Jayawant Shikshan Prasarak Mandal

Table of Contents

Name of the Institute : BSIOTR Campus : Wagholi

Subject : Laboratory Practice - III

Assignment No.1

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

✳ Objective :-
Students should be able to perform non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

✳ Prerequisite :-

1. Basic of Python or Java Programming
2. Concept of Recursive and Non-recursive functions.
3. Execution flow of calculate Fibonacci numbers.
4. Basic of Time and Space complexity.

JSPM
Page No.:

Theory :-

- Introduction to Fibonacci numbers :-

- The Fibonacci series, named after Italian mathematician Leonardo Pisano Bogollo, later known as Fibonacci, is a series (sum) formed by Fibonacci numbers denoted as Fn. The numbers in Fibonacci sequence are given as : 0, 1, 1, 2, 3, 5, 8, 13, 21, 38...

- In a Fibonacci series, every term is the sum of the preceding two terms, starting from 0 and 1 as first and second term. In some old references, the term '0' might be omitted.

- What is the Fibonacci Series?

- The Fibonacci series is the sequence of numbers (also called Fibonacci numbers), where every number is the sum of the preceding two numbers, such that the first two terms are '0' and '1'.

- In some older versions of the series, the term '0' might be omitted. A Fibonacci series can thus be given as, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,... It can be thus be observed that every

term can be calculated by adding the two terms before it.

- Given the first term, F0 and second term, F1 as '0' and '1', the third term here can be given as, F2 = 0 + 1 = 1

Similarly,
F3 = 1 + 1 = 2
F4 = 2 + 1 = 3

Given a number n, print n-th Fibonacci Number.

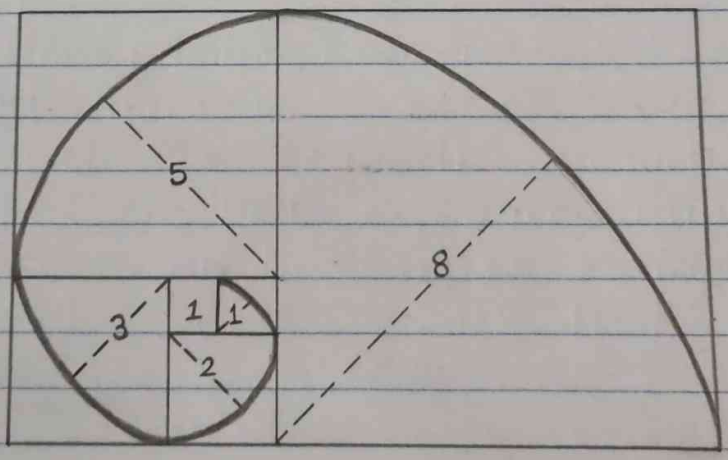


Fig. Fibonacci Number

Fibonacci Sequence Formula

The Fibonacci sequence of numbers "Fn" is defined using the recursive relation the seed values F0 = 0 and F1 = 1:

$F_n = F_{n-1} + F_{n-2}$

Here, the sequence is defined using two different parts, such as kick-off and recursive relation.

The kick-off part is F0 = 0 and F1 = 1.

The recursive relation part is $F_n = F_{n-1} + F_{n-2}$.

It is noted that the sequence starts with 0 rather than 1. So, F5 should be the 6th term of the sequence.

