计算机算法设计与分析

第 2 次作业

主讲教师：卜东波 *Section A*

张倩倩
202128007329011

Problem 1

对应 sep 发布的作业题 1：Money robbing A robber is planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night. 1. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police. 2. What if all houses are arranged in a circle?

the optimal substructure and DP equation

用 *dp*[i] 表示前 i 间房屋能偷窃到的最高总金额。如果只有一间房屋，则可以偷窃到最高总金额就是该间房屋里的金钱：dp[0]=nums[0]。如果只有两间房屋（隐藏条件：两间房屋相邻不能同时偷窃，只能偷窃其中的一间房屋）所以偷窃到最高总金额为两个房间金钱中钱最多的那个房间 dp[1]=max(nums[0],nums[1])。如果房屋数量大于两间，有两个选项：偷或者不偷。如果偷第 m 间房 (m>2) 那就不能偷窃 m-1 间房，偷窃总金额为前 m-2 间房屋的最高总金额与第 m 间房屋的金额之和,dp[i-2]+nums[i]。如果不偷，则总金额为前 m-1 间房屋的最高总金额 dp[i-1]。在两个选项中选择大的那项，即为前 m 间房屋能偷窃到的最高总金额。最终返回 *dp*[n-1]，其中 n 是数组的长度。则对应的 DP 方程为：

$$\text{dp[i]} = \begin{cases} \text{nums[0]} & \text{i} = 0 \\ \text{max(nums[0],nums[1])} & \text{i} = 1 \\ \text{dp[i-2]+nums[i],dp [i-1]} & 1 < i < \text{ nums.length} \end{cases} \tag{1}$$

Solution

Algorithm 1 求解打家劫舍

Input: 一个代表每个房屋存放金额的非负整数数组。
Output: 一夜之内能够偷窃到的最高金额。

```
1: function MONEYROB(nums)
2:     if(n=0)            //如果房间数为 0，那么就直接返回盗窃的金额为 0
3:         return 0
4:     end if
5:     if(n=1)            //如果房间数为 1，则最大收益为第一间房间内藏有的全部金额
6:         return nums[0]
7:     end if
8:     //创建 dp 数组，用来记录状态
9:     //初始化 dp[0] 和 dp[1]
10:    dp[0] = nums[0]
11:    dp[1] = max(nums[0], nums[1])
12:    //根据 dp 公式，依次更新 dp[i]
13:    for (int i = 2; i < n; i++)          //n 为数组长度
14:        dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])
15:    end for
16:    return dp[n - 1]          //最终返回最大收益
17: end function
```

Proof of the correctness

举以下例子来证明算法的正确性：假设从数组 {1,2,3,1} 中寻找一夜之内能够偷窃到的最高金额 (即偷窃到的最高金额 = 1 + 3 = 4 。)

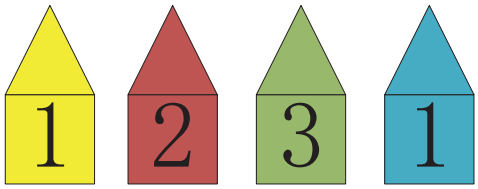

Figure 1: 待偷窃房屋

1. 偷窃第一间房屋：dp[0]=nums[0]=1，最大金额为 1。
2. 偷窃前两间房屋：dp[1]=max(nums[0], nums[1])=2, 最大金额为 2。
3. 偷窃前三间房屋：dp[2]=max(dp[0]+nums[2], dp[1])=max(1+3,2)=4, 最大金额为 4。max(偷窃前 2 间房屋的最高金额, 第 1 间房屋 + 第 3 间房屋的总金额)
4. 偷窃前 4 间房屋：dp[3]=max(dp[1]+nums[3], dp[2])=max(2+1,4)=4, 最大金额为 4。max(偷窃前 3 间房屋的最高金额, 第 2 间房屋 + 第 4 间房屋的总金额)
5. 返回 dp[n-1]=dp[3]=4。和预测的结果一样。

Analysis of Complexity

时间复杂度 $O(n)$。从 0 到 n-1 依次更新每个状态 (此处的 n 为数组长度，即房屋总数) 所以时间复杂度为 $O(n)$。**空间复杂度** $O(n)$。因为使用了一个长度为 n 的数组记录 dp[i]。

第二小问：如果所有的房子都排成一个圆圈呢?

