天津大学

《算法设计与分析》实验报告

题目：多种算法求解子集和数问题

姓名	学号	任务分工	成绩
江熠	3020207160	动态规划算法实现、论文写作	
罗奥成	3019207021	回溯算法、分支限界算法实现	

摘　要

本实验的主要目的是利用多种算法求解子集和数问题。子集和数问题给定了一个正整数c和由n个正整数组成的集合$W = \{w_1, w_2, \dots, w_n\}$，要求我们找到所有满足条件的集合$W_i$，使得$W_i \subseteq W$，且$W_i$中元素之和（即和数）为$c$。若找不到满足条件的集合，则找到使得和数尽量大且不超过c的W的子集。针对子集和数问题的特性，我们分别使用了动态规划算法、回溯算法以及分支限界算法，使用C++语言编写代码对问题进行求解，并利用 Florida State University 提供的 7 个数据集中的 6 个以及 1 个自编数据集（为了验证找不到和数刚好等于c的子集的情况）对算法正确性进行验证。除此以外，我们还分析了不同算法在解决这一问题时理论时间复杂度和实际运行时间的差异，归纳了三种算法各自的优缺点，并对进一步的工作做出展望。

关键词:

子集和数，动态规划，回溯，分支限界

目　录

1. 实验目的

1.1 问题的背景及意义

想象一下如下的场景：假如你是一名超市收银员，现在你需要给一位顾客找出一定数额的钱（例如 87 元）。收银机里有若干张面额不同的纸币（例如 2 张 50 元，3 张 20 元，2 张 10 元，5 张 5 元和 2 张 1 元），你是否能刚好将钱找给顾客？如果能，有几种可行的找零方案？如果不能，你最多能找给顾客多少钱？这个找零问题就可以被抽象为子集和数问题，并利用算法知识和计算机程序解决。

首先我们先给出子集和数问题的定义。在这个问题中，给定一个正整数c和由n个正整数组成的集合$W = \{w_1, w_2, \dots, w_n\}$，要求我们找到所有满足条件的集合$W_i$，使得$W_i \subseteq W$，且$W_i$中元素之和（即和数）为$c$（最优解）。若找不到满足条件的集合，则找到使得和数尽量大且不超过c的W的子集（次优解）。

1.2 课题的任务与目的

本次课题的任务就是利用在本课程中学习到的算法知识解决上述的子集和数问题。通过本次实验，我们可以从实践的角度掌握利用算法求解复杂问题的流程。在分析问题的过程中，我们能够强化对课堂所学的几大基础算法的原理的理解。在编写代码时，我们可以提升利用算法原理进行编码实践的能力。最后，我们还能从本次课题中拓展算法改进的思路，真正掌握《算法设计与分析》这门课的核心思想。

1.3 解决方案的主要思路

子集和数问题是一个 NP-难问题，使用暴力求解的方法需要对其每一个子集进行和数的计算，时间复杂度高达$O(2^n)$。为了避免较高的时间复杂度，我们可以利用算法对问题的求解过程进行优化。常见的解决 NP-难问题的算法有动态规划算法、回溯算法和分支限界算法。这三种算法都可以解决子集和数问题。下面对这三种算法解决子集和数问题的主要思路进行阐述。

1.3.1 动态规划算法

动态规划算法可以解决的另一个经典的 NP-难问题就是 0-1 背包问题。我们发现，子集和数问题可以转化为一个特殊的 0-1 背包问题。在 0-1 背包问题中，向背包中装入某物品i，背包容量减少了w_i，而价值增加了c_i。而子集和数问题

中，将某正整数“装入”子集后，子集剩余能装入的数的和，即“容量”减少了w_i，而目前子集的和数，即“价值”增加了w_i。因此，子集和数问题可以看作是每个物品重量和价值相等的特殊 0-1 背包问题。我们就可以仿照 0-1 背包问题，写出相应的状态转移方程，并编写代码进行求解。

1.3.2 回溯算法

子集和数问题可以被展开为一个解空间树。对于第i层的节点来说，其左孩子表示子集中存在第i个数，右孩子表示不存在。容易得到这个解空间树为一棵$i+1$层的完全正则二叉树，从根节点到叶节点的每一条路径都是解空间的一个元素。回溯算法使用深度优先搜索（DFS）的方法对该树进行搜索。为了剪枝，我们需要对每个节点进行限界。节点的下界即为当前子集的和数，上界为当前子集的和数与W中所剩元素的和数之和。如果某个节点的下界刚好等于c，则找到一个可行解；如果节点的下界大于c或是上界小于c，则这个节点是不可行的，可以将其“杀死”；否则，则需要继续扩展，直到找出可行解。与动态规划算法不同的是，回溯算法不能同时考虑最优解和次优解的情况，因为对于某一固定的c值来说，回溯算法只会找出和数刚好等于c的子集。如果整棵树的节点都被遍历仍没有找出可行解，则证明没有最优解，逐步减小c的值，直到寻找到次优解为止。

