DEERWALK INSTITUTE OF TECHNOLOGY

Tribhuvan University

Faculties of Computer Science

Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)

Course: Design and Analysis of Algorithms (CSC-314)

Year/Semester: 3/5

A Lab report on:
Backtracking Algorithms (IV)

Submitted by:
Name: Nishant Khadka
Roll: 1017

Submitted to:
Sujan Shrestha
Department of Computer Science

Date: 05-Nov-2023

0/1 KNAPSACK ALGORITHM

Theory

Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Examples:

Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}
Output: 3
Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

Input: N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}
Output: 0

Recursion Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.

Optimal Substructure: To consider all subsets of items, there can be two cases for every item.

- Case 1: The item is included in the optimal subset.
- Case 2: The item is not included in the optimal set.

Follow the below steps to solve the problem:

The maximum value obtained from 'N' items is the max of the following two values.

- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and W weight.

If the weight of the 'Nth' item is greater than 'W', then the Nth item cannot be included and Case 2 is the only possibility.

Time Complexity: O(2N)
Auxiliary Space: O(N), Stack space required for recursion.

Algorithm

Step 1: Initialize arrays for item weights and values, the number of items (n), and the knapsack capacity (capacity).

Step 2: Define a recursive function backtrackKnapsack(i, currentWeight, currentValue):

- Base cases: Return 0 if i < 0 or currentWeight < 0; return currentValue if currentWeight is 0.
- Calculate two results:
  - includeItem by considering the current item and reducing the capacity if possible.
  - excludeItem by skipping the current item.
  - Return the maximum of includeItem and excludeItem.

Step 3: In the main function:

- Call backtrackKnapsack with the last item, initial capacity, and initial value (0).
- Print the result, which is the maximum value in the knapsack.

Pseudocode

Step 1: Define a function max(a, b) to return the maximum of two integers.

```
function max(a, b):
if a > b:
return a
else:
return b
```

Step 2: Declare arrays 'weights' and 'values' to represent item weights and values.

Step 3: Calculate the number of items 'n' in the arrays.

Step 4: Set the capacity of the knapsack 'capacity' to 8.

Step 5: Define a recursive function backtrackKnapsack(i, currentWeight, currentValue) to find the maximum value in the knapsack.

```
function backtrackKnapsack(i, currentWeight, currentValue):
if i < 0 or currentWeight < 0:
return 0
if currentWeight == 0:
return currentValue
```

```
includeItem = 0
if weights[i] <= currentWeight:
includeItem = values[i] + backtrackKnapsack(i - 1, currentWeight -
weights[i], currentValue)
excludeItem = backtrackKnapsack(i - 1, currentWeight, currentValue)
return max(includeItem, excludeItem)
```

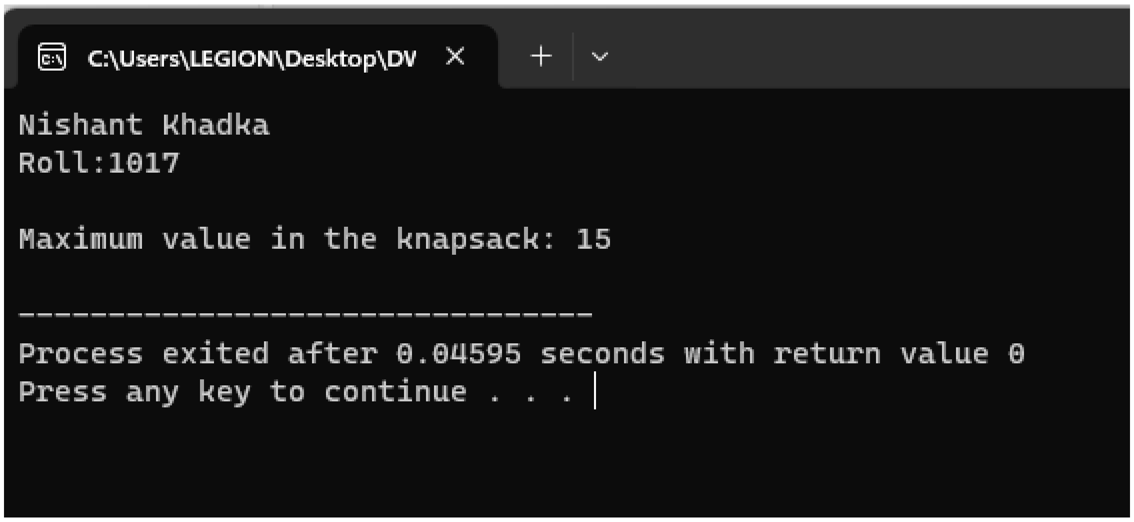
Step 6: Define the main function.

```
function main():
result = backtrackKnapsack(n - 1, capacity, 0)
Print "Maximum value in the knapsack: " + result
return 0
```