Methods:-

1. Non-recursion:

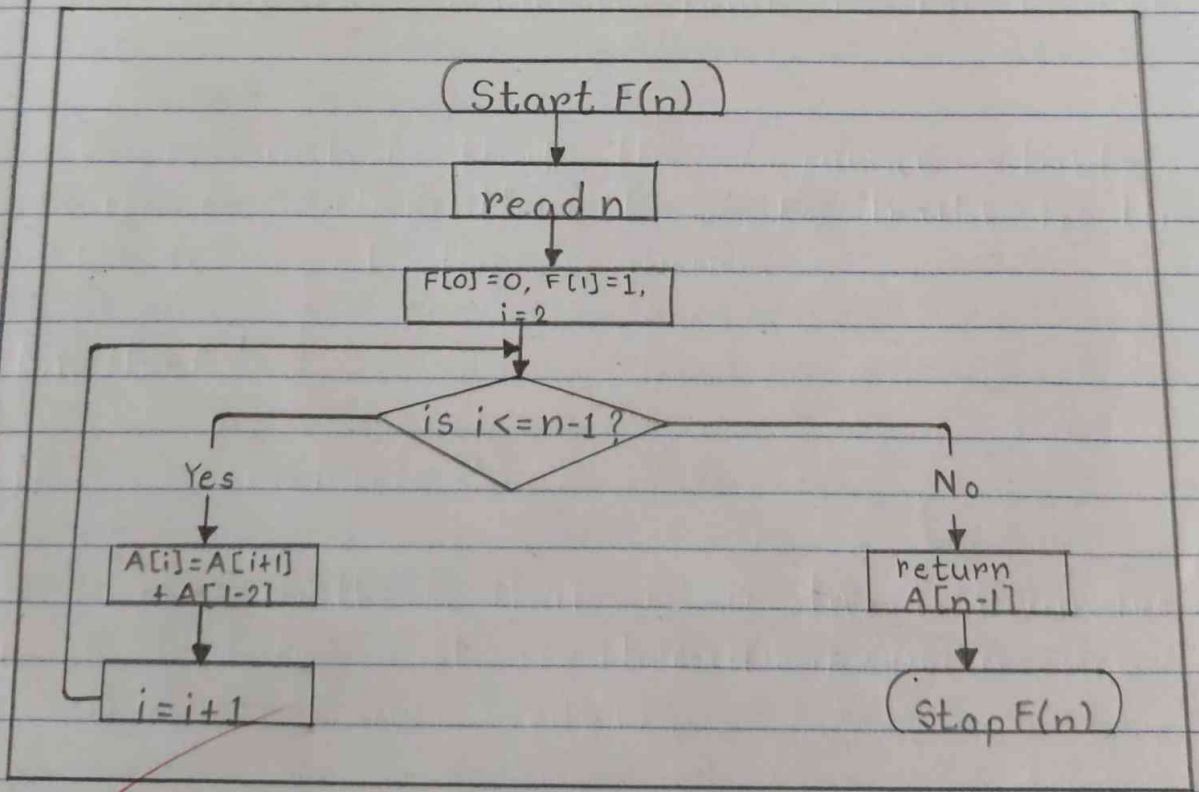
A simple method that is a direct recursive implementation of mathematical recurrence relation is given above.

First, we'll store 0 and 1 in F[0] and F[1], respectively.

Next, we'll iterate through array positions 2 to n-1. At each position i, we store the sum of the two preceding array values in F[i].

Finally, we return the value of F[n-1], giving us the number at position n in the sequence.

Here's a visual representation of this process:



```
# Program to display the Fibonacci sequence
  up to n-th term  nterms = int(input("How
  many terms?"))

# first two terms

 n1, n2 =0,1
 count =0

# check if the number of terms is valid
 if nterms <=0;
     print ("Please enter a positive integer")

#if there is only one term, return n1
 elif nterms ==1;
    print ("Fibonacci sequence upto", nterms,":")
    print (n1)

# generate fibonacci sequence
   else:
    print ("Fibonacci sequence:")
    while count < nterms;
     print (n1)
     nth = n1 +n2

# update values
  n1 = n2
  n2 = nth
 count+=1.
```

JSPM
Page No. :

* Output :-

How many terms? 7

Fibonacci sequence :

0
1
1
2
3
5
8

2. Recursion :

Start by defining F(n) as the function that returns the value of Fn.

To evaluate F(n) for n>1, we reduce our problem into two smaller problems of the same kind : F(n-1) and F(n-2).

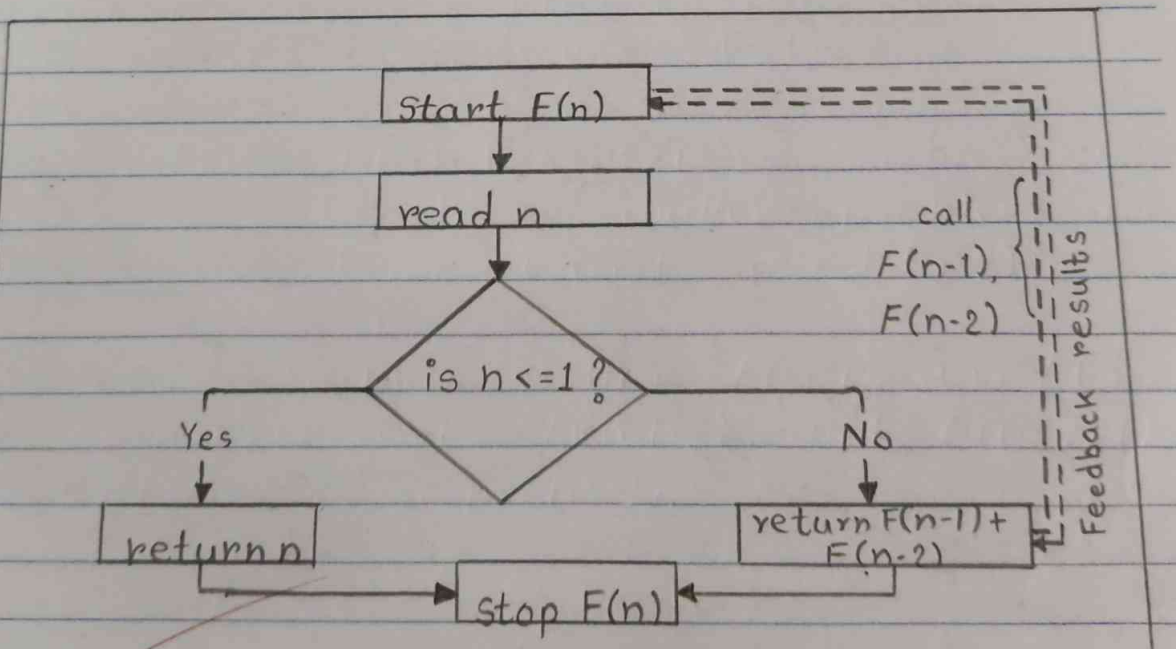
We can further reduce F(n-1) and F(n-2) to F(n-1)-1) and F((n-1-2); and F((n-2)-1) and F((n-2)-2), respectively.

