Design and Analysis of Algorithms (CSC-314)

Department of B.Sc. CSIT

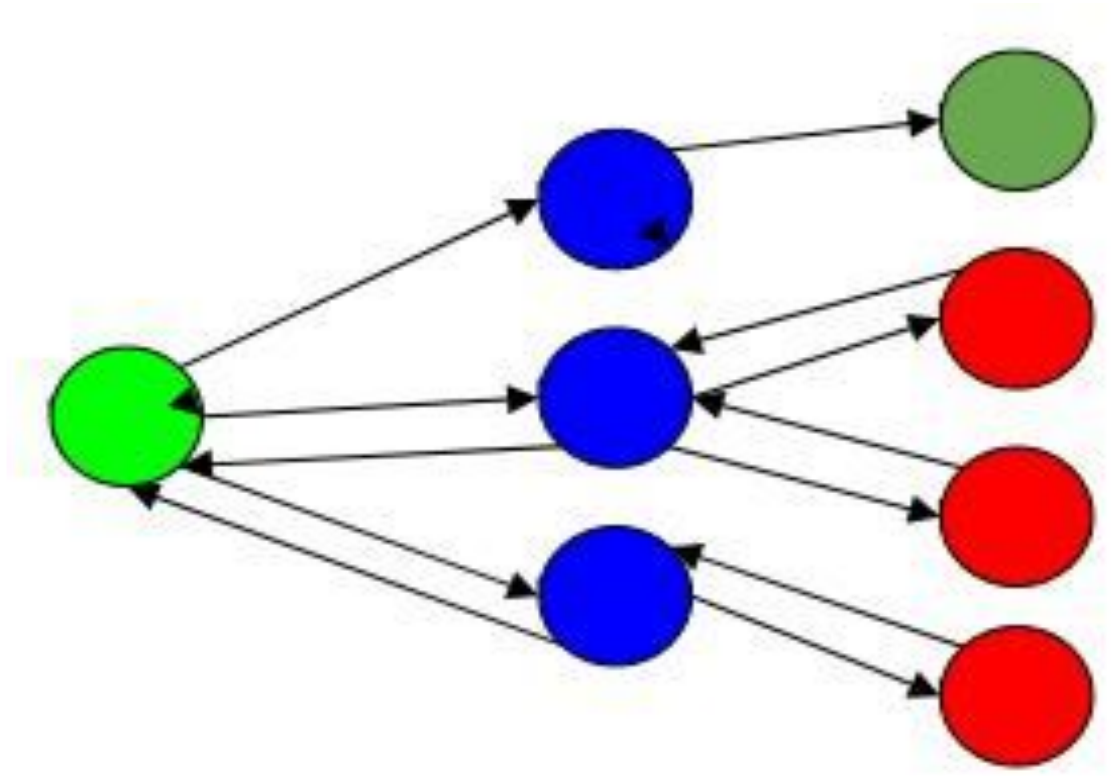
Godawari College

Unit-6: Backtracking

- **Introduction**
- The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- This approach is used to solve problems that have multiple solutions. If you want an optimal solution, you must go for dynamic programming.
- Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach.
- It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.

Unit-6: Backtracking

- **Introduction**
- Example:



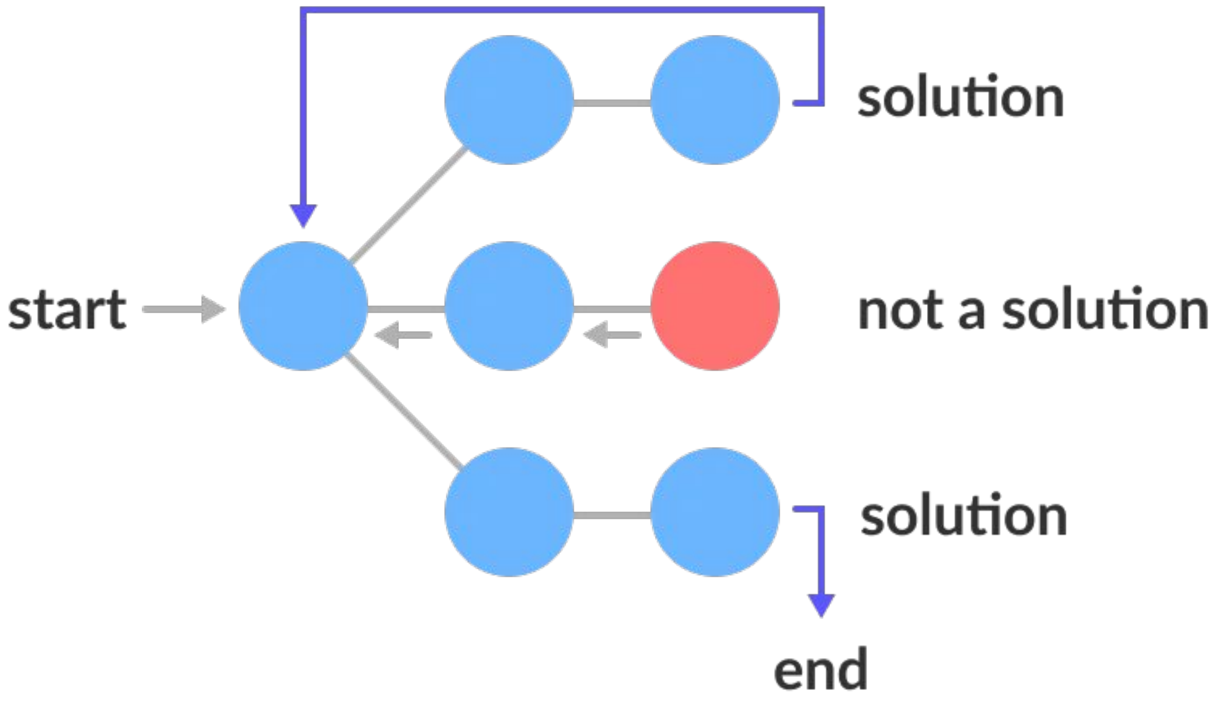
- Here,
- Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution.
- Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution

