfs(grid, row, col - 1);
        if (row != 0)
            dfs(grid, row - 1, col);
    }
    return;
}

int numIslands(vector<vector<char>>& grid) {
    if (grid.size() == 0 || grid[0].size() == 0)
```

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

- **法二：优化的DFS**
  - **注：**其实这个DFS的复杂度仍不变，只不过这里借用了数组`dx和dy`，使得可以只写一个`if`判断来完成4个`if`判断的工作，更加简洁。

```
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] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
        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) 截图



2.【Leetcode 127】单词接龙

（1）算法思路

- 这道题本质考察BFS算法。
- 由于单词之间的转换只发生在只差一个字符的单词之间，所以，可以看成这些单词是图中的**节点**，而满足此条件的单词之间存在一条**无向边**。问题的解转化为求从这个无向无权图的起点（起始单词）到终点（终止单词）的最短路径。
- 这种无向无权图的最短路径就是用BFS来解决问题的。
- 而这道题的难点在于：如何构建图？即如何表示各个单词的可转换集合？
- 有两种方法可以解决：
  - **法一：构建邻接表。**
    - 为了方便起见，用map构建图， map的key是单词的`string`序列，map的value是`vector<string>`,即当前key可以转换到的单词序列。
    - BFS时，需要标记已经访问过的单词元素，所以可以用**set**来储存已经访问过得元素。（这不是`int`，无法用一个`vector<int> visit`来表示）
  - **法二：二重循环遍历查找**
    - 由于构建图的过程复杂度是O(n^2)，复杂度较高，所以可以不构建图，而采用二重循环的方式遍历该单词的每个字母，进行`'a'`-`'z'`的替换，看替换后的单词是否在原来的单词表里且未被访问过，如果在则将这个词加到队列中，同时把该节点移除掉。
    - 此时，相当于是用`set`来储存图（不显式的画出单词转换图，而是遍历来查找），所以可以直接将已经考虑的单词从`set`中**除去**，而**不需要**引入额外的数据结构空间来标记`visit`。

（2）复杂度分析

- **时间复杂度**：
  - **法一：O(MN)**，其中M为单词的长度，N为单词的总数。
    - 由于构建图的时候每个单词需要遍历N遍，每个单词之间的比较需要M遍，所以复杂度为O(MN)。
    - 而由于下面查找的过程中，visit使用`unordered_set<string>`构建的，一般查找复杂度为O(1)，所以下面的BFS复杂度在最坏情况下为O(N)。
    - 总体复杂度为O(MN)。
  - **法二：O(N)**，N为单词的总数。
    - 由于此时使用循环遍历查找可转换的单词数，每个单词遍历`'a'-'z'`共`26`遍，是常数，所以不会像之前计算构建图的时候那样，因为单词的字母数增加而增加。
    - 所以，总体时间复杂度为O(N)。
- **空间复杂度**：
  - **法一：O(N^2)**，N为单词的总数。
    - 要在`unordered_map<string, vector<string> >`中最多使用O(N^2)的空间记录图，`queue<string> Q`最坏情况下有O(N)的空间，visit集合需要O(N)的空间。
  - **法二：O(N)**，N为单词的总数。
    - `unordered_set<string> s`需要O(N)的空间，`queue<pair<string, int>> q`最坏情况下需要O(N)的空间。

