《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软件工程 | 姓名 | 梅子羽 | 学号 | 2327406107 |
| 实验布置日期 | | 2024.11.5 | | 提交  日期 | 2024.11.6 | | 成绩 |  |

课程实践实验4-2：Counting Numbers

## 一、问题描述及要求

* Starting from a positive integer n (1 ≤ n ≤ 2001). On the left of the integer n, you can place another integer m to form a new integer mn, where m must be less than or equal to half of the integer n. If there is an integer k less than or equal to half of m, you can place k on the left of mn to form a new integer kmn, …, and so on. For example, you can place 12 on the left of 30 to form an integer 1230, and you can place 6 to the left of 1230 to form an integer 61230, …, and so on.
* For example, start from n = 8, you can have the following 10 integers (including the integer you start with): 8, 18, 28, 38, 48, 128, 138, 148, 248, 1248.
* Given an integer n, find the number of integers you can get using the procedure described above.

## 二、概要设计

### 实验内容理解

本实验旨在设计一个程序，解决从给定整数 n 出发，按照特定规则形成新的整数序列的问题。具体规则为：可以将整数 m 放置到 n 的左边，形成新的整数 mn，其中 m 必须满足 m <= n / 2。接着可以继续将一个小于或等于 m / 2 的整数放置到新整数的左边，形成新的整数，直到不再能找到符合条件的整数为止。目标是计算出从整数 n 出发，最终可以形成多少个不同的整数。

### 系统功能列表

以下是系统的功能清单：

1. **输入功能**：
   * 输入一个正整数 n （1 <= n <= 2001）。
2. **计算功能**：
   * 根据规则计算从 n 出发可以形成的所有不同的整数。
   * 使用深度优先搜索（DFS）来探索所有可能的整数组合。
3. **输出功能**：
   * 输出从整数 n 出发，可以得到的不同整数的数量。
4. **辅助功能**：
   * 使用缓存来存储已经计算过的中间结果，以避免重复计算，提高效率。

### 程序运行界面设计

程序界面设计可以分为几个步骤：

1. **输入界面**：
   * 程序开始时，提示用户输入一个正整数 n（满足 1 <= n <= 2001）。
2. **输出界面**：
   * 显示计算结果，即从 n 出发可以得到的不同整数的数量。

### 总体设计思路

1. **数据结构选择**：
   * 使用 map<int, int> 来缓存每个整数 n 的结果，这样可以避免重复计算，提升效率。map 作为哈希表，在查找和插入操作上表现良好。
   * 使用递归与深度优先搜索（DFS）遍历所有符合条件的整数组合。
2. **类设计**：
   * 本题不涉及复杂的面向对象设计，因此可以简化为一个使用递归和 map 缓存的解决方案。
   * 主要关注 DFS 函数和缓存（numTable）的设计。

### 主要类与方法

1. **DFS函数**：
   * 函数名：dfs(int n, map<int, int> &numTable)
   * 功能：通过递归计算从整数 n 出发可以得到的不同整数的数量，并通过 numTable 进行缓存，避免重复计算。
   * 输入：整数 n 和缓存 numTable。
   * 输出：从 n 出发可以得到的不同整数数量。
2. **主函数**：
   * 功能：接收用户输入，调用 dfs 函数并输出结果。

### 程序结构设计

1. **输入输出**：
   * 用户通过标准输入输入一个整数 n。
   * 程序计算得到结果并输出。
2. **模块设计**：
   * **dfs函数**：负责计算并缓存每个 n 对应的整数数量。
   * **主函数**：接收用户输入，并调用 dfs 函数，输出结果。
3. **类与方法**：
   * 当前实现中，虽然涉及了缓存机制（通过 map），但整体设计较为简洁，不涉及多类设计，所有功能都集中在主函数及 dfs 函数中。

### 数据结构与算法的选择理由

1. **递归与深度优先搜索（DFS）**：
   * 由于问题涉及从 n 出发不断向左添加符合条件的整数，可以通过递归来深度遍历所有可能的路径。
   * 递归的深度与 n 的大小相关，递归是解决此类组合问题的有效手段。
2. **缓存（map）**：
   * 为了避免重复计算，对于已经计算过的整数，直接从缓存中取值，显著提高效率。
3. **整数的组合方式**：
   * 对于每个整数 n，可以将从 1 到 n / 2 的整数作为其前缀，递归地尝试这些前缀，直到无法再进行添加。