1.3.3 分支限界算法

分支限界算法解决该问题的思想与回溯算法很类似，区别在于分支限界算法使用的是广度优先搜索（BFS）的方法对解空间树进行搜索。用于剪枝的限界函数与上述的回溯算法中的限界函数是一样的。需要注意的是，由于我们需要找到每一个可行解，而不是找出最优解，故不能使用 LC 分支限界算法，只能使用 FIFO 分支限界算法。和回溯算法一样，分支限界算法也不能同时考虑最优解和次优解的情况，若没有最优解，分支限界算法也需要逐步减小c的值直到寻找到次优解。

2. 实验设计流程

下面将解释以上的三种算法具体如何求解子集和数问题。

2.1 准备工作

为了实现接下来的算法，首先我们需要把文件中的数据集读入程序中。为此我们需要包含 fstream 库，并且编写一个 testcase_read()函数，该函数接受一个代

表当前读取数据集编号的字符串作为参数。对于每个数据集，需要读入两个.data文件，分别代表正整数c和正整数集W。例如，第一个数据集包含两个文件subsetsum-1-1.data 和 subsetsum-1-2.data。前者保存的是正整数c的值，读入整型变量 c 中；后者保存的是正整数集W的值，读入整型数组 w[1:n]中。除此以外，还定义了布尔型数组 x[1:n]表示符合条件的子集，x[i]为1表示W的第i个元素在子集当中，为0则表示不在子集当中，我们称之为解向量。初始化 x[1:n]中每个元素为0，代表一开始的子集为一个空集。

2.2 动态规划算法

设$f(i,y)$表示条件为$c'=y, W'=\{w_{i+1}, w_{i+2}, \dots, w_n\}$的子集和数问题的最大子集和数的值，则初始条件为

$$f(n,y)=\begin{cases} w_n & ,y \geq w_n \\ 0 & ,0 \leq y < w_n \end{cases},$$

状态转移方程为

$$f(i,y)=\begin{cases} \max\{f(i+1,y), f(i+1,y-w_i)+w_i\} & ,y \geq w_i \\ f(i+1,y) & ,0 \leq y < w_i \end{cases}.$$

求出$f(1,c)$的值，即可得到符合条件的最大子集和数的值。

我们现在已经能得到最大子集和数的值了，接下来该如何找到相应的解向量呢？我们只需要观察每次状态转移方程中$f(i,y)$的值从何而来即可。例如，对于$f(1,c)$，若$0 \leq y < w_i$，则表明第1个数一定不在子集当中，继续考虑$f(2,c)$。反之则需要分三种情况讨论：如果$f(2,c) > f(2,c-w_1)+w_1$，则表明第1个数不在子集当中，继续考虑$f(2,c)$；如果$f(2,c) < f(2,c-w_1)+w_1$，则表明第1个数在子集当中，继续考虑$f(2,c-w_1)$；如果$f(2,c)=f(2,c-w_1)+w_1$，则表明第1个数在或者不在子集当中都能够得到最优解，$f(2,c-w_1)$和$f(2,c)$这两条路径都需要考虑。这样一步步回溯得出最优解的过程，直到$i=n$，若$0 \leq y < w_n$，则表明第n个数不在子集当中；反之则表示第n个数在子集当中。至此就得到了一个或数个满足条件的解向量 x[1:n]。

为了进行上述的回溯过程，我们需要建立一个二维数组 dp_table[][]来存储所有$f(i,y)$的值以便于回溯。初始化 dp_table[1:n][0]均为0，dp_table[n][0:c]根据初始条件赋值。例如，对于第 3 个数据集，初始化 dp_table 如下：

	0	1	2	3	4	5	6	7	8
1	0								
2	0								
3	0								
4	0	0	0	0	4	4	4	4	4

图 1　dp_table 示意图（1）

初始化之后，根据状态转移方程填充 dp_table。注意填充顺序是从左到右，填充完下一行之后再填充上一行，在上述例子中得到的 dp_table 如下：

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	7
2	0	0	2	2	4	4	6	6	6
3	0	0	0	0	4	4	4	4	4
4	0	0	0	0	4	4	4	4	4

