2022.11.10

每日一题 [864 获取钥匙的最短路径]

三维 BFS 再加状压解决。呜呜呜一写就废。

首先观察要求路径，基本可以确定需要用到 BFS ，来观察其状态需要又几个变量表示：首先是坐标所在位置横纵坐标，另外，当我们遇到一扇门的时候需要判断是否有钥匙，因此走到每一步时所带的是哪些钥匙也必须设计进去。

故有表示的数组 $dist[i][j][k]$ 表示为：当前在 $(x，y)$ 坐标下且其所带的钥匙状态数为 k 时的最小步数，那么钥匙状态一共有几种呢，根据容斥原理应最多有 2^k 种(k 为钥匙的种数)，因此用每一个二进制位代表该钥匙是否已经被取到。

```
class Solution {
public:
    int shortestPathAllKeys(vector<string>& grid) {
        int m = grid.size(), n = grid[0].size();
        int sx = 0, sy = 0;
        unordered_map<char, int> key_to_idx;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '@') {  // 找起点
                    sx = i;
                    sy = j;
                }
                else if (islower(grid[i][j])) {  // 找一共有几把钥匙出现
                    if (!key_to_idx.count(grid[i][j])) {
                        int idx = key_to_idx.size();
                        key_to_idx[grid[i][j]] = idx;
                    }
                }
            }
        }

        queue<tuple<int, int, int>> q;
        vector<vector<vector<int>>> dist(m, vector<vector<int>>(n, vector<int>(1
<< key_to_idx.size(), -1)));
        q.emplace(sx, sy, 0);
        dist[sx][sy][0] = 0;
        while (!q.empty()) {
            auto [x, y, mask] = q.front();
            q.pop();
            for (int i = 0; i < 4; ++i) {
                int nx = x + dirs[i][0];
                int ny = y + dirs[i][1];
                if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] !=
'#') {
                    // 这里确实分类写好一些
                    if (grid[nx][ny] == '.' || grid[nx][ny] == '@') {
                        if (dist[nx][ny][mask] == -1) {
                            dist[nx][ny][mask] = dist[x][y][mask] + 1;
```

```
                            q.emplace(nx, ny, mask);
                        }
                    }
                    else if (islower(grid[nx][ny])) {
                        int idx = key_to_idx[grid[nx][ny]];
                        if (dist[nx][ny][mask | (1 << idx)] == -1) {
                            dist[nx][ny][mask | (1 << idx)] = dist[x][y][mask] +
1;
                            if ((mask | (1 << idx)) == (1 << key_to_idx.size())
- 1) {
                                return dist[nx][ny][mask | (1 << idx)];
                            }
                            q.emplace(nx, ny, mask | (1 << idx));
                        }
                    }
                    else {
                        int idx = key_to_idx[tolower(grid[nx][ny])];
                        if ((mask & (1 << idx)) && dist[nx][ny][mask] == -1) {
                            dist[nx][ny][mask] = dist[x][y][mask] + 1;
                            q.emplace(nx, ny, mask);
                        }
                    }
                }
            }
        }
        return -1;
    }

private:
    static constexpr int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
};
```

像这种纯模拟的写的时候一些细节还是很重要的，比如值得学习的点：

1. 队列里存的是一个 $tuple$ 三元组而不是 $pair$ 二元组，这样转移遍历的时候情况更加针对具体。
2. 在对当前的 $grid[i][j]$ 进行处理时分类讨论，虽然写着繁琐但是更加清晰不容易出错。
3. 位运算时的一些细节需要注意。
4. $dirs$ 数组的应用。

673 最长递增子序列的个数

方法一

区间DP的一道题。

在原来最长子序列的基础上，只需要标记一下这个点可以由哪些点转移过来即可。

再深入思考一步，事实上我们甚至不需要知道哪些点转移过来的，我们只需要知道什么时候能转移过来的时候多加的最长子序列的个数即可，而这个增加的个数可以与最长子序列同步求解。