Unit-6: Backtracking

- **Introduction**
- There are three types of problems in backtracking –
  - Decision Problem – In this, we search for a feasible solution.
  - Optimization Problem – In this, we search for the best solution.
  - Enumeration Problem – In this, we find all feasible solutions.
- In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Unit-6: Backtracking

- **State Space Tree**
- A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.



Unit-6: Backtracking

Backtracking Algorithm

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

Unit-6: Backtracking

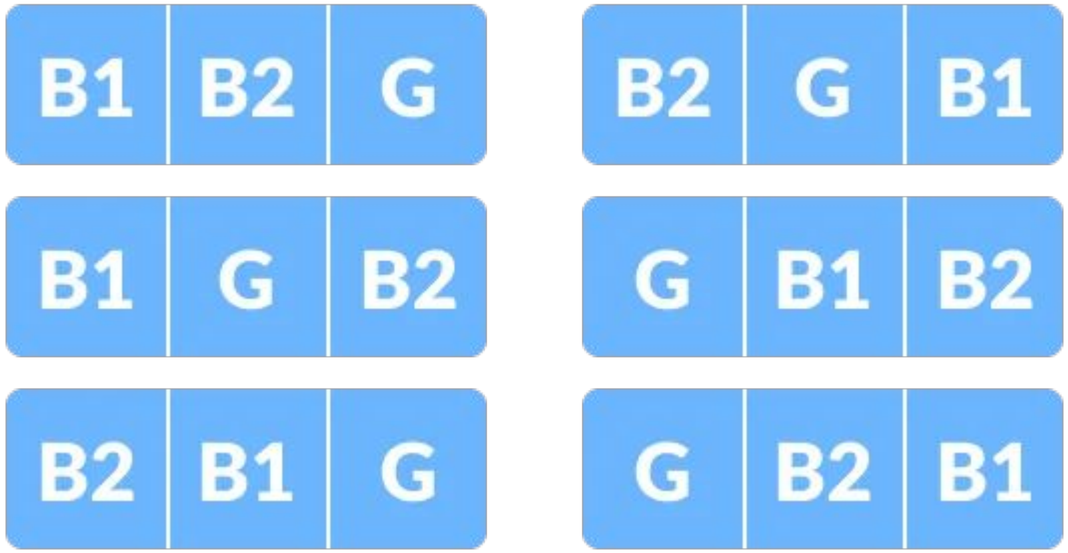
- **Example Backtracking Approach**

- **Problem:** You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle of the bench.

- ?