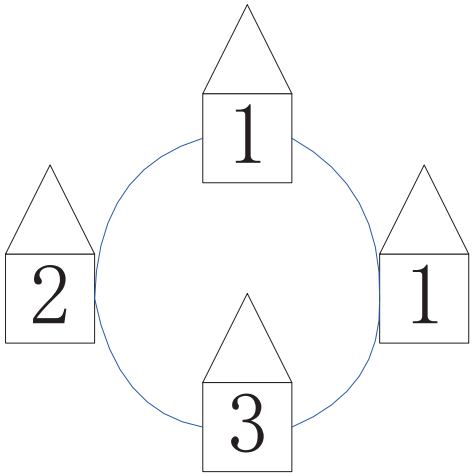

Figure 2: 环形房屋

the optimal substructure and DP equation

只一间房，偷该房可得到最高总金额。只两间房，两间房相邻，不能同时偷，选其中金额高偷。如果房屋数量大于两间，第一间和最后一间房不能同时偷。如果偷了第一间就不能偷最后一间房，偷的范围是第一间房到最后第二间房屋；如果偷了最后一间房，则不能偷第一间，偷的范围是第二间房到最后一间房。

假设数组 nums 的长度为 n。如果不偷最后一间房，偷的下标范围是 [0,n−2]；如果不偷第一间房，偷的下标范围是 [1,n−1]。然后比较两个中那个偷盗的更多，就是偷的最大金额。假设偷的下标范围是 [start,end]，用 dp[i] 表示在下标范围 [start,i] 内可以偷到的最高总金额，dp 方程为：

$$dp[i] = \max(dp[i-2] + \text{ nums } [i], dp[i-1])$$

1. 求出第一家到倒数第二家的最大钱财数量 2. 求出第二家到最后一家的最大钱财数量 3. 求两者的较大值即分别取 (start,end)=(0,n−2) 和 (start,end)=(1,n−1) 进行计算，取两个 dp[end] 中的最大值，即为下标范围 [start,end] 内可以偷到的最高总金额。

特殊情况:

$$\begin{cases} dp[\text{i}] = \text{ nums [start]} & \text{只有一间} \\ dp[\text{i}] = \max(\text{ nums [start], nums [start } + 1]) & \text{只有两间房屋} \end{cases}$$

Solution

Algorithm 2 求解环形打家劫舍

Input: 一个代表每个房屋存放金额的非负整数数组。
Output: 一夜之内能够偷窃到的最高金额。

```
1: function ROB(nums)
2:     //取输入数组长度 n
3:     if(n=1)              //如果只有一家
4:         return nums[0]//直接返回这家的金钱;
5:     end if
6:     if(n=2)              //如果有两家
7:         return max(nums[0], nums[1])//返回两家中金额最多的
8:     end if
9:     //如果有两家以上
10:    return max(robRange(nums, 0, length - 2), robRange(nums, 1, length - 1))//分别取 (start,end)=(0,n−2)
   和 (start,end)=(1,n−1) 进行计算，取两个 dp[end] 中的最大值返回
11: end function
12: function ROBRANGE(nums, start, end)
13:    first = nums[start], second = max(nums[start], nums[start + 1])
14:    for (int i = start + 2; i <= end; i++)
15:        temp = second
16:        second = max(first + nums[i], second)
17:        first = temp
18:    end for
19:    return second
20: end function
```

Proof of the correctness

如果只有一间房，则偷该房可得到最高总金额。如果只有两间房，则由于两间房相邻，不能同时偷，只能偷其中的一间房，选择其中金额较高的房进行偷就是最高总金额。如果房屋数量大于两间，第一间房和最后一间房不能同时偷。如果偷了第一间房就不能偷最后一间房，偷的范围是第一间房到最后第二间房屋；如果偷了最后一间房，则不能偷第一间房，偷的范围是第二间房到最后一间房。进行计算，取两个 dp[end] 中的最大值返回。

Analysis of Complexity

时间复杂度 $O(n)$。**空间复杂度** $O(n)$。

Problem 2

对应 sep 发布的作业题 2：Largest Divisible Subset Given a set of distinct positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si%Sj = 0 or Sj%Si = 0.Please return the largest size of the subset.Note: Si%Sj = 0 means that Si is divisible by Sj.

the optimal substructure and DP equation

dp[i] 表示在输入数组 nums 升序排列的前提下，以 nums[i] 为最大整数的整除子集的大小。存在 2 种情况：如果在 i 之前找不到符合 nums[i]%nums[j]==0 的位置 j，那么 nums[i] 不能接在 i 之前的数后面，只能自己独立作为整除子集的第一个数，此时 dp[i]=1。如果在 i 之前能够找到符合条件的 j，则取符合条件的 dp[j] 的最大值。表示如果找到以 nums[i] 结尾的最长子集，将 nums[i] 接到符合条件的最长 nums[j] 后面。此时 dp[i]=dp[j]+1。则对应的 DP 方程为:

$$dp[i] = \begin{cases} \max(\text{dp[i]}, \text{dp[j]} + 1) & \text{if nums[i]\%nums } [j] == 0 \\ \\ 1 & \text{if nums[i]\%nums } [j]! = 0 \end{cases} \tag{2}$$

