Assignment 1

Aim:
Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Theory:
In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$F_n = F_{n-1} + F_{n-2}$
With seed values

$F_0 = 0$ and $F_1 = 1$.
Program for n-th Fibonacci number Using recursion
Here we will use recursion function. The code defines a function Fibonacci(n) that calculates the nth Fibonacci number recursively. It checks for invalid input and returns the Fibonacci number based on the base cases (0 and 1) or by recursively calling itself with reduced values of n.
To find the fibonacci series without using recursion.
The program takes the first two numbers of the series along with the number of terms needed and prints the fibonacci series.
Problem Solution

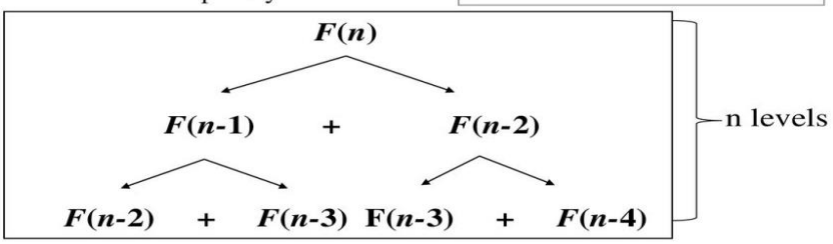
1. Take the first two numbers of the series and the number of terms to be printed from the user.
   2. Print the first two numbers.
   3. Use a while loop to find the sum of the first two numbers and then proceed the fibonacci series.
   4. Print the fibonacci series till n-2 is greater than 0.
   5. Exit.

Algorithm:

Fibonacci Numbers

- Computing the n^{th} Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - **Top-down approach**
    - Time complexity?

```
int Fib(int n)
{
   if (n <= 1)
      return 1;
   else
      return Fib(n - 1) + Fib(n - 2);
}
```



-

Write algorithm for Non recursive:

```
int Fibonacci2 (int N)
{
     if (N <= 2)
       return N;
     Let N1 = 1, N2 = 2, Total = 0;
     for (i = 3 to N)
     {
          Total = N1 + N2;
          N1 = N2;
          N2 = Total;
     }
     return Total;
}
```

Flowchart:

Time Complexity: $O(2^N)$ Find out timecomplexity
Auxiliary Space: O(N)

Code with Output: Both recursive and non recursive

Conclusion:

This is how we implement program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Assignment 2

Aim: Write a program to implement Huffman Encoding using a greedy strategy.

Theory: Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calls minHeapify(). So, the overall complexity is O(nlogn).
If the input array is sorted, there exists a linear time algorithm. We will soon be discussing this in our next post.
Space complexity :- O(N)
Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding(to be more precise the prefix codes).

Algorithm:

```
Algorithm Huffman (c)
{
  n= |c|

  Q = c
  for i<-1 to n-1

  do
  {

     temp <- get node ()

    left (temp] Get_min (Q) right [temp] Get Min (Q)
```

```
    a = left [templ b = right [temp]
    F [temp]<- f[a] + [b]
    insert (Q, temp)
  }
return Get_min (0)
}
```

Flowchart:

Program with output:

Conclusion: We implement Huffman Encoding using a greedy strategy.

Assignment 3

Aim: Write a program to solve a fractional Knapsack problem using a greedy method.

Theory: What is Greedy Algorithm?

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.

Fractional Knapsack Problem using Greedy algorithm:

An efficient solution is to use the Greedy approach.

The basic idea of the greedy approach is to calculate the ratio **profit/weight** for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it). This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

Algorithm:

greedy fractional-knapsack (P[1…n], W[1…n], X[1..n]. M)

/*P[1…n] and W[1…n] contain the profit and weight of the n-objects ordered such that X[1…n] is a solution set and M is the capacity of knapsack*/

{

For j ← 1 to n do
X[j]← 0
profit ← 0 // Total profit of item filled in the knapsack
weight ← 0 // Total weight of items packed in knapsacks
j ← 1
While (Weight < M) // M is the knapsack capacity

{

if (weight + W[j] =< M)
X[j] = 1
weight = weight + W[j]

```
else{
X[j] = (M – weight)/w[j]
weight = M

}
Profit = profit + p[j] * X[j]
j++;
} // end of while
}
```

Time Complexity: O(N * logN)
Auxiliary Space: O(N)

Flowchart:

Program with output:

Conclusion: Thus we solve a fractional Knapsack problem using a greedy method.

Assignment 4

Aim: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Theory:

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming. In this problem, we have a Knapsack that has a capacity m.

- There are items i1,i2,….,in each having weight w1,w2,…..wn and some benefit(value or profit) associated with it p1,p2,….,pn
- Our objective is to maximize the benefit such that the total weight inside the Knapsack is at most m.
- Since this is a 0-1 Knapsack problem, we can either take an entire item or reject it completely. We cannot break an item and fill the Knapsack.
- To solve 0/1 knapsack problem using dynamic programming use the following recursive formula:-

$fn(m)=max[fn-1(m);fn-1(m-wn)+pn]$

$fn(-m)=-\infty$

$f0(m)=0$

Example:-

- Assume that we have a Knapsack with max weight capacity m=8.