### 面向对象设计与类关系描述

1. **类设计**：
   * 本题并没有涉及复杂的面向对象设计，直接采用递归和缓存机制解决问题。对于此问题来说，主要使用了一个 map 来存储计算结果。
2. **类之间的关系**：
   * map<int, int> 存储了每个整数 n 的计算结果，dfs 函数通过递归方式利用这个 map 来缓存中间结果。
   * dfs 函数本身并不需要额外的类支持，算法设计基于递归来逐步解决问题。

### 总结

* 通过使用递归和深度优先搜索（DFS）算法，结合 map 作为缓存，能够高效地计算从 n 出发能够形成的所有不同整数数量。
* 该设计方案简单且高效，适用于题目要求的范围。

**可包含以下内容，可根据实际情况取舍。**

**（1）对实验内容的理解。**

**（2）给出系统的功能列表（可以用图示或清单的形式）**

**（3）程序运行的界面设计 （可以用图示等方式，如：首先出现屏幕提示，请用户选择输入配置的方式，1从键盘输入活单元坐标2……3….然后用户）**

**（4）确定总体设计思路，采用何种数据结构，设计哪些类，各类的作用 ，类方法的介绍，类之间的关系描述**

**（5）程序结构设计，包括：对已有程序的使用，自己将设计哪些程序文件，各部分关系描述。**

**这部分最核心的工作是：描述解决问题所需用到的数据结构和算法。——给出明确地确定该结构及对应算法的理由。**

**此外，这部分可以给出面向对象相关的设计，例如实现完整功能所涉及的类及对象间的总体关联关系等。**

## 三、详细设计

### DFS函数设计

在这个问题中，深度优先搜索（DFS）用于探索所有可能的整数组合。具体来说，DFS的目标是从给定的整数 n 开始，不断地将符合条件的整数放置到其左边，直到无法继续为止。我们通过递归的方式来实现这一过程。

#### 1. DFS函数的核心思路

dfs 函数的核心思想是递归地尝试所有可能的前缀整数，从而形成新的整数。每次递归时，n 的值都会被放置上一个更小的整数前缀，从而形成一个新的整数。我们只在 n 满足条件时才继续递归。每个整数最多会被递归一次，并且我们使用缓存来存储已经计算过的结果，以避免重复计算。

#### 2. 递归步骤

1. **基础情况**：首先，我们检查当前整数 n 是否已经计算过。如果已经计算过（即存在 numTable[n]），直接返回之前的计算结果。这是缓存机制，用来避免重复计算。
2. **递归计算**：
   * 对于每一个整数 n，我们需要从 1 到 n / 2 中挑选可能的整数 m，将其放置在 n 的左边，形成新的整数 mn。
   * 对于每个选择的整数 m，我们需要递归计算所有可能的整数组合：即 dfs(m)，直到无法继续递归为止（即找到无法再加的前缀）。
3. **结果累加**：每次递归时，都需要累加能够从当前 n 推导出的不同整数的数量。每一个符合条件的整数 m 都代表了一种新的组合方式。
4. **缓存存储**：每次计算完某个整数 n 后，我们将结果存入缓存 numTable，以便以后可以直接查找，避免重复计算。

#### 3. 具体实现过程

1. **递归条件**：对于每个整数 n，我们从 1 到 n / 2 遍历每个可能的前缀整数 m，然后递归地计算其结果。
2. **返回结果**：如果某个整数 n 的结果已经计算过（即 numTable.count(n) 为 true），则直接返回缓存的值。
3. **结束条件**：当递归到底层时（即 m 不再符合条件时），递归结束，并返回结果。