If we repeat this reduction, we'll eventually reach our known base cases and thereby, obtain a solution to F(n).

Employing this logic, our algorithm for F(n) will have two steps;

1. Check if $n \leq 1$. If so, return n.
2. Check if $n > 1$. If so, call our function F with inputs n-1 and n-2, and return the sum of the two results.

Here's a visual representation of this algorithm:



```
# Python program to display the Fibonacci sequence

def recur_fibo(n):
 if n <= 1:
     return n
 else:
     return (recur_fibo(n-1) + recur_fibo(n-2))

nterms = 7

# check if the number of terms is valid

if nterms <= 0:
   prit ("Please enter a positive integer")
else:
     print ("Fibonacci sequence:")
     for i in range (nterms):
       print (recur_fibo(i))
```

✳ Output:

Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8

✳ Conclusion:- In this way we have learned concept of fibonacci series.

```
import java.io.*;
class FibonacciNonRecursive {
  // Function to print N Fibonacci Numbers
  static void Fibonacci(int N) {
    int num1 = 0, num2 = 1;
    for (int i = 0; i < N; i++) {
      // Print the number
      System.out.print(num1 + " ");
      // Swap
      int num3 = num2 + num1;
      num1 = num2;
      num2 = num3;
    }
  }
  // Driver Code
  public static void main(String args[]) {
    int N = 10;
    Fibonacci(N);
  }
}
```

```
import java.io.*;
class FibonacciRecursive {
  // Function to print the Fibonacci series
  static int fib(int n) {
    if (n <= 1)
      return n;
    return fib(n - 1) + fib(n - 2);
  }
  // Driver Code
  public static void main(String args[]) {
    int N = 10;
    for (int i = 0; i < N; i++) {
      System.out.print(fib(i) + " ");
    }
  }
}
```

user@user: ~
user@user:~$ javac p.java
user@user:~$ java p
0 1 1 2 3 5 8 13 21 34 user@user:~$

```
user@user:~
user@user:~$ javac p.java
user@user:~$ java p
0 1 1 2 3 5 8 13 21 34 user@user:~$
```

JSPM
Page No. :

Assignment No. 2.

* **Title :-** Write a program to implement Huffman Encoding using a greedy strategy.

* **Objective :-** Students should be able to understand and solve Huffman Encoding and analyze time and space complexity using a greedy strategy.

* **Theory :**

- **What is a Greedy Method ?**

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverse the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

- **Advantages of Greedy Approach :-**
  - The algorithm is easier to describe.
  - This algorithm can perform better than other algorithms (but, not in all cases).

- **Disadvantages of Greedy Approach :-**
  - As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm.
  - For example, suppose we want to find the longest path in the graph below from root to leaf.

- **Greedy Algorithm :-**

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Huffman Encoding

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 frequency of corresponding characters.

The most frequent character gets the smallest code and the least frequent character gets the largest code. The variable-length codes assigned to input characters are Prefix Codes means the largest code. 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 prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

» 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:

Input is array of unique characters along with their frequency of occurances & output is Huffman Tree.

1. Create a leaf nodes for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the leaf frequent character is at root).

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted nodes as its leaft child & the other extracted node as a right child. Add this node to the min heap.

4. Repeat steps #2 and #3 until the heap contains only one node. The remaining node is the root node and the tree

is complete.

Algorithm for Huffman code:

Input :- Number of message with frequency count.

Output :- Huffman merge tree.

1. Begin
2. Let Q be the priority queue.
3. Q = {initialize prority queue with frequenci-
   -es of all symbol or message}
4. Repeat n-1 times.
5. Create a new node Z
6. X = exact_min (Q)
7. Y = exact_min (Q)
8. Frequency (Z) = Freqeuency (X) + Frequency (Y);
9. Insert (Z, Q)
10. End repeat
11. Return (extract_min (Q))
12. End.

Time Complexity-

O(nlogn) where n is the number of unique characters. If there are n nodes, extract Min() is called 2*(n-1) times. extractMin() takes O(logn) time as it calles minHeapify().

So, overall complexity is O(nlogn).

Thus, Overall time complexity of Huffman Coding becomes O(nlogn).

If the input array is sorted, there exists a linear time algorithm.