设数组 $dp[i][2]$ 表示遍历到第 i 位的时候的状态，其中第 0 维表示以这一位结尾时的最长子序列的长度，而第 1 维表示以这一位结尾时满足的最长子序列的个数。则遍历前 $i-1$ 位后可得转移方程：

$$dp[i][0] = max\{dp[j][0]+1, dp[i][0]\}, j = 0, 1, \ldots, i-1$$

$$dp[i][1] = \sum_{j=0}^{i-1} cost_{ij}$$

$$\text{其中} cost_{ij} = \begin{cases} dp[j][1] & , dp[j][0]+1 = dp[i][0] \;\&\&\; j < i \\ 0 & , else \end{cases}$$

最后遍历一遍数组，寻求满足最长子序列长度的结尾的数字有几个，把每一个对应的个数相加即可。时间复杂度 $O(n^2)$。

```
class Solution {
public:
    int findNumberOfLIS(vector<int>& nums) {
        int len = nums.size();
        vector<vector<int>> dp(len+1,vector<int>(2,1));
        int maxLen = 1;
        for(int i = 1; i < len; i++)
            for(int j = 0; j < i; j++) {
                if(nums[i] > nums[j]) {
                    if(dp[j][0] + 1 > dp[i][0]) {  // 此时表示不是最长子序列，更新最优
解
                        dp[i][0] = dp[j][0] + 1;
                        dp[i][1] = dp[j][1];
                        maxLen = max(maxLen, dp[i][0]);
                    }
                    else if(dp[j][0] + 1 == dp[i][0])  // 此时表示有多条路可以到达该
最优解，相加结果
                        dp[i][1] += dp[j][1];
                }
            }
        int ans = 0;
        for(int i = 0; i < len; i++)
            if(dp[i][0] == maxLen) ans += dp[i][1];
        return ans;
    }
};
```

方法二

对比 LIS 还有 $O(nlogn)$ 解法，这个题应该也有。

结合传统的 LIS 解法和法一的新增的那一维度，即可得到该做法。

```
class Solution {
    int binarySearch(int n, function<bool(int)> f) {
        int l = 0, r = n;
        while (l < r) {
            int mid = (l + r) / 2;
            if (f(mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l;
```

```
    }

public:
    int findNumberOfLIS(vector<int> &nums) {
        vector<vector<int>> d, cnt;
        for (int v : nums) {
            int i = binarySearch(d.size(), [&](int i) { return d[i].back() >= v;
});
            int c = 1;
            if (i > 0) {
                int k = binarySearch(d[i - 1].size(), [&](int k) { return d[i -
1][k] < v; });
                c = cnt[i - 1].back() - cnt[i - 1][k];
            }
            if (i == d.size()) {
                d.push_back({v});
                cnt.push_back({0, c});
            } else {
                d[i].push_back(v);
                cnt[i].push_back(cnt[i].back() + c);
            }
        }
        return cnt.back().back();
    }
};
```

522 最长特殊子序列二

嗯这个题表述挺恶心的。说不清楚，建议看英文。

有一个重要性质：**如果一个字符串本身是其他字符串的子串的话，那么这个字符串的子串也一定是其他字符串的子串。**

因此我们只需要比较每个子串是否是其他字符串的子串即可。那么如何比较一个字符串是否是另一个字符串的子串呢?

1. 法一：求两个字符串的最长公共子序列比长度。
2. 法二：双指针，遍历被比较的字符串一遍后看是否另一个字符串完全出现。本题采取这种做法。

时间复杂度 $O(n^2l)$ 可过，其中 l 为每个字符串的最大长度。

```
class Solution {
public:
    int findLUSlength(vector<string>& strs) {
        int ans = -1, len = strs.size();
        for(int i = 0; i < len; i++) {
            bool flag = true;
            for(int j = 0; j < len; j++)
                if(i != j && notcheck(strs[i],strs[j])) {
                    flag = false;
                    break;
                }
            if(flag) ans = max(ans, static_cast<int>(strs[i].size()));
        }
        return ans;
    }
```

```
    bool notcheck(string str, string sub) {
        int tmp = 0;
        for(auto ch : sub)
            if(tmp < str.size() && ch == str[tmp])
                tmp++;
        return tmp == str.size();
    }
};
```