天津大學

算法综合实验

子集和数问题求解

石云天（3020205015） 分工：回溯法、报告撰写

张冬驰（3020001361） 分工：分支限界法

汪业杭（3020001056） 分工：动态规划

目　录

一、摘要

子集和数问题是在真实世界应用中经常出现的一种简单且基础的问题。直接的枚举搜索可能遍历问题的所有2^n个解空间，即直接搜索最坏情况下的时间复杂性为$O(2^n)$。本文提出了三种解决子集和数问题的方法——回溯法、分支限界法与动态规划。实验过程中使用 C++实现了这三种算法，并利用提供的数据集样例进行了性能测试。回溯算法以深度优先搜索的形式展开，最坏情况下时间复杂度为$O(2^n)$；分支限界算法针对某些样例表现了较好的时间性能，但其最坏情形的时间复杂度仍为$O(2^n)$；动态规划算法是以空间换时间，将每个子问题求解结果保存在一个 $C*n$的表格中（C 指所求子集和数，n 为输入自然数个数），使得遇到重复子问题时不再求解，其时间复杂度为$O(C*n)$。

二、实验设计与实现

2.1 回溯法

2.1.1 回溯法简介

回溯法（Backtracking）是一种常用的算法思想，用于解决在给定约束条件下的求解问题。它通过逐步构造解空间，并在搜索过程中剪枝，以减少不必要的搜索，从而找到问题的解。回溯法通常适用于组合优化问题、排列组合问题、子集问题和搜索问题等。

回溯法的基本思想是通过试探和回溯的方式进行搜索。它的搜索过程类似于深度优先搜索（DFS），但与普通的 DFS 不同，回溯法在搜索过程中具有剪枝的能力，以避免不必要的搜索。它会尝试一种可能的选择，并继续向下搜索，如果发现当前选择不能满足约束条件或达到期望的解，则回溯到上一步进行其他选择，直到找到问题的解或穷尽所有可能的选择。

下面是利用回溯法解决问题的一般步骤：

1. 确定问题的解空间：根据问题的特点，确定问题的解空间，通常可以使用树形结构或图形结构来表示。

2. 定义状态空间：将解空间映射为状态空间，状态空间中的每个状态代表问题的一个部分解。

3. 确定约束条件：定义约束条件，用于判断当前状态是否满足问题的要求。

4. 定义搜索顺序：确定搜索的顺序，可以根据问题的特点选择深度优先搜索或广度优先搜索。

5. 进行搜索：从初始状态开始，根据搜索顺序选择一个状态进行扩展，生成下一层的状态，然后再进行进一步的搜索。

6. 判断是否满足约束条件：在搜索过程中，对每个状态进行约束条件的判断，如果满足约束条件，则保留该状态作为部分解，否则剪枝。

7. 判断是否达到最终解：当搜索到达最终解或无法继续搜索时，根据需要输出或记录找到的解。

需要注意的是，在实际应用中，回溯法通常通过递归的方式实现。在每一步的搜索过程中，递归调用自身来处理下一层的状态，并在递归返回时进行回溯。

总结来说，回溯法通过试探和回溯的方式搜索解空间，它的搜索策略可以根据问题的性质选择深度优先搜索、广度优先搜索、最佳优先搜索或启发式搜索等方法。通过合理选择搜索策略和剪枝技巧，可以提高回溯法的效率，并解决各种约束求解、组合优化和搜索问题。

