数据分析与算法设计：自选编程作业 1

3190102060 黄嘉欣 信工 1903 班

一、题目：312.戳气球

有 n(1<=n<=500)个气球，编号 0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums(0<=nums[i]<=100)中。现在要求你戳破所有的气球。当戳破第 i 个气球时，你可以获得 nums[i-1]*nums[i]*nums[i+1]枚硬币，其中 i-1 和 i+1 分别代表和 i 相邻的两个气球的序号。如果 i-1 或 i+1 超出了数组的边界，那么就当它是一个数字为 1 的气球。求所能获得硬币的最大数量。

二、算法设计

由题意，我们需要求出所能获得硬币的最大数量，则必须将所有的可能情况一一列举，从中找出最大值。面对穷举问题，主要有回溯算法和动态规划两种思路，前者是暴力穷举，即穷举戳气球的顺序，得到不同顺序下的硬币数，从中找出最高的一种；后者根据状态转移方程推导状态，利用递归解决问题。两种算法的具体思路分析如下：

① 回溯算法

根据问题，对于所给数组，可以利用循环求解。假设每次循环从第 i 个气球开始戳破，首先求解出该次戳球所得硬币数，再从数组中将该气球移除，递归回溯，求解出剩余 n-1 个气球可得的硬币数，其伪代码为：

算法：
```
int maxCoins(int* nums){
  int maxCoins = 0; // 初始化最大硬币数
  BackTrack(nums, 0, maxCoins);
  return maxCoins;
}

void BackTrack(int* nums, int coins, int maxCoins)
// 求解戳破气球可得的硬币数
// 输入：气球数组 nums，硬币数 coins，已计算得到的最大硬币数 maxCoins
{
  if (sizeof(nums)/sizeof(int) == 0){ // 递归终点
      maxCoins = max(coins, maxCoins);
      return ;
  }
  for (int i=0; i<sizeof(nums)/sizeof(int); i++){
      int points = nums[i-1]*nums[i]*nums[i+1];
      delete(nums[i]); // 从数组中移除 nums[i]
```

```
            BackTrack(nums, coins+points, maxCoins);
            add(nums[i]); // 将 nums[i]复原到 nums 中
        }
    }
```

由伪代码可以发现，由于对所有情况进行了计算，采用回溯算法的效率为 O(n!)，难以应用于 n 的规模较大的情况，故不做进一步的考虑。

② 动态规划算法

为了推导状态转移方程，我们考虑对问题进行转换。由于“i-1 或 i+1 超出了数组的边界，则当它是一个数字为 1 的气球”，即 nums[-1]=nums[n]=1，因此，可以将数组 nums 进行扩充，使得到的新数组中，balls[0]=balls[n+1]=1，balls[i]=nums[i-1](1<=i<=n)，此时问题即转换为：求戳破气球 0 和气球 n+1 之间的所有气球（不含边界）可得的最大硬币数。设 F(i,j)为戳破气球 i 和气球 j 之间的所有气球（不含边界）可得的最大硬币数，k 为 i 和 j 之间最后戳破的那个气球，则 i 和 k 之间可得最大硬币数为 F(i,k)，k 和 j 之间为 F(k,j)，故有递推关系：

$$F(i,j)=\max_{i<k<j}\{F(i,k)+F(k,j)+balls[i]*balls[k]*[j]\},\ 0<=i<j<=n+1$$

$$F(i,j)=0,\ i=j$$

为了在计算 F(i,j)之前确保 F(i,k)与 F(k,j)的值已被算出，我们需要通过从左往右、从下至上的遍历方式先行计算与 F(i,j)同行的所有 F(i,k)及其同列的所有 F(k,j)，逐步填充二维数组，最终得到 F(0,n+1)的值，此算法的伪代码为：

```
算法：int maxCoins(int* nums, int numSize)
    // 求解戳破气球可得的最大硬币数
    // 输入：气球数组 nums，气球数 numSize
    // 输出：最大硬币数
    {
      int balls[] = create_new_array(nums) // 扩充 nums
      int F[numSize+2][numSize+2] = 0; // 初始化硬币数矩阵
      for (int i=numSize; i>=0; i--){ // i 从下至上
          for (int j=i+1; j<numSize+2; j++){ // j 从左往右
             for (int k=i+1; k<j; k++){
                 F[i][j] = max(F[i][j],
                              F[i][k]+F[k][j]+balls[i]*balls[k]*[j]);
             }
          }
      }
```

```
}
```