图 2　dp_table 示意图（2）

至此我们已经得到了最大子集和数$f(1,c)$。接下来，从 dp_table 的右上角元素 dp_table[1][c]开始，按照之前所叙述的方法，对得到最优解$f(1,c)$的过程回溯：

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	7
2	0	0	2	2	4	4	6	6	6
3	0	0	0	0	4	4	4	4	4
4	0	0	0	0	4	4	4	4	4

图 3　dp_table 示意图（3）

即得到解向量 x[1:n]。上述例子中得到的解向量为$x = (1,1,0,1)$。

整个动态规划算法的流程图表示为：

图 4　动态规划算法流程图

2.3 回溯算法

1.3.2 节中说到，子集和数问题可以被展开为一个解空间树。对于第i层的节点来说，其左孩子表示子集中存在第i个数，右孩子表示不存在。例如，对于第 3 个数据集来说，解空间树展开如下：

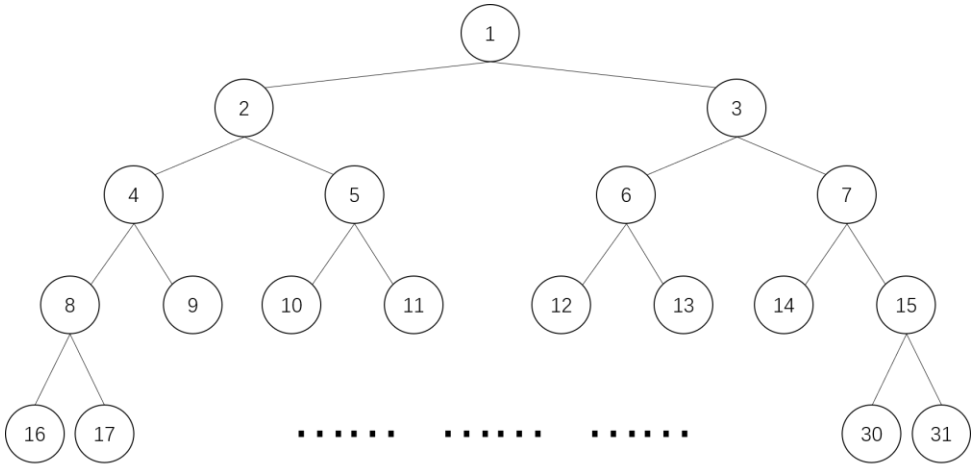

图 5　解空间树示意图

回溯算法使用 DFS 对于该解空间树进行搜索。如果搜索到了叶节点，证明已经到了树的底部仍没有搜索出可行解，可以向树的根部进行回溯。对于除了叶节点以外的其他节点，需要考虑是否继续搜索其左孩子和右孩子。对于左孩子，若其子集的值刚好等于c，则表明它是一个可行解，将其输出，再继续搜索左孩子是没有意义的，将其杀死；若其下界超过了c或上界低于c，则表明搜索其左孩子一定得不到可行解，将其杀死；否则可以继续搜索左孩子。对于右孩子，其下界一定是低于c的，如果其上界低于c，则将其杀死，否则继续搜索右孩子。如果这个节点的孩子都已经被杀死了，那么这个节点也就无法活动了，故杀死该节点并向树的根部回溯。当树中已经没有活的节点时，就得到了所有可行的最优解。