2.1.2 算法设计思想

利用回溯法解决子集与数问题的算法设计思想如下：

1. 定义回溯函数 backtrack，该函数的参数包括：

nums：输入的数字样本。

target：目标和。

currSet：当前的子集。

index：当前处理的数字的索引。

currSum：当前子集的和。

result：保存所有满足条件的子集的结果集。

2. 在回溯函数中，首先检查当前和 currSum 是否等于目标和 target，如果是，则将当前子集 currSet 添加到结果集中，并返回。

3. 从当前索引 index 开始，遍历剩余的数字。对于每个数字，进行以下操作：

• 判断当前和 currSum 加上当前数字是否小于等于目标和 target，如果是，则将当前数字添加到当前子集 currSet 中。

• 递归调用 backtrack 函数，将当前数字的索引更新为 i+1，当前和更新为 currSum+nums[i]。

• 递归结束后，回溯，将刚刚添加的数字从当前子集中移除。

4. 在主函数 main 中，首先接收用户输入的数字个数、数字样本以及目标和。

5. 调用 subsetSum 函数，传入数字样本和目标和，得到满足条件的子集。

6. 最后，遍历结果集并输出所有满足条件的子集。

在基础版回溯法代码的基础上可以进行剪枝优化，于是添加两个剪枝条件：

1. 在回溯函数的开头，我们检查当前和 currSum 是否已经等于目标和 target，如果是，直接将当前子集添加到结果集中并返回。这样可以避免继续进行不必要的递归调用。

2. 在循环中，我们添加了一个剪枝条件来避免处理重复的元素。如果当前数字和上一个数字相同，并且上一个数字已经被处理过（即 i > index），则跳过当前数字的处理。

这些剪枝条件能够减少不必要的操作和递归调用，从而降低时间复杂度。但是，这些优化方法可以提高算法的效率，但在某些情况下仍可能达到指数级复杂度。此外，剪枝优化后的代码在思想上与分支限界法有一些相似之处，但也有一些区别：

（1）共同点：

1. 剪枝优化和分支限界法都是通过减少搜索空间来提高算法效率。

2. 两种方法都使用了类似的剪枝条件来排除不必要的状态或分支。

3. 都是基于回溯的思想，通过遍历状态空间来找到满足条件的解。

（2）区别：

1. 剪枝优化主要集中在回溯过程中对当前状态进行剪枝，通过一些条件判断来减少递归调用和操作的次数，从而降低复杂度。

2. 分支限界法更侧重于在搜索过程中选择最有希望的分支，以优先搜索可能包含解的子集，同时通过界限函数（即优先级函数）进行剪枝，剔除不可能达到最优解的分支。它使用优先队列或优先级队列来维护候选解的优先级，以便选择最有希望的分支进行扩展。

在剪枝优化后的代码中，我们通过一些简单的条件判断来避免不必要的递归调用和操作，以减少搜索空间。与分支限界法相比，剪枝优化更加简单直接，没有显式地引入优先级队列或界限函数，而是通过条件判断来切除不必要的搜索分支。

总体上说，剪枝优化是一种简单而常见的优化方法，适用于一些简单的问题。而分支限界法则是一种更为细致和高级的搜索算法，适用于更复杂的问题，并且可以通过合理选择分支和设置优先级函数来进一步提高效率。

2.1.3 算法复杂度分析

基础版回溯法代码：

（1）时间复杂度分析：

在回溯函数中，我们使用一个循环来遍历剩余的数字。由于每个数字都有两种选择（选中或不选中），所以在最坏情况下，我们需要遍历所有的子集，时间复杂度为$O(2^n)$，其中 n 是数字的个数。

在主函数中，我们调用了回溯函数，因此主要的时间复杂度是由回溯函数决定的，即$O(2^n)$。

（2）空间复杂度分析：

空间复杂度分为两个方面考虑：首先是递归调用栈的空间，由于回溯法使用递归遍历解空间，在最坏情况下，即遍历整个解空间时，递归深度将达到集合大小，即 n，因此递归调用栈的空间复杂度为$O(n)$；其次是存储当前解的空间，最坏情况下，即每个元素都被包含在解中时，解的大小将达到集合大小 n，因此存储当前解的空间复杂度为$O(n)$，故整体空间复杂度为$O(n)$。

综上所述，该代码的时间复杂度为$O(2^n)$，空间复杂度为$O(n)$。但是，这里的指数级时间复杂度是由于在最坏情况下，需要遍历所有的子集。在实际应用中，可能存在剪枝等优化方法，可以减少实际执行的操作数量，从而提高效率。

剪枝优化后回溯法代码：

剪枝优化后的代码的时间复杂度和空间复杂度与未优化的代码基本相同。

（1）时间复杂度分析：

在回溯函数中，我们仍然需要遍历所有可能的子集，因为我们不知道哪些子集的和等于目标和。因此，最坏情况下，时间复杂度仍然为$O(2^n)$，其中 n 是数字的个数。虽然剪枝条件可以减少一些不必要的递归调用和操作，但在最坏情况下，仍然需要遍历所有的子集。