Conclusion :-

In this way concept of Huffman Encoding is explored using greedy method.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;

class HuffmanCoding {
  // Print Huffman codes
  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");
  }

  // Main function
  public static void main(String[] args) {
    char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int[] charfreq = { 5, 9, 12, 13, 16, 45 };
    PriorityQueue<HuffmanNode> q = new PriorityQueue<>(6, new MyComparator());

    for (int i = 0; i < charArray.length; 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.poll();
          HuffmanNode y = 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;
      }
    }
```

user@user:~
user@user:~$ javac p.java
user@user:~$ java p
f:0
c:100
d:101
a:1100
b:1101
e:111
user@user:~$

Assignment No. 3.

* Title :- Write a program to solve a fractio-nal knapsack problem using a greedy method.

* Objective :- To analyze time and space comple-xity of fractional knapsack problem using a greedy method.

* Theory :-

- Knapsack Problem :

Your are given the following :-

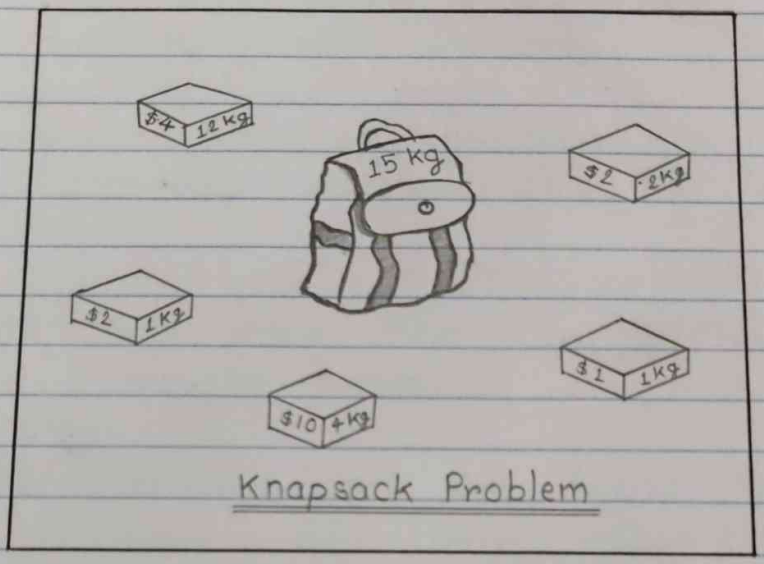
- A knapsack (Kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.
- The problem states :-

Which items should be placed into the knapsack such that :-

- The value or profit obtained by putting

the items into the knapsack is maximum.

- And the weight limit of the knapsack does not exceed.



Knapsack Problem

✳ Knapsack Problem Variants:
Knapsack problem has the following two variants

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

⟷ Knapsack Problem (Fractional):

In Fractional knapsack Problem,

- As the name suggests, items are divisible here.

- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using the Greedy Method.

Fractional Knapsack Problem:-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01 : For each item, compute its value/weight ratio.

Step-02 : Arrange all the items in decreasing order of their value / weight ratio.

Step-03 : Start putting the items into the knapsack beginning from the item with the highest ratio. as many items as you can into the knapsack.

- Example:- Find the optimal solution for the fractional knapsack problem making use of greedy approach.

Given :-

$n = 5$

$w = 60\,kg$

$(w_1, w_2, w_3, w_{4,5}) = (5, 10, 15, 22, 25)$

$(b_1, b_2, b_3, b_4, b_5) = (30, 40, 45, 77, 90)$

→ Solution -

Step-01: Compute the value/weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step-02: Sort all the items in decreasing order of their value/weight ratio.

I1, I2, I3, I4, I5

(6) (4) (3.6) (3.5) (3)

Step-03: Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	Items in Knapsack	Cost
60	∅	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

- Knapsack weight left to be filled is 20kg but item-4 has weight of 22kg.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items-

$< I1, I2, I5, (20/22)\ I4 >$

Total cost of the knapsack

$= 160 + (20/27) \times 77$

$= 160 + 70$

$= 230$ units

Algorithm - Fractional Knapsack:

- Greedy-fractional-knapsack (w, v, W)

```
1. for i = 1 to n
2. do x[i] = 0
3. weight = 0
4. While weight < W
5. do i = best remaining item
6. If weight + w[i] ≤ W
7. then x[i] = 1
8. weight = weight + w[i]
9. else
10. x[i] = [w - weight) / w[i]
11. weight = W
12. return x
```

Time Complexity -

- The main time taken step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of Quick Sort is O(n log n).

- Therefore, total time taken including the sort is O (n log n).

✳ Conclusion :-

In this way concept of Fractional Knapsack is explained using greedy method.

