Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 7: Backtracking (N-Queens Problem and Sum of Subsets)

Aim – Implement solutions to the N-Queens Problem and Sum of Subsets problem using backtracking algorithms.

Part – 1: N-Queens Problem

Problem statement:

Find all possible ways to place **N** chess queens on an **N×N** chessboard so that no two queens attack each other.

Pseudocode :

```
Algorithm N-Queens(n)
  Input: The number of queens n
  Output: All solutions to the n-queens problem

  1. Initialize an array column[1..n] with all elements as -1
  2. for i = 1 to n
  3.     Place-Queens(column, 1, i)

Function Place-Queens(column, i, j)
  Input: The column array, the current row i, and the current column j
  Output: None

  1. column[i] = j
```

2. if i > 1

3. for x = i-1 down to 1

4. if column[x] == j - (i - x) or column[x] == j + (i - x)

5. column[i] = -1

6. return

7. if i == n

8. Print-Queens(column, n)

9. column[i] = -1

10. for x = 1 to n

11. if column[x] == 0

12. Place-Queens(column, i+1, x)

13. column[i] = -1

The algorithm places queens one by one in different columns, starting from the leftmost column. When placing a queen in a column, it checks for clashes with already placed queens. If a queen can be placed in the current column, the function recurs for the next column. If no place can be found for a queen, it returns false and the placement of queens in the previous columns is changed.

Source code(C language):

```c
#include<stdio.h>
#include<stdlib.h>
#include<wchar.h>
#include<locale.h>

int count=1;

int find(int x, int* column, int n){
    for(int i=0;i<n;i++){
        if(column[i]==x){
            return 1;
        }
    }
    return 0;
}
```

```
void printqueens(int* column,int n){
    wprintf(L"Solution %d:\n",count);
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(j!=column[i]){
                wprintf(L"□ ");
            }
            else{
                wprintf(L"♕ ");
            }
        }
        wprintf(L"\n");
    }
    wprintf(L"\n");
}

void calculate(int n, int* column,int i,int j){
    column[i]=j;
    if(i>0){
        for(int x=i-1;x>=0;x--){
            if(column[x]==j-(i-x) || column[x]==j+(i-x)){
                column[i]=-1;
                return;
            }
        }
    }
    if(i==n-1){
        printqueens(column,n);
        count++;
        column[i]=-1;
    }
    for(int x=0;x<n;x++){
        if(find(x,column,n)==0){
            calculate(n,column,i+1,x);
        }
    }
    column[i]=-1;
}

int main(){
    setlocale(LC_CTYPE,"");
    int n;
```

```
    wprintf(L"Enter value of N : ");
    scanf("%d",&n);
    int column[n];
    for(int i=0;i<n;i++){
        column[i]=-1;
    }
    for(int i=0;i<n;i++){
        calculate(n,column,0,i);
    }

return 0;
}
```

Output:

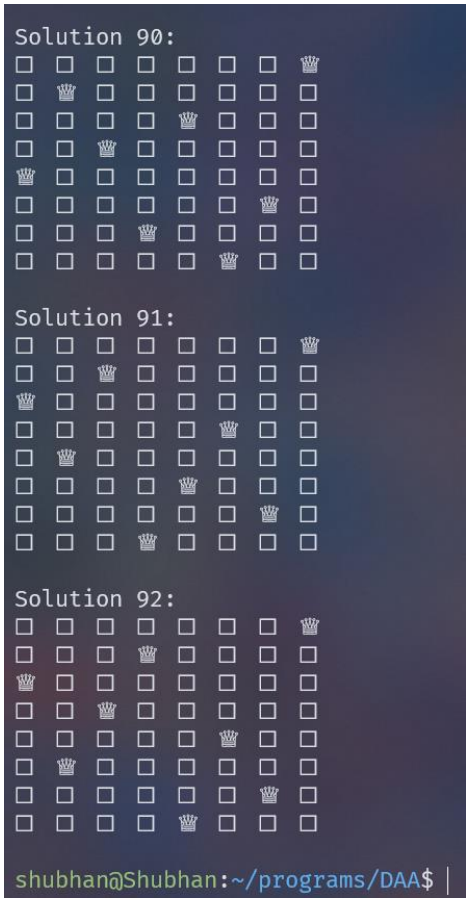
For n=4:

```
shubhan@Shubhan:~/programs/DAA$ ./nqueens
Enter value of N : 4
Solution 1:
□ ♛ □ □
□ □ □ ♛
♛ □ □ □
□ □ ♛ □

Solution 2:
□ □ ♛ □
♛ □ □ □
□ □ □ ♛
□ ♛ □ □

shubhan@Shubhan:~/programs/DAA$
```