（2）空间复杂度分析：

改进后影响空间复杂度的两部分因素均未发生改变，空间复杂度不变，仍为$O(n)$。

综上所述，剪枝优化后的代码的时间复杂度仍为$O(2^n)$，空间复杂度仍为$O(n)$。尽管剪枝优化可以减少一些不必要的操作和递归调用，但在最坏情况下仍需要遍历所有的子集，因此复杂度没有显著变化。

2.1.4 运行测试结果

利用说明文档中提供的数据集对回溯法代码（剪枝优化后）进行测试，测试结果如下：

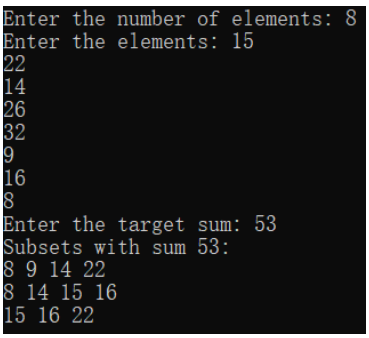

图 1 数据集 P01 测试结果

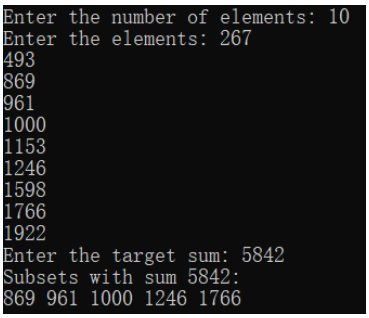

图 2 数据集 P02 测试结果

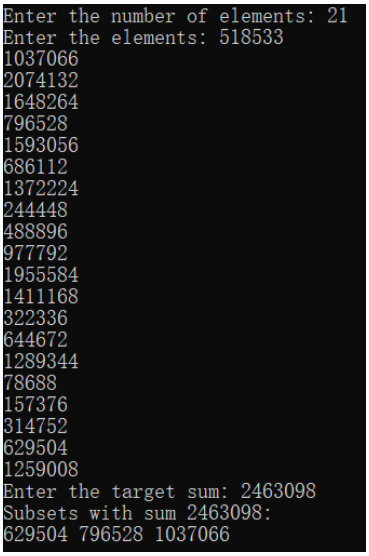

图 3 数据集 P03 测试结果

与网站提供结果进行对比，全部正确，代码实现成功。

2.2 分支限界法

2.2.1 分支限界法简介

分支限界法（Branch and Bound）是一种常用的求解优化问题的算法，它通过穷举搜索解空间并根据问题的特性进行剪枝，以达到快速找到最优解的目的。该方法在组合优化问题、整数规划、图论等领域具有广泛应用。

下面是分支限界法的基本步骤：

1. 初始化：设定问题的目标函数，定义问题的约束条件，并初始化当前最优解为无穷大。

2. 构造搜索树：将问题的解空间构造成一棵搜索树，每个节点表示一个问题的解。

3. 分支操作：选择一个未扩展的节点，进行分支操作。分支操作是指根据问题的约束条件，生成当前节点的子节点，将问题的解空间划分成更小的子空

间。

4. 限界操作：对于生成的子节点，进行限界操作，即计算子节点的目标函数值，并与当前最优解进行比较。如果子节点的目标函数值已经超过当前最优解，则可以剪枝，不再对该子节点进行进一步的搜索。

5. 更新当前最优解：如果生成的子节点的目标函数值优于当前最优解，则更新当前最优解。

6. 结束条件：重复执行步骤 3 至 5，直到搜索树中没有可扩展的节点，或者搜索到满足问题约束的最优解。如果搜索树为空，则问题无解。

7. 输出最优解：搜索结束后，输出当前最优解。

分支限界法的核心思想是通过剪枝操作减少搜索空间，从而提高算法的效率。在搜索树的构建过程中，通过限界操作确定哪些子节点不再需要进一步搜索，从而减少了搜索的复杂度。通过不断更新当前最优解，可以及时发现更好的解，避免不必要的搜索。

需要注意的是，分支限界法的效率和剪枝策略密切相关。不同的问题可能需要设计不同的剪枝策略才能达到较好的效果。另外，分支限界法的搜索空间大小取决于问题的规模，较大的问题可能需要更多的时间和资源来求解。