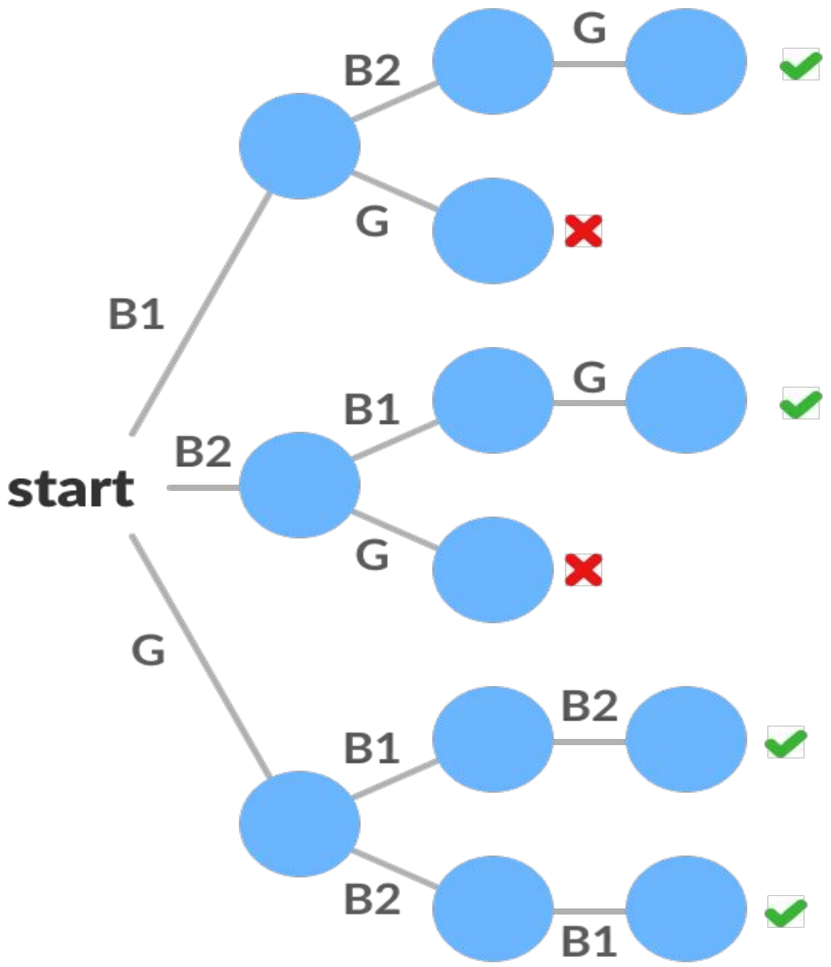
Unit-6: Backtracking

- **Example Backtracking Approach**
- **Solution:** There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.
- All the possibilities are:



Unit-6: Backtracking

- **Example Backtracking Approach**
- The following state space tree shows the possible solutions.



Unit-6: Backtracking

- **Recursion vs Backtracking**
- In recursion, the function calls itself until it reaches a base case.
- In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Unit-6: Backtracking

- **Concept of Subset Sum Algorithm:**

- Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number.
- The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.
- A subset A of n positive integers and a value sum is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

Unit-6: Backtracking

- **Concept of Subset Sum Algorithm:**
- **Example:**
- Given the following set of positive numbers:
- { 2, 9, 10, 1, 99, 3}
- We need to find if there is a subset for a given sum say 4:
- { 1, 3 }
- For another value say 5, there is another subset:
- { 2, 3}
- Similarly, for 6, we have {2, 1, 3} as the subset.
- For 7, there is no subset where the sum of elements equal to 7.

Unit-6: Backtracking

- **Concept of Subset Sum Algorithm:**
- Start with an empty set
- Add the next element from the list to the set
- If the subset is having sum M, then stop with that subset as solution.
- If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
- If the subset is feasible (sum of subset < M) then go to step 2.
- If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Concept of Subset Sum Algorithm:Example

