ASSIGNMENT NO.: 1

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

INTRODUCTION:

❖ FIBONACCI NUMBERS

The Fibonacci numbers are the numbers in the following integer sequence. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …….

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

Fn = Fn-1 + Fn-2 with seed values .
F0 = 0 and F1 = 1.

❖ PROGRAM FOR NON-RECURSIVE FIBONACCI NUMBERS

```
class FibonacciExample1{
public static void main(String args[])
{
int n1=0,n2=1,n3,i,count=10;
System.out.print(n1+" "+n2);     //printing 0 and 1
for(i=2;i<count;++i)     //loop starts from 2 because 0 and 1 are already printed
{
        n3=n1+n2;
        System.out.print(" "+n3);
        n1=n2;
        n2=n3;
}    }    }
```

❖ OUTPUT:

```
0 1 1 2 3 5 8 13 21 34
```

Time Complexity and Space Complexity

- The time complexity of the above code is T(N), i.e., linear. We have to find the sum of two terms, and it is repeated n times depending on the value of n.
- The space complexity of the above code is O(N).

❖ PROGRAM FOR RECURSIVE FIBONACCI NUMBERS

```
class FibonacciExample2{
static int n1=0,n2=1,n3=0;
static void printFibonacci(int count){
   if(count>0){
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
        System.out.print(" "+n3);
        printFibonacci(count-1);
     }
 }
public static void main(String args[]){
int count=10;
System.out.print(n1+" "+n2);//printing 0 and 1
printFibonacci(count-2);//n-2 because 2 numbers are already printed
}
}
```

❖ OUTPUT:

```
0 1 1 2 3 5 8 13 21 34
```

Time and Space Complexity of Recursive Method

- The time complexity of the above code is T(2^N), i.e., exponential.
- The Space complexity of the above code is O(N) for a recursive series.

CONCLUSION: We have successfully studied java program for non-recursive and recursive to calculate Fibonacci numbers and about their time and space complexity.

ASSIGNMENT NO.: 2

AIM: Write a program to implement Huffman encoding using a greedy strategy.

INTRODUCTION:

❖ HUFFMAN CODING

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 most frequent character gets the smallest code and the least frequent character gets the largest code.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

❖ STEPS TO BUILD HUFFMAN TREE

Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

❖ PROGRAM FOR HUFFMAN ENCODING WITH GREEDY STRATEGY:

```
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;
class Huffman
{
```

```
        public static void printCode(HuffmanNode root, String s)
        {
                if (root.left == null && root.right == null  && Character.isLetter(root.c))
                   {
                        System.out.println(root.c + ":" + s);
                        return;
                }
                printCode(root.left, s + "0");
                printCode(root.right, s + "1");
        }
        public static void main(String[] args)
        {
                Scanner s = new Scanner(System.in);
                int n = 6;
                char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
                int[] charfreq = { 5, 9, 12, 13, 16, 45 };
                PriorityQueue<HuffmanNode> q = new PriorityQueue<HuffmanNode>(n,
new MyComparator());
                for (int i = 0; i < n; i++)
                   {
                        HuffmanNode hn = new HuffmanNode();
                        hn.c = charArray[i];
                        hn.data = charfreq[i];
                        hn.left = null;
                        hn.right = null;
                        q.add(hn);
                }
                HuffmanNode root = null;
                while (q.size() > 1)
                    {
                        HuffmanNode x = q.peek();
                        q.poll();
                        HuffmanNode y = q.peek();
                        q.poll();
                        HuffmanNode f = new HuffmanNode();
                        f.data = x.data + y.data;
                        f.c = '-';
                        f.left = x;
                        f.right = y;
                        root = f;
                        q.add(f);
                }
                printCode(root, "");
        }
}
class HuffmanNode
{
        int data;
        char c;
        HuffmanNode left;
```

```
        HuffmanNode right;
}
class MyComparator implements Comparator<HuffmanNode>
{
        public int compare(HuffmanNode x, HuffmanNode y)
        {
                return x.data - y.data;
        }
}
```

❖ **OUTPUT:**

CONCLUSION: We have successfully studied Java program to implement Huffman encoding using a greedy strategy.

ASSIGNMENT NO. : 3

AIM: Write a program to solve the fractional knapsack problem using greedy method.