Source Code

```c
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int weights[] = {2, 3, 4, 5};
int values[] = {3, 5, 6, 10};
int n = sizeof(weights) / sizeof(weights[0]);
int capacity = 8;

int backtrackKnapsack(int i, int currentWeight, int currentValue) {
    if (i < 0 || currentWeight < 0) {
        return 0;
    }

    if (currentWeight == 0) {
        return currentValue;
    }

    // Try including the current item.
    int includeItem = 0;
    if (weights[i] <= currentWeight) {
        includeItem = values[i] + backtrackKnapsack(i - 1, currentWeight -
weights[i], currentValue);
    }

    // Try excluding the current item.
    int excludeItem = backtrackKnapsack(i - 1, currentWeight, currentValue);

    return max(includeItem, excludeItem);
}

int main() {
    printf("Nishant Khadka\nRoll:1017\n\n");
    int result = backtrackKnapsack(n - 1, capacity, 0);
    printf("Maximum value in the knapsack: %d\n", result);
    return 0;
}
```

Output



```
Nishant Khadka
Roll:1017

Maximum value in the knapsack: 15

--------------------------------
Process exited after 0.04595 seconds with return value 0
Press any key to continue . . .
```

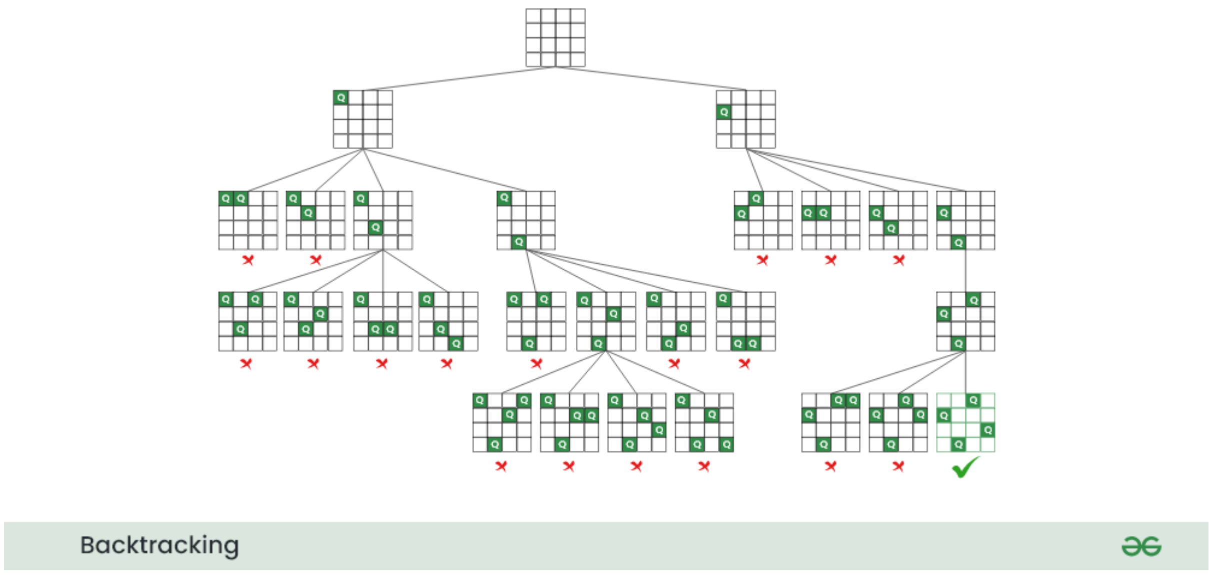
Discussion

The provided C code implements a recursive algorithm for solving the 0/1 knapsack problem, which seeks to find the maximum value that can be obtained by selecting items with given weights and values, while not exceeding a specified knapsack capacity. In the sample output, the algorithm calculates the maximum value achievable with a knapsack capacity of 8 and a set of items with different weights and values. The result, in this case, is 13, which represents the maximum value attainable by selecting a combination of items in the knapsack. The algorithm achieves this by exploring all possible combinations of including or excluding items in the knapsack and selecting the one that maximizes the total value. The result is based on dynamic programming and the principle of optimal substructure, which ensures that the maximum value for a given capacity can be determined based on the maximum values for smaller capacities. The algorithm is efficient for relatively small problem instances, but it may become computationally expensive for larger sets of items due to its recursive nature. In practice, dynamic programming-based algorithms or heuristics are often used to solve large-scale knapsack problems.