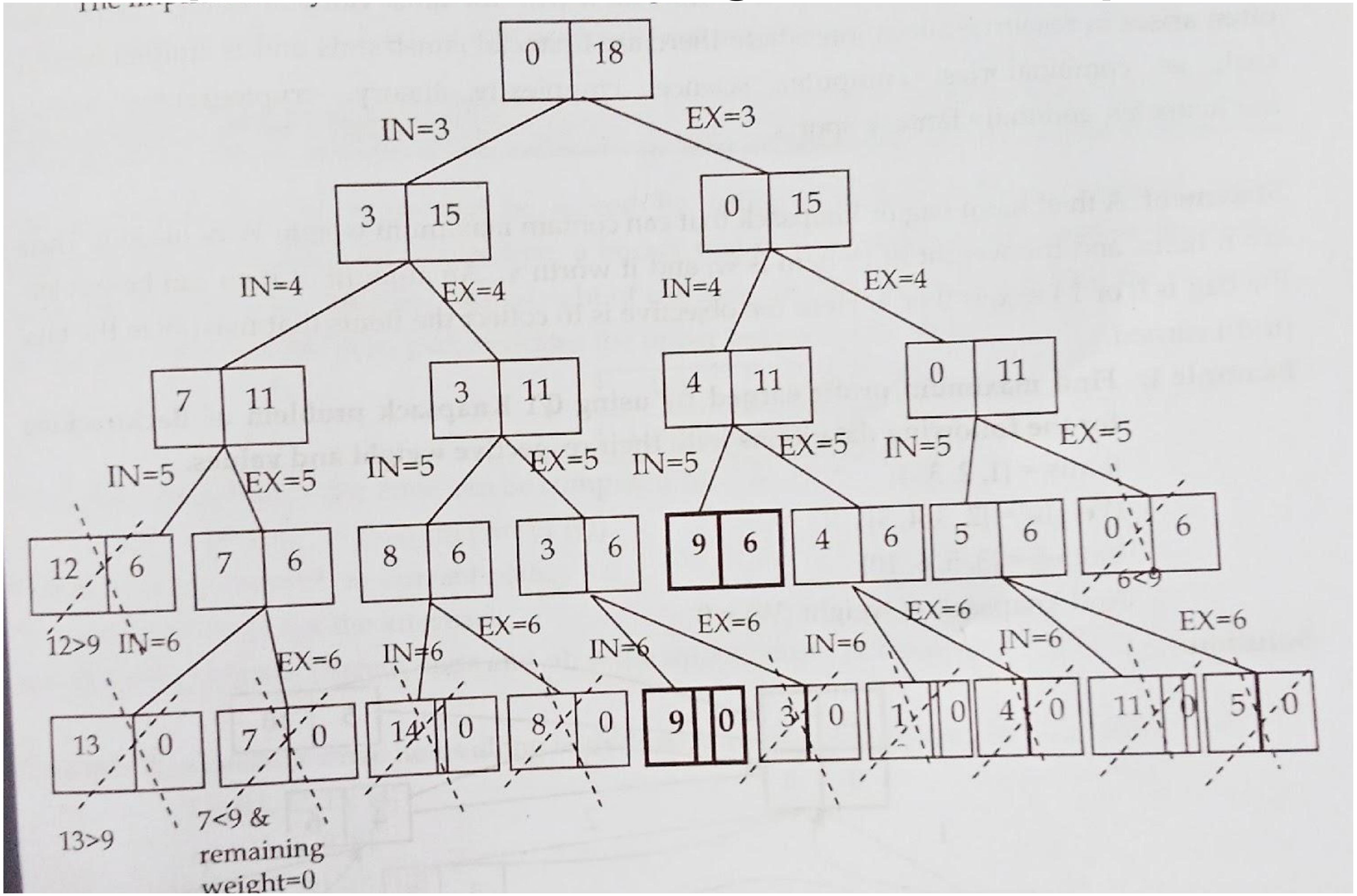
- Consider the following array/ list of integers: {1, 3, 2}
- We want to find if there is a subset with sum 3.
- Note that there are two such subsets of feasible solutions {1, 2} and {3}. We will follow our backtracking approach.
  - Consider our empty set {}
  - We add 1 to it {1} (sum = 1, 1 < 3)
  - We add 3 to it {1, 3} (sum = 4, 4>3, not found)
  - We remove 3 from it {1} (sum = 1, 1 < 3)
  - We add 2 to it {1, 2} (sum = 3, 3 == 3, found)

Concept of Subset Sum Algorithm:Example

- Given a set s={4,5,6,3} and X= 9. obtain the solutions subset using backtracking approach.
- Solution:
  - Initially, we arrange elements in given sets in increasing order.
  - Like: s={3,4,5,6}
  - X=9 (i.e. sum of subset, problem constraints in our case)
  - S'={ } // empty initially

Unit-6: Backtracking

Concept of Subset Sum Algorithm:Example



Unit-6: Backtracking

- **Concept of Subset Sum Algorithm:Example (try yourself)**
- Given a set s={5,10,12,13,15,18} and X= 30. obtain the solutions subset using backtracking approach.

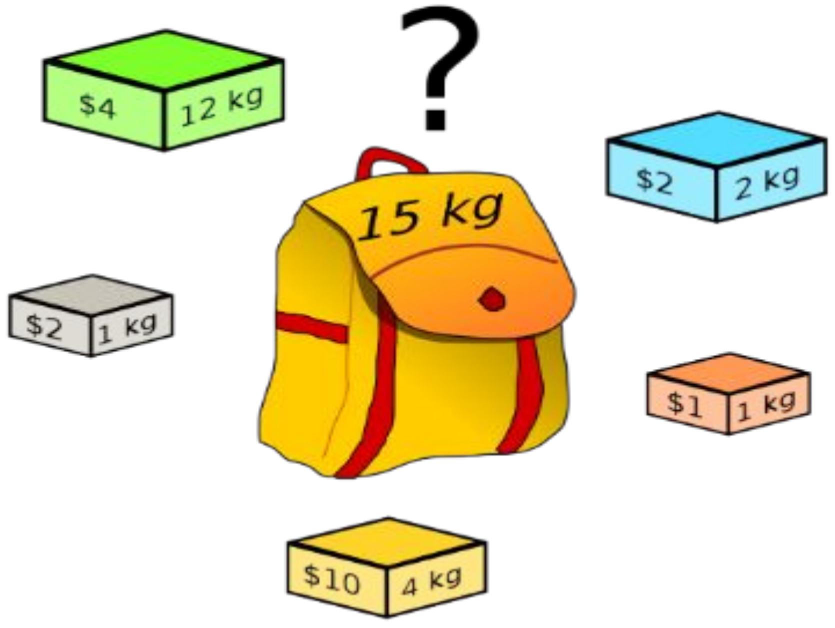
Unit-6: Backtracking

Concept of Subset Sum Algorithm:

- Complexity
  - Worst case time complexity: Θ(2^n)
  - Space complexity: Θ(1)

Unit-6: Backtracking

- **0/1 Knapsack Problem:**
- The problem states-
  - Which items should be placed into the knapsack such that-
  - The value or profit obtained by putting the items into the knapsack is maximum.
  - And the weight limit of the knapsack does not exceed.