2.2.2 算法设计思想

1. 定义全局变量：代码中定义了全局变量 `bestSubset` 和 `bestSum`，用于存储最佳解的子集和和子集本身。这样做是为了在递归过程中能够记录最佳解，并在计算完成后进行输出。

2. 递归函数：`subsetSum` 函数使用递归的方式来进行分支限界法求解。函数接收参数 `numbers`（原始数据集合）、`subset`（当前处理的子集）、`currentIndex`（当前处理的元素索引）、`currentSum`（当前子集的和）和 `targetSum`（目标和）。

3. 判断最佳解：在每次递归调用前，代码检查当前子集的和是否等于目标和，如果是，则更新 `bestSubset` 和 `bestSum`。

4. 剪枝条件：在递归调用前，代码检查两个剪枝条件：

• 如果当前索引超过了数据集合的大小，即已经处理完所有元素，则直接返回。

• 如果当前子集的和已经大于目标和，则也可以直接返回，因为无论如何添加后续元素，子集的和只会越来越大。

5. 递归调用：在递归调用时，有两种情况：

• 包含当前元素：将当前元素添加到子集中，更新当前索引和当前子集的和，然后递归调用函数。

• 不包含当前元素：直接递归调用函数，不对子集和索引进行修改。

6. 主函数：主函数负责接收用户输入的数据，并调用 `subsetSum` 函数来求解子集和问题。最后输出最佳解的子集和。

2.2.3 算法复杂度分析

（1）时间复杂度分析：

时间复杂度因存在剪枝而不确定，因为传入数组中整数的顺序不同，故剪枝后的状态也不同，遍历到的结点个数也不一样。但由于 n 个整数的解空间树总结点数为2^n+1-1=2^n个，所以对应的最坏时间复杂度 O(2^n)