INTRODUCTION:

❖ KNAPSACK PROBLEM

The knapsack problem is in combinatorial optimization problem. It appears as a sub problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

❖ APPLICATIONS

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

❖ FRACTIONAL KNAPSACK

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of i^{th} item wi>0wi>0
- Profit for i^{th} item pi>0pi>0 and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

0⩽xi⩽10⩽xi⩽1

The i^{th} item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

❖ ALGORITHM: GREEDY-FRACTIONAL-KNAPSACK (W[1..N], P[1..N], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

❖ ANALYSIS

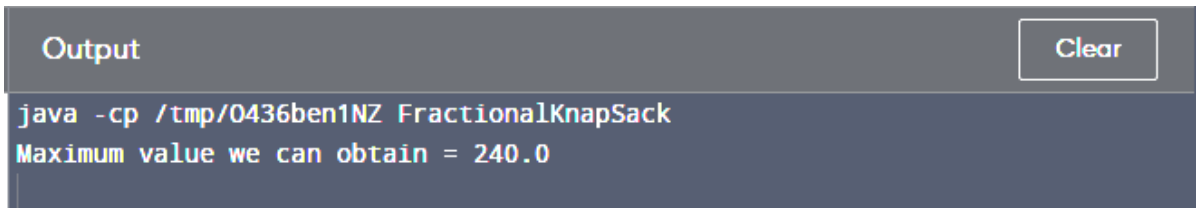
If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

❖ PROGRAM FOR FRACTIONAL KNAPSACK PROBLEM USING GREEDY METHOD:

```
// Java program to solve fractional Knapsack Problem
import java.util.Arrays;
import java.util.Comparator;
public class FractionalKnapSack {
        // function to get maximum value
        private static double getMaxValue(ItemValue[] arr,int capacity)
        {
         Arrays.sort(arr, new Comparator<ItemValue>() {
                        @Override
                        public int compare(ItemValue item1,ItemValue item2)
                        {
                        double cpr1= new Double((double)item1.value/
(double)item1.weight);
                        double cpr2= new Double((double)item2.value/
(double)item2.weight);
                        if (cpr1 < cpr2)
                                return 1;
                        else
                                return -1;
                        }
                });
                double totalValue = 0d;
                for (ItemValue i : arr) {
                        int curWt = (int)i.weight;
                        int curVal = (int)i.value;
                        if (capacity - curWt >= 0) {
```

```
                    capacity = capacity - curWt;
                    totalValue += curVal;
                }
                else {
                    double fraction= ((double)capacity / (double)curWt);
                    totalValue += (curVal * fraction);
                    capacity= (int)(capacity - (curWt * fraction));
                    break;
                }
            }
            return totalValue;
        }
        static class ItemValue {
        int value, weight;
         public ItemValue(int val, int wt)
            {
                this.weight = wt;
                this.value = val;
            }
        }
        // Driver's code
        public static void main(String[] args)
        {
         ItemValue[] arr = { new ItemValue(60, 10),new ItemValue(100, 20),
                                    new ItemValue(120, 30) };
          int capacity = 50;
          double maxValue = getMaxValue(arr, capacity);
          System.out.println("Maximum value we can obtain = "+ maxValue);
        }
}
```

❖ **OUTPUT:**



CONCLUSION: We have successfully studied Java program to solve the fractional knapsack problem using greedy method.

ASSIGNMENT NO.: 4

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

INTRODUCTION:

❖ KNAPSACK PROBLEM

The knapsack problem is in combinatorial optimization problem. It appears as a sub problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

❖ DYNAMIC PROGRAMMING:

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping sub-problems and optimal substructure property.
- The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler sub-problems, solving each sub-problem just once, and then storing their solutions to avoid repetitive computations.

❖ 0/1 KNAPSACK USING BRANCH AND BOUND

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.