Knapsack Problem

Unit-6: Backtracking

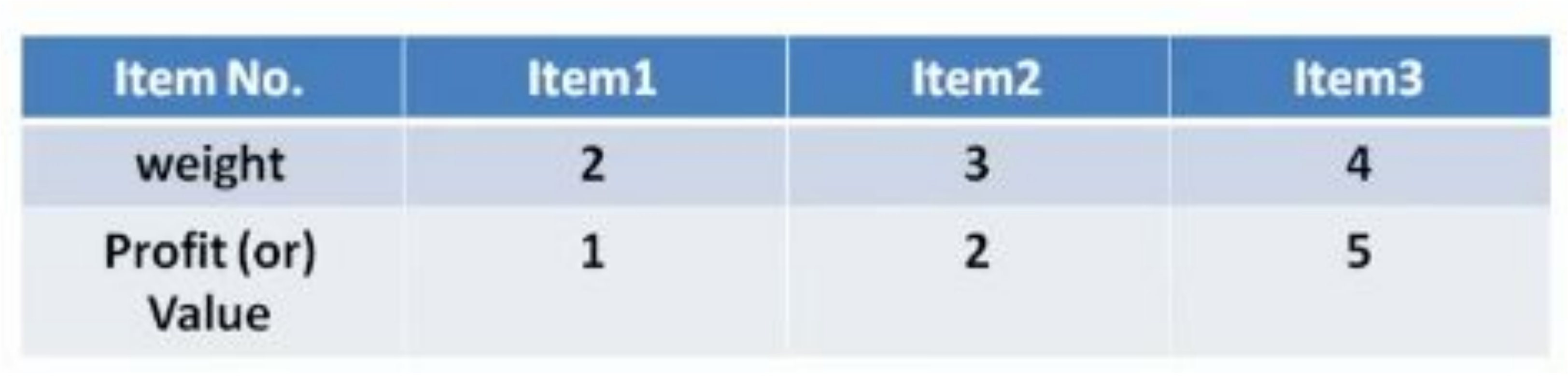
- **0/1 Knapsack Problem:**
- In 0/1 Knapsack Problem,
  - As the name suggests, items are indivisible here.
  - We can not take the fraction of any item.
  - We have to either take an item completely or leave it completely.
  - It is solved using dynamic programming approach.
  - **Statement:** A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is Wi and it worth Vi . An amount of item can be put into the bag is 0 or 1 i.e. x i is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.

Unit-6: Backtracking

- **0/1 Knapsack Problem:Solution steps**
- 1) For the given problem, define the solution space of the problem;
- 2) Determine the solution space structure that is easy to search;
- 3) Search the solution space in a depth-first manner, and use a pruning function to avoid invalid searches during the search.

Unit-6: Backtracking

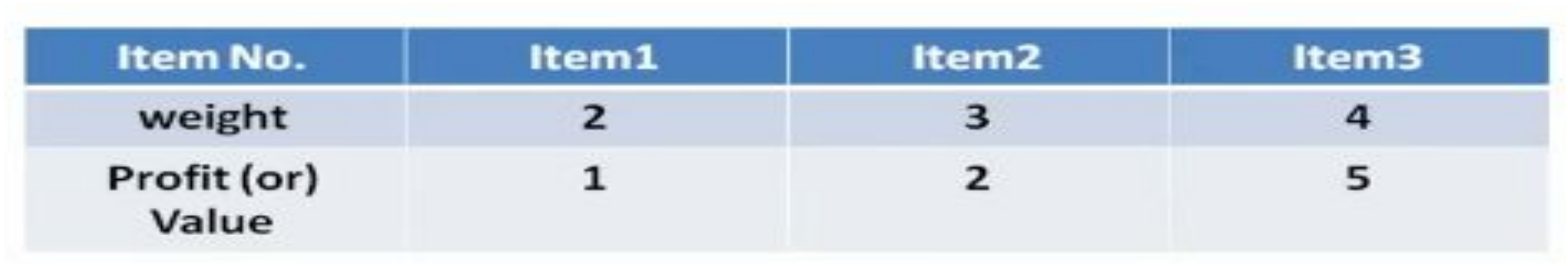
- **0/1 Knapsack Problem:Example**
- **Find the maximum profit earned by using 0/1 knapsack problem using backtracking approach. And the knapsack weight is 6.**