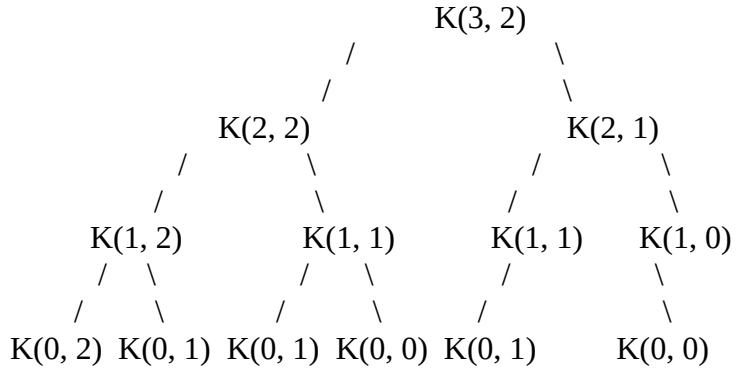
Our objective is to fill the Knapsack with items such that the benefit (profit) is maximum.

Algorithm:

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
```

```
    if vi + c[i-1, w-wi] then
      c[i, w] = vi + c[i-1, w-wi]
    else c[i, w] = c[i-1, w]
  else
    c[i, w] = c[i-1, w]
```

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.
n the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
weight[] = {1, 1, 1}, W = 2, profit[] = {10, 20, 30}



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Time complexity: O(n^2) because the while loop runs n times, and inside the while loop, there is another loop that also runs n times in the worst case.
Space complexity : O(n) because the queue stores the nodes, and in the worst case, all nodes are stored, so the size of the queue is proportional to the number of items, which is n.

Time Complexity: O(N), as only one path through the tree will have to be traversed in the best case and its worst time complexity is still given as O(2^N)

Flowchart:

Coding:

Conclusion: Thus we solve a 0-1 Knapsack dynamic strategy.

Assignment 5

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place Queens to generate the final n-queen‘s matrix

Theory:

In backtracking, we start with one possible move out of many available moves. We then try to solve the problem.

If we are able to solve the problem with the selected move then we will print the solution. Else we will backtrack and select some other move and try to solve it.

If none of the moves works out we claim that there is no solution for the problem.

What is N-Queen problem?

The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.
For example, the following is a solution for the 4 Queen problem.
The expected output is in the form of a matrix that has ‘**Q**‘s for the blocks where queens are placed and the empty spaces are represented by **‘.’** . For example, the following is the output matrix for the above 4-Queen solution.
. Q . .
. . . Q
Q . . .
. . Q .
N Queen Problem using Backtracking**:**
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**.
Below is the recursive tree of the above approach:
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:

```
IS-ATTACK(i, j, board, N)
 // checking in the column j
 for k in 1 to i-1
  if board[k][j]==1
   return TRUE

 // checking upper right diagonal
 k = i-1
 l = j+1
 while k>=1 and l<=N
  if board[k][l] == 1
   return TRUE
  k=k+1
  l=l+1

 // checking upper left diagonal
 k = i-1
 l = j-1
 while k>=1 and l>=1
  if board[k][l] == 1
   return TRUE
  k=k-1
  l=l-1

return FALSE
```

Time Complexity: O(N!)

Flowchart:

Coding:

Conclusion: Thus we solve Design n-Queens matrix having first Queen placed. Use backtracking to place Queens to generate the final n-queen‘s matrix

Assignment 6

Aim: Write a program for analysis of quick sort by using deterministic and randomized variant.

Theory:

QuickSort we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater than the pivot. Then we recursively call the same procedure for left and right subarrays.

Unlike merge sort, we don't need to merge the two sorted arrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort. Using a randomly generated pivot we can further improve the time complexity of QuickSort.

Algorithm:

```
partition(arr[], lo, hi)
   pivot = arr[hi]
   i = lo     // place for swapping
   for j := lo to hi – 1 do
      if arr[j] <= pivot then
         swap arr[i] with arr[j]
         i = i + 1
   swap arr[i] with arr[hi]
   return i
partition_r(arr[], lo, hi)
   r = Random Number from lo to hi
   Swap arr[r] and arr[hi]
   return partition(arr, lo, hi)
quicksort(arr[], lo, hi)
   if lo < hi
      p = partition_r(arr, lo, hi)
      quicksort(arr, lo , p-1)
      quicksort(arr, p+1, hi)
```

Time Complexity: O(N*N)
Auxiliary Space: O(N) // due to recursive call stack

Flowchart:

Coding:

Conclusion: Thus we implement program for analysis of quick sort by using deterministic and randomized variant.