❖ PROGRAM FOR HUFFMAN ENCODING WITH GREEDY STRATEGY:

```
import java.util.*;
class Item {
        float weight;
        int value;
        int idx;
        public Item() {}
        public Item(int value, float weight, int idx)
        {
                this.value = value;
                this.weight = weight;
                this.idx = idx;
        }
```

```
}
class Node {
	// Upper Bound: Best case (Fractional Knapsack)
	float ub;
	// Lower Bound: Worst case (0/1)
	float lb;
	// Level of the node in the decision tree
	int level;
	// Stores if the current  item is selected or not
	boolean flag;
	// Total Value: Stores the sum of the values of the items included
	float tv;
	// Total Weight: Stores the sum of the weights of included items
	float tw;
	public Node() {}
	public Node(Node cpy)
	{
		this.tv = cpy.tv;
		this.tw = cpy.tw;
		this.ub = cpy.ub;
		this.lb = cpy.lb;
		this.level = cpy.level;
		this.flag = cpy.flag;
	}
}
class sortByC implements Comparator<Node> {
	public int compare(Node a, Node b)
	{
		boolean temp = a.lb > b.lb;
		return temp ? 1 : -1;
	}
}
class sortByRatio implements Comparator<Item> {
	public int compare(Item a, Item b)
	{
		boolean temp = (float)a.value / a.weight > (float)b.value / b.weight;
		return temp ? -1 : 1;
	}
}
class knapsack {
	private static int size;
	private static float capacity;
	// Function to calculate upper bound (includes fractional part of the items)
	static float upperBound(float tv, float tw, int idx, Item arr[])
	{
		float value = tv;
		float weight = tw;
		for (int i = idx; i < size; i++) {
			if (weight + arr[i].weight
				<= capacity) {
```

```
                        weight += arr[i].weight;
                        value -= arr[i].value;
                    }
                    else {
                        value -= (float)(capacity- weight) / arr[i].weight* arr[i].value;
                        break;
                    }
                }
                return value;
            }
            // Calculate lower bound (doesn't include fractional part of items)
            static float lowerBound(float tv, float tw, int idx, Item arr[])
            {
                float value = tv;
                float weight = tw;
                for (int i = idx; i < size; i++) {
                    if (weight + arr[i].weight
                        <= capacity) {
                        weight += arr[i].weight;
                        value -= arr[i].value;
                    }
                    else {
                        break;
                    }
                }
                return value;
            }
            static void assign(Node a, float ub, float lb, int level, boolean flag, float tv, float tw)
            {
                a.ub = ub;
                a.lb = lb;
                a.level = level;
                a.flag = flag;
                a.tv = tv;
                a.tw = tw;
            }
            public static void solve(Item arr[])
            {
                // Sort the items based on the profit/weight ratio
                Arrays.sort(arr, new sortByRatio());
                Node current, left, right;
                current = new Node();
                left = new Node();
                right = new Node();
// min_lb -> Minimum lower bound of all the nodes explored final_lb -> Minimum lower
bound of all the paths that reached the final level
                float minLB = 0, finalLB= Integer.MAX_VALUE;
                current.tv = current.tw = current.ub= current.lb = 0;
                current.level = 0;
                current.flag = false;
```

```
                // Priority queue to store elements based on lower bounds
                PriorityQueue<Node> pq= new PriorityQueue<Node>(new sortByC());
                // Insert a dummy node
                pq.add(current);
// curr_path -> Boolean array to store at every index if the element is included or not
// final_path -> Boolean array to store the result of selection array when it reached the last
level
                boolean currPath[] = new boolean[size];
                boolean finalPath[] = new boolean[size]
                while (!pq.isEmpty()) {
                        current = pq.poll();
                        if (current.ub > minLB|| current.ub >= finalLB) {
                                continue;
                        }
                        if (current.level != 0)
                                currPath[current.level - 1] = current.flag;
                        if (current.level == size) {
                                if (current.lb < finalLB) {
                                        // Reached last level
                                        for (int i = 0; i < size; i++)
                                                finalPath[arr[i].idx]= currPath[i];
                                        finalLB = current.lb;
                                }
                                continue;
                        }
                        int level = current.level;
                        // right node -> Excludes current item
//Hence, cp, cw will obtain the value  of that of parent
                            assign(right, upperBound(current.tv, current.tw, level + 1, arr),
                                lowerBound(current.tv, current.tw, level + 1, arr),
                                level + 1, false,
                                current.tv, current.tw);
                        if (current.tw + arr[current.level].weight<= capacity) {
                                // left node -> includes current item
                                // c and lb should be calculated including the current item.
                                left.ub = upperBound(current.tv- arr[level].value, current.tw+
arr[level].weight, level + 1, arr);
                                left.lb = lowerBound(current.tv- arr[level].value, current.tw+
arr[level].weight, level + 1, arr);
                                assign(left, left.ub, left.lb, level + 1, true, current.tv -
arr[level].value, current.tw+ arr[level].weight);
                        }
                        // If the left node cannot be inserted
                        else {
                                  left.ub = left.lb = 1;
                        }
                        // Update minLB
                        minLB = Math.min(minLB, left.lb);
                        minLB = Math.min(minLB, right.lb);
                        if (minLB >= left.ub)
```

```
                        pq.add(new Node(left));
                    if (minLB >= right.ub)
                        pq.add(new Node(right));
                }
                System.out.println("Items taken"+ "into the knapsack are");
                for (int i = 0; i < size; i++) {
                    if (finalPath[i])
                        System.out.print("1 ");
                    else
                        System.out.print("0 ");
                }
                System.out.println("\nMaximum profit"+ " is " + (-finalLB));
        }
        // Driver code
        public static void main(String args[])
        {
                size = 4;
                capacity = 15;
                Item arr[] = new Item[size];
                arr[0] = new Item(10, 2, 0);
                arr[1] = new Item(10, 4, 1);
                arr[2] = new Item(12, 6, 2);
                arr[3] = new Item(18, 9, 3);

                solve(arr);
        }
}
```

❖ **OUTPUT:**

```
Items taken into the knapsack are :
1 1 0 1
Maximum profit is : 38
```

CONCLUSION: We have successfully studied Java program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

PRACTICAL NO.: 6

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

INTRODUCTION:

❖ QUICK SORT

Quicksort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot.
There are many different versions of quicksort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot.
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quicksort** is a partition (). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

❖ ALGORITHM FOR QUICK SORT

```
QUICKSORT (array A, start, end)    {
if (start < end)      {
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
```

❖ PROGRAM FOR QUICK SORT USING DETERMINISTIC VARIANT:

```
class QuickSort
{
        int partition(int arr[], int low, int high)
        {
                int pivot = arr[high];
                int i = (low-1); // index of smaller element
                for (int j=low; j<high; j++)
                {
```

```
                        // If current element is smaller than or equal to pivot
                        if (arr[j] <= pivot)
                        {
                                i++;
                                // swap arr[i] and arr[j]
                                int temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                        }
                }
                int temp = arr[i+1];
                arr[i+1] = arr[high];
                arr[high] = temp;
                return i+1;
        }
        void sort(int arr[], int low, int high)
        {
                if (low < high)
                {
                        //pi is partitioning index, arr[pi] is now at right place
                        int pi = partition(arr, low, high);
                        // Recursively sort elements before partition and after partition
                        sort(arr, low, pi-1);
                        sort(arr, pi+1, high);
                }
        }
        static void printArray(int arr[])
        {
                int n = arr.length;
                for (int i=0; i<n; ++i)
                        System.out.print(arr[i]+" ");
                System.out.println();
        }
        // Driver program
        public static void main(String args[])
        {
                int arr[] = {10, 7, 8, 9, 1, 5};
                int n = arr.length;
                QuickSort ob = new QuickSort();
                ob.sort(arr, 0, n-1);
                System.out.println("sorted array");
                printArray(arr);
        }
}
```

❖ OUTPUT:

```
java -cp /tmp/Bl317q8css QuickSort

Sorted array :1 5 7 8 9 10
```

❖ RANDOMIZED QUICK SORT

Randomized quicksort solves this problem by first randomly shuffling the values in the array, and then running quicksort, using the first value as the pivot every time. Now, shuffling might produce a sorted array, and this pivot choice would then result in a slow O(n^2) quicksort. But as experience tells you, shuffling almost never produces sorted data! In fact, since you can re-arrange n values in n! ways (n factorial), the probability that one re-arrangement gives you sorted data is 1/n!, which is effectively zero for n bigger than 10.

❖ PROGRAM FOR QUICK SORT USING RANDOMIZED VARIANT:

```
import java.util.Random;
public class Randomized_Quick_Sort
{
   public static int N = 20;
   public static int[] sequence = new int[N];
   public static void QuickSort(int left, int right)
   {
      if (right - left <= 0)
         return;
      else
      {
         Random rand = new Random();
         int pivotIndex = left + rand.nextInt(right - left + 1);
         swap(pivotIndex, right);
         int pivot = sequence[right];
         int partition = partitionIt(left, right, pivot);
         QuickSort(left, partition - 1);
         QuickSort(partition + 1, right);
      }
   }
   public static int partitionIt(int left, int right, long pivot)
   {
      int leftPtr = left - 1;
      int rightPtr = right;
      while (true)
      {
         while (sequence[++leftPtr] < pivot);
         while (rightPtr > 0 && sequence[--rightPtr] > pivot);
         if (leftPtr >= rightPtr)
            break;
         else
            swap(leftPtr, rightPtr);
      }
      swap(leftPtr, right);
      return leftPtr;
   }
   public static void swap(int dex1, int dex2)
```

```
    {
        int temp = sequence[dex1];
        sequence[dex1] = sequence[dex2];
        sequence[dex2] = temp;
    }
    static void printSequence(int[] sorted_sequence)
    {
        for (int i = 0; i < sorted_sequence.length; i++)
            System.out.print(sorted_sequence[i] + " ");
    }
    public static void main(String args[])
    {
        System.out
                .println("Sorting of randomly generated numbers using RANDOMIZED QUICK
SORT");
        Random random = new Random();
        for (int i = 0; i < N; i++)
            sequence[i] = Math.abs(random.nextInt(100));
        System.out.println("\nOriginal Sequence: ");
        printSequence(sequence);
        System.out.println("\nSorted Sequence: ");
        QuickSort(0, N - 1);
        printSequence(sequence);
    }
}
```

- **OUTPUT:**

```
java -cp /tmp/Bl317q8css Randomized_Quick_Sort
Sorting of randomly generated numbers using RANDOMIZED QUICK
    SORTOriginal Sequence: 43 91 42 9 35 63 20 13 66 88 28 15 69 97 19
    57 14 62 65 30
Sorted Sequence:
9 13 14 15 19 20 28 30 35 42 43 57 62 63 65 66 69 88 91 97
```

CONCLUSION: We have successfully studied Java program for analysis of quick sort by using deterministic and randomized variant.