Solution

Algorithm 3 最大整除子集

Input: 无重复正整数组成的集合 nums
Output: 最大整除子集 res

```
1:  function LARGESTDIVISIBLESUBSET(nums)
2:      len = nums.size()              //取输入数组的长度
3:      sort(nums.begin(),nums.end())          //用 sort 函数升序排序，算法复杂度为 O(nlogn)
4:      //找出最大子集、最大子集中的最大整数
5:      //初始化长度为 len 长的 dp，初始化值 dp[i]=1
6:      maxSize = 1            //最大子集尺寸赋初始值为 1
7:      maxVal = dp[0];            //最大子集中的最大整数初始值为 dp[0]
8:      for (int i = 1; i < len; i++)
9:          for (int j = 0; j < i; j++)
10:             if (nums[i] % nums[j]==0)          //如果存在整除关系，取所有符合条件的 dp 的最大值
11:                 dp[i] = max(dp[i], dp[j] + 1)
12:             end if
13:         end for
14:         if(dp[i] > maxSize)          //如果找到了，更新两个最大值
15:             maxSize = dp[i]
16:             maxVal = nums[i]
17:         end if
18:     end for
19:     //然后倒着遍历得到最大整除子集
20:     if (maxSize == 1)          //如果最大整除子集大小为 1，则直接返回 nums[0]
21:         res.push_back(nums[0])
22:         return res
23:     end if
24:     for (int i = len - 1; i >= 0 && maxSize > 0; i–)          //倒着遍历
25:         if (dp[i] == maxSize && maxVal % nums[i] == 0)      //满足条件，就装入，直到全部遍历完
```

```
26:                 res.push_back(nums[i])
27:                 maxVal = nums[i]
28:                 maxSize–
29:             end if
30:         end for
31:         return res          //返回最大整除子集
32: end function
```

Proof of the correctness

本题中定义 dp[i] 表示输入数组 nums 升序排序的前提下，以 nums[i] 为最大整数的整除子集的大小, 初始化时,dp[i]=1,(其中 i=0,1...n-1) 循环遍历 nums[j], 如果满足 nums[i] % nums[j]==0，则说明 nums[i] 可以扩充在以 nums[j] 为最大整数的整数子集成为一个更大的整数子集。找到最大子集、最大子集中的最大整数之后，倒序遍历数组 dp，直到找到 dp[i]=maxSize 为止，把此时对应的 nums[i] 加入结果集，此时 maxVal=nums[i]；然后将 maxSize 的值减 1，继续倒序遍历找到 dp[i]=maxSize，且 nums[i] 能整除 maxVal 的 i 为止，将此时的 nums[i] 加入结果集，maxVal 更新为此时的 num[i]；重复上述操作，直到 maxSize 的值变成 0，此时的结果集即为目标子集。
举个例子进行详细说明：

nums[i]	2	7	8	4	12	9	16	18
排序后	2	4	7	8	9	12	16	18
dp[i]	1	2	1	3	1	3	4	3

Figure 3: 排序并计算 dp[i] 后的结果表

倒序遍历求得最大整除子集：
1. 根据 dp 结果 (表格第三行),maxSize=4,maxVal=16, 即大小为 4 的最大整除子集包含的最大整数为 16。
2.maxSize–查找 maxSize=3 的最大整除子集, 有多个, 可以知道的是最大整除子集一定包含 8, 因为 16%8=0。
3.maxSize–查找 maxSize=2 的最大整除子集,maxVal=4 对应 maxSize=2, 即最大整除子集一定包含 4。
4.maxSize–查找 maxSize=1 的最大整除子集,maxVal=2 对应 maxSize=1, 即最大整除子集一定包含 2。
5. 即得到最大整除子集 [16,8,4,2], 和预测结果一致。

Analysis of Complexity

时间复杂度 $O\left(n^2\right)$,n 为输入数组的长度。对数字 nums 排序的时间复杂度为 $O(nlogn)$, 第一步找出最大子集以及最大子集中的整数的时间复杂度为 $O\left(n^2\right)$，第二步倒序遍历得到最大子集的时间复杂度为 $O(n)$。所以此题的时间复杂度为 $O\left(n^2\right)$。空间复杂度为 $O(n)$, 因为需要创建长度为 n 的数组 dp。

