Session 9

Week -4

Understanding the Game First

https://www.mathsisfun.com/games/towerofhanoi.html

```
void towerOfHanoi(int n, char from_rod,
    char to_rod, char aux_rod)
{
    if (n == 1)
    {
        cout << "Move disk 1 from rod " << from_rod <<
            " to rod " << to_rod << endl;
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod <<
        " to rod " << to_rod << endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

According to the wiki definition,

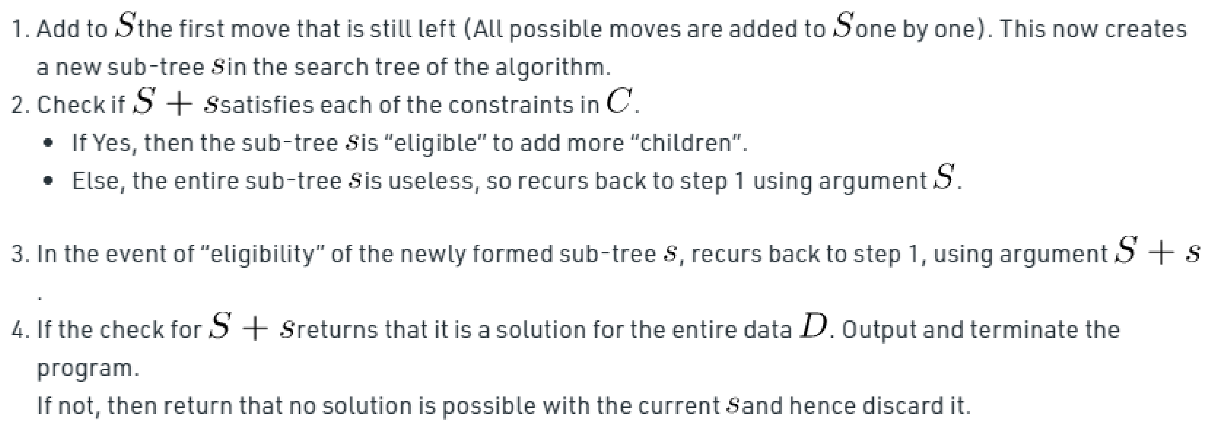
Backtracking *can be defined as a general algorithmic technique that considers searching ever y possible combination in order to solve a computational problem.*

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

How to determine if a problem can be solved using Backtracking?

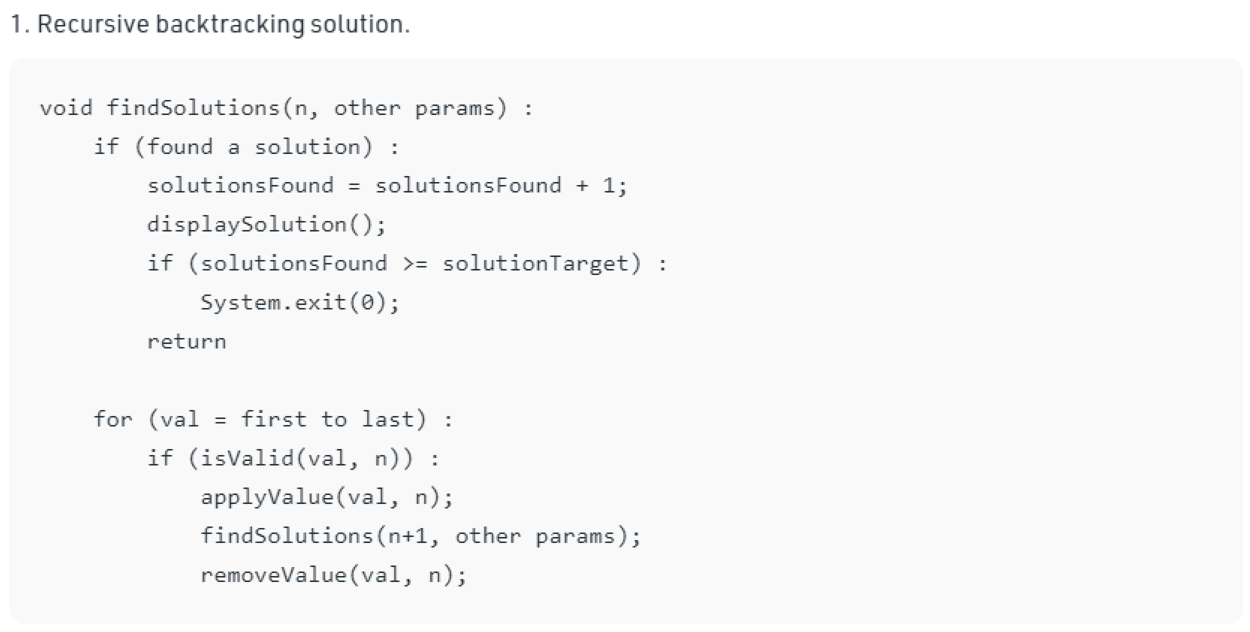
Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem and data corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by . A backtracking algorithm will then work as follows:



1. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree s in the search tree of the algorithm.
2. Check if $S + s$ satisfies each of the constraints in C.
   - If Yes, then the sub-tree s is "eligible" to add more "children".
   - Else, the entire sub-tree s is useless, so recurs back to step 1 using argument S.

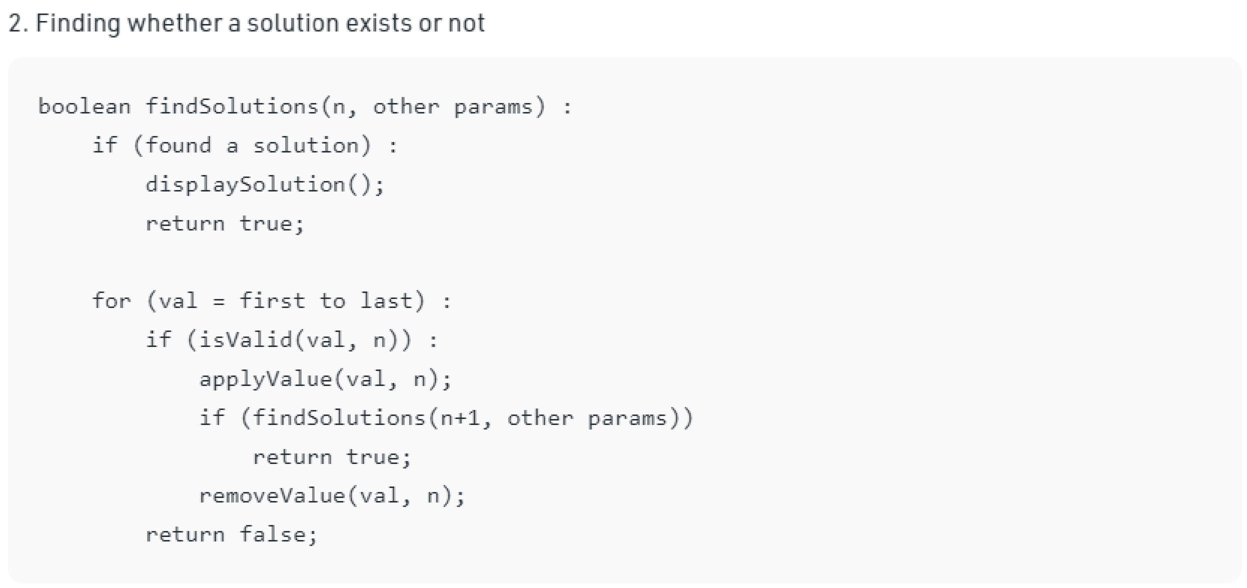
3. In the event of "eligibility" of the newly formed sub-tree s, recurs back to step 1, using argument $S + s$.
4. If the check for $S + s$ returns that it is a solution for the entire data D. Output and terminate the program.
   If not, then return that no solution is possible with the current s and hence discard it.



1. Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```



2. Finding whether a solution exists or not

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```

Solving a Maze (Only 1 solution maze)

Backtracking

How recursion supports natural backtracking?