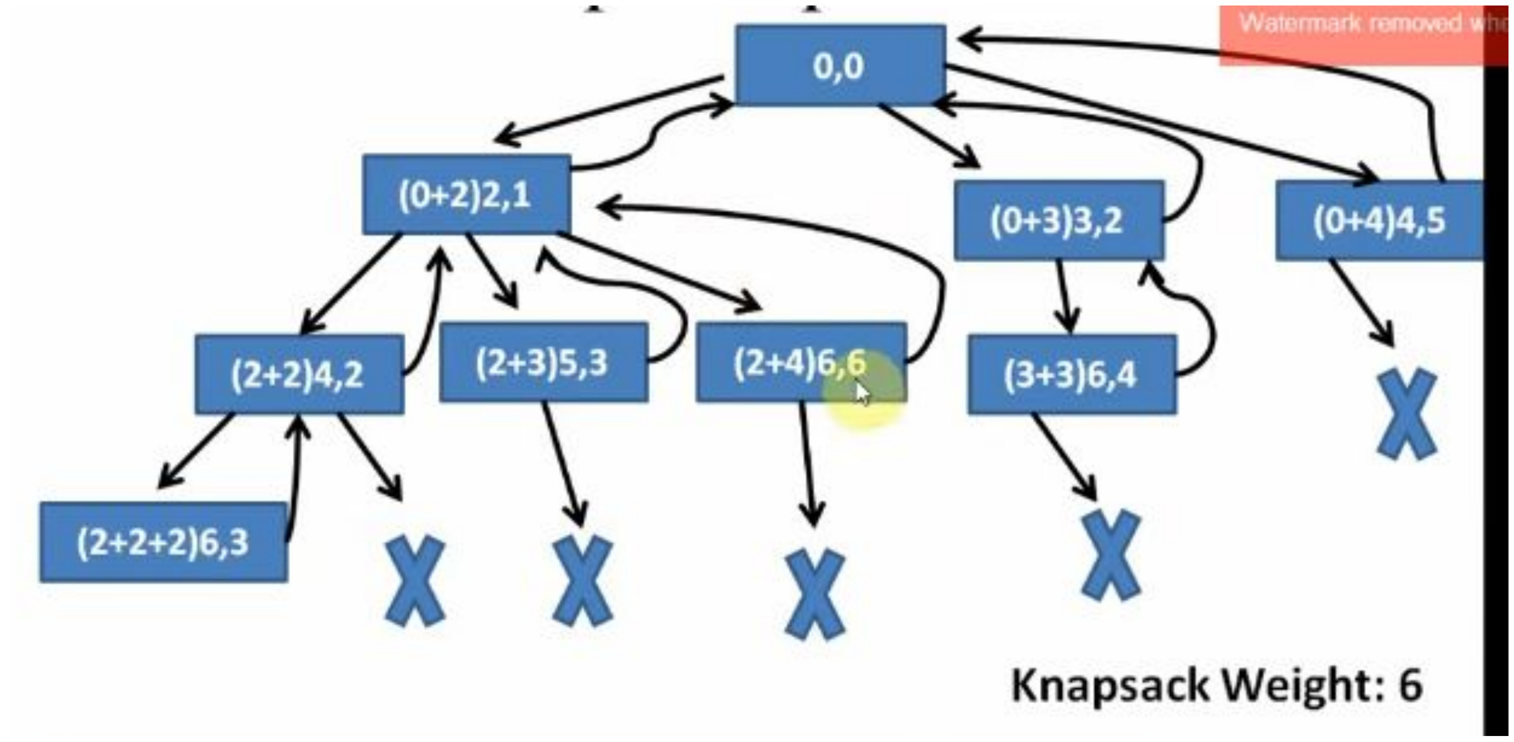
Item No.	Item1	Item2	Item3
weight	2	3	4
Profit (or) Value	1	2	5

Unit-6: Backtracking

0/1 Knapsack Problem:Example



Item No.	Item1	Item2	Item3
weight	2	3	4
Profit (or) Value	1	2	5



Unit-6: Backtracking

- **0/1 Knapsack Problem:Example (try yourself)**
- Find the maximum profit earned by using 0/1 knapsack problem using backtracking approach. And the knapsack weight is 8.
  - Weight ={2,3,4,5}
  - Values ={3,5,6,10}

