

深圳大学实验报告

课程名称： 算法设计与分析

实验名称： 动态规划—金罐游戏问题

学院： 计算机与软件学院 专业： 软件工程

报告人： 郑杨 学号： 2020151002 班级： 腾班

同组人： 陈敏涵

指导教师： 李炎然

实验时间： 2022.5.26~2022.5.30

实验报告提交时间： 2022.5.30

教务处制

目录

一、实验目的.....	3
二、实验内容与要求.....	3
1、实验内容.....	3
2、实验要求.....	4
三、实验步骤与结果.....	4
1 问题分析.....	4
1.1 博弈问题.....	4
1.2 博弈问题分析.....	5
2 蛮力法.....	5
2.1 算法思想.....	5
2.2 算法流程.....	6
2.3 记录选择方案.....	7
2.4 实现细节.....	8
2.5 伪代码.....	8
2.6 复杂度分析.....	8
3 记忆化搜索.....	9
3.1 算法思想.....	9
3.2 算法流程.....	10
3.3 伪代码.....	10
3.4 复杂度分析.....	10
4 动态规划.....	10
4.1 算法思想.....	10
4.2 状态转移方程.....	11
4.3 算法流程.....	12
4.4 伪代码.....	13
4.5 正确性验证.....	13
4.6 复杂度分析.....	13
4.7 记录选择方案.....	13
4.8 空间复杂度优化.....	14
5 相关实验.....	15
5.1 蛮力法在不同数据规模下的理论时间与实际运行时间比较.....	15
5.2 记忆化搜索与蛮力法在不同数据规模下的运行效率比较.....	15
5.3 记忆化搜索在不同数据规模下的运行效率测试.....	16
5.4 动态规划方法在不同数据规模下的理论时间与实际运行时间比较.....	16
5.5 能在有限时间处理完的最大规模数据.....	17
四、实验心得.....	17
五、附件说明.....	17

一、实验目的

- 1、掌握动态规划算法设计思想。
- 2、掌握金罐游戏问题的动态规划解法。

二、实验内容与要求

1、实验内容

金罐游戏中所有的金罐排成一排，每个罐子里都装有金币。两个玩家 A 和 B，可以看到每个金罐中有多少硬币。游戏要求 A 和 B 两个玩家交替轮流打开金罐，但是必须从一排的某一端开始挑选，玩家可以从一排罐的任一端挑选一个罐打开。游戏规则参考表 1 和表 2 的金罐例子。获胜者是最后拥有更多硬币的玩家。我们是 A 玩家，问如何才能使 A 收集的硬币数量最大。假设 B 也是按照“最佳”策略玩，并且 A 开始游戏。

表 1：4 个金罐游戏例子

金罐中金币个数（已排列）	A	B
4, 6, 2, 3	3	
4, 6, 2		4
6, 2	6	
2		2
	9 coins	6 coins

表 2：6 个金罐游戏例子

金罐中金币个数（已排列）	A	B
6, 1, 4, 9, 8, 5	6	
1, 4, 9, 8, 5		5
1, 4, 9, 8	8	
1, 4, 9		9
1, 4	4	
1		1
	18 coins	15 coins

2、实验要求

- 1) 给出解决问题的动态规划方程；
- 2) 随机产生金罐的个数和金币值，在小数据规模下利用蛮力法验证算法的正确性，并记录A选择的是哪个金罐；
- 3) 随机产生金罐的个数和金币值，对不同数据规模（ n 的值）测试算法效率，并与理论效率进行比对，请提供能处理的数据最大规模，注意要在有限时间内处理完；
- 4) 该算法是否有效率提高的空间？包括空间效率和时间效率。

三、实验步骤与结果

1 问题分析

1.1 博弈问题

在提出问题解决思路之前，本人分析了问题的本质含义并提前解释一些相关概念。

金罐游戏本质上是一个轮流取数字的博弈游戏，该问题可以抽象为，在如图 1 中的数字序列上，两个玩家轮流（Alice 和 Bob）从数字序列 X 的某一端取出数字，两个玩家的目的都是使得自己取出的数字总和最大，也即“**使用最优策略**”。最终想要让 Alice 取出的数字总和最大，也就是 Alice 在最优策略下取出数字的总和。



图 1：数字序列 1

当前轮次取数的玩家称为先手，下一轮次取数的玩家称为后手。例如图 2 所示，第一轮游戏中 Alice 选择了当前序列最左边的数字 4，此时 Alice 称为先手，Bob 称为后手。



图 2：第一轮游戏

而在第二轮游戏中，Bob 也选择了当前序列最左边的数字 6，此时 Bob 称为先手，Alice 称为后手。于是，先手玩家与后手玩家并不是绝对的，先手与后手的定义是相对的，随着轮次的变化而变换。

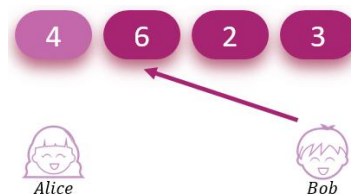


图 3：第二轮游戏

1.2 博弈问题分析

观察上一小节的博弈问题，显然，盲目的取出当前序列两端的两个数中较大的数并不是所谓的“最优策略”。例如，对于图 1 中的数字序列使用盲目取较大数的策略，如图 4(a)所示，最终，Alice 只能得到数字 4 和数字 3，总和为 7，而实际上，使用“最优策略”可以得到总和为 9 的数字和（如图 4(b)所示）。本质上，盲目策略就是只考虑局部最优而忽略了全局最优。