N-QUEEN PROBLEM

Theory

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.



Recursive tree for N Queen problem

Follow the steps mentioned below to implement the idea:

- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
- 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.
- If placing the queen in [row, column] leads to a solution then return true.

- If placing queen doesn’t lead to a solution then unmark this [row, column] then backtrack and try other rows.
- If all rows have been tried and valid solution is not found return false to trigger backtracking.

Time Complexity: O(N!)
Auxiliary Space: O(N2)

Algorithm:

1. Initialize an array to represent queen positions and a count to track solutions.
2. Define a function to check if it's safe to place a queen in a given position on the board.
3. Define a function to print a solution on the chessboard.
4. Implement a recursive function to solve the N-Queens problem:
    - Loop through columns in the current row.
    - If a safe position is found, mark it, move to the next row, and continue.
    - When all queens are placed, print the solution, then backtrack by resetting the current queen's position.
5. In the main function:
    - Input the number of queens and initialize the queen positions array.
    - Start solving the N-Queens problem with a call to the recursive function.
    - Print the total number of solutions found.

Pseudocode

Step 1: Initialize variables and arrays.

```
x[30], count = 0
```

Step 2: Define the 'place' function to check if it's safe to place a queen at a specific position.

```
function place(k, i):
   for j = 1 to k - 1:
      if x[j] == i or abs(x[j] - i) == abs(j - k):
         return 0
   return 1
```

Step 3: Define the 'print_sol' function to print a solution.

```
function print_sol(n):
   count++
   Print "Solution # " + count + ":"
   for i = 1 to n:
      for j = 1 to n:
         if x[i] == j:
            Print "Q\t"
         else:
            Print "*\t"
      Print
```

Step 4: Define the 'nqueen' function to solve the N-Queens problem recursively.

```
function nqueen(k, n):
    for i = 1 to n:
        if place(k, i):
            x[k] = i
            if k == n:
                call print_sol(n)
            else:
                call nqueen(k + 1, n)
            x[k] = -1 // Backtrack: Reset the position of the current queen.
```