三、代码实现

由于回溯算法的复杂度过大，无法通过 LeetCode 所有测试用例，故采用动态规划算法。根据(二)中伪代码，可以得到其 C 语言代码实现为：

代码：
```c
int maxCoins(int* nums, int numSize){
    int balls[numSize+2]; // 新气球数组
    int F[numSize+2][numSize+2]; // 最大硬币数矩阵
    int i, j, k;
    // 扩充 nums
    balls[0] = balls[numSize+1] = 1;
    for (i=1; i<numSize+1; i++){
        balls[i] = nums[i-1];
    }
    for (i=0; i<numSize+2; i++){
        for (j=0; j<numSize+2; j++){
            F[i][j] = 0; // 初始化
        }
    }
    // 遍历求解
    for (i=numSize; i>=0; i--){ // i 从下至上
        for (j=i+1; j<numSize+2; j++){ // j 从左往右
            for (k=i+1; k<j; k++){
                if (F[i][j]<F[i][k]+F[k][j]+balls[i]*balls[k]*balls[j])
                  F[i][j] = F[i][k]+F[k][j]+balls[i]*balls[k]*balls[j];
            }
        }
    }
    return F[0][numSize+1];
}
```

四、运行结果

如图 4.1，将动态规划算法代码提交至 LeetCode，得其执行用时为 200ms，内存消耗 7.1MB，通过全部 70 个测试用例。当测试输入为[3,1,5,8]时，有最大硬币获得数 167，程序输出为 167，与正确结果一致，如图 4.2 所示。采用自测试示例，若输入的气球数组为[1,1]，则输出为 2=1*1*1+1*1*1，正确；若输入数组为[1,3,1]，得输出为 9=1*1*3+3*1*1+1*3*1，正确，如图 4.3 所示。对于更多的输入可能，无法一一列举，但由 LeetCode 测试结果可知，算法设计正确。