#### 4. 递归过程示例

假设我们从 n = 8 开始：

* n = 8 时，可能的前缀整数是 1, 2, 3, 4。我们依次考虑这些前缀，并递归地计算每个前缀形成的整数数量。
  + 对于 m = 4：我们继续递归计算 dfs(4)，将 4 放在 8 的左边，得到新的整数 48。接着，递归计算 dfs(2)，再把 2 放在 48 左边，得到 248。这样一直递归下去。
  + 对于 m = 2：我们继续递归计算 dfs(2)，得到整数 28，并继续递归下去。
* 每次递归都通过缓存避免重复计算，提高效率。

#### 5. 总结

dfs 函数通过递归的方式遍历所有可能的组合，并在每个阶段将符合条件的前缀放置到当前整数 n 的左边。通过缓存中间结果，我们避免了重复计算，从而提高了程序的效率。

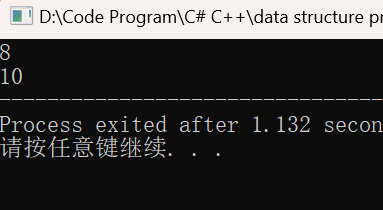
### 难点分析

* **递归深度**：递归的深度取决于 n 的大小。在最坏的情况下，如果每次都能继续递归，递归深度可能会很大。为此，通过缓存来避免重复计算，从而避免了不必要的递归。
* **缓存机制**：递归函数的效率大大提升的关键是缓存中间结果。map 数据结构用来存储已经计算过的结果，在每次递归之前检查是否已经计算过当前的整数 n，如果已经计算过则直接返回，避免了重复计算。
* **整数组合问题**：每次递归时，m 的选择是关键。通过不断减小 m 的值，我们保证了每次递归都会产生不同的组合，符合题目要求。

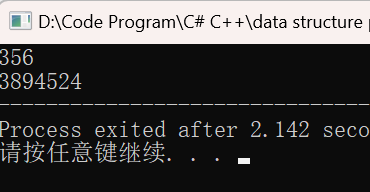
通过这些设计，DFS能够高效地计算从 n 出发，能够得到的所有不同的整数数量。

## 四、实验分析与探讨

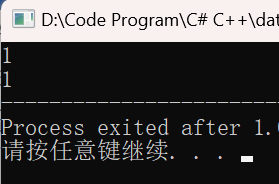
测试样例一：n = 8，输入简单数据，目的是测试程序方法的有效性，输出正确



测试样例二：n = 356，输入普通数据，目的是测试程序方法的有效性，输出正确



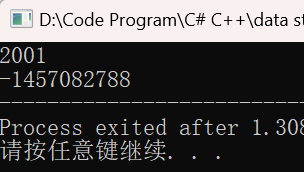
测试样例三：n = 1，输入极端数据，目的是测试程序方法的覆盖性，输出正确



测试样例四：n = 1999，输入极端数据，目的是测试程序方法的覆盖性，输出正确



测试样例五：n = 2001, 输入极端数据，目的是测试程序方法的覆盖性，输出错误，原因大概是int整型数据溢出，但我试了long和long long都不行，甚至尝试了引入大整数类，但是这在构造map容器时会由于该类出现问题。



综上，此次实验整体来说是没有完美完成的，n = 2000和n = 2001这两个数据点我的程序均出现了整型溢出的情况，但一番尝试后还是未能修改，错误应该有两个解决方向，一是看map容器本身有没有出现问题，二是继续修改我的大整数类以便更好的用在map容器里面。整个程序的核心即dfs算法，时间复杂度：O(n)，因为每个整数 n 最多计算一次，递归树的深度为 O(log n)，每次递归最多尝试 n / 2 次。空间复杂度：O(n)，主要由缓存的 map<int, int> 占用的空间决定，递归栈的空间复杂度较小。