（2）空间复杂度分析：

如果存在解，则一定会遍历到底层叶子结点，递归过程中函数压栈，空间复杂度O(n)。

但在实际求解时，求解过程并没有真的建立这棵空间树。并且传入数组中大的整数越靠前，剪枝效果越明显。因此可以考虑在算法中先加入将传入数组从大到小排序。

2.2.4 运行测试结果

利用说明文档中提供的数据集对分支限界法代码进行测试，测试结果如下：

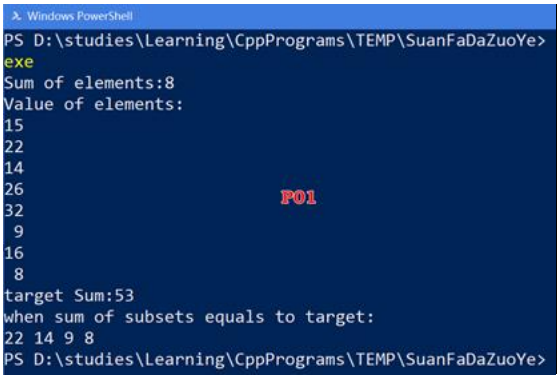

图 4 数据集 P01 测试结果

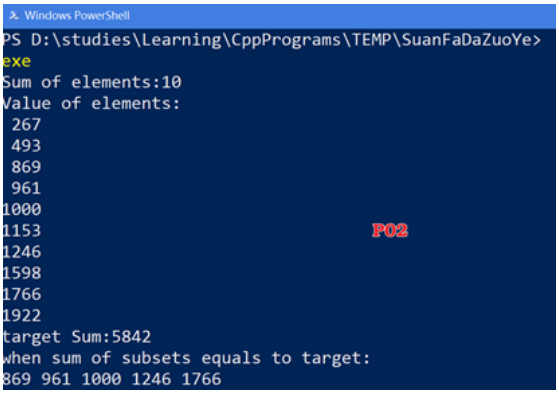

图 5 数据集 P02 测试结果

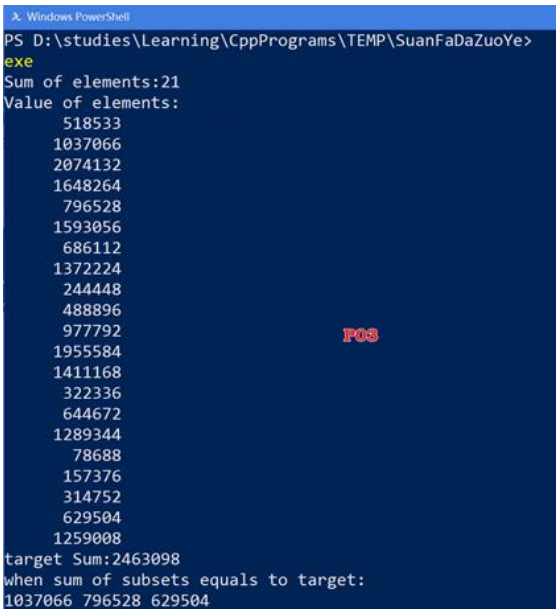

图 6 数据集 P03 测试结果

与网站提供结果进行对比，全部正确，代码实现成功。

2.2.5 小结

首先对回溯法与分支限界法的区别进行简要总结：

1. 回溯法：深度优先搜索，搜索所有的解。回溯法深度优先搜索堆栈活结点的所有可行子结点被遍历后才被从栈中弹出找出满足约束条件的所有解。

2. 分支限界法：广度优先搜索，搜索一个解或者是最优解。分支限界法广度优先或最小消耗优先搜索队列、优先队列每个结点只有一次成为活结点的机会找出满足约束条件的一个解或特定意义下的最优解。

结论：分支限界法实际上并不适合于子集和数问题的求解。因子集和数问题经常出现多个子集组合的和均可满足目标和的情况，而分支限界法仅可得出一种情况。分支限界法是暴力穷举法的一种优化，对于子集和数问题这一特定问题，如果将输入数组进行从大至小排序，可以有效降低可能的时间消耗。

2.3 动态规划

2.3.1 动态规划简介

动态规划（Dynamic Programming）是一种用于解决多阶段决策问题的算法思想，它将问题分解为多个重叠的子问题，并通过保存子问题的解来避免重复计算，从而提高算法的效率。

动态规划的基本思想是将原问题划分为若干个子问题，通过求解子问题的

解来逐步推导出原问题的解。在求解子问题时，动态规划会保存子问题的解，避免重复计算，这是动态规划与分治法的主要区别之一。

动态规划通常适用于具有重叠子问题和最优子结构性质的问题。重叠子问题意味着在问题的求解过程中，同一个子问题可能被多次遇到，为了避免重复计算，动态规划会使用一个表格（通常是数组）来保存子问题的解。最优子结构性质意味着原问题的最优解可以由子问题的最优解推导出来。

动态规划算法的一般步骤如下：

1. 定义状态：明确问题的子问题以及子问题之间的关系，并定义状态表示子问题的解。

2. 确定状态转移方程：利用子问题之间的关系，确定问题的状态转移方程。这个方程描述了问题的当前状态与下一个状态之间的关系。

3. 初始化：设置问题的初始状态，通常是最简单的子问题的解。

4. 递推求解：根据状态转移方程，从初始状态逐步递推到问题的最终状态，求解问题的最优解。

5. 提取最优解：根据求解过程中保存的信息，提取出最优解。

动态规划算法的关键是找到适合的状态表示和状态转移方程。一般来说，状态表示应该具备无后效性，即当前状态只依赖于之前的状态而不依赖于之后的状态。状态转移方程应该能够准确描述问题的状态之间的转移关系。

2.3.2 算法设计思想

动态规划方法的核心思想是将多阶段过程转化为一系列相互影响的单阶段问题，进而逐个求解获得最优决策。在该过程中，对每个子问题只求解一次，并将结果保存在一个$C*n$的表格中，之后重复该子问题时不再求解，直接查表以减少求解时间，以空间换时间。

在利用动态规划思想求解子集和问题时，可将数字视为 W 与 V 相等的 0/1 背包问题，使用二维数组保存子问题的解，并使用如下方法填表：

$$\begin{cases} m[i][j]=m[i-1][j], j<w[j]; \\ m[i][j]=\max\{m[i-1][j-w[i]]+v[i], m[i-1][j]\}, j\geq w[i] \end{cases}$$

2.3.3 算法复杂度分析

（1）时间复杂度分析：

由于动态规划方法每个子问题只求解一次并存储在一张$C*n$大小的表中（C 指所求子集和数，n 为输入自然数个数），随后直接查找即可，因此时间复杂度主要受生成表耗费时间影响，为$O(C*n)$。