如果没有得到可行的最优解，则证明不存在和数刚好等于c的子集。为了得到和数不超过c且最大的子集，令 c=c-1，重复以上搜索过程，若找到了可行解即

可输出可行解并退出算法，若仍未找到则再将 c 减去 1，以此类推，直到找到可行解，即可确定和数尽量大且不超过c的次优解。

回溯算法的流程图表示为：

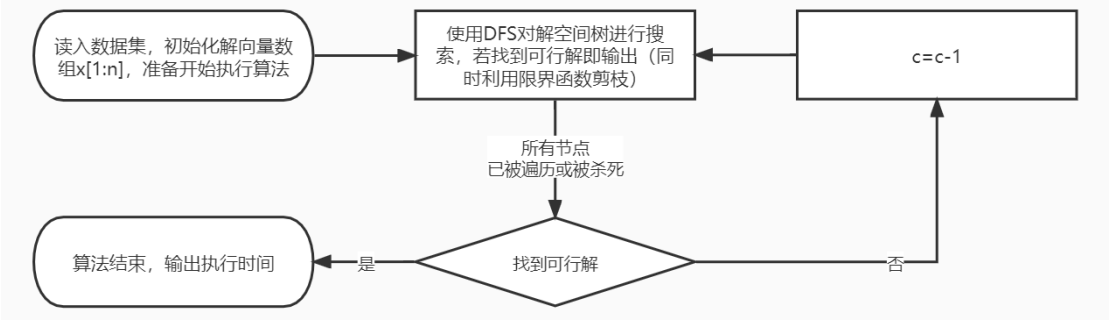

图 6　回溯算法流程图

2.4 分支限界算法

分支限界算法使用 BFS 对于该解空间树进行搜索。在这里因为我们需要搜索所有的可行解，故不能采用 LC 分支限界算法，只能采用 FIFO 分支限界算法。先将根节点加入 FIFO 队列。取出队列头部的节点，若为叶节点，则直接将其舍弃，否则需要判断其能否被扩展。对于左孩子，若其子集的值刚好等于c，则表明它是一个可行解，将其输出，不需要将其加入队列；若其下界超过了c或上界低于c，则表明搜索其左孩子一定得不到可行解，也不需要加入队列；否则可以继续搜索左孩子，将左孩子加入队列。对于右孩子，其下界一定是低于c的，如果其上界低于c，则其不能被扩展，否则将右孩子加入队列。当队列为空时，就得到了所有的可行的最优解。在具体实现中，为了代码的简洁和实现的简便，我们采用了递归而不是迭代的方式表示节点出入队，但实际的出入队过程是一致的。

若没有找到可行的最优解，则需继续搜索次优解，搜索策略与回溯法相同。

分支限界算法的流程图表示为：

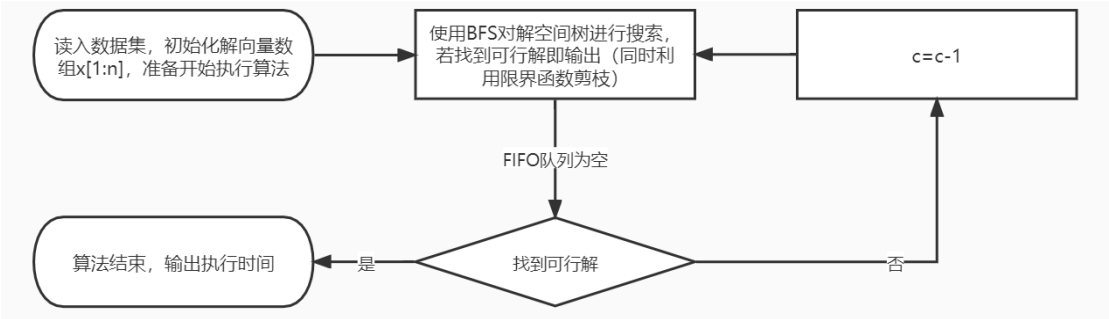

图 7　分支限界算法流程图

3. 实验结果及复杂性分析

3.1 实验环境设置

本实验在基于 Linux 系统的 Ubuntu 虚拟机下进行，Ubuntu 版本号为 20.04。使用的编译器为 g++，版本号为 9.3.0。为了更加方便地编写代码，我们还使用了 Visual Studio Code 的 Remote-SSH 扩展连接虚拟机。

为了便于编译和测试程序，我们使用了 Makefile 文件实现自动化编译和链接。在 Linux 系统中工程的根目录下执行 make 命令，将自动编译 src 文件夹下的 dp-subsetsum.cpp、bt-subsetsum.cpp 和 bb-subsetsum.cpp 三个代码文件并逐个运行，输出运行的结果。

3.2 测试方法及性能评价指标

3.2.1 测试方法

为了验证算法的正确性，我们使用 Florida State University 提供的 7 个数据集中的 6 个（P01、P02 和 P04~P07）以及 1 个自编数据集（为了验证找不到和数刚好等于c的子集的情况）对算法正确性进行验证。所有数据集的c存储于 data 文件夹下文件名以“-1”结尾的.data 文件中，W存储于文件名以“-2”结尾的.data 文件中，并以 0 作为数据结尾的标志。

