Fatimaa Khan

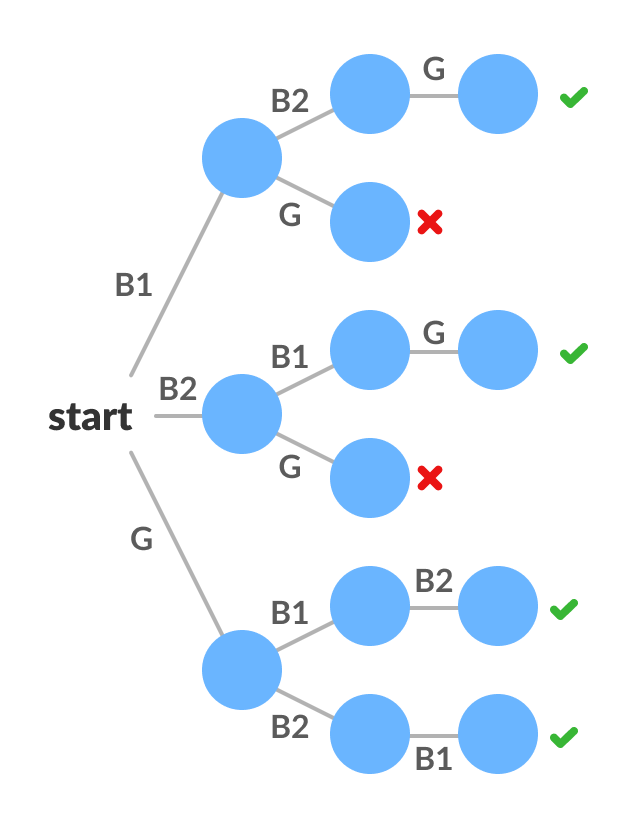
CT-23024

*BackTracking*

#### Backtracking

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

#### Diagram:



Pseudocode

void FIND\_SOLUTIONS( parameters):

if (valid solution):

store the solution

Return

for (all choice):

if (valid choice):

APPLY (choice)

FIND\_SOLUTIONS (parameters)

BACKTRACK (remove choice)

Return

#### Difference between recursion and backtracking

| **Recursion** | **Backtracking** |
| --- | --- |
| Recursion does not always need backtracking | Backtracking always uses recursion to solve problems |
| Solving problems by breaking them into smaller, similar subproblems and solving them recursively. | Solving problems with multiple choices and exploring options systematically, backtracking when needed. |
| Controlled by function calls and call stack. | Managed explicitly with loops and state. |
| **Applications of Recursion:** Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search. | **Application of Backtracking:**N Queen problem, Rat in a Maze problem, Knight’s Tour Problem, Sudoku solver, and Graph coloring problems. |

#### Code to find all subsets

#include <iostream>

using namespace std;

*// Function to generate all subsets of a given set*

void generateSubsets(int set[], int currentSubset[], int n, int subsetSize, int index) {

*// Print the current subset*

    cout << "{";

    for (int i = 0; i < subsetSize; i++) {

        cout << currentSubset[i];

        if (i < subsetSize - 1) cout << ", ";

    }

    cout << "}" << endl;

*// Generate subsets by including each element starting from 'index'*

    for (int i = index; i < n; i++) {

        currentSubset[subsetSize] = set[i]; *// Include the element*

        generateSubsets(set, currentSubset, n, subsetSize + 1, i + 1); *// Recursively generate subsets for the remaining elements*

    }

}

int main() {

    int inputSet[] = {1, 2, 3}; *// Input set*

    int n = sizeof(inputSet) / sizeof(inputSet[0]); *// Size of the input set*

    int currentSubset[10]; *// Array to store the current subset (size set to a safe upper limit)*

*// Generate all subsets*

    generateSubsets(inputSet, currentSubset, n, 0, 0);

    return 0;

}

#### Binary Sequence

#include <iostream>

using namespace std;

*// Function to generate and print all binary sequences of length 'possibleOutcomes'*

void binarySequence(int outcomes[], int currentOutcome[], int n, int depth, int possibleOutcomes) {

    if (depth == possibleOutcomes) {

*// Base case: if we have generated a sequence of length 'possibleOutcomes', print it*

        for (int i = 0; i < possibleOutcomes; i++) {

            cout << currentOutcome[i];

        }

        cout << endl;

        return;

    }

*// Recursively generate sequences by choosing either 0 or 1*

    for (int i = 0; i < n; i++) {

        currentOutcome[depth] = outcomes[i]; *// Choose the current outcome (0 or 1)*

        binarySequence(outcomes, currentOutcome, n, depth + 1, possibleOutcomes); *// Recurse for the next position*

    }

}

int main() {

    int outcomes[] = {0, 1}; *// Possible outcomes for each position in the sequence (binary)*

    int n = sizeof(outcomes) / sizeof(outcomes[0]); *// Calculate the size of the outcomes array*

    int possibleOutcomes = 4; *// Length of the binary sequence*

    int currentOutcome[4]; *// Array to store the current sequence being generated*

*// Generate all binary sequences of length 'possibleOutcomes'*

    binarySequence(outcomes, currentOutcome, n, 0, possibleOutcomes);

    return 0;

}