(3) 代码

- **法一：邻接表构建图，储存可转换的单词**

```
bool connect(const string& word1, const string &word2)//判断两个单词是否仅一个字母不同
{
    int dif = 0; //不同的字母数量
    for (int i = 0; i < word1.length(); i++)
    {
        if (word1[i] != word2[i])
            dif++;
        if (dif > 1)
            return false;
    }
    return dif == 1;//! ! ! 这样返回
}

//构建图
void construct_graph(string &beginWord, vector<string> &wordList, unordered_map<string, vector<string> > &graph)
{
    wordList.push_back(beginWord);
    for (int i = 0; i < wordList.size(); i++)
        for (int j = i + 1; j < wordList.size(); j++)//避免重复
        {
            if (connect(wordList[i], wordList[j]) == true)
            {
                graph[wordList[i]].push_back(wordList[j]);
                graph[wordList[j]].push_back(wordList[i]);
            }
        }
}
int ladderLength(string beginWord, string endWord, vector<string>& wordList)
{
    unordered_map<string, vector<string> > graph;
    construct_graph(beginWord, wordList, graph);
    queue<string> Q;
    unordered_set<string> visit;//记录已经访问过的元素
    Q.push(beginWord);
    visit.insert(beginWord);
    int result = 1;
    while (!Q.empty())
    {
        int size = Q.size();//记录此时queue大小
        for (int i = 0; i < size; i++) //将这一层的节点全部处理掉
        {
            string tmp = Q.front();
            Q.pop();
            vector<string> neighbors = graph[tmp]; //邻居
            for (int i = 0; i < neighbors.size(); i++)
            {
                if (visit.find(neighbors[i]) == visit.end())  //还没加入
                {
                    Q.push(neighbors[i]);
```

```
                    visit.insert(neighbors[i]);
                }
                if (neighbors[i] == endWord)
                    return result + 1;
            }
        }
        result++;  //遍历一层则+1
    }
    return 0;
}
```

- **法二：二重循环遍历查找**

```
int ladderLength(string beginWord, string endWord, vector<string>& wordList)
{
        //加入所有节点，访问过一次，删除一个。
        unordered_set<string> s;
        for (auto &i : wordList) s.insert(i);

        queue<pair<string, int>> q;
        q.push({beginWord, 1});//加入beginword

        string tmp; //每个节点的字符
        int step;    //抵达该节点的step

        while ( !q.empty() ){
            if ( q.front().first == endWord){
                return (q.front().second);
            }
            tmp = q.front().first;
            step = q.front().second;
            q.pop();

            //寻找下一个单词了
            char ch;
            for (int i = 0; i < tmp.length(); i++){
                ch = tmp[i];//更改
                for (char c = 'a'; c <= 'z'; c++){
                    if ( ch == c) continue;
                    tmp[i] = c ;
                    //如果找到的到
                    if (  s.find(tmp) != s.end() ){
                        q.push({tmp, step+1});
                        s.erase(tmp) ; //删除该节点
                    }
                tmp[i] = ch; //复原
                }

            }
        }
        return 0;
    }
```

(4) 截图

- 其中，下面的是法一的时间，上面的是法二的时间。

提交时间	提交结果	运行时间	内存消耗	语言
30 分钟前	通过	164 ms	11.4 MB	Cpp
1 小时前	通过	996 ms	29 MB	Cpp

3.【Leetcode 847】访问所有节点的最短路径

(1) 算法思路

- 本题考察的是BFS。（也可以用BFS+DP的思路，但此处只介绍纯BFS的方法）
- 一开始，我们很容易可以想到"暴力法"，即对每一个节点都使用BFS，然后终止条件是所有点都被遍历过了，转化为"最短路径问题"。结果就是所有的BFS里面最小的。但是，这个暴力方法的复杂度非常大。
- 所以，可以改为多个节点**同时开始BFS**，当有某一个节点的BFS到达终止条件时，该结果就是最短路径。降低时间复杂度。
- 而最短路径问题的难点之一在于**如何标记已访问的节点。**可以很容易想到，使用`unordered_set<int> visited`来进行标记。然而，由于题目中有条件`"0<=N<=12"`，所以，总的状态数最多只有2^12种，也就是最多是2^12个状态，是可以用整型存下来的，所以可以使用**位压缩**来代替上面的`unordered_Set`，即使用一个`int visit`来表示已遍历的节点（包括之前遍历的节点+当前节点）。因此，状态表示如下：

```
typedef struct state {
    int visit;//用位运算符来表示：当前节点+已访问节点（N有范围）
    int cur;//当前节点
    state(int v, int c)
    {
        cur = c;
        visit = v;
    }
}state;
```