For N=5:

```
shubhan@Shubhan:~/programs/DAA$ ./nqueens
Enter value of N : 5
Solution 1:
♛ □ □ □ □
□ □ ♛ □ □
□ □ □ □ ♛
□ ♛ □ □ □
□ □ □ ♛ □

Solution 2:
♛ □ □ □ □
□ □ □ ♛ □
□ ♛ □ □ □
□ □ □ □ ♛
□ □ ♛ □ □

Solution 3:
□ ♛ □ □ □
□ □ □ ♛ □
♛ □ □ □ □
□ □ ♛ □ □
□ □ □ □ ♛

Solution 4:
□ ♛ □ □ □
□ □ □ □ ♛
□ □ ♛ □ □
♛ □ □ □ □
□ □ □ ♛ □

Solution 5:
□ □ ♛ □ □
♛ □ □ □ □
□ □ □ ♛ □
□ ♛ □ □ □

Solution 6:
□ □ ♛ □ □
□ □ □ □ ♛
□ ♛ □ □ □
□ □ □ ♛ □
♛ □ □ □ □

Solution 7:
□ □ □ ♛ □
♛ □ □ □ □
□ □ ♛ □ □
□ □ □ □ ♛
□ ♛ □ □ □

Solution 8:
□ □ □ ♛ □
□ ♛ □ □ □
□ □ □ □ ♛
□ □ ♛ □ □
♛ □ □ □ □

Solution 9:
□ □ □ □ ♛
□ ♛ □ □ □
□ □ □ ♛ □
♛ □ □ □ □
□ □ ♛ □ □

Solution 10:
□ □ □ □ ♛
□ □ ♛ □ □
♛ □ □ □ □
□ □ □ ♛ □
□ ♛ □ □ □
```

For N=8:

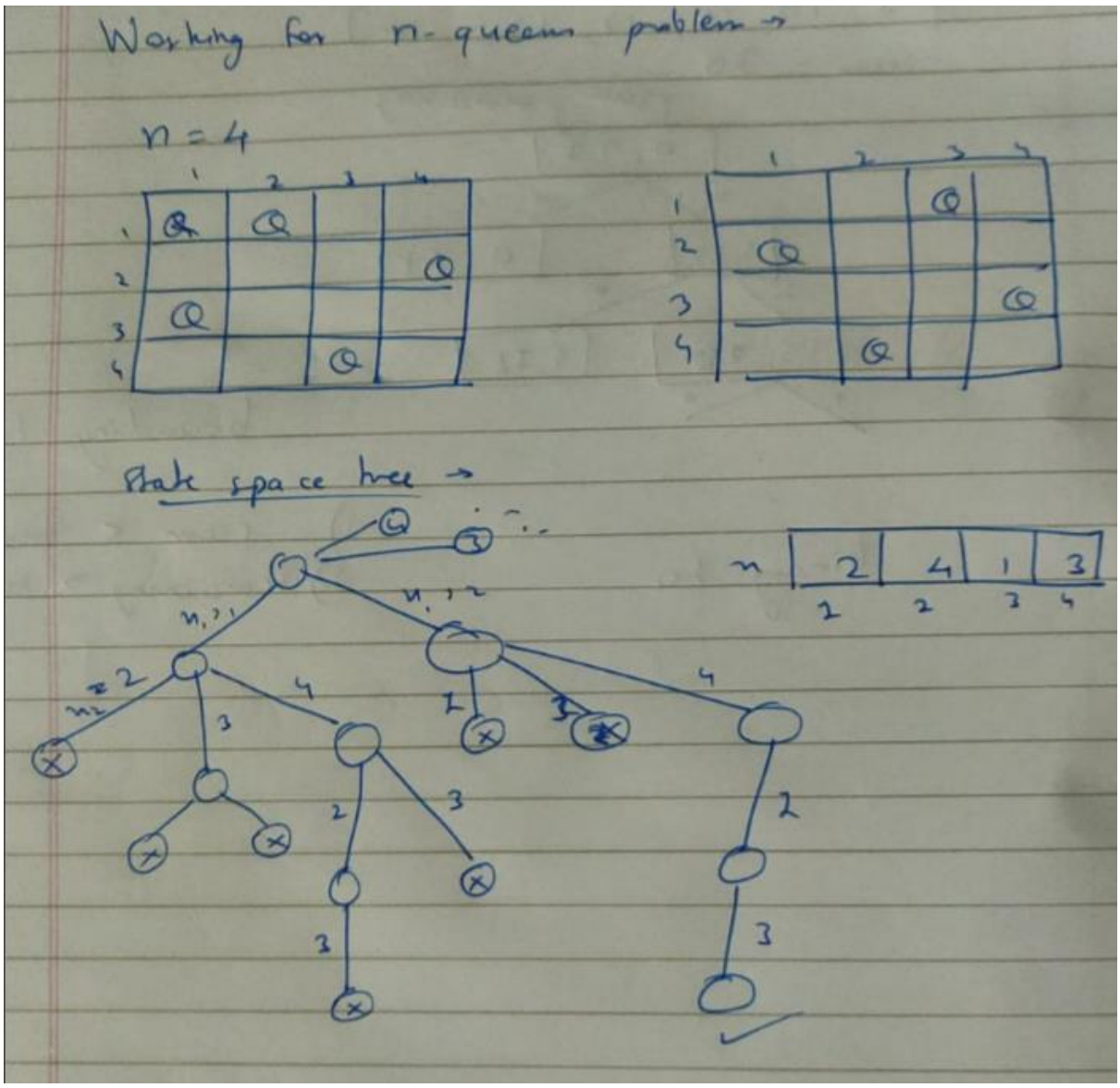


(92 solutions printed, only some shown)

Conclusion:

- We solved the N-Queens problem using backtracking in O(n^4) time complexity.
- For solving the problem, we placed queens from the top to bottom row, marking occupied columns, then for each subsequent row, we check the unoccupied columns and check if the square lies on any occupied diagonals.
- For subsequent rows, we check for n-1,n-2…1 squares, and each check takes O(n) time, so the time complexity comes out to be O(n^3), now this is repeated for all of the n positions for the queen in the initial row, making the final time complexity O(n^4).

Rough Working :



Part – 2: Sum of subsets problem

Problem statement:

Given a set of non-negative integers and a value sum, print all subsets of the given set whose sum is equal to the given sum.

Pseudocode :

Algorithm Subset-Sum

Input: An array arr[1..n] of integers, a target sum target

Output: All subsets of arr that add up to target

1. Initialize an array state[1..n] with all elements as 0
2. remains = sum of all elements in arr
3. Calculate(arr, n, target, state, 1, 0, remains)

Function Calculate(arr, n, target, state, i, curr, remains)

Input: The array arr, the number of elements n, the target sum target, the state array, the current index i, the current sum curr, and the remaining sum remains

Output: None

1. if curr == target
2. Print-Subset(state, n)
3. if i <= n and curr + remains >= target
4. state[i] = 1
5. Calculate(arr, n, target, state, i+1, curr+arr[i], remains-arr[i])
6. if i <= n and curr + remains - arr[i] >= target
7. state[i] = 0
8. Calculate(arr, n, target, state, i+1, curr, remains-arr[i])
9. state[i] = 0

The algorithm generates all subsets of the given set and checks if the sum of elements in each subset is equal to the target sum. The time complexity of this algorithm is O(2^n), where n is the number of elements in the set.