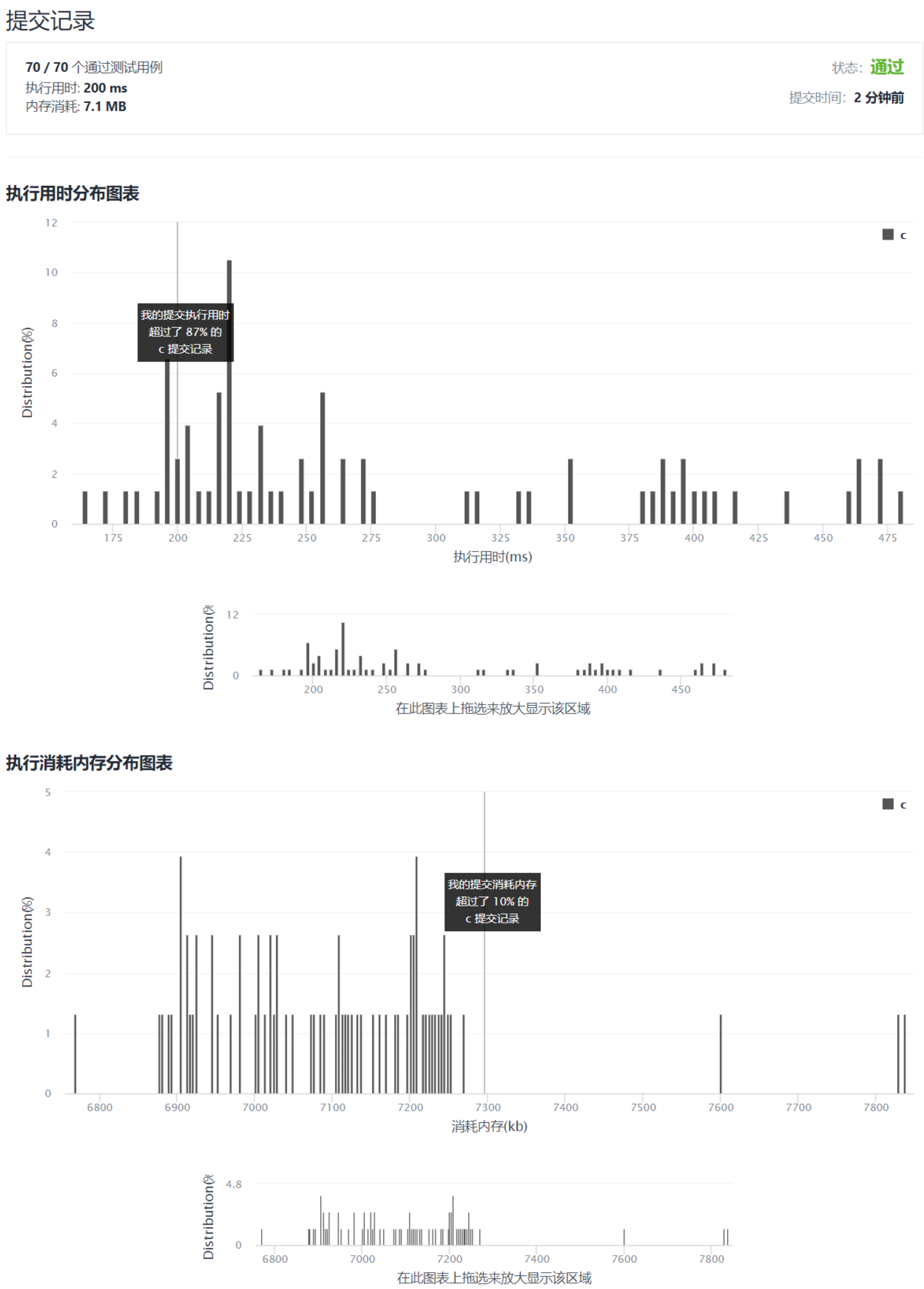

图 4.1 LeetCode 算法运行结果

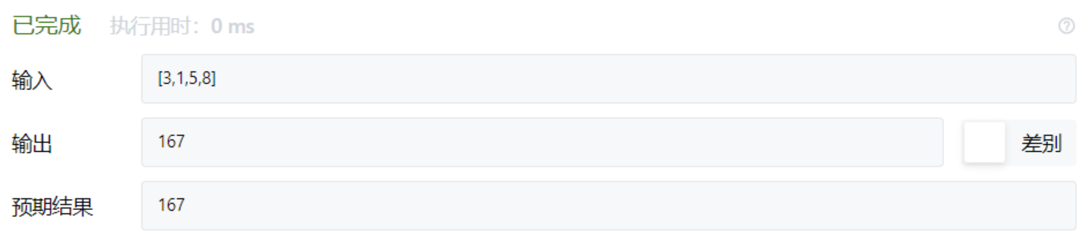

图 4.2 LeetCode 测试实例

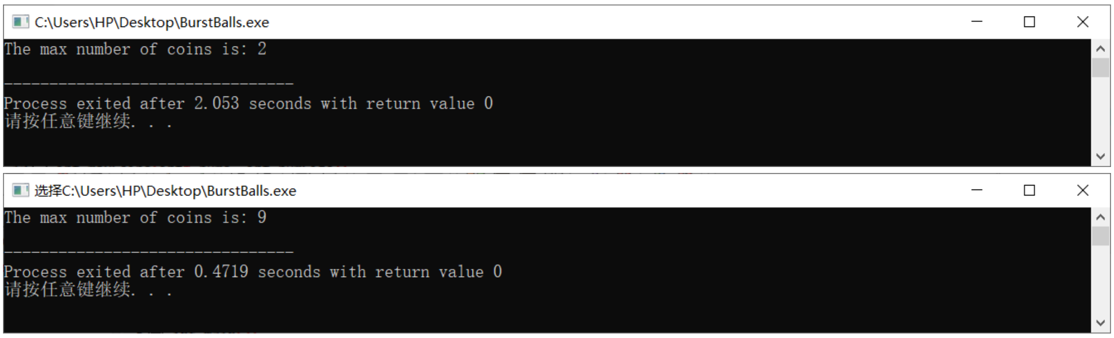

图 4.3 自测试实例

五、效率分析

① 回溯算法

由伪代码，设对输入规模为 n 的数组，其基本操作（回溯递归）执行次数为 C(n)，则有递推式：

$$\begin{cases} C(n) = nC(n-1), n > 1 \\ C(1) = 1 \end{cases}$$

于是：C(n)=nC(n-1)=n(n-1)C(n-2)=……=n(n-1)……2C(1)=n!

即回溯算法的时间效率为 O(n!)。

同理，由于在每次循环当中，我们都需要将移除的 nums[i]保存，以供后续恢复数组，若设空间复杂度为 V(n)，则有：

$$\begin{cases} V(n) = n[1 + V(n-1)], n > 1 \\ V(1) = 1 \end{cases}$$

故：V(n)=n+nV(n-1)]=n+n(n-1)+n(n-1)V(n-2)=……=n+n(n-1)+……+n!。可见，无论是时间效率还是空间效率，回溯算法都存在有极大的不足，不能用于大规模输入情况下的计算。

② 动态规划算法

由伪代码，设对输入规模为 n 的数组，其基本操作（比较）执行次数为 C(n)，则有：

$$C(n)=\sum_{i=0}^{n}\sum_{j=i+1}^{n+1}\sum_{k=i+1}^{j-1}1=\sum_{i=0}^{n}\sum_{j=i+1}^{n+1}(j-i-1)=\sum_{i=0}^{n}(0+1+\cdots+n-i)\in O(n^3)$$

即动态规划算法的时间效率为 $O(n^3)$。

显然，由于在动态规划算法中，我们只需要对最大硬币数组 F[n+2][n+2]进行填充，同时构建新的气球数组 balls[n+2]，故所需的额外空间为：$V(n)=(n+2)^2+n+2=(n+2)(n+3)$，即算法的空间效率为 $O(n^2)$。

通过比较可以发现，动态规划算法在时空效率上远远优于回溯算法。通过合理定义子问题、巧妙寻找递推式，我们可以把原问题分解成为相互独立、相对简单的小规模问题，并将其结果记录在表中，最终从表中得到原始问题的解。以此题为例，在面对较大规模的输入时，动态规划算法的时空效率优势体现更大。总的来说，定义好 F(i,j)及算法的遍历方向、构建递推关系，是动态规划算法中异常重要的一步，需要我们灵活思考，多加练习。