- 然后，需要考虑的是**如何储存当前节点和已访问节点的最短路径长度。**经过大佬题解里面的提示，采用一个`vector<vector<int>> dist`数组，数组**行**是已访问的节点（位表示法），**列**是表示当前节点的索引，**各单元内容**为访问当前节点时的最短路径。
- 采用**队列**来实现**BFS**，队列中存的是`state`。每次利用队头`pop`出的`State`，更新`dist`数组。
- *这道题应该是本次作业里面最难的一道吧，不看题解的提示根本想不到要这样处理。*

(2) 复杂度分析

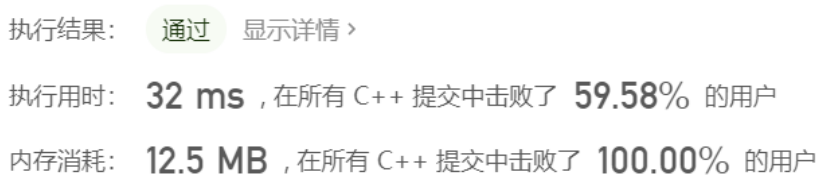
- **时间复杂度**：**O(2^N*N)**。位运算表示状态的visit一共有2^N个，而cur一共有N个，所以状态一共有O(2^N *N)。

- **空间复杂度**：同理，也是**O(2^N*N)**，对应状态数。

(3) 代码

```
typedef struct state {
    int visit;//用位运算符来表示：当前节点+已访问节点（N有范围）
    int cur;//当前节点
    state(int v, int c)
    {
        cur = c;
        visit = v;
    }
}state;

int shortestPathLength(vector<vector<int>>& graph) {
    int n = graph.size();//图的节点数
    if (n == 0)
        return 0;
    int state_num = 1 << n;
    queue<state> myqueue;
    vector<vector<int>> dist(1 << state_num, vector<int>(n, INT_MAX));

    int end = (1 << n) - 1;//终止状态：用位运算表示

    queue<state> q;
    //队列初始化
    for (int i = 0; i < n; i++)
    {
        q.push(state(1 << i, i));
        dist[1 << i][i] = 0;
    }
    while (!q.empty())
    {
        state pre = q.front();
        q.pop();
        int visit1 = pre.visit;
        int dist1 = dist[visit1][pre.cur];
        if (visit1 == end)
            return dist1;
        for (int node : graph[pre.cur])//遍历当前节点的邻接点
        {
            int visit_new = visit1 | (1 << node);//新状态
            if (dist[visit_new][node] > dist1 + 1)
            {
                dist[visit_new][node] = dist1 + 1;
                q.push(state(visit_new, node));
            }
        }
    }
    return INT_MAX;

}
```

(4) 截图



4.【Leetcode 55】跳跃游戏

(1) 算法思路

- 由于这道题我曾在寒假的时候做过，所以以下部分内容来源于我的题解。

- 这道题考察的是**贪心算法。**
- 对于这道题，其实可以这样理解：对于 数组中的任意一个位置`y`，能否跳跃到达y，意味着是否存在一个位置`x`，有`x+nums[x] >=y`。如果存在，则`y`可以到达。
- 所以，可以在遍历每一个位置的时候，**实时维护当前可以达到的最远的位置k。**如果在遍历到`x`的时候，`k<x`，则说明不能到达`x`，返回`false`；否则，说明可以到达`x`，继续下一轮遍历。
- 如果`k>nums.size()`，则可以直接返回true。且这个条件可以放到**for循环条件**里面判断。

