Experiment Title: Implementation of Programs on Sudoku Solver-Backtracking approach,

Aim/Objective: To understand the concept and implementation of Basic programs on Sudoku Solver-Backtracking approach

Description:

The students will understand and able to implement programs on Sudoku Solver-Backtracking approach.

Pre-Requisites:

Knowledge: Sudoku Solver-Backtracking approach in C/C++/Python

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

You are given an integer N. You need to create and output to the console all the divisors of this integer in Descending order.

Input Format

- The first line of input will contain a single integer T, denoting the number of test cases.
- Each test case consists of a single line of input - the integer N.

Input

2

12

21

Output

12 6 4 3 2 1

21 7 3 1

- **Procedure/Program:**

```
#include <stdio.h>

void find_divisors(int n) {
   int divisors[100], count = 0;

   for (int i = 1; i * i <= n; i++) {
      if (n % i == 0) {
         divisors[count++] = i;
```

```
            if (i != n / i) {
                divisors[count++] = n / i;
            }
        }
    }

    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            if (divisors[i] < divisors[j]) {
                int temp = divisors[i];
                divisors[i] = divisors[j];
                divisors[j] = temp;
            }
        }
    }

    for (int i = 0; i < count; i++) {
        printf("%d ", divisors[i]);
    }
    printf("\n");
}

int main() {
    int T;
    scanf("%d", &T);

    while (T--) {
        int N;
        scanf("%d", &N);
        find_divisors(N);
    }

    return 0;
}
```

- **Data and Results:**

Data:

Input contains multiple test cases, each with a number `N`.

Result:

Divisors of `N` are printed in descending order for each test.

- **Analysis and Inferences:**

Analysis:

The algorithm finds divisors efficiently and sorts them in descending order.

Inferences:

Output demonstrates divisors of numbers in descending order, fulfilling requirements.

In-Lab:

The Clique Decision Problem belongs to NP-Hard. Prove that the Boolean Satisfiability problem reduces to the Clique Decision Problem

• **Procedure/Program:**

Proof: SAT Reduces to Clique Decision Problem

1. **SAT**: Given a Boolean formula, determine if there is a satisfying assignment for variables.
2. **Clique Decision**: Given a graph G and k, check if there is a clique of size k.

Reduction:

- From a SAT formula with n variables and m clauses, construct a graph G where:
  - Vertices represent literals (variables or their negations).
  - Edges exist between non-conflicting literals.
- Set $k = n$ (number of variables).

Result:

- If SAT is satisfiable, there is a clique of size n in the graph.
- If SAT is unsatisfiable, no such clique exists.

Post-Lab:

Given an array of positive elements, you must flip the sign of some of its elements such that the resultant sum of the elements of array should be minimum non-negative (as close to zero as possible). Return the minimum no. of elements whose sign needs to be flipped such that the resultant sum is minimum non-negative. Note that the sum of all the array elements will not exceed 104.

Input

arr [] = {15, 10, 6}

Output

1

Here, we will flip the sign of 15and the resultant sum will be 1.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>

int minFlips(int arr[], int n) {
   int total_sum = 0;
   for (int i = 0; i < n; i++) {
      total_sum += arr[i];
   }

   bool dp[total_sum + 1];
   dp[0] = true;
   for (int i = 1; i <= total_sum; i++) {
      dp[i] = false;
   }

   for (int i = 0; i < n; i++) {
      for (int j = total_sum; j >= arr[i]; j--) {
         dp[j] |= dp[j - arr[i]];
      }
   }

   int half_sum = total_sum / 2;
   int closest_sum = 0;
```

```
    for (int s = half_sum; s >= 0; s--) {
        if (dp[s]) {
            closest_sum = s;
            break;
        }
    }

    int result_sum = total_sum - 2 * closest_sum;
    int flips = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] <= result_sum) {
            flips++;
            result_sum -= arr[i];
        }
    }

    return flips;
}

int main() {
    int arr[] = {15, 10, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int flips = minFlips(arr, n);

    printf("Output: %d\n", flips);
    return 0;
}
```

- **Data and Results:**

Data: Array elements are `{15, 10, 6}` with a total sum of `31`.

Result: Minimum flips required to achieve the closest sum to zero is `1`.

- **Analysis and Inferences:**

Analysis: Flipping the sign of `15` results in a sum of `1`.

Inferences: Flipping only one element minimizes the sum closest to zero.

- **Sample VIVA-VOCE Questions:**

1) Describe the basic steps involved in the Sudoku backtracking algorithm.

- Find an empty cell.
- Try all digits (1-9) in the empty cell.
- Check if the digit is valid (no conflicts).
- If valid, move to the next empty cell.
- If no valid digit is found, backtrack to previous cell.

2) What is the termination condition for the Sudoku backtracking algorithm?

- The algorithm terminates when the entire board is filled correctly or if all possibilities have been exhausted (in case of no solution).

3) Differentiate between NP-Hard and NP-Complete?

- **NP-Hard**: Problems that are at least as hard as the hardest problems in NP (may or may not be in NP).
- **NP-Complete**: Problems that are in NP and are as hard as any problem in NP (can be verified in polynomial time).

4) Draw the ven diagram of P and NP class problems?

- **P**: Problems solvable in polynomial time.
- **NP**: Problems verifiable in polynomial time.
- The Venn diagram shows P is a subset of NP, and it's unknown if they are equal.

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.