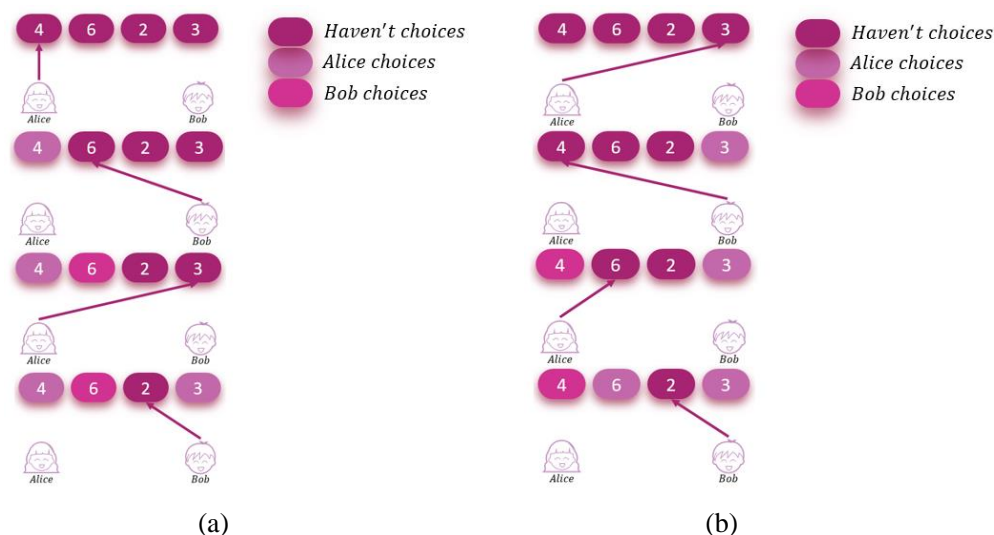


图 4：盲目策略与最优策略

如何解决盲目选取最大值的问题呢，一个最简单的思想就是预测对方的取数操作，就像下象棋一样，在走一步棋之前需要想好之后的几步。具体解决问题的思路将在下一节进行阐述。

2 蛮力法

蛮力法的主要思想就是去暴力搜索所有的可能情况，但是这道题有一个要求，取数双方必须遵循最优策略。直接求解原问题比较困难，于是我把原问题拆分为许多个小问题，逐个求解之后合并成原问题的解。

2.1 算法思想

以在图 1 所示的数字序列上的游戏为例，蛮力法需要搜索所有的可能情况，最终得出所谓的“最优策略”。一开始 Alice 可以选择序列两端的任何一个数字（4 或 3），此时的问题可以描述为：Alice 采用“最优策略”在序列 $X_{1,4}$ （其中， $X_{l,r}$ 表示序列 X 位于区间 $[l, r]$ 的一段子串）上能够取得的最大数字和。如图 5 所示。在枚举 Alice 选完的第一个数字之后，产生的两个子问题分别可以描述为：Bob 采用“最优策略”在序列 $X_{2,4}$ 上能够取得的最大数字和、Bob 采用“最优策略”在序列 $X_{1,3}$ 上能够取得的最大数字和。可以发现，三个问题的描述的不同之处只有序列的区间范围与取数的主角不同，可以把取数的主角统一称为先手玩

家，那么问题就可以统一转化为先手玩家在序列 $X_{l,r}$ 上能够取得的最大数字和。

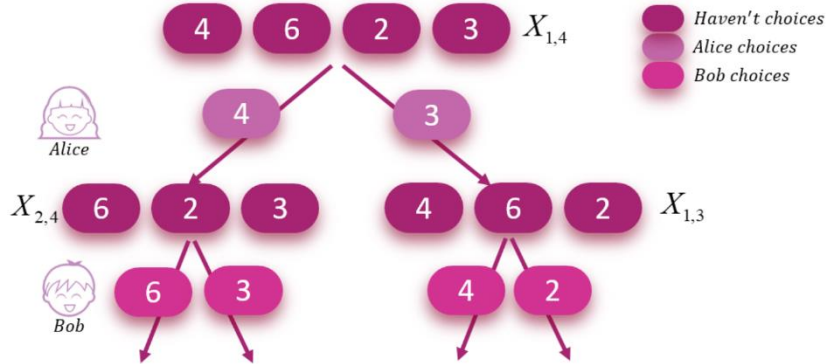


图 5: 博弈树

由于原问题与其子问题存在相同的问题描述，故可以利用递归进行搜索，这里我将求解先手玩家在序列 $X_{l,r}$ 上能够取得的最大数字和这个问题的方法称为 $solve(l, r)$ ，在下一节进行蛮力法具体流程的阐述。

2.2 算法流程

在问题规模为 n ，序列中第 i 个数为 $golds[i]$ 时，具体算法流程如下：

- 首先，原问题的求解方法表示为 $solve(l, r)$
- 向下递归至子问题 $solve(l + 1, r)$ 和 $solve(l, r - 1)$
- 当达到递归边界（假设子问题为 $solve(l, r)$ ，那么到达边界时满足 $l = r$ ）时，返回答案 $golds[l]$
- 设 $lans = solve(l + 1, r)$, $rans = solve(l, r - 1)$ ，由于这两个子问题的答案，对应的是先手在子问题中取得的最大数字和，而对于原问题 $solve(l, r)$ 中的先手，在子问题中就是后手，那么其实只需要使用子问题的所有数字和减去子问题先手取得的最大数字和，就可以得出原问题先手在子问题中取得的数字和了。故 $solve(l, r)$ 的答案为 $\max(golds[l] + sum[l + 1, \dots, r] - lans, golds[r] + sum[l, \dots, r - 1] - rans)$