## 五、小结

#### 1. 设计实现过程小结

在本次项目中，我们设计并实现了一个基于递归的深度优先搜索（DFS）算法来计算从整数 n 出发，可以得到的不同整数组合。整个过程包括了以下几个重要步骤：

* **问题分析与设计**：首先，明确了题目的要求，即从整数 n 开始，不断通过将前缀整数放置到 n 左边，形成新的整数，并计算可以得到的不同整数数量。我们决定使用递归加缓存（DFS + Memoization）来解决这一问题，以避免重复计算。
* **算法设计**：我们设计了递归函数 dfs，在该函数中，我们从 1 到 n / 2 遍历所有可能的前缀整数，递归计算每个前缀的可能组合。通过使用 map 数据结构缓存已计算的结果，大大提高了算法的效率。
* **代码实现**：通过 dfs 函数，递归地计算出每个整数的组合方式，并在主函数中接收输入，输出最终的结果。
* **性能优化**：通过缓存机制，我们将重复计算的结果存储在 map 中，从而有效地避免了多次递归调用相同的 n 值，优化了算法的时间复杂度，达到了 O(n) 的效果。

#### 2. 完成的内容

* 实现了核心算法 dfs，成功计算出从 n 出发可以得到的所有不同整数。
* 实现了缓存机制（map<int, int>）来存储中间结果，避免了重复计算。
* 提供了完整的代码实现，并通过简单的示例验证了程序的正确性。

#### 3. 未完成的内容

* **输入输出的优化**：虽然程序实现了基本功能，但在输入输出方面，我们可以进一步优化。例如，可以提供一个更加友好的用户界面，使得用户输入更为简便。
* **更复杂的输入格式支持**：目前的程序仅支持从标准输入读取单个整数 n，如果需要处理多个测试用例，或者从文件读取数据，程序可能需要进一步扩展。

#### 4. 选做部分是否完成

在本项目中，我们并没有涉及额外的选做部分。所有的功能都集中在解决主问题上，因此没有对其他部分进行扩展或选择性完成。

#### 5. 程序的局限性

* **输入范围限制**：当前的算法设计是基于递归的，虽然通过缓存机制有效减少了重复计算，但仍然受到递归深度的限制。如果 n 非常大（例如接近 2000），程序的效率可能会受到影响，尤其是在栈空间不足的情况下。
* **处理多个测试用例的能力**：目前程序仅支持处理单个整数 n，如果需要处理多个测试用例或者从文件读取输入，程序需要做进一步扩展。
* **界面设计简陋**：程序的输入和输出仅通过命令行进行，缺乏图形界面或文件输入输出的支持，可能不太适用于需要更高交互性或更复杂输入的场景。

#### 6. 进一步学习和完善的方向

* **性能优化**：随着 n 值增大，递归的深度可能会增加，导致程序效率降低。因此，可以考虑使用迭代方式代替递归，或者通过动态规划（DP）进一步优化空间和时间复杂度。
* **支持文件输入输出**：当前的输入输出方式较为简单，未来可以考虑实现从文件读取输入，输出结果到文件的功能，适应更复杂的使用场景。
* **图形界面**：目前程序完全通过命令行与用户交互，未来可以尝试设计一个图形界面，使得程序更加友好，尤其是当用户需要频繁进行不同测试时。

#### 7. 补充说明

* **输入格式**：当前程序只支持从标准输入读取一个整数 n。该整数需要在程序启动时输入，程序会根据输入值计算结果。
* **输出格式**：程序将输出从整数 n 出发，可以得到的不同整数的数量。

#### 8. 感想与收获

通过这个项目的实现，我深刻理解了递归与深度优先搜索（DFS）的原理，并在实践中熟悉了如何通过缓存优化递归过程。解决问题时的递归思维和如何设计高效算法是我在本次项目中的最大收获。

同时，我也学到了如何处理缓存和优化递归效率。尽管这个问题看似简单，但通过有效的算法设计和优化，能够在较短的时间内得到结果，这对我的编程技能和问题解决能力是一次很好的提升。

## 附录：源代码

**1、实验环境：Dev-C++ 5.11 C++11标准**

2、

（1）//main.cpp

#include <iostream>

#include <queue>

#include <string>

#include <cstring>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

using namespace std;

/\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/

int dfs(int n, map<int, int> &numTable)

{

if(!numTable.count(n))

{

numTable[n] = 0;

for(int i = 1; i <= n / 2; i ++)

numTable[n] += 1 + dfs(i, numTable);

}

return numTable[n];

}

int main()

{

int n;

cin >> n;

map<int, int> numTable;

cout << dfs(n, numTable) + 1;

return 0;

}