3.2.2 性能评价指标

为了测试三种算法的实际耗时，我们还需要计算每种算法测试每一个样例时的运行时间。由于运行时间太短，不能用自然时间来表示运行时间，我们利用了 ctime 库中的 clock()函数来计算运行时间。clock()函数的返回值为从开始此程序到调用 clock()函数之间的 CPU 时钟计时单元（clock tick）数。对于每一个样例，在数据集读入完毕时将当前 clock()函数的返回值存储至 time1 中，直至算法执行完毕时，计算当前 clock()的返回值与 time1 的差值，此差值即可表示算法对于该样例的实际运行时间。

3.3 实验结果分析

3.3.1 算法的正确性检验

在工程的根目录下执行 make 命令，输出结果如下：（受篇幅限制，这里只展示了一次 make 命令的结果）

```
g++ -c src/bb-subsetsum.cpp
g++ -o bb-subsetsum bb-subsetsum.o
g++ -c src/bt-subsetsum.cpp
g++ -o bt-subsetsum bt-subsetsum.o
g++ -c src/dp-subsetsum.cpp
g++ -o dp-subsetsum dp-subsetsum.o
NOW RUNNING...
=====Dyanamic Programming Algorithm=====
Testing case 1:
0 1 1 0 0 1 0 1
1 0 1 0 0 0 1 1
1 1 0 0 0 0 1 0
Time cost: 28
Testing case 2:
0 0 1 1 1 0 1 0 1 0
Time cost: 322
Testing case 3:
1 1 0 1
Time cost: 5
Testing case 4:
0 0 1 0 1 1 1 1 0 1
0 0 1 0 1 1 1 1 1 0
Time cost: 23
Testing case 5:
0 1 0 0 0 0 0 1 1
Time cost: 12
Testing case 6:
0 1 1 0 1 0
Time cost: 7
Testing case 7:
1 0 0 0 1 0 0 0 0 1
Time cost: 9
========================================
```

图 8　动态规划算法运行结果

```
==========Backtrack  Algorithm==========
Testing case 1:
1 1 0 0 0 0 1 0
1 0 1 0 0 0 1 1
0 1 1 0 0 1 0 1
Time cost: 21
Testing case 2:
0 0 1 1 1 0 1 0 1 0
Time cost: 14
Testing case 3:
1 1 0 1
Time cost: 6
Testing case 4:
0 0 1 0 1 1 1 1 1 0
0 0 1 0 1 1 1 1 0 1
Time cost: 14
Testing case 5:
0 1 0 0 0 0 0 1 1
Time cost: 7
Testing case 6:
0 1 1 0 1 0
Time cost: 6
Testing case 7:
1 0 0 0 1 0 0 0 0 1
Time cost: 19
========================================
```

图 9　回溯算法运行结果

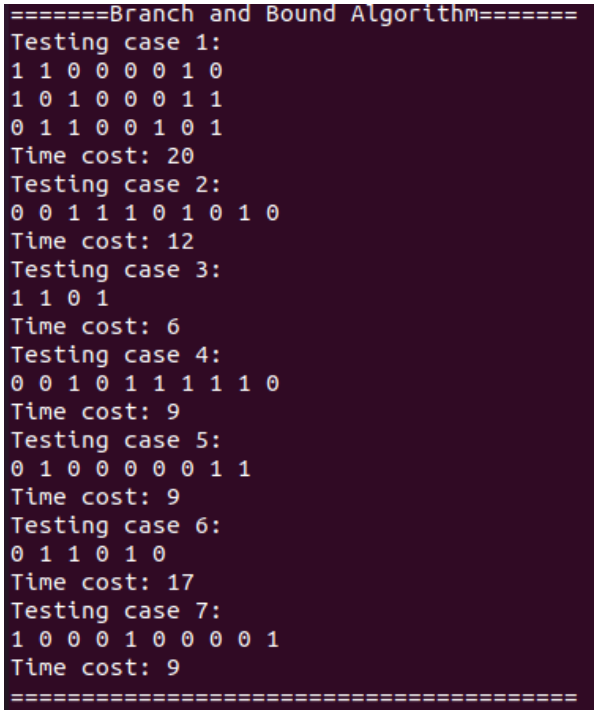

图 10　分支限界算法运行结果

经检验，三种算法输出的结果是一致且正确的，这证明了我们编写的三种算法在当前的子集和数问题下都是正确的。

3.3.2 理论复杂度分析

动态规划算法：此算法的核心任务是填充 dp_table。dp_table 为一个n行$c+1$列的二维数组，填充该数组的每一项的时间复杂度为$O(1)$，故填充整个数组的时间复杂度为$O[n(c+1)]$。除此以外，还需要进行时间复杂度为$O(n)$的回溯以确定解向量。故理论时间复杂度为$O[n(c+1)]+O(n)=O(nc)$。