仍以表 1 中的金罐例子为例叙述算法流程，对应的序列如图 1 所示。向下递归的过程如图 6 所示，每次枚举当前轮次的先手选取了两端的那个数字，分别对应了两种决策，分别向下递归到两个不同的子问题。随着子问题的规模不断缩小，最终到达递归边界。

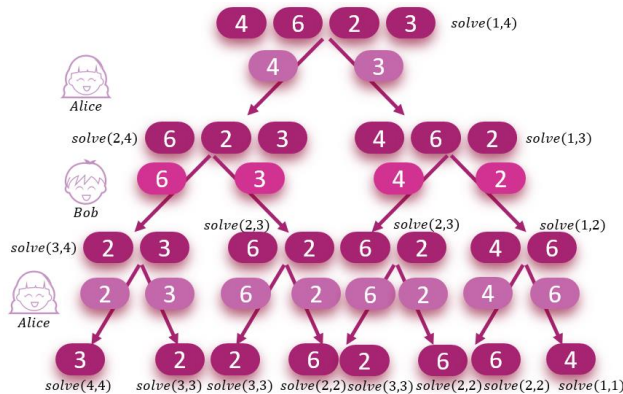


图 6: 蛮力法示意图 (1)

到达递归边界之后，返回子问题答案并根据“最优策略”更新原问题答案。显然，最后一层的子问题对应着 Bob 最后的唯一决策，返回的时候，计算两个子问题的最佳取值作为原问题的答案。如图 7 所示左下角的子树返回的情况，此时 $lans = solve(4, 4) = 3, rans = solve(3, 3) = 2$ ，故原问题 $solve(3, 4)$ 的答案为 $\max(golds[3] + sum[4, 4] - lans, golds[4] + sum[3, 3] - rans) = 3$ ，即图中红色箭头所指的策略为 Alice 此时的最佳策略。

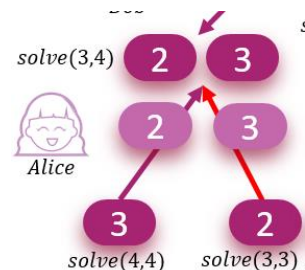


图 7：蛮力法示意图（2）

整个递归返回过程如图 8 所示，可以看出，利用最佳策略，最终的最优路径是唯一的，答案也是唯一的（不考虑数字被取出的先后顺序）。

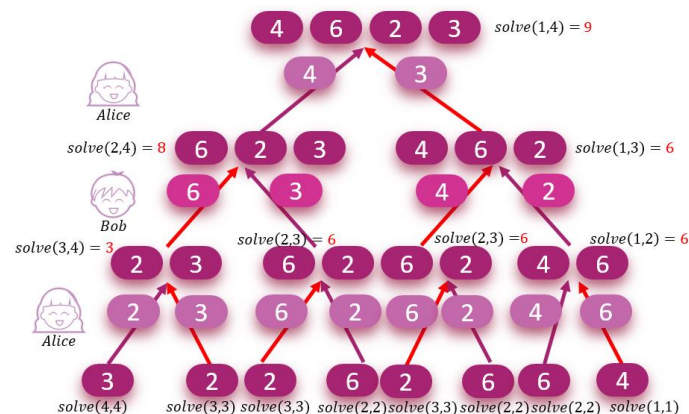


图 8：蛮力法示意图（3）

2.3 记录选择方案

从图 8 中可以看出，利用最佳策略进行游戏之后，每一个局面的后续局面都是唯一的，于是我使用 $next[l][r]$ 记录 $solve(l, r)$ 问题中先手玩家做出的取数选择。当 $next[l][r] = 1$ 时，先手玩家在当前轮次取出的数字为序列中最后一个数字；当 $next[l][r] = 0$ 时，先手玩家在当前轮次取出的数字为序列中的第一个数字，如图 9 所示。

于是，只需要从 $l = 1, r = 4$ 时开始（此时先手玩家为 Alice），根据 $next[1][4] = 1$ ，记录此时 Alice 取出的数字为第 4 个数字，进入下一个局面 $solve(1, 3)$ （此时先手玩家为 Bob）。之后的过程类似，最终当 $l = r$ 时即可停止记录。

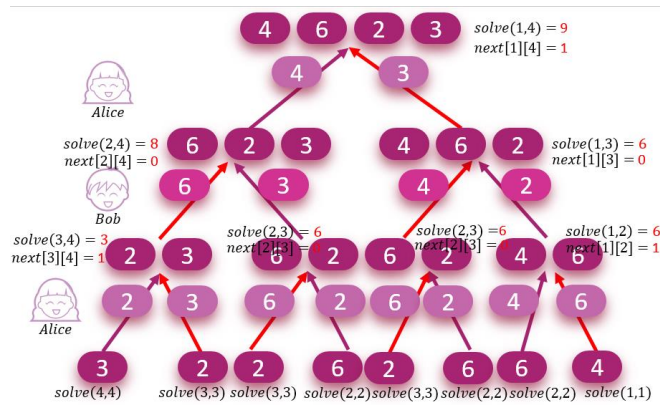


图 9：记录选择方案

2.4 实现细节

在算法流程中，提到了一个 $sum[l, \dots, r]$ 的概念，表示序列 $X_{l,r}$ 所有的序列和，为了快速计算这个数字和，可以在进入算法流程之前预处理出序列 X 的所有前缀和。记前缀和 $S[i]$ 表示 $sum[1, \dots, i]$ ，于是 $S[i] = S[i - 1] + golds[i]$ ，可以通过 $O(n)$ 的时间复杂度预处理出所有的前缀和。而 $sum[l, \dots, r] = S[r] - S[l - 1]$ ，则可以在 $O(1)$ 时间内快速计算出来。