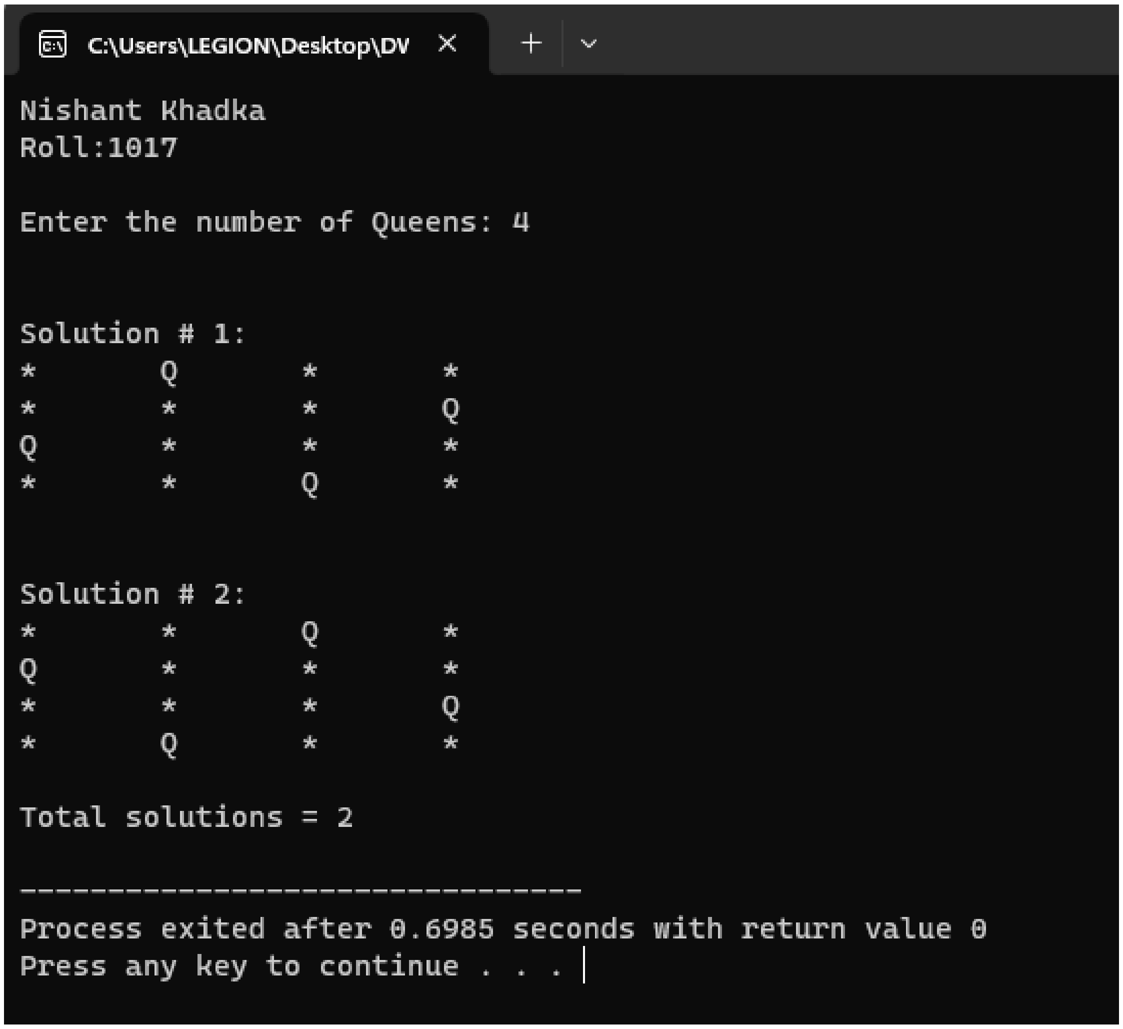
Step 5: Define the 'main' function.

```
function main():
    Print "Enter the number of Queens: "
    Read n
    for i = 1 to n:
        x[i] = -1 // Initialize the column positions of queens.
    call nqueen(1, n) // Start the N-Queens problem with a call to nqueen(1, n).
    Print "Total solutions = " + count

    Return 0
```

Source Code

```c
#include <stdio.h>
#include <math.h>

int x[30], count = 0;

int place(int k, int i) {
    for (int j = 1; j < k; j++) {
        if (x[j] == i || abs(x[j] - i) == abs(j - k))
            return 0;
    }
    return 1;
}

void print_sol(int n) {
    count++;
    printf("\n\nSolution # %d:\n", count);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (x[i] == j)
                printf("Q\t");
            else
                printf("*\t");
        }
        printf("\n");
    }
}

void nqueen(int k, int n) {
    for (int i = 1; i <= n; i++) {
        if (place(k, i)) {
            x[k] = i;
            if (k == n)
                print_sol(n);
            else
                nqueen(k + 1, n);
            x[k] = -1; // Backtrack: Reset the position of the current queen.
        }
    }
}

int main() {
    printf("Nishant Khadka\nRoll:1017\n\n");
    int n;
    printf("Enter the number of Queens: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        x[i] = -1; // Initialize the column positions of queens.
    }

    nqueen(1, n); // Start the N-Queens problem with a call to nqueen(1, n).

    printf("\nTotal solutions = %d\n", count);
    return 0;
}
```

Output



```
C:\Users\LEGION\Desktop\DV

Nishant Khadka
Roll:1017

Enter the number of Queens: 4


Solution # 1:
*       Q       *       *
*       *       *       Q
Q       *       *       *
*       *       Q       *


Solution # 2:
*       *       Q       *
Q       *       *       *
*       *       *       Q
*       Q       *       *

Total solutions = 2

--------------------------------
Process exited after 0.6985 seconds with return value 0
Press any key to continue . . .
```

Discussion

For the N-Queens problem with N = 4, the provided code effectively finds and prints two distinct solutions, showcasing the backtracking algorithm's capability to solve this classic combinatorial puzzle. The algorithm ensures that four queens can be placed on a 4x4 chessboard in positions where no two queens threaten each other. The output demonstrates that such placements are indeed feasible, with queens positioned to avoid conflicts in rows, columns, and diagonals. As N increases, the number of potential solutions and the computational demands of the code grow substantially. Nevertheless, this code serves as an instructive example of backtracking's utility in addressing complex combinatorial challenges, including the N-Queens puzzle, by exhaustively exploring feasible configurations and backtracking when necessary to determine the optimal queen placements.