Source code(C language):

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

int count=0;
void printans(int* state, int* arr, int n){
    printf("Solution %d : ",count);
    for(int i=0;i<n;i++){
        if(state[i]){
            printf("%d + ",arr[i]);
        }
```

```
    }
    printf("\b\b \n");
}

void calculate(int* arr,int n, int target,int* state, int
i,int curr,int remains){
    if(curr==target){
        count++;
        printans(state,arr,n);
        state[i]=0;
        return;
    }
    if(i==n-1){
        if(curr+arr[i]!=target){
            state[i]=0;
            return;
        }
        else{
            state[i]=1;
            count++;
            printans(state,arr,n);
            state[i]=0;
            return;
        }
    }
    if(curr+arr[i]<=target){
        state[i]=1;
        calculate(arr,n,target,state,i+1,curr+arr[i],remains-
arr[i]);
    }
    if(curr+remains>=target){
        state[i]=0;
        calculate(arr,n,target,state,i+1,curr,remains-
arr[i]);
    }
    state[i]=0;
}

int main(){
    int n,remains=0;
    printf("Enter number of elements in array : ");
    scanf("%d",&n);
```

```
    int arr[n];
    printf("Enter the elements of the array : \n");
    for(int i=0;i<n;i++){
        scanf("%d",&arr[i]);
    }
    printf("Enter the target sum : ");
    int target;
    scanf("%d",&target);
    printf("\nThe subarrays which add up to the target sum
are :\n");
    int state[n];
    for(int i=0;i<n;i++){
        remains+=arr[i];
        state[i]=0;
    }
    calculate(arr,n,target,state,0,0,remains);

