Design and Analysis of Algorithms

- ***Course Code***: BCSE204L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 2

Module:2	Design Paradigms: Dynamic Programming, Backtracking and Branch & Bound Techniques	10 hours

Dynamic programming: Assembly Line Scheduling, Matrix Chain Multiplication, Longest Common Subsequence, 0-1 Knapsack, TSP

Backtracking: N-Queens problem, Subset Sum, Graph Coloring

Branch & Bound: LIFO-BB and FIFO BB methods: Job Selection problem, 0-1 Knapsack Problem

Backtracking

Backtracking is a general algorithmic technique used for solving optimization and decision problems. The idea is to explore all possible solutions to a problem systematically, but in a way that avoids unnecessary work by abandoning a partial solution as soon as it is determined that the solution cannot be extended to a valid one.

Detailed breakdown of the backtracking process:

- **Choose**: Make a choice for the next decision point in the solution.
- **Explore:** Recursively explore all possible choices that can be made from the current decision point.
- **Backtrack**: If a choice leads to an invalid or unsuccessful solution, undo that choice and go back to the previous decision point to explore other alternatives.

This process continues until a valid solution is found, or it is determined that no valid solution exists.

Backtracking is often used in combination with constraint satisfaction problems, where decisions must satisfy a set of constraints.

Applications of Backtracking:

Common examples of problems that can be solved using backtracking include

- N Queens Problem
- Sum of subsets problem
- Graph colouring
- Bin packing
- Hamiltonian cycles.

***Note*: State-space tree** *is the tree with the root node of no component solutions and from that construction of partial solutions, which are successive configurations or states of an instance.*

State-space tree in Backtracking:

- In the context of backtracking, the state-space tree represents the exploration of possible solutions to a problem in a structured way.
- It is a tree structure where each node corresponds to a specific state or configuration in the process of finding a solution.
  - The root of the tree typically represents the initial state or the starting point of the problem,
  - Branches emanating from each node represent the choices or decisions made at that particular state.

Sum of subsets problem

- **Definition**:
  - The Sum of Subsets problem is a classic combinatorial optimization problem where the goal is to find all possible subsets of a given set whose sum is equal to a specified target sum.

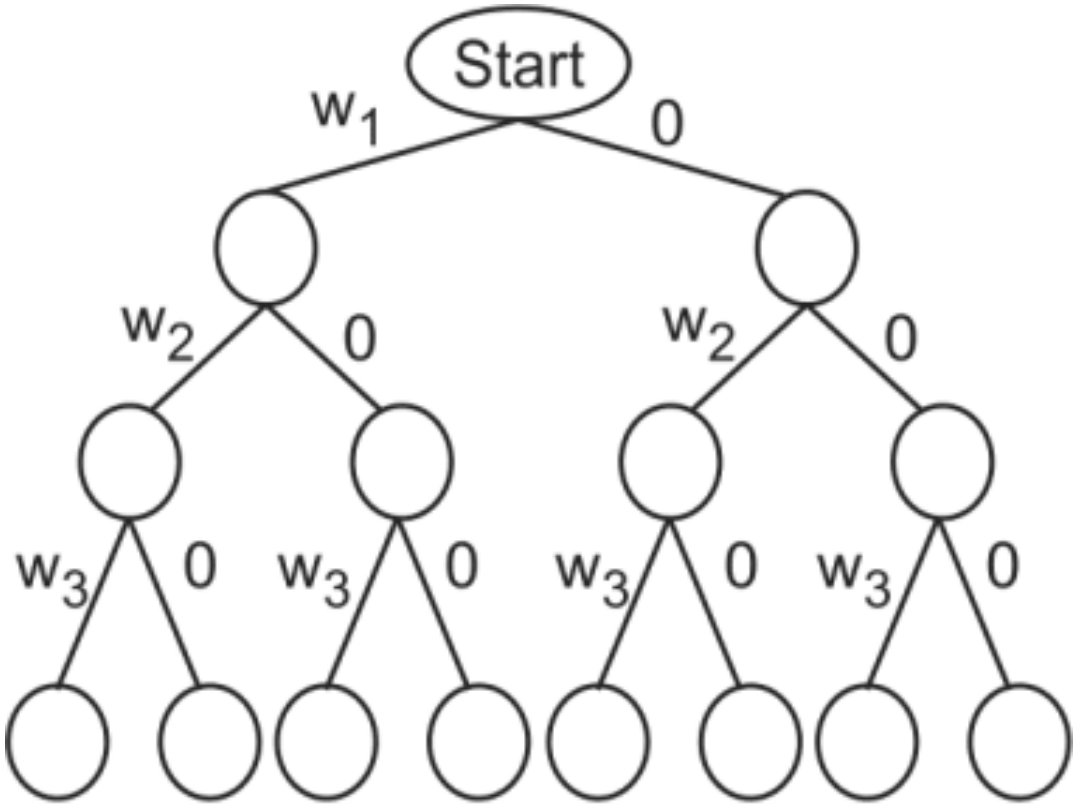
    Simply: Given a set of positive integers and a target sum, find all possible combinations of elements from the set that add up to the given sum.