```
import java.util.Arrays;
import java.util.Comparator;

public class FractionalKnapsack {
  // Inner class to represent an item with weight, value, and value-to-weight ratio
  static class Item {
    double weight, value, ratio;

    // Constructor to initialize an item with weight, value, and ratio
    public Item(double weight, double value) {
      this.weight = weight;
      this.value = value;
      this.ratio = value / weight;
    }
  }

  // Method to calculate the maximum value that can be carried in the knapsack
  public static double getMaxValue(Item[] items, double capacity) {
    // Sort items by their value-to-weight ratio in descending order
    Arrays.sort(items, new Comparator<Item>() {
      @Override
      public int compare(Item o1, Item o2) {
        return Double.compare(o2.ratio, o1.ratio);
      }
    });

    double totalValue = 0;

    // Iterate through the sorted items
    for (Item item : items) {
      if (capacity == 0) break;
```

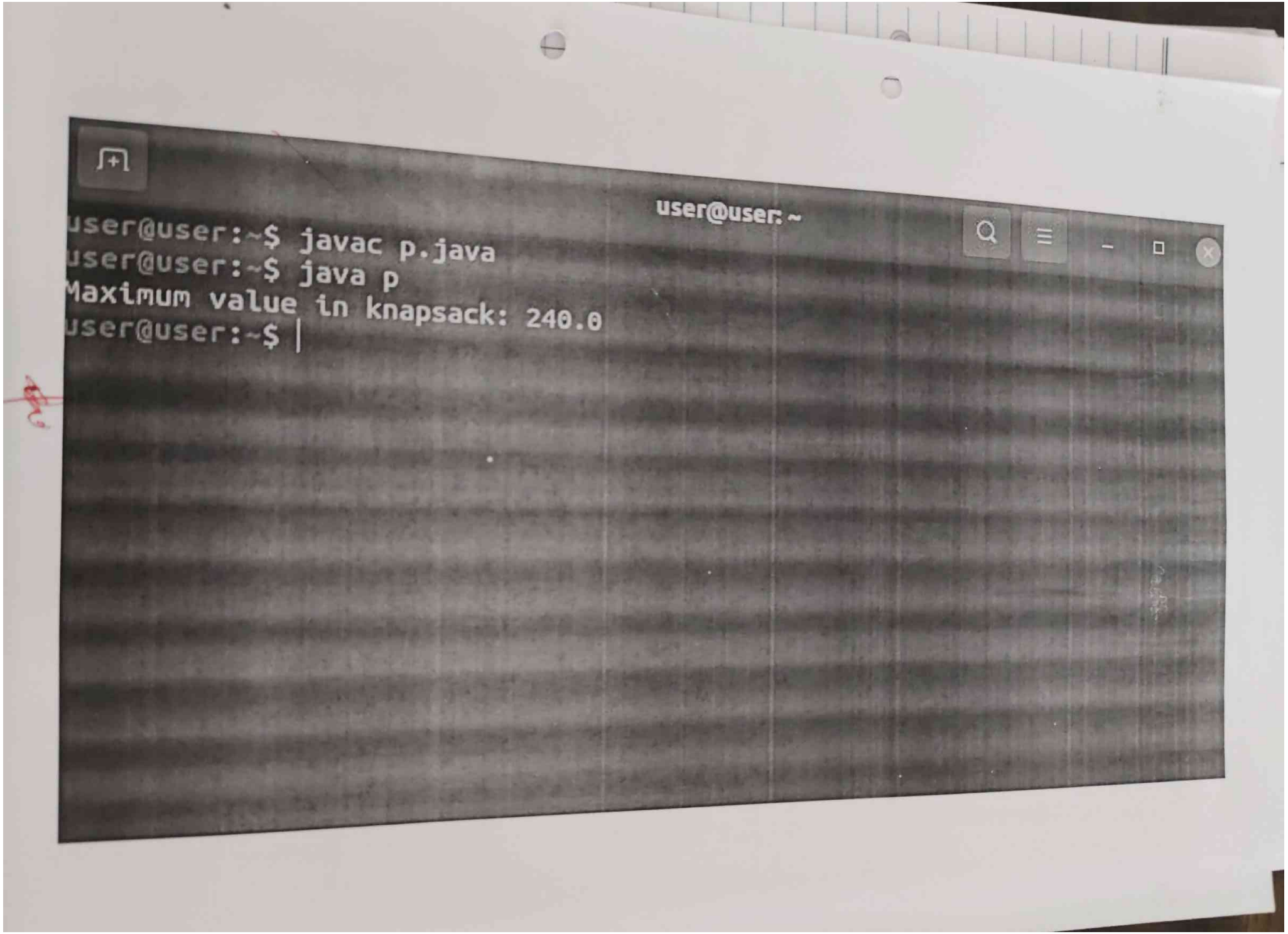
```
            // If the item can be completely added to the knapsack
            if (capacity >= item.weight) {
              capacity -= item.weight;
              totalValue += item.value;
            } else {
              // If only part of the item can be added to the knapsack
              totalValue += item.value * (capacity / item.weight);
              capacity = 0;
            }
          }

          return totalValue;
        }

        // Main method
        public static void main(String[] args) {
          // Array of items with specified weights and values
          Item[] items = {
            new Item(10, 60),
            new Item(20, 100),
            new Item(30, 120)
          };