Problem 3

对应 sep 发布的作业题 5：Distinct Sequences Given two strings S and T , return the number of distinct subsequences of S which equals T .A string’ s subsequence is a new string formed from the original string by deleting some (can be none) of the characters without disturbing the remaining characters’ relative positions. (i.e., ” ACE” is a subsequence of ” ABCDE” while ” AEC” is not).

the optimal substructure and DP equation

假设字符串 s 和字符串 t 的长度分别为 m 和 n。如果 m<n，则 t 一定不是 s 的子序列，直接 return0。如果 m 大于等于 n，则通过动态规划计算 s 的子序列中 t 出现的个数。二维数组 dp[i][j] 表示 s 从下标 i 到末尾的子字符串的子序列中 t 从下标 j 到末尾的子字符串出现的个数（为了后面表示方便，写成 dp[i][j] 表示在 s[i:] 的子序列中 t[j:] 出现的个数）。
1. 当 j=n 时，t[j:] 为空字符串，因为空字符串是任何字符串的子序列，所以 dp[i][n]=1(其中 0≤i≤m)
2. 当 i=m 时，非空字符串 t[j:] 不是空字符串 s[i:] 的子序列，所以 dp[m][j]=0(其中 j<n)
3. 当 i<m,j<n 时: 如果 s[i] 和 t[j] 相等, 则 dp[i][j]=dp[i+1][j+1]+dp[i+1][j]; 如果 s[i] 和 t[j] 不相等, 则 dp[i][j]=dp[i+1][j]。最终计算得到 dp[0][0] 即为在 s 的子序列中 t 出现的个数则对应的 DP 方程为:

$$dp[i][j] = \begin{cases} dp[i+1][j+1] + dp[i+1][j], & s[i] = t[j] \\ dp[i+1][j], & s[i] \neq t[j] \end{cases} \tag{3}$$

Solution

Algorithm 4 不同的子序列

Input: 字符串 s 和字符串 t。
Output: s 的子序列中 t 出现的个数。

```
1: function numDistinct(s,t)
2:     if (m < n)            //如果 s 字符串小于 n，那一定没有，返回 0
3:         return 0
4:     end if
5:     //初始化一个 (m+1,n+1) 的二维数组 dp[i][j]
6:     for (int i = 0; i <= m; i++) //当 j=n 时的特殊情况
7:         dp[i][n] = 1
8:     end for
9:     for (int i = m - 1; i >= 0; i–) //i<m,j<n 时的情况
10:        for (int j = n - 1; j >= 0; j–)
11:            if (sChar == tChar) //如果满足 s[i]=t[j]
12:                dp[i][j] = dp[i + 1][j + 1] + dp[i + 1][j]
13:            end if
14:            else //如果不相等
15:                dp[i][j] = dp[i + 1][j]
16:            end else
17:    return dp[0][0]//返回最终结果
18: end function
```

Proof of the correctness

举以下例子来证明算法的正确性：
字符串 s=”babgbag”, 字符串 t=”bag”, 预测的输出应该是 5。

初始化一个 (7+1) 行 (3+1) 列的二维数组 dp[i][j], 并赋初始值为 0。

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 4: 初始状态

当 j=n 时，空字符串是任何字符串的子序列，所以 dp[i][n]=1

0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1

Figure 5: dp[i][n]=1

计算 dp[6][2]:s[i]=t[j]=g, 所以 dp[i][j]=dp[i+1][j+1]+dp[i+1][j]

0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	1	1
0	0	0	1

Figure 6: dp[6][2]=dp[7][3]+dp[7][2]=1

计算 dp[6][1]:s[i]=g 而 t[j]=a, 两者不相等, 所以 dp[i][j]=dp[i+1][j]

0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	0	1
0	0	1	1
0	0	0	1

Figure 7: dp[6][1]=dp[7][1]=0

其他位置同理，此处省略。最终：

5	3	2	1
2	3	2	1
2	1	2	1
1	1	2	1
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

Figure 8: 最终结果返回 dp[0][0]=5

和预测结果一样，最终结果为 5。

Analysis of Complexity

时间复杂度 $O(mn)$,m,n 分别是字符串 s 和 t 的长度，二维数组 dp 有 m+1 行和 n+1 列，需对 dp 中每个元素进行计算，所以时间复杂度为 $O(mn)$。空间复杂度也为 $O(mn)$，因为创建了 m+1 行 n+1 列的二维数组 dp[i][j]。