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.

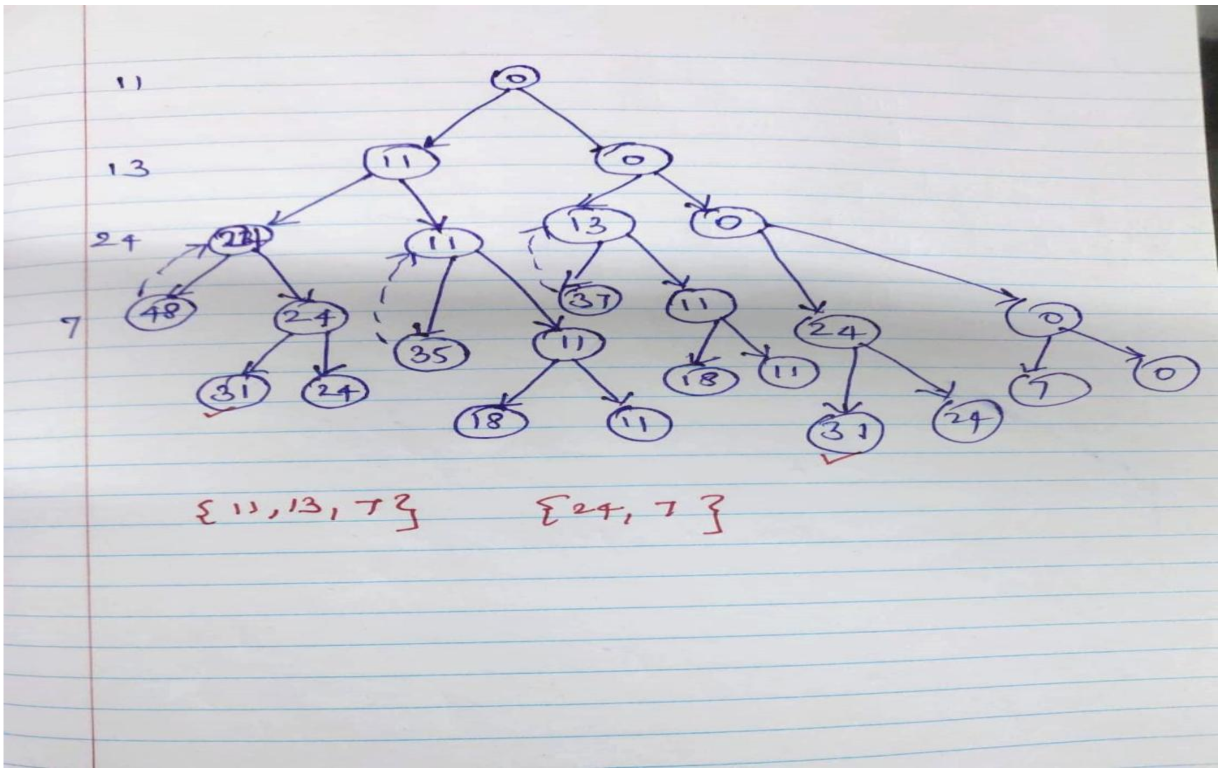
Solving Subset Sum with Backtracking

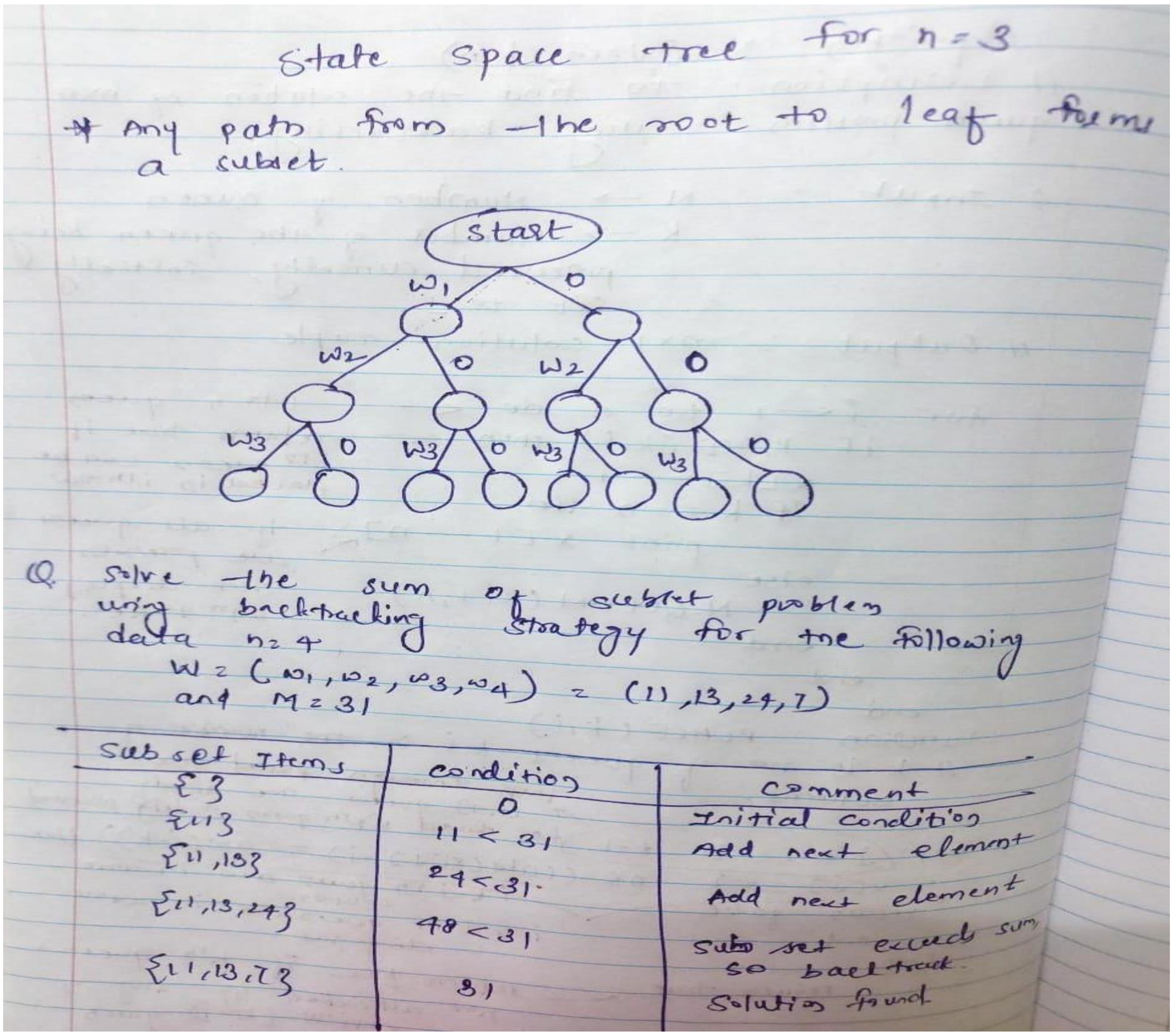
To solve the subset, sum problem using backtracking, we will follow a recursive approach. Here's an outline of the algorithm:

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.

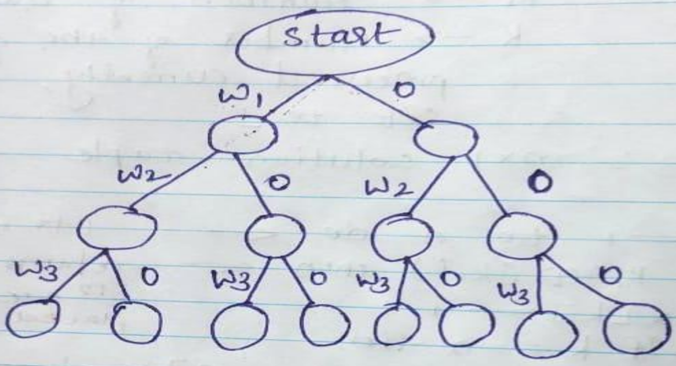
1. If we have exhausted all the integers and none of the subsets sum up to the target, we conclude that there is no valid subset.





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

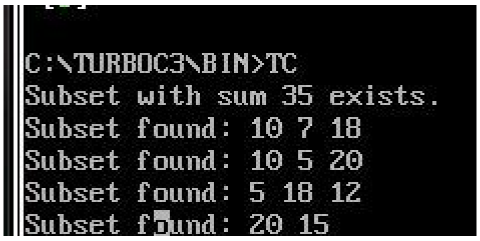
Subset Items	condition	comment
{ }	0	Initial condition
{11}	11 < 31	Add next element
{11, 13}	24 < 31	Add next element
{11, 13, 24}	48 < 31	Sub set exceeds sum so backtrack
{11, 13, 7}	31	Solution found

Implementation:

```
#include <stdio.h>

#define MAX_SIZE 100

// Function to check if there is a subset with given sum
int isSubsetSum(int set[], int n, int sum) {
int i;    // Base Cases
    if (sum == 0)
    return 1;
    if (n == 0 && sum != 0)
    return 0;
```

```c
    // If last element is greater than sum, then ignore it
    if (set[n - 1] > sum)
    return isSubsetSum(set, n - 1, sum);

    // Check if sum can be obtained by including the last element or
excluding it
    return isSubsetSum(set, n - 1, sum) || isSubsetSum(set, n - 1, sum -
set[n - 1]);
}

// Function to find subsets with the given sum
void findSubsets(int set[], int n, int sum, int subset[], int
subsetSize, int idx) {
int i;
    if (sum == 0) {
    // Print the subset
    printf("Subset found: ");
    for (i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    if (idx == n)
        return;

    // Include the current element
    subset[subsetSize] = set[idx];
    findSubsets(set, n, sum - set[idx], subset, subsetSize + 1, idx + 1);

    // Exclude the current element
    findSubsets(set, n, sum, subset, subsetSize, idx + 1);
}

int main() {
    int set[] = {10, 7, 5, 18, 12, 20, 15};
    int n = sizeof(set) / sizeof(set[0]);
    int sum = 35;
    int subset[MAX_SIZE];
    if (isSubsetSum(set, n, sum)) {
    printf("Subset with sum %d exists.\n", sum);
    findSubsets(set, n, sum, subset, 0, 0);
    } else {
        printf("No subset with sum %d exists.\n", sum);
    }
    return 0;
}
```

Output:



```
C:\TURBOC3\BIN>TC
Subset with sum 35 exists.
Subset found: 10 7 18
Subset found: 10 5 20
Subset found: 5 18 12
Subset found: 20 15
```

Conclusion: The implemented backtracking solution effectively determined whether a subset with a specified sum exists within a given set. It showcased the versatility of backtracking algorithms in solving combinatorial optimization problems like the Subset Sum Problem efficiently. This approach provides a foundational method for addressing similar challenges with varying constraints or objectives.