Sample Input and Output:
For example, given the set {1, 2, 3, 4, 5} and the target sum of **8**,
a solution might include subsets like {3, 5} or {1, 2, 5} because their sums equal the target sum.

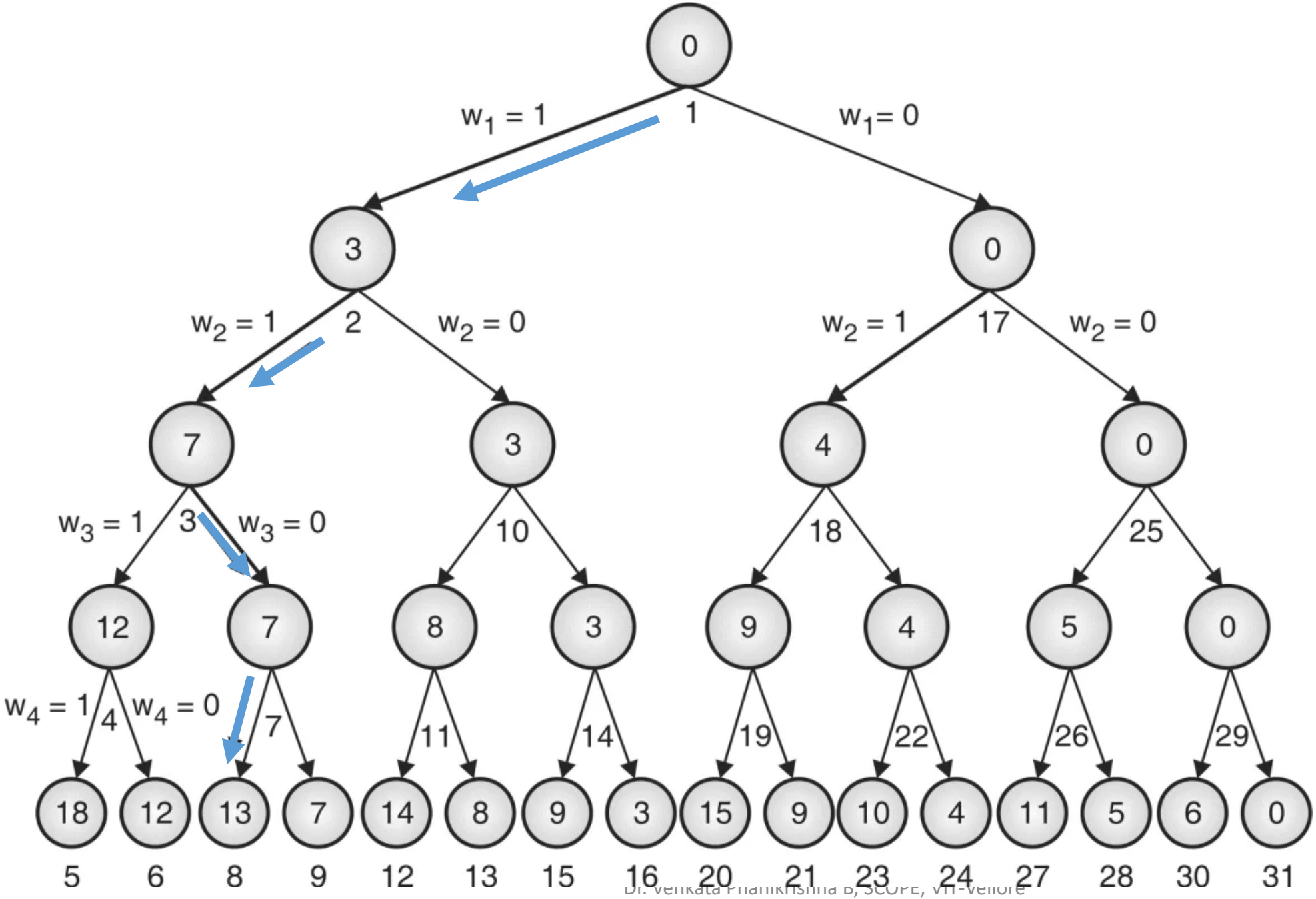
Time Complexity in Brute Force Approach:

The brute force approach involves checking all possible subsets, resulting in a time complexity of O(2^N), where N is the number of elements in the set.

This is because, for each element, you have two choices: include or exclude.



State-space tree



Sum of M = 13 for given
W = {3, 4, 5, 6}

The solution vector for [3, 4, 6]

would be
X = [1, 1, 0, 1]

Sum of subsets problem using Backtracking

- In the context of backtracking, the exploration of the solution space for the Subset Sum problem is guided by equations that efficiently prune the search space and identify valid solutions.
- The Subset Sum problem entails finding all subsets of a given set whose sums equal a specified target sum M.

  - W_i represents the weight of item i.
  - M denotes the capacity of the bag (subset).
  - X_i is an element of the solution vector, taking values of either one or zero.
    - If $X_i = 1$, it signifies that W_i is chosen;
    - if $X_i = 0$, W_i is not chosen.

Sum of subsets problem using Backtracking

- W_i represents the weight of item i.
- M denotes the capacity of the bag (subset).
- X_i is an element of the solution vector, taking values of either one or zero.
  - If $X_i = 1$, it signifies that W_i is chosen;
  - if $X_i = 0$, W_i is not chosen.

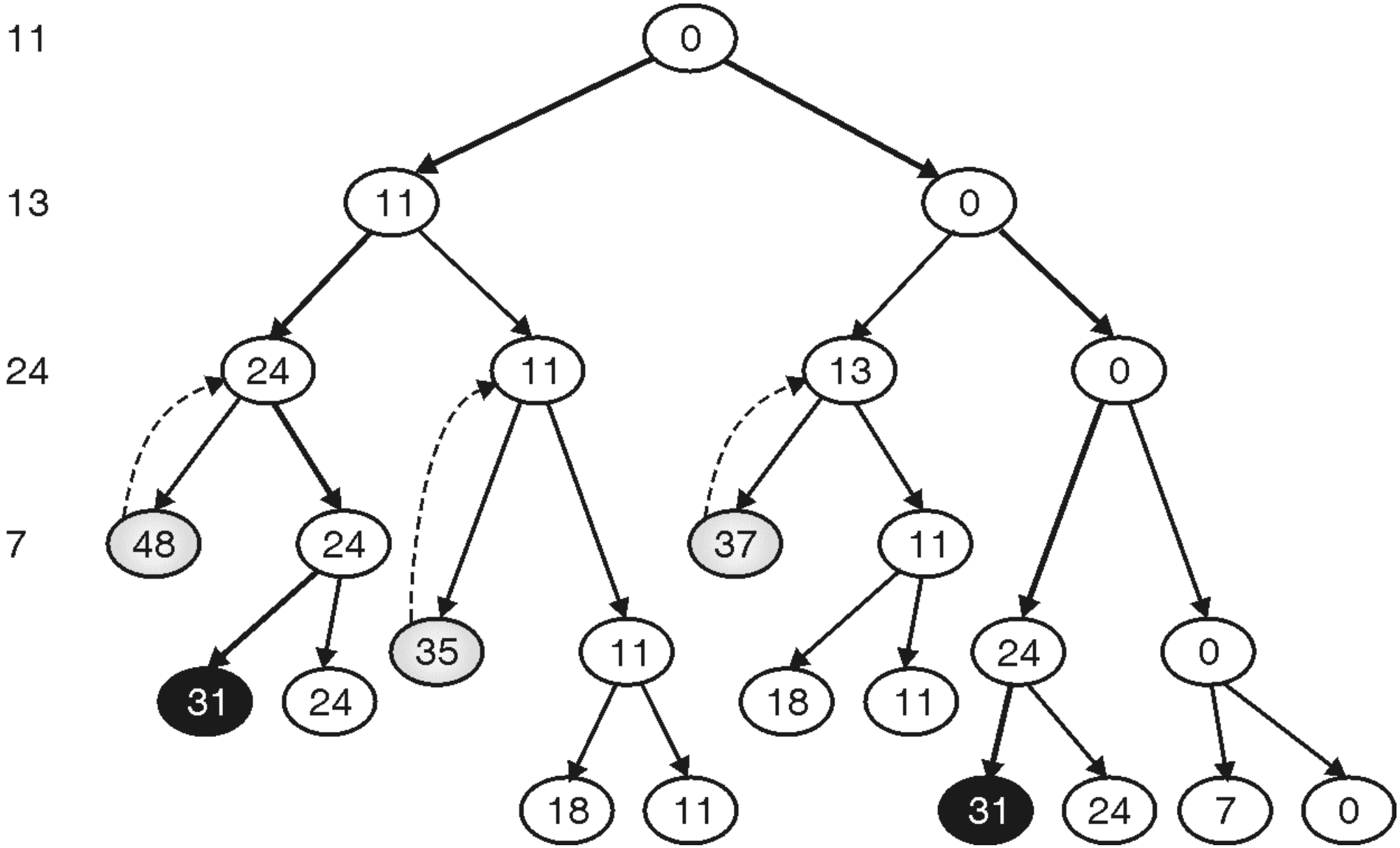
$$= \sum_{k=0}^{n} W_i \cdot X_i + W_{k+1} > M$$

This equation serves as a pruning criterion, stating that if the sum of weights chosen in the current subset, Denoted by

$$\sum_{k=0}^{n} W_i \cdot X_i$$

X_i, along with the weight of the next item W_{k+1}, exceeds the target sum M , then including the next item would already surpass the target sum. Consequently, there is no need to explore further with the inclusion of the next item, and backtracking can be applied for optimization.