2.5 伪代码

Algorithm1 1 Brute Force

Input: $golds[1, \dots, n]$

Output: *Maximum sum of numbers taken by the first player*

```

1:
2: function  $solve(l, r)$ 
3:   if  $l = r$  then
4:     return  $golds[l]$ 
5:   end if
6:    $lans \leftarrow solve(l + 1, r)$ 
7:    $rans \leftarrow solve(l, r - 1)$ 
8:   return  $\max(golds[l] + sum[l + 1, \dots, r] - lans, golds[r] + sum[l, \dots, r - 1] - rans)$ 
9: end function

```

2.6 复杂度分析

由以上分析过程，可以得出，递归树是一颗 n 层的完全二叉树，如图 10 所示。

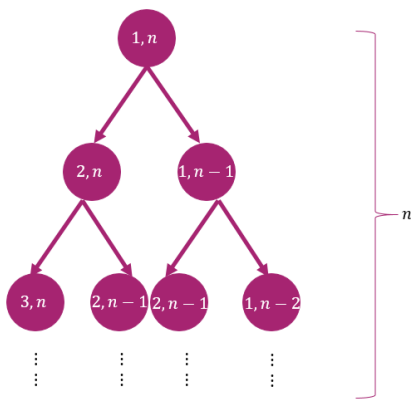


图 10: 递归树

那么可以得到，节点数共有：

$$\sum_{i=0}^{n-1} 2^i = \frac{1-2^n}{1-2} = 2^n - 1$$

故：蛮力法解决这个问题时间复杂度为 $O(2^n)$ 。由于需要使用前缀和数组作为辅助空间，故空间复杂度为 $O(n)$ 。

3 记忆化搜索

蛮力法求解轮流取数问题的效率较低，但是可以对其进行优化。

3.1 算法思想

在蛮力法中，对整个状态空间进行搜索必然会产生较多的冗余信息，如图 11 所示，搜索树中阴影部分的两颗子树状态完全相同。当搜索树规模增大时，这种冗余信息会变得越来越，使得蛮力法搜索效率低下。

因此，对于相同的冗余状态，没必要重新搜索一遍，只需要保存之前搜索过这个状态之后得到的最佳答案即可（因为两颗搜索子树完全相同，答案也是相等的）。于是我令 $f[l][r]$ 表示问题 $solve(l, r)$ 的答案，最开始把 f 当中的所有值初始化为 -1 ，当进入某一颗搜索子树 $solve(l, r)$ 中时，判断 $f[l][r]$ 是否等于 -1 ，若不等于 -1 则说明这颗子树在之前已经搜索过了，那么直接返回 $f[l][r]$ 即可；否则进行子树的搜索并更新 $f[l][r]$ 。

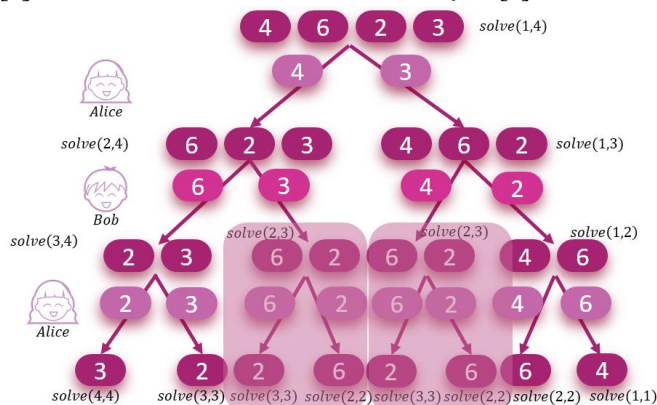


图 11: 示例 1 搜索树

3.2 算法流程

算法大体流程与蛮力法流程一致，故不再赘述。记忆化搜索主要体现在记忆了状态，如图 11 所示，当搜索到右边阴影对应的那颗子树时，左边阴影对应的那颗子树必然已经搜索过了，即 $f[2][3] = 6$ ，故不需要再进行搜索，直接返回 $f[2][3]$ 即可。

在第 5 小节中，进行了蛮力法与记忆化搜索在不同规模数据上运行效率的比较，可以看出，记忆化搜索对于蛮力法的优化还是很有效的，是一种空间换取时间的优化手段。

3.3 伪代码

Algorithm 1 Optimized Brute Force

Input: $golds[1, \dots, n]$, $f[1, \dots, n][1, \dots, n]$

Output: *Maximum sum of numbers taken by the first player*

```
1: function Solve( $l, r$ )
2:   if  $l = r$  then
3:     return  $golds[l]$ 
4:   end if
5:   if  $f[l][r] \neq -1$  then
6:     return  $f[l][r]$ 
7:   end if
8:    $lans \leftarrow solve(l + 1, r)$ 
9:    $rans \leftarrow solve(l, r - 1)$ 
10:  return  $f[l][n] \leftarrow \max(golds[l] + \text{sum}[l + 1, \dots, r] - lans, golds[r] + \text{sum}[l, \dots, r - 1] - rans)$ 
11: end function
```

3.4 复杂度分析

由于每一个状态只会被搜索一次，故时间复杂度与状态数成正比，由于状态数为 $\frac{n(n+1)}{2}$ ，故时间复杂度为 $O(n^2)$ 。

由于需要额外空间记录所有状态，故空间复杂度为 $O(n^2)$ 。

4 动态规划

4.1 算法思想

我所理解的动态规划思想，就是在证明问题具有最优子结构性性质之后，把每一个子问题的状态表示出来，利用不同决策找到不同状态之间的联系，推导出状态转移方程之后进行状态转移，通过组合子问题的解来求解原问题。