（2）空间复杂度分析：

一般情况下，采用一张$C*n$的表存放各子问题的解，因此空间复杂度通常也为$O(C*n)$，但事实上通过优化可仅用两个一维数组进行存放更新子问题的解来进行解决，优化空间复杂度为$O(C)$，但时间复杂度将因此上升，得不偿失，因此本次实验采用常规方法建表进行存放，空间复杂度为$O(C*n)$。

2.3.4 运行测试结果

利用说明文档中提供的数据集对分支限界法代码进行测试，测试结果如下：

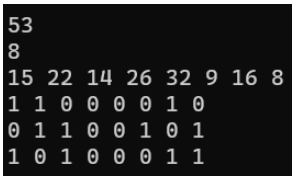

图 7 数据集 P01 测试结果

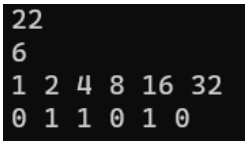

图 8 数据集 P06 测试结果

图 9 数据集 P07 测试结果

与网站提供结果进行对比，全部正确，代码实现成功。

三、总结

本实验针对子集和数求解问题，利用回溯、分支限界与动态规划三种方法分别进行求解，每种方法选取 3 组数据集进行性能测试，得到如下结论：

• 动态规划方法将原问题转换为多个子问题的求解并保存，随后重复子问题直接查表即可，时间复杂度较低，为$O(C*n)$，（C 表示输入整数的大小，n 表示输入整数的个数），但在 C 比较大时将生成一张列数较大的表，空间复杂度较高。

• 回溯法与分支限界法的本质不同是在于搜索解空间的遍历方式不同。回溯法是深度优先，穷尽解空间的所有可能，找到最优解。分支限界法是广度优先，本质上也是穷尽了解空间的所有可能，找到最优解。在本实验中两者时间复杂度相差不大，受数据影响，难以确定，最坏的情况可达到指数级$O(2^n)$。

四、附录

4.1 附录 1 基础版回溯法代码

```
#include <bits/stdc++.h>

using namespace std;

// 回溯函数
void backtrack(vector<int>& nums, int target, vector<int>& currSet, int index, int currSum,
vector<vector<int>>& result) {
    // 如果当前和等于目标和，将当前子集添加到结果集中
    if (currSum == target) {
        result.push_back(currSet);
        return;
    }

    // 从当前索引开始遍历剩余的数字
    for (int i = index; i < nums.size(); i++) {
        // 如果当前和加上当前数字小于等于目标和
        if (currSum + nums[i] <= target) {
            currSet.push_back(nums[i]);  // 将当前数字添加到当前子集中
            // 递归调用回溯函数，更新参数：index 为 i+1，currSum 为 currSum+nums[i]
            backtrack(nums, target, currSet, i + 1, currSum + nums[i], result);
            currSet.pop_back();  // 回溯，将刚刚添加的数字从当前子集中移除
        }
    }
}

vector<vector<int>> subsetSum(vector<int>& nums, int target) {
    vector<vector<int>> result;
    vector<int> currSet;
    backtrack(nums, target, currSet, 0, 0, result);
    return result;
}

int main() {
    int n;  // 数字个数
    cout << "Enter the number of elements: ";
```

```
    cin >> n;

    vector<int> nums(n);  // 数字样本
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    int target;  // 所需求和的值
    cout << "Enter the target sum: ";
    cin >> target;

    vector<vector<int>> subsets = subsetSum(nums, target);

    cout << "Subsets with sum " << target << ":" << endl;
    for (const auto& subset : subsets) {
        for (const auto& num : subset) {
            cout << num << " ";
        }
        cout << endl;
    }

    return 0;
}
```