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



M = 35 and

w = {5, 7, 10, 12, 15, 18, 20}

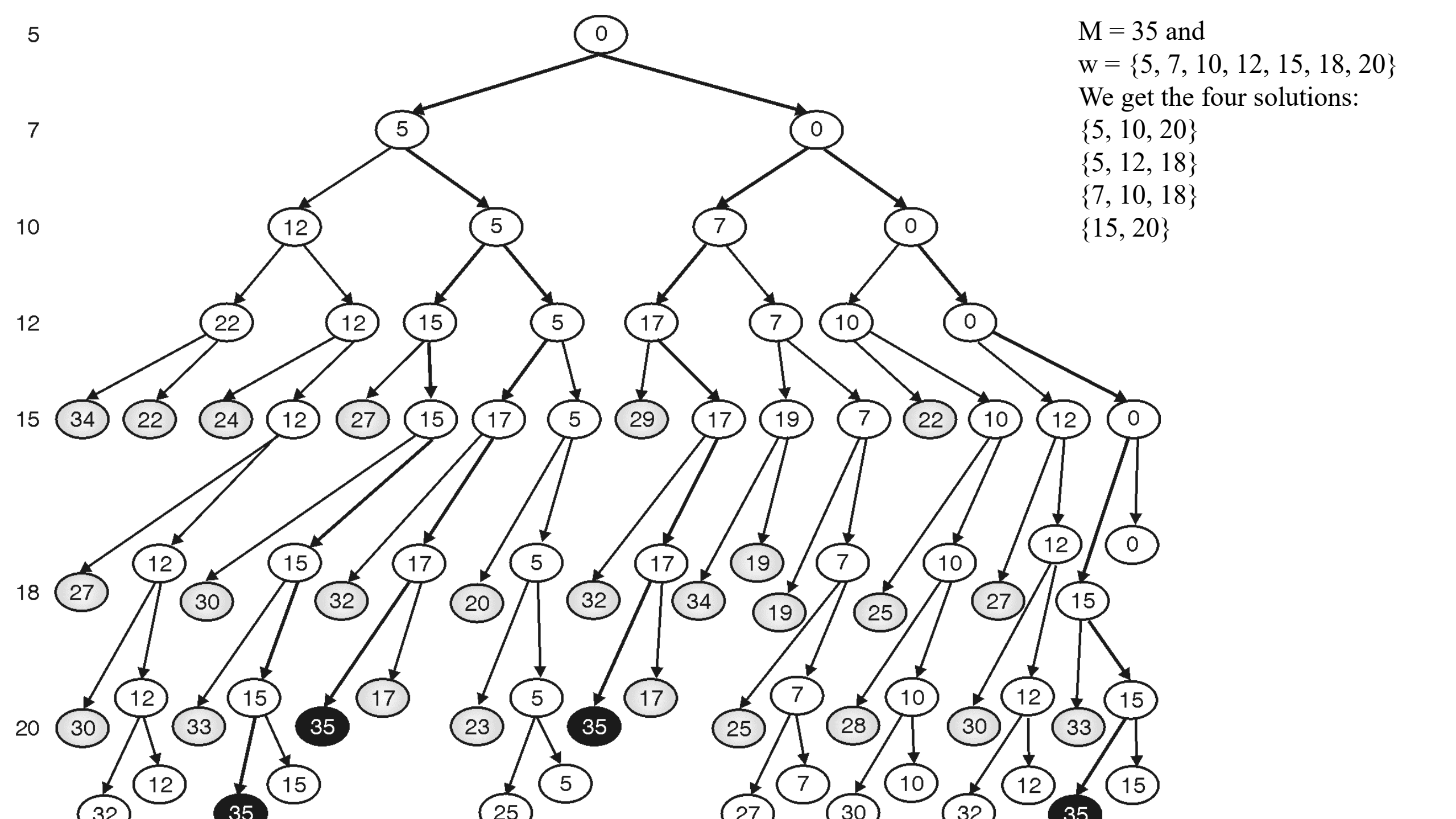
We get the four solutions:

{5, 10, 20}

{5, 12, 18}

{7, 10, 18}

{15, 20}


M = 35 and
w = {5, 7, 10, 12, 15, 18, 20}
We get the four solutions:
{5, 10, 20}
{5, 12, 18}
{7, 10, 18}
{15, 20}

```
Algorithm subsetSumBacktrack(index, currentSubset, weights, targetSum, result):
    if sum(currentSubset) == targetSum:
        result.append(copy_of_currentSubset)

    if index == len(weights):
        return

    # Include the current weight in the subset
    currentSubset.append(weights[index])
    subsetSumBacktrack(index + 1, currentSubset, weights, targetSum, result)
    currentSubset.pop()  # Backtrack

    # Exclude the current weight from the subset
    subsetSumBacktrack(index + 1, currentSubset, weights, targetSum, result)

Algorithm subsetSum(weights, targetSum):
    result = []
    subsetSumBacktrack(0, [], weights, targetSum, result)
    return result
```

Time Complexity in Backtracking

- Backtracking allows for pruning the search space by avoiding unnecessary explorations. The time complexity is better than the brute force approach but still exponential, often depending on the structure of the problem instance. In practice, backtracking can significantly reduce the number of subsets to explore.

Recurrence Relation:

The recurrence relation for the time complexity in backtracking can be expressed as follows:

- $T(N)=2 \cdot T(N-1)$

Any Question?

Thank You!

FOR YOUR ATTENTION