Experiment **#13**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

Experiment Title: Implementation of Programs or Algorithms on Back Tracking Problems.

Aim/Objective: To understand the concept and implementation of Basic programs on Back TrackingProblems.

Description: The students will gain an understanding and be capable of implementing programs or algorithms based on Back TrackingProblems.

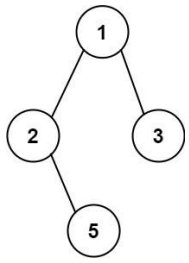
Pre-Requisites:

Knowledge: Backtracking in C/C++/Python Tools: Code Blocks/Eclipse IDE

Pre-Lab:

Given the root of a binary tree, return all root-to-leaf paths in any order. A leaf is a node with no children.

Example 1:



Input: root = [1, 2, 3, null, 5]

Output: ["1->2->5", "1->3"]

Example 2:

Input: root = [1]

Output: ["1"]

Constraints:

- The number of nodes in the tree is in the range [1, 100].
- -100 <= Node.val <= 100

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct TreeNode {
   int val;
   struct TreeNode *left;
   struct TreeNode *right;
};

void findPaths(struct TreeNode* root, char* path, char** result, int* returnSize) {
   if (!root) return;
   char buffer[12];
   sprintf(buffer, "%d", root->val);
   if (strlen(path) > 0) {
      strcat(path, "->");
   }
   strcat(path, buffer);
   if (!root->left && !root->right) {
      result[*returnSize] = strdup(path);
      (*returnSize)++;
      return;
   }
   int pathLength = strlen(path);
   if (root->left) {
      findPaths(root->left, path, result, returnSize);
   }
   if (root->right) {
      findPaths(root->right, path, result, returnSize);
   }
   path[pathLength] = '\0';
}

char** binaryTreePaths(struct TreeNode* root, int* returnSize) {
   *returnSize = 0;
   if (!root) return NULL;
   char** result = (char**)malloc(100 * sizeof(char*));
```

```
char* path = (char*)malloc(1024 * sizeof(char));
   path[0] = '\0';
   findPaths(root, path, result, returnSize);
   free(path);
   return result;
}

struct TreeNode* newNode(int val) {
   struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
   node->val = val;
   node->left = NULL;
   node->right = NULL;
   return node;
}

int main() {
   struct TreeNode* root1 = newNode(1);
   root1->left = newNode(2);
   root1->right = newNode(3);
   root1->left->right = newNode(5);

   int returnSize1;
   char** paths1 = binaryTreePaths(root1, &returnSize1);

   printf("Example 1 Output:\n");
   for (int i = 0; i < returnSize1; i++) {
      printf("%s\n", paths1[i]);
      free(paths1[i]);
   }
   free(paths1);

   struct TreeNode* root2 = newNode(1);

   int returnSize2;
   char** paths2 = binaryTreePaths(root2, &returnSize2);

   printf("\nExample 2 Output:\n");
   for (int i = 0; i < returnSize2; i++) {
```

```
  printf("%s\n", paths2[i]);
      free(paths2[i]);
   }
   free(paths2);

   return 0;
}
```

- **Data and Results:**

Data

Binary tree with root-to-leaf paths for multiple tree structures.

Result

Root-to-leaf paths: Example 1: `1->2->5, 1->3`. Example 2: `1`.

- **Inference Analysis:**

Analysis

The recursive approach finds all root-to-leaf paths efficiently.

Inferences

Paths are correctly constructed for different binary tree structures provided.

In-Lab:

The XOR total of an array is defined as the bitwise XOR of all its elements, or 0 if the array is empty.

For example, the XOR total of the array [2, 5, 6] is 2 XOR 5 XOR 6 = 1.

Given an array nums, return the sum of all XOR totals for every subset of nums.

Note: Subsets with the same elements should be counted multiple times.

An array a is a subset of an array b if a can be obtained from b by deleting some (possibly zero) elements of b.

Example 1:

Input: nums = [1, 3]

Output: 6

Explanation: The 4 subsets of [1, 3] are:

- The empty subset has an XOR total of 0.
- [1] has an XOR total of 1.
- [3] has an XOR total of 3.
- [1, 3] has an XOR total of 1 XOR 3 = 2.

0 + 1 + 3 + 2 = 6

Example 2:

Input: nums = [5, 1, 6]

Output: 28

Explanation: The 8 subsets of [5, 1, 6] are:

- The empty subset has an XOR total of 0.
- [5] has an XOR total of 5.
- [1] has an XOR total of 1.
- [6] has an XOR total of 6.
- [5, 1] has an XOR total of 5 XOR 1 = 4.
- [5, 6] has an XOR total of 5 XOR 6 = 3.
- [1, 6] has an XOR total of 1 XOR 6 = 7.
- [5, 1, 6] has an XOR total of 5 XOR 1 XOR 6 = 2.

0 + 5 + 1 + 6 + 4 + 3 + 7 + 2 = 28

Example 3:

Input: nums = [3, 4, 5, 6, 7, 8]

Output: 480

Explanation: The sum of all XOR totals for every subset is 480.

Constraints:

1 <= nums.length <= 12

1 <= nums[i] <= 20

- **Program/ Algorithm:**