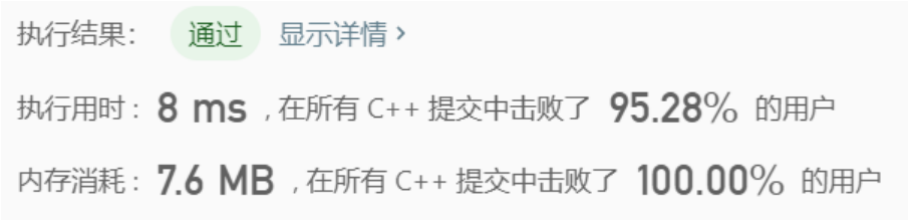
(2) 复杂度分析

- **时间复杂度：O(n)**，其中n为数组的大小。
  - 最多主需要访问`nums`数组一遍，所以复杂度为O(n)。
- **空间复杂度：O(1)。**不需要额外的空间开销。

(3) 代码

```
bool canJump(vector<int>& nums)
{
    int k = 0;
    for (int i = 0; k < nums.size()-1; i++)//条件是k<nums.size()
    {
        if (i > k) return false;
        k = max(k, i + nums[i]);
    }
    return true;
}
```

(4) 截图



5.【Leetcode 134】加油站

（1）算法思路

- 同样，这道题我曾在寒假的时候完成过，所以以下部分内容来自我的题解。

- 这道题考察的同样是贪心算法。
- 一开始看到这道题，我的想法就是用**双指针**，模拟**循环数组**的操作。
- 对于任意一个位置x，他能否到达x意味着：当前汽油剩余量temp+gas[x]-cost[x]是否大于等于0？如果是，则可以到达x加油站；否则，不可到达x加油站。
- 然而，本题的**关键在于找到循环行使的起点。**因为并不是从每一个点开始，都可以成功行使一周。所以，我设置了**双指针**，初始的时候，`left=0，right=1`。然后，当`left！=right`的时候，一直`while`循环。循环内判断当前汽油剩余量`temp`是否大于0？
  - 是，则说明以`left`作为行使起点，可以继续往后走，`right++`；
  - 否则，说明以`left`作为行使起点，无法继续往后走，即当前left不符合起点的条件，且`[left,right]`之间的点也不符合起点的条件（证明如下）。所以，此时`left--`，也就是检查看left之前的节点是否满足起点的条件，使得`temp>0`。
    - 因为前面`[left,right]`之间的节点已经证明是可达的，也就是这些点的`temp`都是大于`0`的，即使从left的后面开始作为起点，则这种情况下对应点的`temp`值只会更小，所以`[left+1,right]`这些节点也不会可以到达终点。
- 并且，本题还可以做一个**优化**：可以在一开始，通过`stl`**的**`accumulate`**函数**计算可以加油的总量`gas_sum`，并且同样计算消耗的总量`cost_sum`，比较两者之间的值，如果`cost_sum>gas_sum`，则不需要进行下面的循环判断了，直接返回`-1`.

（2）复杂度分析

- **时间复杂度：O(N)**，其中N为数组的大小。
  - 由于只会把数组遍历1遍，且accumulate的复杂度同样是O(N)，所以总的复杂度为O(N)。
- **空间复杂度：O(1)**，只用到了常数个变量。

（3）代码

```
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    if (gas.size() == 1)
```

```
        return gas[0] >= cost[0] ? 0 : -1;
    int gas_sum = accumulate(gas.begin(), gas.end(),0);//0为累加的初值
    int cost_sum = accumulate(cost.begin(), cost.end(), 0);
    if (gas_sum < cost_sum )//这里进行初始判断，如果总汽油量小于总花费量直接返回-1.
        return -1;
    int left = 0;//由于条件以满足，故left可以任意赋初值。
    int right = 1;
    int temp = gas[left]-cost[left];
    int N=gas.size();
    while (right != left)
    {
        if (temp >= 0)
        {
            temp += gas[right] - cost[right];//这里要先加再改right，因为之前还没有加
            right = right == N - 1 ? 0 : right + 1;
        }
        else
        {
            left = left == 0 ? N - 1 : left - 1;//这里要先改left再加。
            temp += gas[left] - cost[left];
        }
    }
    return left;
}
```

(4) 截图