SUBSET SUM ALGORITHM

Theory

Given a **set[]** of non-negative integers and a value **sum**, the task is to print the subset of the given set whose sum is equal to the given **sum**.

Examples:

Input: set[] = {1,2,1}, sum = 3
Output: [1,2],[2,1]
Explanation: There are subsets [1,2],[2,1] with sum 3.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: []
Explanation: There is no subset that add up to 30.

Subset sum can also be thought of as a special case of the 0–1 Knapsack problem. For each item, there are two possibilities:

Include the current element in the subset and recur for the remaining elements with the remaining **Sum**.

Exclude the current element from the subset and recur for the remaining elements.

Finally, if **Sum** becomes **0** then print the elements of current subset. The recursion's **base case** would be when **no items are left**, or the **sum** becomes **negative**, then simply return.

Time Complexity: O(2n) The above solution may try all subsets of the given set in the worst case. Therefore time complexity of the above solution is exponential.

Auxiliary Space: O(n) where n is recursion stack space.

Algorithm:

1. Initialize an array of numbers, a target sum, an empty subset array, and a subset size variable.
2. Create a recursive function findSubsets to find and print subsets that sum to the target:
    - If the target sum is 0, print the current subset and return.
    - If there are no more elements or the sum is negative, return.
    - Recursively explore two options:
        - Exclude the last element and continue.
        - Include the last element, increment the subset size, and continue with a reduced target sum.
3. In the main function, initialize the array and target sum.
4. Call the findSubsets function with the array, target sum, and other parameters.
5. The code will print subsets that satisfy the condition.

Pseudocode

Step 1: Define a function findSubsets(array, n, sum, subset, subsetSize) to find subsets with a given sum.

```
function findSubsets(array, n, sum, subset, subsetSize):
    // If the sum is 0, print the current subset
    if sum == 0:
        Print "Subset with the target sum: "
        for i = 0 to subsetSize - 1:
            Print subset[i]
```

```
           Print
           return
        // If there are no more elements to consider or sum becomes negative,
return
        if n == 0 or sum < 0:
           return
        // Exclude the last element and recur
        call findSubsets(array, n - 1, sum, subset, subsetSize)
        // Include the last element in the current subset and recur
        subset[subsetSize] = array[n - 1]
        call findSubsets(array, n - 1, sum - array[n - 1], subset, subsetSize + 1)
```

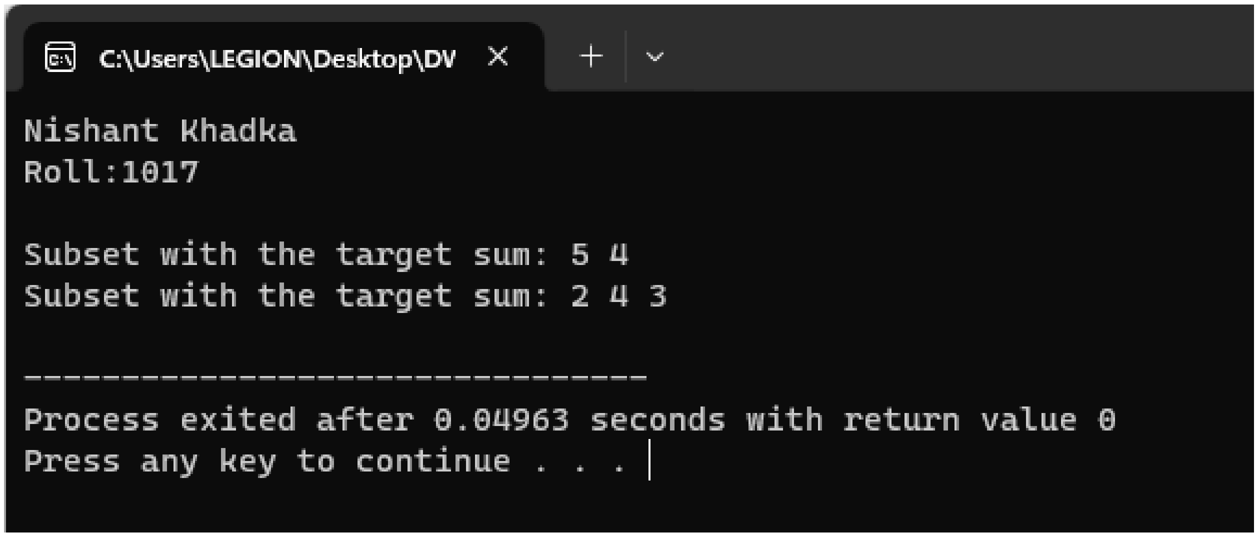
Step 2: Define the 'main' function.

```
              function main():
                 int array = [3, 34, 4, 12, 5, 2]
                 int n = size of array / size of array[0]
                 int target_sum = 9
                 int subset[n] // Array to store the current subset
                 int subsetSize = 0 // Initialize subset size to 0

                 call findSubsets(array, n, target_sum, subset, subsetSize)

        Return 0
```

