Experiment No. 10
Sum of Subset using Backtracking
Date of Performance:
Date of Submission:

Title: Sum of Subset

Aim: To study and implement Sum of Subset problem

Objective: To introduce Backtracking methods

Theory:

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again.

The subset sum problem is a classic optimization problem that involves finding a subset of a given set of positive integers whose sum matches a given target value. More formally, given a set of non-negative integers and a target sum, we aim to determine whether there exists a subset of the integers whose sum equals the target.

Let's consider an example to better understand the problem. Suppose we have a set of integers [1, 4, 6, 8, 2] and a target sum of 9. We need to determine whether there exists a subset within the given set whose sum equals the target, in this case, 9. In this example, the subset [1, 8] satisfies the condition, as their sum is indeed 9.

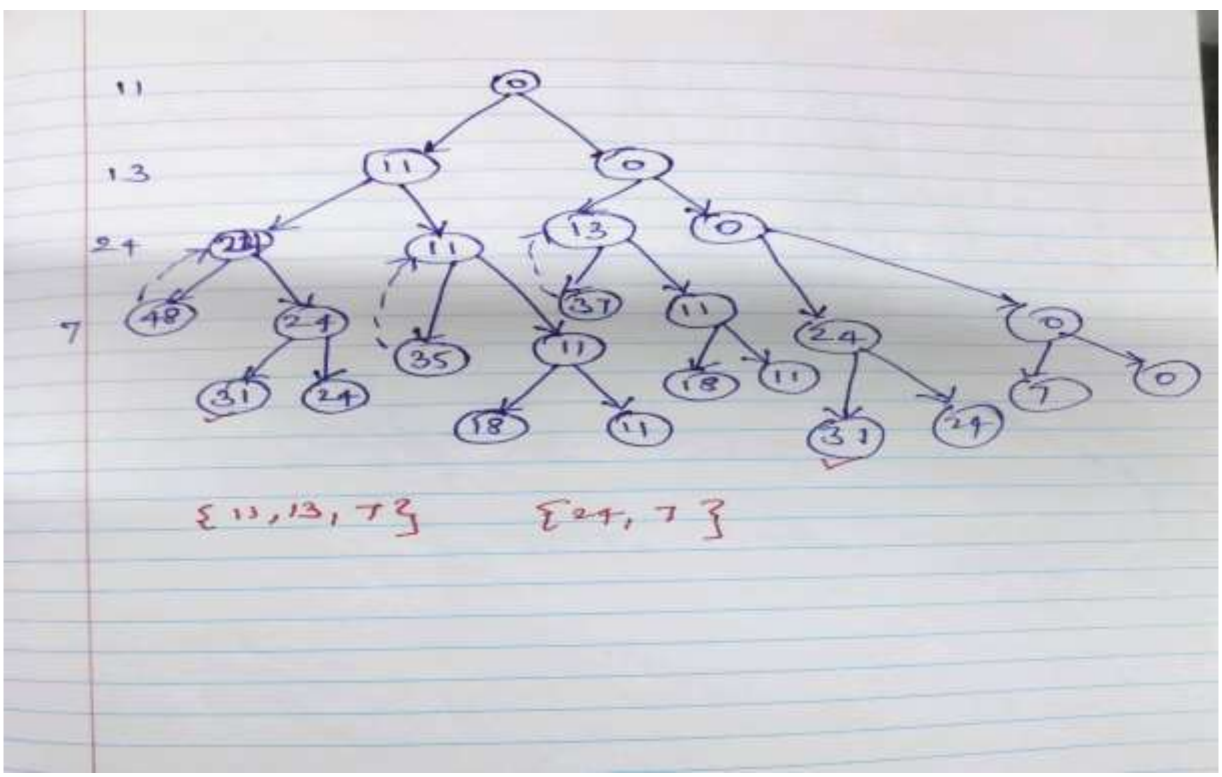
Solving Subset Sum with Backtracking

To solve the subset, sum problem using backtracking, we will follow a recursive approach. Here's an outline of the algorithm:

1. Sort the given set of integers in non-decreasing order.
2. Start with an empty subset and initialize the current sum as 0.
3. Iterate through each integer in the set:

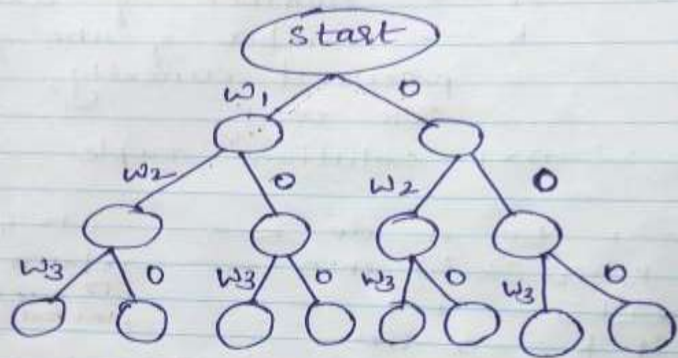
- Include the current integer in the subset.
- Increment the current sum by the value of the current integer.
- Recursively call the algorithm with the updated subset and current sum.
- If the current sum equals the target sum, we have found a valid subset.
- Backtrack by excluding the current integer from the subset.
- Decrement the current sum by the value of the current integer.

1. If we have exhausted all the integers and none of the subsets sum up to the target, we conclude that there is no valid subset.



State space tree for n = 3

* Any path from the root to leaf forms a subset.



Q. Solve the sum of subset problem using backtracking strategy for the following data $n = 4$
$W = (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$
and $M = 31$

Subset Items	condition	comment
{ }	0	Initial condition
{11}	11 < 31	Add next element
{11, 13}	24 < 31.	Add next element
{11, 13, 24}	48 < 31	Sub set exceeds sum so backtrack
{11, 13, 7}	31	Solution found

Implementation:

```c
#include <stdio.h>

#define MAX 100

// Function to find subsets with given sum using backtracking
void findSubsetSum(int set[], int n, int targetSum, int subset[], int
subsetSize, int sum, int index) {
    if (sum == targetSum) {
        // Print the subset
```

```
        printf("Subset with sum %d: ", targetSum);
        for (int i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    // If sum exceeds targetSum or index is out of bounds, return
    if (sum > targetSum || index >= n) {
        return;
    }

    // Include the current element in the subset
    subset[subsetSize] = set[index];
    // Recur with the next element and increase the sum
    findSubsetSum(set, n, targetSum, subset, subsetSize + 1, sum +
set[index], index + 1);

    // Exclude the current element from the subset and recur
    findSubsetSum(set, n, targetSum, subset, subsetSize, sum, index + 1);
}

int main() {
    int set[] = {10, 7, 5, 18, 12, 20, 15};
    int n = sizeof(set) / sizeof(set[0]);
    int targetSum = 35;
    int subset[MAX];

    printf("Subsets with sum %d:\n", targetSum);
    findSubsetSum(set, n, targetSum, subset, 0, 0, 0);

    return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SEARCH ERROR

PS C:\TURBOC3\BIN> cd "c:\TURBOC3\BIN\" ; if ($?) { gcc TRAVELLING.C -o TRAVELLING } ; if ($?) { .\TRAVELLING }
The shortest path length is 80
PS C:\TURBOC3\BIN> cd "c:\TURBOC3\BIN\" ; if ($?) { gcc subsets.c -o subsets } ; if ($?) { .\subsets }
Subsets with sum 35:
Subset with sum 35: 10 7 18
Subset with sum 35: 10 5 20
Subset with sum 35: 5 18 12
Subset with sum 35: 20 15
PS C:\TURBOC3\BIN>
```

Conclusion: In conclusion, we have presented a C program that utilizes backtracking to find all subsets of a given set whose sum is equal to a specified target sum. The program efficiently explores the solution space by including and excluding elements from the set, ensuring that all possible combinations are considered. By employing backtracking, the program achieves a time complexity that is proportional to the number of valid subsets. This solution provides a practical implementation for solving the subset sum problem, demonstrating the effectiveness of backtracking algorithms in solving combinatorial optimization tasks.