          // Knapsack capacity
          double capacity = 50;

          // Calculate the maximum value that can be carried and print the result
          double maxValue = getMaxValue(items, capacity);
          System.out.println("Maximum value in knapsack: " + maxValue);
        }
```


user@user: ~
user@user:~$ javac p.java
user@user:~$ java p
Maximum value in knapsack: 240.0
user@user:~$

Assignment No. 4

* Title : Write a program to solve a 0-1 Knapsack problem using dynamic programming strategy.

* Objective : To analyze time and space complexity of 0-1 Knapsack problem using dynamic programming.

* Theory :-

- What is Dynamic Programming?

Dynamic Programming is also used in optimization programes. Like divide and conquer method, Dynamic Programming solves problems by combining the solutions of subproblems.

Dynamic Programming algorithm solves each sub-problem just onee and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming.

These properties are overlapping sub-problems and optimal substracture.

Dynamic Programming also combines solutions to sub-problems. is needed repeatedly. The computed solutions are stored in a table, so that three don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

- Steps of Dynamic Programming Approach:

Dynamic Programming algorithm is designed using the following four steps-

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion
- Construct an optimal solution from the computed information.

→» Applications of Dynamic Programming Approach :

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

→» Time Complexity :-

- Each entry of the table requires constant time $\theta(1)$ for its computation
- It takes $O(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $O(n)$ time for tracing the solution since tracing process traces the n rows.
- Thus, overall $O(nw)$ time is taken to solve 0/1 Knapsack problem using dynamic programming.

✳ Conclusion :- In this way we have explored Concept of 0/1 Knapsack using Dynamic approach.

```
public class KnapsackDP {
  // Function to solve the 0-1 Knapsack problem
  public static int knapsack(int[] weights, int[] values, int capacity) {
    int n = weights.length;
    int[][] dp = new int[n + 1][capacity + 1];

    // Building the DP table
    for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
        if (i == 0 || w == 0) {
          dp[i][w] = 0; // Base case: no items or zero capacity
        } else if (weights[i - 1] <= w) {
          // Include the item or exclude it, choose the maximum value
          dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
        } else {
          dp[i][w] = dp[i - 1][w]; // Cannot include the item
        }
      }
    }

    // The bottom-right corner contains the maximum profit
    return dp[n][capacity];
  }

  // Main method
  public static void main(String[] args) {
    int[] weights = {10, 20, 30}; // Weights of the items
    int[] values = {60, 100, 120}; // Values of the items
    int capacity = 50; // Maximum capacity of the knapsack

    // Solve the knapsack problem
```

```
        int maxProfit = knapsack(weights, values, capacity);
        System.out.println("Maximum profit: " + maxProfit);
    }
}
```