4.2 附录 2 剪枝优化后回溯法代码

```
#include <bits/stdc++.h>

using namespace std;

void backtrack(vector<int>& nums, int target, vector<int>& currSet, int index, int currSum, vector<vector<int>>& result) {
    if (currSum == target) {
        result.push_back(currSet);
        return;
    }

    // 剪枝条件：当前和加上剩余数字的最小值大于目标和，或当前和加上剩余数字的最大值小于目标和
```

```
    if (currSum > target || (index < nums.size() && currSum + nums[index] > target)) {
        return;
    }

    for (int i = index; i < nums.size(); i++) {
        if (i > index && nums[i] == nums[i - 1]) {
            continue;  // 剪枝条件：避免重复元素导致的重复解
        }
        currSet.push_back(nums[i]);
        backtrack(nums, target, currSet, i + 1, currSum + nums[i], result);
        currSet.pop_back();
    }
}

vector<vector<int>> subsetSum(vector<int>& nums, int target) {
    vector<vector<int>> result;
    vector<int> currSet;
    sort(nums.begin(), nums.end());  // 首先对数字样本进行排序
    backtrack(nums, target, currSet, 0, 0, result);
    return result;
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    int target;
    cout << "Enter the target sum: ";
    cin >> target;

    vector<vector<int>> subsets = subsetSum(nums, target);

    cout << "Subsets with sum " << target << ":" << endl;
```

```
    for (const auto& subset : subsets) {
        for (const auto& num : subset) {
            cout << num << " ";
        }
        cout << endl;
    }

    return 0;
}
```

4.3 附录 3 分支限界法代码

```
#include <iostream>
#include <vector>

using namespace std;

// 定义全局变量，用于存储最佳解
vector<int> bestSubset;
int bestSum = 0;

// 计算子集的和
int calculateSum(const vector<int>& subset) {
    int sum = 0;
    for (int num : subset) {
        sum += num;
    }
    return sum;
}

// 分支限界法求解子集和数问题
void subsetSum(const vector<int>& numbers, vector<int>& subset, int currentIndex, int currentSum, int targetSum) {
    // 如果当前子集的和等于目标和，更新最佳解
    if (currentSum == targetSum) {
        bestSubset = subset;
        bestSum = currentSum;
    }

    // 如果已经处理完所有元素或者当前子集的和大于目标和，则剪枝返回
```

```
    if (currentIndex >= numbers.size() || currentSum > targetSum) {
        return;
    }

    // 包含当前元素的情况
    subset.push_back(numbers[currentIndex]);
    subsetSum(numbers, subset, currentIndex + 1, currentSum + numbers[currentIndex], targetSum);
    subset.pop_back(); // 回溯，移除刚添加的元素

    // 不包含当前元素的情况
    subsetSum(numbers, subset, currentIndex + 1, currentSum, targetSum);
}

int main() {
    int n; // 元素个数
    cout << "Sum of elements:";
    cin >> n;
    vector<int> numbers(n);
    cout << "Value of elements:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> numbers[i];
    }

    int targetSum;
    cout << "target Sum:";
    cin >> targetSum;

    vector<int> subset;
    subsetSum(numbers, subset, 0, 0, targetSum);

    // 输出最佳解
    cout << "when sum of subsets equals to target:" << endl;
    for (int num : bestSubset) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

4.4 附录 4 动态规划代码

```
#include<iostream>

using namespace std;

const int C = 53, N = 8, Max = 32;
extern int table[100][1000] = { 0 };
int Larger(int a, int b){
	if (a > b)  return a;
	else return b;
}

int check(int a, int b) {
	int c = a - b;
	for (int i = 0; i <= N; i++) {
		for (int j = 0; j <= Max; j++) {
			if (c == table[i][j])return 1;
		}
	}
	return 0;
}

int main() {

	int W[10] = {};


	for (int i = 1; i <= N; i++) {
		cin >> W[i];
	}

	for (int i = 1; i <= N; i++) {
		for (int j = 1; j <=C; j++) {
			if (j < W[i])table[i][j] = table[i - 1][j];
			else table[i][j] = Larger(table[i - 1][j - W[i]] + W[i], table[i - 1][j]);
		}
	}

	cout << endl << endl;
```

```
    int times = 0;
    for (int i = 0; i <= N; i++) {
        if (table[i][C] == C) {
            times++;
        }
    }

    for (int i = 0; i < times; i++) {
        int count[N + 1] = {}; int temp = C;
        for (int j = 1; j <= N; j++) {
            if (check(temp, W[j]==1)){
                count[j] = 1;
                temp = temp - W[j];
            }
        }
        for (int j = 1; j <= N; j++)cout << count[j] << ' ';
        cout << endl;
    }
    return 0;
}
```