由于每个玩家都必须采取最优策略，故问题满足最优子结构性性质。上一节的记忆化搜索方法，本质上也是一种动态规划算法，它利用了冗余的信息加速算法的运行效率。记忆化搜索通过不断把问题拆分成若干个子问题，自顶向下进行搜索，返回时进行状态的更新，本质

上也是通过组合子问题的解来求解原问题，那么可以设计一个等价的算法，直接从底层往上更新状态，就是本节提出的动态规划算法。

4.2 状态转移方程

我使用集合角度来推导动态转移方程。

首先定义 $f(l, r)$ 表示先手玩家在序列 $X_{l,r}$ 上取得所有可能的数字总和的集合，定义集合的属性 $f[l][r]$ 为所有可能的数字总和的**最大值**（也就是最终的答案）。以图 1 所示的数字序列（示例一，长度为 4）为例， $f(2, 4)$ 所表示的集合如图 12 所示，图中还给出了对应的选数方案。

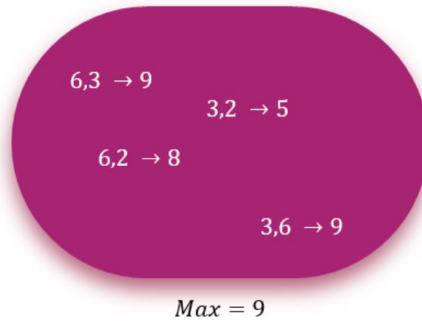


图 12：集合表示

表示完集合之后，开始考虑当前集合如何划分为有规律的小集合，通常以不同决策进行划分。在这里，先手玩家只有两个决策，即取出序列最左端的数或者取出序列最右端的数。那么，就可以按照这两个决策对集合进行划分，把先手玩家取出序列最左端数的可能情况划分在一起，把先手玩家取出序列最右端数的可能情况划分在一起，以 $f(2, 4)$ 为例，划分情况如图 13 所示。

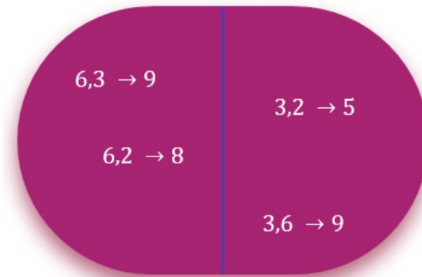


图 13：集合划分

那么集合 $f(l, r)$ 的最大值 $f[l][r]$ （也就是它的属性）可以表示为左右两个集合的属性最大值。观察得到，左边集合中所有情况共同点就是先手玩家取到了 $golds[l]$ ，可以将共同点提取出来，左边集合的属性就转化为： $golds[l] + sum[l + 1, \dots, r] - f[l + 1][r]$ ；同理右边集合中的所有情况共同点就是先手玩家取到了 $golds[r]$ ，于是右边集合的属性就转化为： $golds[r] + sum[l, \dots, r - 1] - f[l][r - 1]$ 。由于 $f[l][r]$ 表示的是整个集合的属性，也就是左右两边集合属性的最大值，故状态转移方程为：

$$f[l][r] = \max(golds[l] + sum[l + 1, \dots, r] - f[l + 1][r], golds[r] + sum[l, \dots, r - 1] - f[l][r - 1])$$

4.3 算法流程

首先，需要求出最底层的初始状态 $f[i][i]$ ，也就是只有一个数的情况，那么先手取出一个数的最大和就是那个数本身，即初始化所有的 $f[i][i] = \text{golds}[i]$ 。

通过状态转移方程可以看出，状态 $f[l][r]$ 依赖于序列长度比其小 1 的状态 $f[l+1][r]$ 和 $f[l][r-1]$ ，那么可以通过从小到大枚举长度，再通过枚举左端点得到右端点从而得到状态，利用状态转移方程进行转移，详细见伪代码。

最终，要求解的状态答案为 $f[1][n]$ 。

仍以图 1 的简单问题为例，采用填表法叙述具体流程。首先，初始化所有对角线元素也就是我们的初始状态 $f[i][i] = \text{golds}[i]$ ，如图 14 所示。

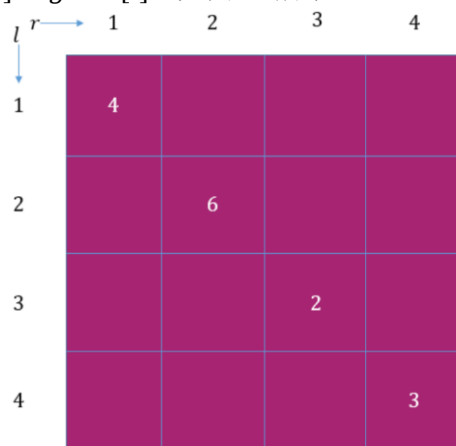


图 14：动态规划流程图示意图（1）

之后按照长度枚举区间，根据状态转移方程进行状态转移，转移过程如图 15 所示。三种不同颜色表示的是不同长度的区间的转移情况。

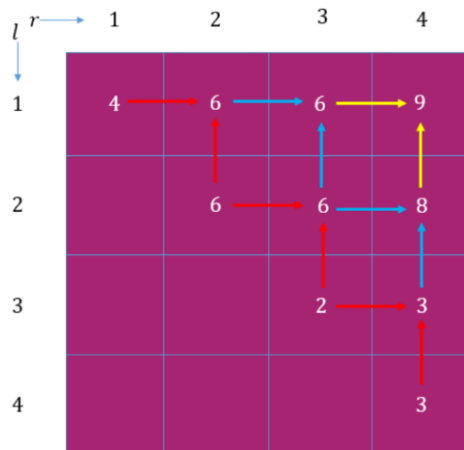


图 15：动态规划流程图示意图（2）

可以看到，上述填表法过程具有一个特点，每一个待填数值都是通过它下方和左方的表项转移过来的，根据这一特点，可以对状态空间进行压缩以优化算法的空间复杂度。这一点将在 4.8 节中进行详细叙述。