```
user@user:~
user@user:~$ javac p.java
user@user:~$ java p
220
user@user:~$
```

Assignment No. 5

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

* Objective :- To analyze time and space complexity of quick sort by using deterministic and randomized variant.

* Theory :-

→ What is a Randomized Algorithm?

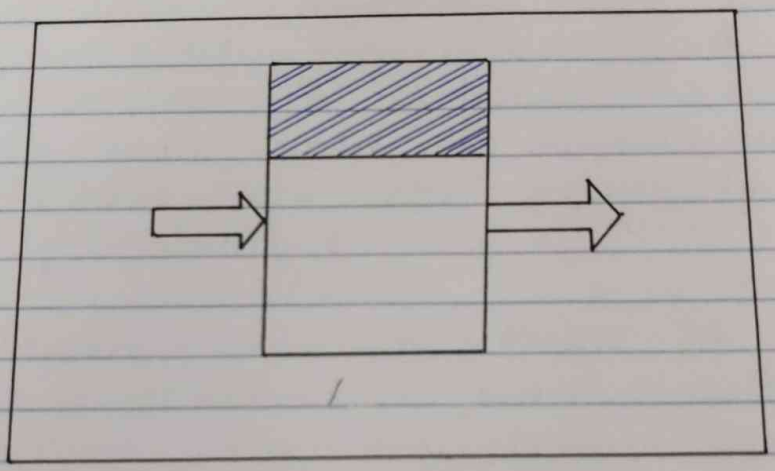
- An algorithm that uses random number to decide what to do next anywhere in its logic is called Randomized Algorithm.

- For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array).

- Typically, this randomness is used to reduce time complexity or space

space complexity in other standard algorithms.

- Randomized algorithm for a problem is usually simpler and more efficient than its deterministic counterpart.
- The output or the running time are functions of the input and random bits chosen.



✳ Types of Randomized Algorithms:-

1. Las Vegas Algorithms:

- These algorithm always produce correct or optimum result.

- Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value.
- For example, Randomized QuickSort always sorts an input array and expected worst case time completexity of QuickSort is O(nLogn).
- A Las Vegas algorithm fails with some probability, but we can tell when it fails. In particular, we can run it again until it succeeds, which means that we can eventually succeed with probability 1.
- Alternatively, we can think of a Las Vegas algorithm as an algorithm that runs for an unpredictable amount of time but always succeeds.

2. Monte Carlo Algorithms:

- Produce correct or optimum result with some probability.
- These algorithms have deterministic running time and it is generally easier

to find out worst case time complexity.

- For example karger's Algorithm produces minimum cut with probability greater than or equal to $1/n^2$ (n is number of vertices) and has worst case time complexity as O(E).
- A Monte carlo algorithm fails with some probability, but we can't tell when it fails.
- The polynomial equility-testing algorithm in an example of a Monte carlo algorithm.

✳ Randomized Quick Sort Algorithm:

- Randomized algorithm have huge applications in Cryptography.
- Load Balancing:
- Number-Theoretic Applications : Primality Testing.
- Data Structure : Hashing, Sorting, Searching.

Analysis of Randomized Quick Sort:

The running time of quicksort depends mostly on the number of comparisons performed in all calls to the Randomized-Partition routine. Let X denotes the random variable counting the number of comparisons in all to Randomized-Partition.

Let z_i denote the i-th smallest element of A[1..n],
Thus A[1..n] sorted is $\langle z_1, z_2, \ldots, z_n \rangle$.

Let $z_{ij} = [z_i, \ldots, z_j]$ denote that the set of elements between z_i and z_j, including these elements.

$X_{ij} = 1(z_i, \ldots, z_j)$ denote the set of elements between z_i and z_j, including these elements.

$X_{ij} = 1(z_i$ is compared to $z_j)$.

Thus, X_{ij} is an indicator random variable for the event that the i-thv smallest and the j-th smallest elements of A are compared in an execution of quicksort.

Number of Comparisons :-

Since each pair of elements is compared at most once by quicksort, the number X of comparisons is given by

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Therefore, the expected number of comparisons is

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ is compared to } z_j]$$

Expected Number of Comparisons.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$

It follows the expected running time to Randomized - Quicksort is $O(n \log n)$.

It is unlikely that this algorithm will choose a terribly unbalanced partition each time, so the performance is very good almost all the time.

✳ Conclusion :-

In this way we have explored Concept of quick sort by using deterministic and randomized variant.

```
import java.util.Random;

public class QuickSortAnalysis {
  // Deterministic QuickSort
  public static void deterministicQuickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = partition(arr, low, high);
      deterministicQuickSort(arr, low, pi - 1);
      deterministicQuickSort(arr, pi + 1, high);
    }
  }

  // Randomized QuickSort
  public static void randomizedQuickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = randomizedPartition(arr, low, high);
      randomizedQuickSort(arr, low, pi - 1);
      randomizedQuickSort(arr, pi + 1, high);
    }
  }

  // Partition method used in deterministic QuickSort
  private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high]; // Pivot element
    int i = low - 1; // Index of smaller element

    for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
        i++;
        swap(arr, i, j);
      }
```

```
            }
            swap(arr, i + 1, high);
            return i + 1;
        }

        // Randomized partition for Randomized QuickSort
        private static int randomizedPartition(int[] arr, int low, int high) {
            Random rand = new Random();
            int randomIndex = low + rand.nextInt(high - low + 1);
            swap(arr, randomIndex, high); // Move random element to the end
            return partition(arr, low, high);
        }

        // Utility method to swap elements in the array
        private static void swap(int[] arr, int i, int j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }

        // Utility method to generate a random array
        public static int[] generateRandomArray(int size, int range) {
            Random rand = new Random();
            int[] arr = new int[size];
            for (int i = 0; i < size; i++) {
                arr[i] = rand.nextInt(range);
            }
            return arr;
        }

// Method to copy an array (used to ensure same input for both variants)
```

```
    public static int[] copyArray(int[] arr) {
        int[] newArr = new int[arr.length];
        System.arraycopy(arr, 0, newArr, 0, arr.length);
        return newArr;
    }

    // Main method to analyze the performance
    public static void main(String[] args) {
        int size = 10000; // Size of the array
        int range = 10000; // Range of random numbers

        // Generate a random array for testing
        int[] originalArray = generateRandomArray(size, range);

        // Test Deterministic QuickSort
        int[] arr1 = copyArray(originalArray);
        long startTime = System.nanoTime();
        deterministicQuickSort(arr1, 0, arr1.length - 1);
        long endTime = System.nanoTime();
        System.out.println("Deterministic QuickSort time: " + (endTime - startTime) + " ns");