return 0;
}
```

Output:

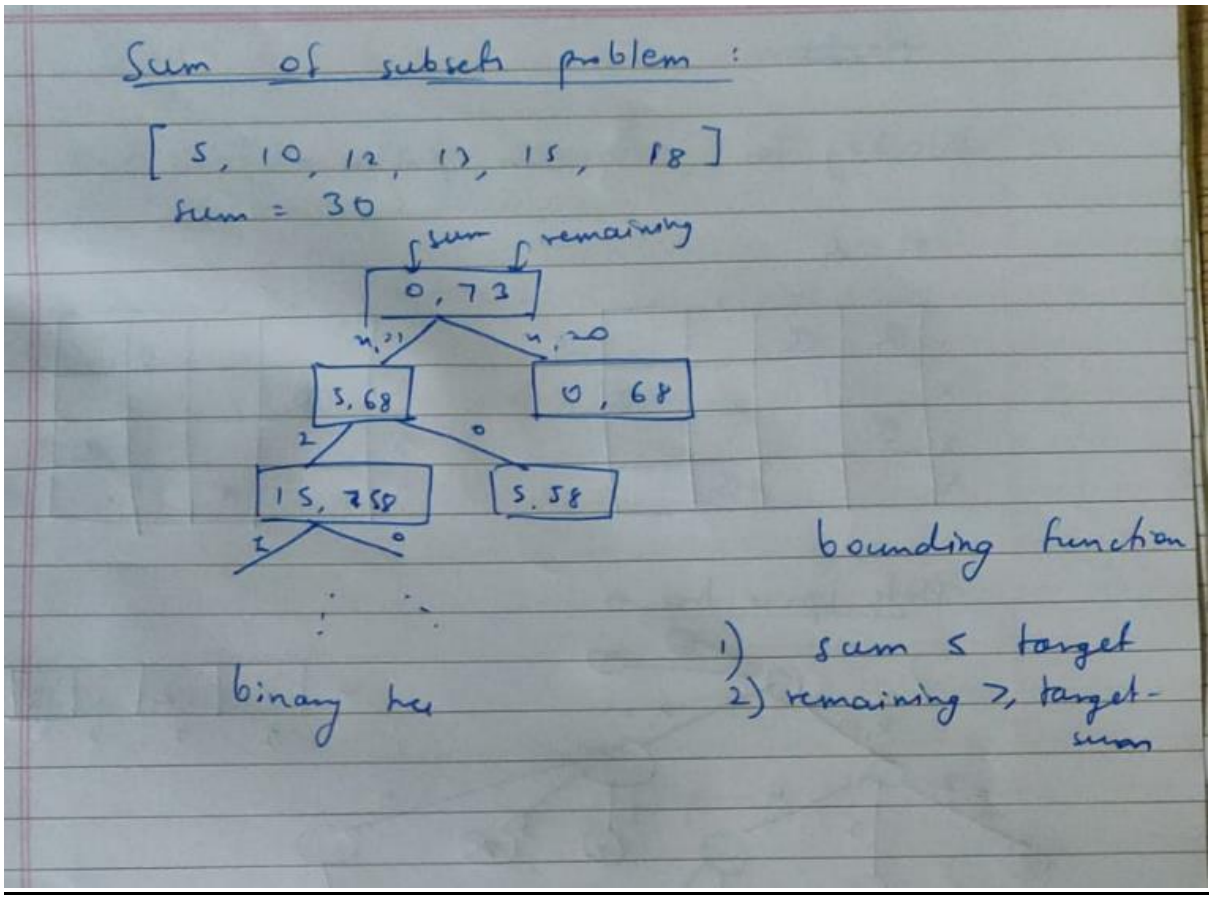
```
C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>.\subsum
Enter number of elements in array : 6
Enter the elements of the array :
5 10 12 13 15 18
Enter the target sum : 30

The subarrays which add up to the target sum are :
Solution 1 : 5 + 10 + 15
Solution 2 : 5 + 12 + 13
Solution 3 : 12 + 18

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>
```

```
C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>.\subsum
Enter number of elements in array : 8
Enter the elements of the array :
25 10 15 5 10 14 6 50
Enter the target sum : 25

The subarrays which add up to the target sum are :
Solution 1 : 25
Solution 2 : 10 + 15
Solution 3 : 10 + 5 + 10
Solution 4 : 15 + 10
Solution 5 : 5 + 14 + 6

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>
```

Conclusion:

- We used backtracking to solve the subset sum problem in O(2^n) time complexity.
- We used 2 bounding functions to improve the performance of our algorithm.

Rough Working :



Theory:

1. **Definition**: Backtracking is a general algorithmic technique that involves exploring all possible options to find a solution to a problem. It is often used for problems where the solution requires a sequence of steps or choices.
2. **Depth-First Search**: Backtracking is essentially a depth-first search (DFS) in a problem's solution space. It tries to exhaust all possibilities until a solution is found or all paths have been tried.
3. **Recursive Nature**: Backtracking is often implemented recursively, where the solution to a problem depends on solutions to smaller instances of the same problem.
4. **Pruning**: A key feature of backtracking is that it prunes the search tree by ruling out the sequences (of choices) that cannot possibly lead to a valid solution, thereby reducing the search space.
5. **Applications**: Backtracking is used in a variety of problems, including the Eight Queens problem, the Knight's Tour problem, the Rat in a Maze, N-Queens problem, Subset Sum, Hamiltonian Cycle, Sudoku, and many others.
6. **Efficiency**: While backtracking can be efficient in some cases, in the worst case it can involve exploring all possible sequences of steps or choices, leading to exponential time complexity. However, it is often more efficient than brute force enumeration of all complete sequences.
7. **Optimization**: Backtracking can often be optimized using techniques like memoization, or by using heuristics to guide the search, or by using problem-specific knowledge.