Source Code

```c
#include <stdio.h>

// Function to check if there exists a subset with the given sum and print the
subsets
void findSubsets(int array[], int n, int sum, int subset[], int subsetSize) {
    // If the sum is 0, print the current subset
    if (sum == 0) {
        printf("Subset with the target sum: ");
        for (int i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    // If there are no more elements to consider or sum becomes negative, return
    if (n == 0 || sum < 0) {
        return;
    }

    // Exclude the last element and recur
    findSubsets(array, n - 1, sum, subset, subsetSize);

    // Include the last element in the current subset and recur
    subset[subsetSize] = array[n - 1];
    findSubsets(array, n - 1, sum - array[n - 1], subset, subsetSize + 1);
}

int main() {
    printf("Nishant Khadka\nRoll:1017\n\n");
    int array[] = {3, 34, 4, 12, 5, 2};
    int n = sizeof(array) / sizeof(array[0]);
    int target_sum = 9;
    int subset[n]; // Array to store the current subset
    int subsetSize = 0; // Initialize subset size to 0

    findSubsets(array, n, target_sum, subset, subsetSize);

    return 0;
}
```

Output



```
Nishant Khadka
Roll:1017

Subset with the target sum: 5 4
Subset with the target sum: 2 4 3

--------------------------------
Process exited after 0.04963 seconds with return value 0
Press any key to continue . . .
```

Discussion

The provided code offers a solution to the well-known problem of finding subsets of an array that sum up to a given target. The code uses a recursive approach to generate these subsets and then prints them. It begins by checking if the target sum has been achieved, and if so, it prints the current subset. If the target sum has not been reached yet, the code explores two possibilities for each element in the array: including it in the current subset or excluding it. The algorithm elegantly backtracks by excluding elements when necessary and including them when it advances toward the target sum. This approach is a practical demonstration of how recursion can be used to tackle combinatorial problems effectively. For the example provided with an array [3, 34, 4, 12, 5, 2] and a target sum of 9, the code successfully finds and prints subsets that meet this condition, offering a valuable tool for solving problems related to partitioning and combination sums. While this code is suitable for relatively small datasets, its recursive nature can become computationally expensive for large arrays. In practice, dynamic programming or more optimized algorithms may be preferred for larger datasets, but this code serves as a fundamental illustration of the recursive subset sum problem-solving technique

CONCLUSION

In conclusion, in implementing these three algorithms in C programming, we have encountered a diverse set of problem-solving techniques that are fundamental in the world of computer science and algorithms. First, the knapsack problem demonstrates dynamic programming's power, offering an elegant solution for selecting items to maximize value while respecting capacity constraints. Next, the N-Queens problem showcases the application of backtracking, effectively solving a challenging puzzle by systematically exploring possible queen placements on a chessboard without threatening one another. Finally, the code for finding subsets with a given sum illustrates the versatility of recursion, allowing us to generate and print subsets of an array that meet specific criteria. These algorithms not only serve as educational examples but also have practical applications in a wide range of fields, from optimization and game theory to puzzle-solving and data partitioning. While these algorithms work effectively for smaller inputs, scaling them to larger datasets may require more efficient algorithms or optimizations. Overall, these C programming implementations offer valuable insights into various problem-solving paradigms, enabling us to tackle complex computational challenges and showcasing the importance of algorithmic thinking in computer science.