4.4 伪代码

Algorithm 1 Dynamic Programming

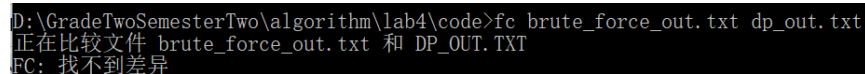
Input: $golds[1, \dots, n]$, $f[1, \dots, n][1, \dots, n]$

Output: *Maximum sum of numbers taken by the first player*

```
1: function Dp()
2:   for  $i$  from 1 to  $n$  do
3:      $f[i][i] \leftarrow golds[i]$ 
4:   end for
5:   for  $len$  from 1 to  $n$  do
6:     for  $l$  from 1 to  $n - len + 1$  do
7:        $r \leftarrow l + len - 1$ 
8:        $f[l][r] \leftarrow \max(golds[l] + \text{sum}[l + 1, \dots, r] - f[l + 1][r], golds[r] + \text{sum}[l, \dots, r - 1] - f[l][r - 1])$ 
9:     end for
10:  end for
11:  return  $f[1][n]$ 
12: end function
```

4.5 正确性验证

以 $n = 20$ 随机生成 100 组数据，使用蛮力法进行正确性验证，我采用将答案输出在文件中，再使用 window 控制台自带的文本比对命令 `fc`，对蛮力法生成的答案和动态规划法生成的答案进行比对，比对结果如图 16 所示。可以看出，动态规划法运行结果与蛮力法运行结果一致。



```
D:\GradeTwoSemesterTwo\algorithm\lab4\code>fc brute_force_out.txt dp_out.txt
正在比较文件 brute_force_out.txt 和 DP_OUT.TXT
FC: 找不到差异
```

图 16: 输出结果对比

4.6 复杂度分析

易得，时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(n^2)$ 。

4.7 记录选择方案

从状态转移方程中看出，每一个状态只依赖于两个子状态，类似蛮力法的方案记录方法，我记录了某个状态 $f[l][r]$ 的前驱状态 $pre[l][r]$ 。当 $pre[l][r] = 1$ 时，表示 $f[l][r]$ 由 $f[l + 1][r]$ 转移而来，说明先手玩家此时选取了序列最左边的数字；当 $pre[l][r] = 0$ 时，表示 $f[l][r]$ 由 $f[l][r - 1]$ 转移而来，说明先手玩家此时选取了序列最右边的数字。

从最终状态 $f[1][n]$ 出发，根据 pre 进行状态变换即可构造出一组最优解。

4.8 空间复杂度优化

由之前的分析可以看出，需要记录 $f[l][r](1 \leq l \leq r \leq n)$ 共 $\frac{n(n+1)}{2}$ 个状态，这需要耗费 $O(n^2)$ 的空间，使得动态规划方法无法处理较大规模的数据。(4GB内存最多装下1073741824个int类型的数据)。由于取数问题的状态转移方程具有顺序性，可对空间复杂度进行优化，优化之后的空间复杂度为 $O(n)$ 。

由算法流程中的填表过程可以看出，填表过程极具特点，就是每一个待填表项都是从它左边和下边的表项转移过来的，也就是说从比它长度小1的两个状态转移过来，不会涉及其他长度的状态。根据这一特点，可以将所有的状态保存在对角线上，仍以图1为例，当计算完长度为2的状态时（状态表如图17(a)所示），将长度为2的状态向下平移覆盖到对角线元素上，如图17(b)所示

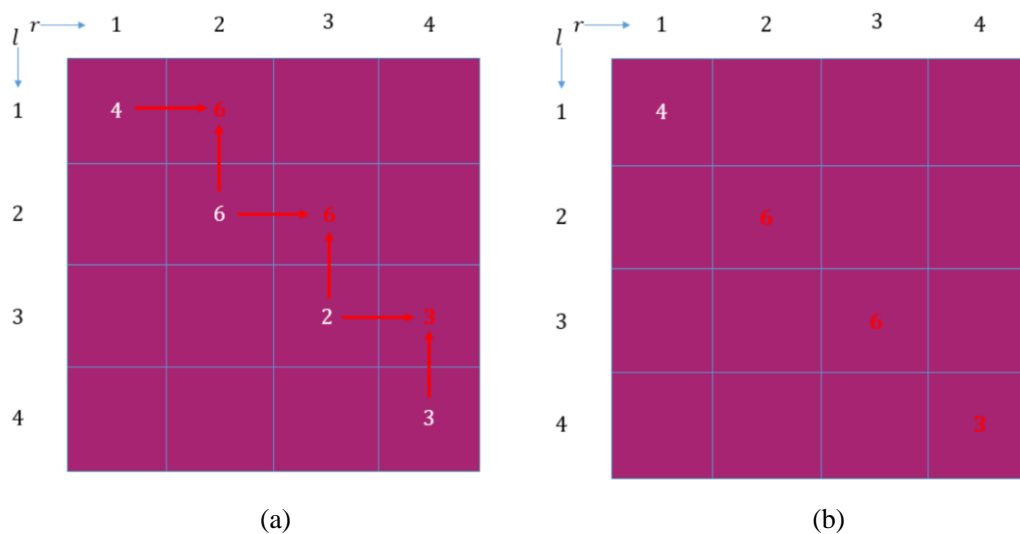


图 17：优化空间复杂度示意图（1）

在计算长度为3的状态时，原始计算方式如图18(a)所示，而此时状态 $f[1][2]$ 、 $f[2][3]$ 、 $f[3][4]$ 分别保存于 $f[2][2]$ 、 $f[3][3]$ 、 $f[4][4]$ 中，故计算方式如图18(b)所示，计算完成之后，依然把状态保存于对角线上。