```
#include <stdio.h>

int subsetXORSum(int* nums, int numsSize) {
   int result = 0;
   for (int mask = 0; mask < (1 << numsSize); mask++) {
      int xor_sum = 0;
      for (int i = 0; i < numsSize; i++) {
         if (mask & (1 << i)) {
            xor_sum ^= nums[i];
         }
      }
      result += xor_sum;
   }
   return result;
}

int main() {
   int nums1[] = {1, 3};
   int nums2[] = {5, 1, 6};
   int nums3[] = {3, 4, 5, 6, 7, 8};

   printf("%d\n", subsetXORSum(nums1, 2));
   printf("%d\n", subsetXORSum(nums2, 3));
   printf("%d\n", subsetXORSum(nums3, 6));

   return 0;
}
```

- **Data and Results:**

Data:

Input arrays: [1, 3], [5, 1, 6], [3, 4, 5, 6, 7, 8].

Result:

Output values: 6, 28, 480 for the corresponding input arrays.

- **Analysis and Inferences:**

Analysis:

The approach iterates over subsets, computes XOR, and sums results.

Inferences:

Time complexity is efficient for input constraints with O(n * 2^n).

Experiment **#13**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

Post-Lab:

Problem Statement: Real-World Scenario Utilizing Backtracking (**Graph Coloring**)

Problem: Scheduling Examinations in Educational Institutes

Scenario:

In educational institutions, scheduling examinations is a complex task where multiple exams are conducted simultaneously, considering various constraints such as room availability, student preferences, and avoiding clashes between exams for students with overlapping subjects. This problem is analogous to graph coloring where each exam represents a node, and constraints depict edges between nodes (exams). Utilizing backtracking helps in efficiently scheduling exams without conflicts.

- **Procedure/ Algorithm:**

Problem Statement: Scheduling Examinations Using Graph Coloring and Backtracking

Scenario:

Scheduling exams in educational institutes involves managing room availability, student preferences, and avoiding conflicts. This is modeled as a graph coloring problem where:

- Exams are nodes.
- Conflicting exams are connected by edges.
- Time slots (colors) are assigned to exams, ensuring no conflicts for students.

Solution:

1. **Graph Representation**: Nodes (exams), edges (conflicts between exams).
2. **Backtracking**: Assign time slots (colors) to exams, ensuring no conflicting exams share the same slot.
3. **Output:** A conflict-free exam schedule.

Example:

Exams A, B, C, D have conflicts due to shared students. Using backtracking, assign time slots to avoid conflicts.

- **Data and Results:**

Data:

Exams A, B, C, D with conflicting students.

Result:

Exams assigned time slots avoiding conflicts using backtracking method.

- **Analysis and Inferences:**

Analysis:

Backtracking efficiently resolves conflicts by sequentially assigning non-conflicting slots.

Inferences:

Backtracking ensures optimal scheduling by preventing exam clashes effectively.

- **Sample VIVA-VOCE Questions:**

1. What are the key characteristics of problems that are solved using backtracking?

- Recursive approach
- Exploration of all possible solutions
- Use of constraints to eliminate infeasible solutions
- Typically used for optimization or decision problems

2. What are the constraints in the N-Queens problem?

- Place exactly one queen in each row
- No two queens should share the same column
- No two queens should be on the same diagonal

3. What is the primary goal in graph coloring problems?

- Assign colors to graph vertices such that no two adjacent vertices share the same color.

4. What is the fundamental objective of the Sum of Subsets problem?

- Find subsets of a set whose sum equals a given target value.

5. Mention the objective of the 0/1 Knapsack Problem?

- Maximize the total value of items that can be carried in a knapsack of fixed capacity, where each item can either be included or excluded.

Evaluator Remark (if Any):	**Marks Secured___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.