回溯算法：该问题的解空间是一棵完全正则二叉树，其节点个数为2^{n+1}。最坏情况下需要遍历每一个节点，时间复杂度为$O(2^{n+1})=O(2^n)$。使用限界函数进行剪枝并不能在形式上降低时间复杂度，故算法的时间复杂度为$O(2^n)$。

分支限界算法：因为在这里使用的是 FIFO 分支限界算法，故其与回溯算法相似，最坏情况下也需要对解空间的每一个节点进行遍历，时间复杂度为$O(2^{n+1})=O(2^n)$。使用限界函数进行剪枝并不能在形式上降低时间复杂度，故算法的时间复杂度为$O(2^n)$。

3.3.3 实际性能评价

为保证实验结果的准确性，减少系统中其它因素对于算法执行时间的影响，执行 3 次 make 命令，得到三种算法对于每个样例的平均执行时间：

表 1 不同数据集下三种算法的平均执行时间

	数据集 1	数据集 2	数据集 3	数据集 4	数据集 5	数据集 6	数据集 7
动态规划算法	29.67	324.67	5.33	17.33	11.33	6.67	9.00
回溯算法	19.67	13.33	5.33	13.33	7.00	6.00	12.00
分支限界算法	18.67	11.00	6.00	7.33	7.00	9.33	8.33

可以看到，除了数据集 2 以外，其它的数据集对于三种算法的运行时间差别不太大。为什么会造成这样的现象呢？这是因为其他的数据集的c值都较小，$O(nc)$和$O(2^n)$相差不大，所以三种算法的运行时间差别也不大。而数据集 2 的c值比较大，导致$O(nc) \gg O(2^n)$，所以动态规划算法的运行时间远长于回溯算法和分支限界算法。

4. 结论与展望

4.1 解决方案的整体评价

4.1.1 解决方案的优势

在本次实验中，我们分别采用了三种不同的算法对于子集和数问题进行求解。对于动态规划算法，我们使用了迭代填写 dp_table 并回溯的方式进行求解，巧妙地避免了因为递归求解导致难于找出解向量的问题，并将时间复杂度降为伪多项式复杂度$O(nc)$。对于回溯算法和分支限界算法，我们使用了有效的限界函数对于解空间树进行了较为充分的剪枝，提高了搜索效率，降低了实际运行时间。

4.1.2 解决方案的不足

对于动态规划算法，虽然形式上来说时间复杂度为多项式，但实际上其为一个伪多项式，算法的时间复杂度由n和c的大小同时决定。如果c和n的数量级差距过大，如数据集 2，则会导致nc的值偏离多项式的程度较大。此种情况下$O(nc)$的时间复杂度可能还不如暴力求解的$O(2^n)$的时间复杂度，不仅没有起到优化的作用，反而增加了实际运行时间。另外，c值过大也有可能导致 dp_table 过大而超出存储限制，这也是在测试中我们没有采用 Florida State University 的 7 个数据集中的 P03 的原因。

4.2 进一步工作的展望

在研读了 Konstantinos Koiliaris 和 Chao Xu 的 Subset Sum Made Simple 以及 A faster pseudopolynomial time algorithm for subset sum (In Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms)两篇文献后，我们了解了伪多项式时间复杂度算法(pseudopolynomial time algorithms)与 NP-难问题。我们希望可以进一步完善算法，使其更接近伪多项式时间复杂度算法。同时，对于c值过大的数据集（如 Florida State University 提供的 7 个数据集中的 P03）使用动态规划算法时会导致 dp_table 超出存储限制而不能通过回溯得到所有可行解的问题。在之后的学习中，我们希望可以提出可行的解决办法，目前来说以此动态规划算法得到子集和数问题的c值过大的数据集的所有解是不具备可能的。

除此以外，针对其他两种需要搜索解空间树并对其进行剪枝的算法，我们希望能够找到更好的限界函数，能够更加充分地剪去不会产生可行解的分支，这可以大大加快搜索解空间树的速度。

5. 参考文献

[1] Sartaj Sahni. Data Structures, Algorithms, and Applications in C++. Gainesville, Florida. 2004.

[2] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein. Introduction to Algorithms. Massachusetts Institute of Technology. 2009.

[3] 严蔚敏,吴伟民.数据结构[M].北京:清华大学出版社,1997.