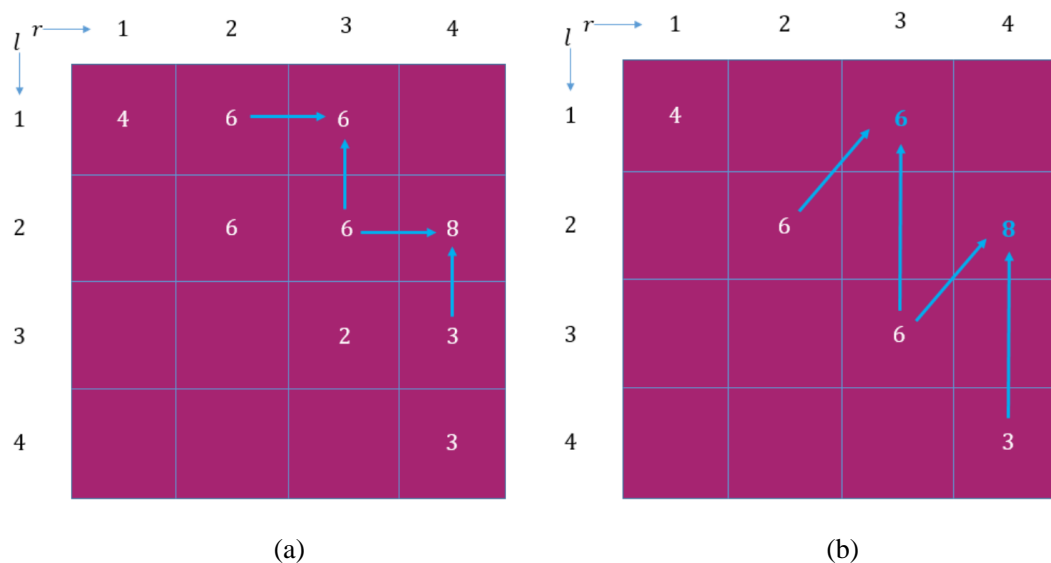


图 18：优化空间复杂度示意图（2）

最后长度为 4 的状态计算也是类似的，就不再赘述。于是，原始方法的 $\frac{n(n+1)}{2}$ 个状态就被转化成了状态表对角线上的 n 个状态，空间复杂度优化为 $O(n)$ 。

5 相关实验

5.1 蛮力法在不同数据规模下的理论时间与实际运行时间比较

由于蛮力法运行效率较慢，故使用较小的数据规模对蛮力法进行测试，数据规模从 10~35，对于每一个数据规模，随机 20 组测试数据，取 20 次运行时间的平均时间作为实际运行时间，以规模为 25 时的实际运行时间为基准计算理论值，测试结果如表 3 所示。

表 3：蛮力法在不同数据规模下的运行效率

数据规模	10	15	20	25	30	35
理论运行时间(ms)	0.004	0.134	4.275	136.812	4377.993	140095.775
实际运行时间(ms)	0.005	0.147	4.295	136.812	4290.501	138283.818
误差	8.12%	9.69%	0.45%	0.00%	-2.00%	-1.29%

图 19 给出了理论运行时间与实际运行时间关于数据规模变化的曲线图，可以看出，在数据规模较小时，理论值与实际值拟合较差，因为运行时间较快，误差较大。随着数据增大，理论值与实验值拟合得都比较好，误差较小，可以看到曲线呈指数增长趋势。

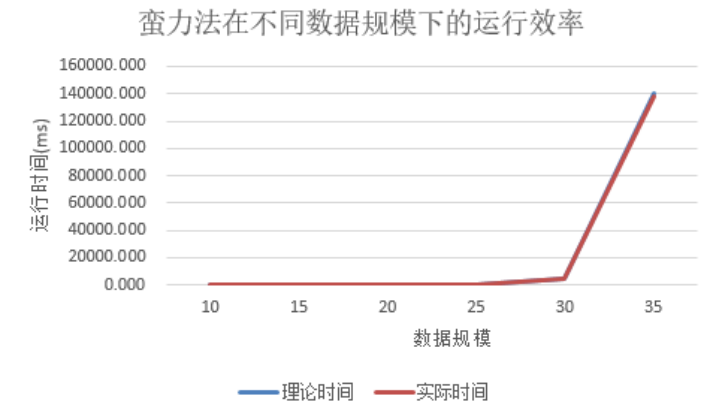


图 19：蛮力法在不同数据规模下的运行效率

5.2 记忆化搜索与蛮力法在不同数据规模下的运行效率比较

使用记忆化搜索在 5.1 节生成的六种不同数据规模下的运行时间与表 3 所示蛮力法的运行时间进行比较，测试结果如表 4 所示。可以看到，用空间换取时间的方式十分有效，在小规模数据上对蛮力法的优化程度非常大，也再次说明了蛮力法冗余信息非常多。

表 4：记忆化搜索与蛮力法在不同数据规模下的运行效率比较

数据规模	10	15	20	25	30	35
记忆化运行时间(ms)	0.003	0.005	0.008	0.012	0.012	0.017
蛮力法运行时间(ms)	0.005	0.147	4.295	136.812	4290.501	138283.818