Unit-6: Backtracking

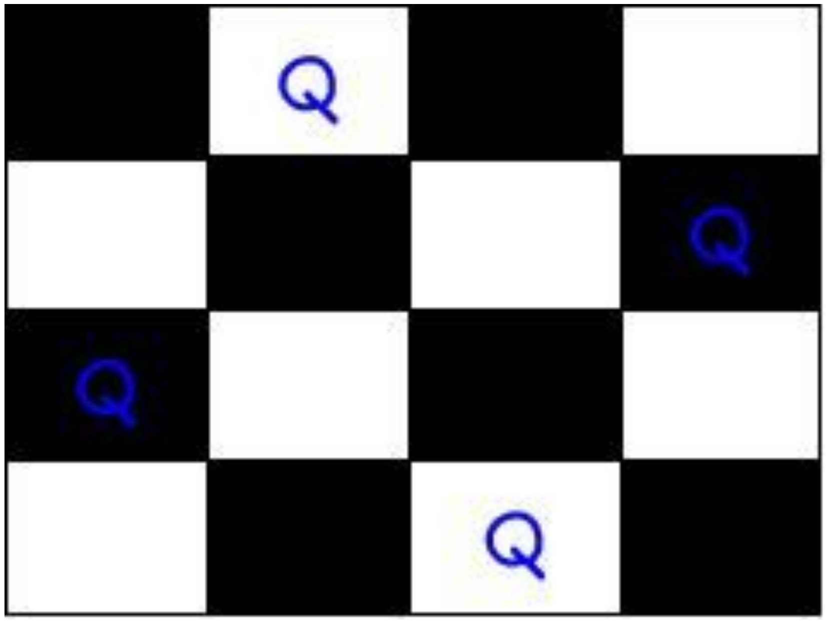
- **N-Queen Problem**
- A classic example of backtracking is the n-Queens problem, first proposed by German chess enthusiast Max Bezzel in 1848. Given a chessboard of size n, the problem is to place n queens on the n*n chessboard, so no two queens are attacking each other.
- It is clear that for this problem, we need to find all the arrangements of the positions of the n queens on the chessboard, but there is a constraint: no queen should be able to attack another queen.
- A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way.

- Unit-6: Backtracking

- **N-Queen Problem**
- 1) Start in the leftmost column
- 2) If all queens are placed
- return true
- 3) Try all rows in the current column. Do following for every tried row.
- a) If the queen can be placed safely in this row then mark this [row,
- column] as part of the solution and recursively check if placing
- queen here leads to a solution.
- b) If placing the queen in [row, column] leads to a solution then return

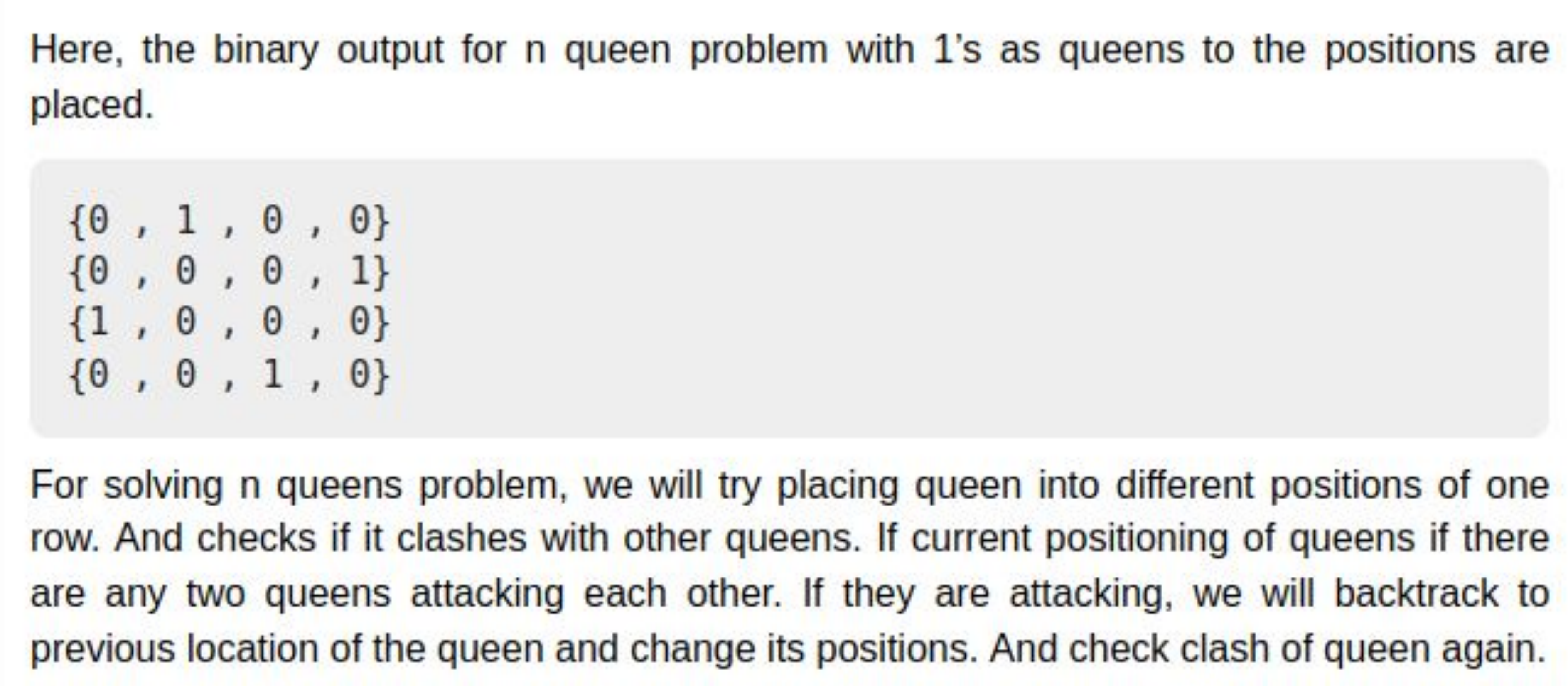
Unit-6: Backtracking

- **N-Queen Problem: Example**
- Here, we will do 4-Queen problem.
- Solution:



Unit-6: Backtracking

- **N-Queen Problem: Example**
- Here, we will do 4-Queen problem.
- Solution:



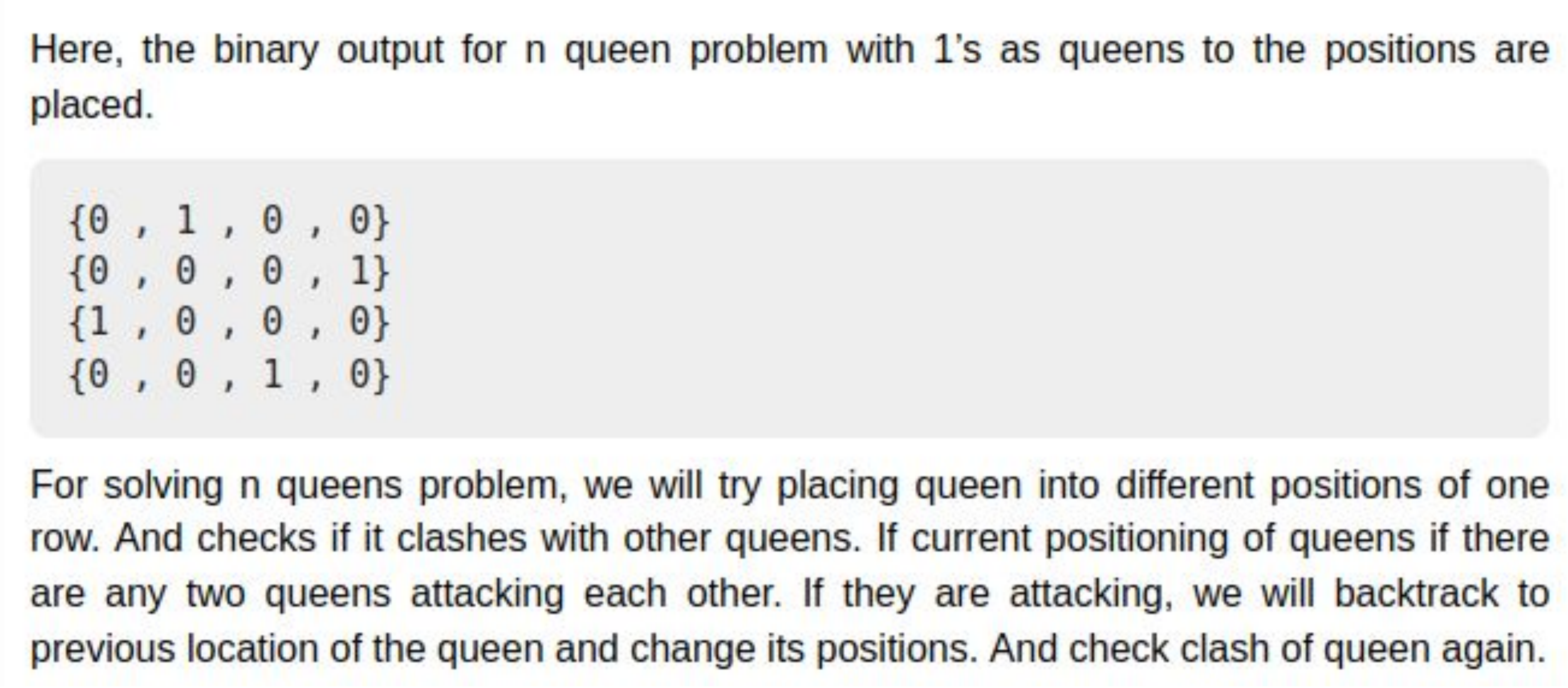
Here, the binary output for n queen problem with 1's as queens to the positions are placed.

```
{0 , 1 , 0 , 0}
{0 , 0 , 0 , 1}
{1 , 0 , 0 , 0}
{0 , 0 , 1 , 0}
```

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

Unit-6: Backtracking

- **N-Queen Problem: Example**
- Here, we will do 4-Queen problem.
- Solution:



Here, the binary output for n queen problem with 1's as queens to the positions are placed.

```
{0 , 1 , 0 , 0}
{0 , 0 , 0 , 1}
{1 , 0 , 0 , 0}
{0 , 0 , 1 , 0}
```

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

Unit-6: Backtracking

Thank You!