5.3 记忆化搜索在不同数据规模下的运行效率测试

使用了 100~10000 的较大数据规模对记忆化搜索进行运行效率测试，对于每一个数据规模，随机 20 组测试数据，取 20 次运行时间的平均时间作为实际运行时间，以规模为 7500 时的实际运行时间为基准计算理论值，测试结果如表 5 所示。

表 5: 记忆化搜索在不同数据规模下的运行效率

数据规模	100	500	1000	2500	5000	7500	10000
实际运行时间(ms)	0.107	2.480	8.440	53.882	199.367	454.868	791.050
理论运行时间(ms)	0.081	2.022	8.087	50.541	202.164	454.868	808.654
误差	32.7%	22%	4.37%	6.61%	-1.38%	0.00%	-2.18%

图 20 给出了理论运行时间与实际运行时间关于数据规模变化的曲线图，可以看出，在数据规模较小时，理论值与实际值拟合较差，因为递归调用在数据规模较小时对运行时间影响较大。随着数据增大，理论值与实验值拟合得都比较好，误差较小，可以看到曲线呈多项式增长趋势。

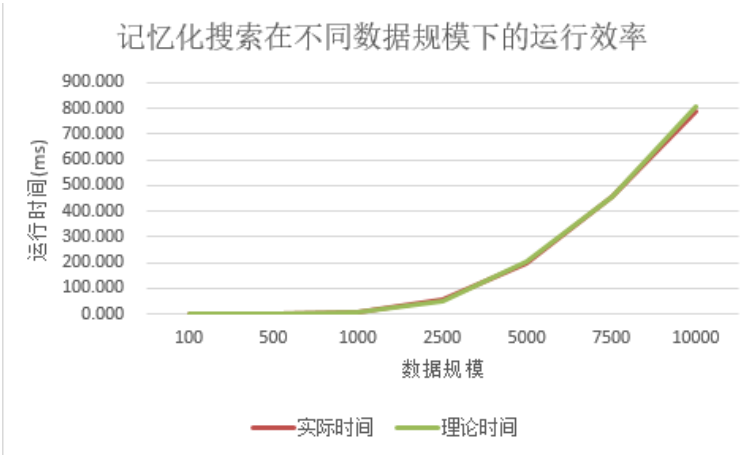


图 20: 记忆化搜索在不同数据规模下的运行效率

5.4 动态规划方法在不同数据规模下的理论时间与实际运行时间比较

测试了空间优化前与空间优化后动态规划法在 100~10000 的较大数据规模下的运行效率，对于每一个数据规模，随机 20 组测试数据，取 20 次运行时间的平均时间作为实际运行时间，以规模为 7500 时的实际运行时间为基准计算理论值，测试结果如表 6 与表 7 所示。可以看出，经过空间优化之后的动态规划算法运行效率较好于优化前的动态规划法，因为对一维数组进行寻址速度略快于二维数组寻址。

表 6: 优化前动态规划法在不同数据规模下的运行效率

数据规模	100	500	1000	2500	5000	7500	10000
实际运行时间(ms)	0.080	2.089	9.141	67.108	273.791	629.255	1129.16
理论运行时间(ms)	0.110	2.738	10.952	68.448	273.791	616.030	1095.16
误差	-26%	-23%	-16%	-1.96%	0.00%	2.15%	3.10%

表 7：空间优化后动态规划法在不同数据规模下的运行效率

数据规模	100	500	1000	2500	5000	7500	10000
实际运行时间(ms)	0.058	1.473	6.067	41.686	174.309	402.363	716.141
理论运行时间(ms)	0.070	1.743	6.972	43.577	174.309	392.195	697.236
误差	-16%	-15%	-12%	-4.34%	0.00%	2.59%	2.71%

图 21 给出了优化前与优化后的动态规划算法运行速率随着数据规模的变化图。可以看出，两者都大致满足二次函数增长。优化后动态规划法的曲线比优化前略低。

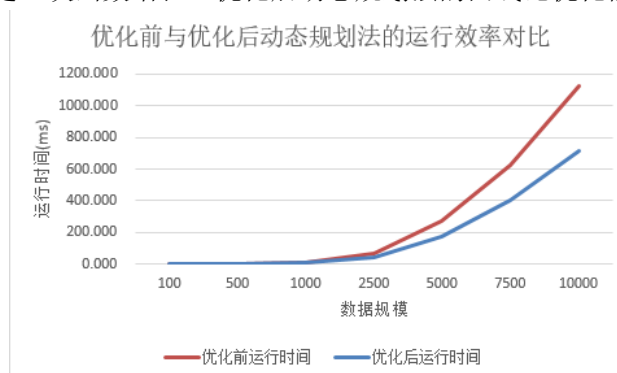


图 21：优化前与优化后动态规划法的运行效率对比

5.5 能在有限时间处理完的最大规模数据

最大测试了规模为 3000000 的数据，实际运行时间为：39242.468s，约为 11 个小时。

四、实验心得

1. 蛮力法也可使用类似动态规划的思想来解决冗余步骤的问题以极大程度提升效率。
2. 动态规划法采用自底向上的方式对状态进行更新，以优秀的运行效率取胜。
3. 动态规划法有时候可以利用问题的特性，进行时间复杂度与空间复杂度的优化。

五、附件说明

- code（实验所编写代码）
 - brute_force.cpp（蛮力法）
 - dp.cpp（未进行空间优化前的动态规划法）
 - generatedata.cpp（随机数据生成器）
 - optimized_brute_force.cpp（优化后的蛮力法，即记忆化搜索）
 - optimized_dp.cpp（空间优化后的动态规划法）
- result（实验结果数据表）
- PPT（演讲 PPT）
- PDF（实验报告 PDF 版本，更好的阅读体验）

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。