        // Test Randomized QuickSort
        int[] arr2 = copyArray(originalArray);
        startTime = System.nanoTime();
        randomizedQuickSort(arr2, 0, arr2.length - 1);
        endTime = System.nanoTime();
        System.out.println("Randomized QuickSort time: " + (endTime - startTime) + " ns");
    }
}
```

user@user:~
user@user:~$ javac p.java
user@user:~$ java p
Deterministic QuickSort time: 4387768 ns
Randomized QuickSort time: 7189222 ns
user@user:~$

JSPM
Page No. :

Assignment No. 6
Mini Project

✳ Title :-
Implementation of Naive String Matching and Rabin-Karp Algorithm for String Matching.

✳ Objective :-

- To implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching.
- To observe the difference in the working of both algorithms for the same input.

✳ Theory :-

→ Naive String Matching Algorithm :-

The Naive string matching algorithm is a simple string matching algorithm that checks for a pattern in a text by sliding the pattern over the text one character at a time. It has a time complexity

of O((n-m+1) * m), where n is the length of the text and m is the length of the pattern.

→ Rabin - Karp Algorithm :-

The Rabin - Karp algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text. It has an average time complexity of O(n+m), making it more efficient than the Naive algorithm for large texts.

✳ Implementation :-

→ Naive String Matching Algorithm:

```
def naive_string_matching(text, pattern):
n = len(text)
m = len(pattern)
for i in range(n-m+1):
match = True
for j in range(m):
    if text[i+j] ≠ pattern[j]:
```

```
            match = False
            break
    if match
        return i
    return -1.
```

→ Rabin - Karp Algorithm:

```
def rabin_karp (text, pattern):
    n = len(text)
    m = len (pattern)
    d = 256
    q = 101
    pattern_hash = 0
    text_hash = 0
    h = 1

for _ in range (m-1):
    h = (h * d) % q

for i in range (m):
    pattern_hash = (d * pattern_hash +
                    ord (pattern [i])) % q
    text_hash = (d * test_hash +
                 ord (text[i])) % q
for i in range (n-m+1):
    if pattern_hash = text_hash
    match = True
    for j in range (m):
```

```
        if text[i+j] ≠ pattern[j]:
            match = False
            break

        if match
            retun i

    if i < n-m:
        text_hash = (d+ (text_hash - ord(text[i])
            if text_s hash <0: text
        hash <0
        text_human t=a

    return -1.
```

* Conclusion:
We have successfully demonstrated the Naive String matching algorithm and the Rabin-Karp algorithm for String matching.

mini-projet-group-b-1

October 15, 2024

[36]:
```
#Name: Shriharsh J. Deshmukh
#Roll No. 71 Div. 'A'
```

[37]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

[38]:
```
# Load the dataset
df = pd.read_csv('Stock_data.csv')
```

[39]:
```
# Perform EDA
print(df.head())
print(df.info())
print(df.describe())
```

```
         Date    Open    High     Low    Last   Close  Total Trade Quantity  \
0  2018-09-28  234.05  235.95  230.20  233.50  233.75                3069914
1  2018-09-27  234.55  236.80  231.10  233.80  233.25                5082859
2  2018-09-26  240.00  240.00  232.50  235.00  234.25                2240909
3  2018-09-25  233.30  236.75  232.00  236.25  236.10                2349368
4  2018-09-24  233.55  239.20  230.75  234.00  233.30                3423509

   Turnover (Lacs)
0          7162.35
1         11859.95
2          5248.60
3          5503.90
4          7999.55
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2035 entries, 0 to 2034
Data columns (total 8 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 2035 non-null   object
```

```
 1   Open                  2035 non-null   float64
 2   High                  2035 non-null   float64
 3   Low                   2035 non-null   float64
 4   Last                  2035 non-null   float64
 5   Close                 2035 non-null   float64
 6   Total Trade Quantity  2035 non-null   int64
 7   Turnover (Lacs)       2035 non-null   float64
dtypes: float64(6), int64(1), object(1)
memory usage: 127.3+ KB
None
```

	Open	High	Low	Last	Close
count	2035.000000	2035.000000	2035.000000	2035.000000	2035.00000
mean	149.713735	151.992826	147.293931	149.474251	149.45027
std	48.664509	49.413109	47.931958	48.732570	48.71204
min	81.100000	82.800000	80.000000	81.000000	80.95000
25%	120.025000	122.100000	118.300000	120.075000	120.05000
50%	141.500000	143.400000	139.600000	141.100000	141.25000
75%	157.175000	159.400000	155.150000	156.925000	156.90000
max	327.700000	328.750000	321.650000	325.950000	325.75000

	Total Trade Quantity	Turnover (Lacs)
count	2.035000e+03	2035.000000
mean	2.335681e+06	3899.980565
std	2.091778e+06	4570.767877
min	3.961000e+04	37.040000
25%	1.146444e+06	1427.460000
50%	1.783456e+06	2512.030000
75%	2.813594e+06	4539.015000
max	2.919102e+07	55755.080000

[40]:
```
# Plot the stock prices over time
plt.figure(figsize=(10,6))
plt.plot(df['Date'], df['Close'])
plt.title('Stock Prices Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.show()
```

[41]:
```
# Prepare the data for time series analysis
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
```

[42]:
```
# Split the data into training and testing sets
train_data, test_data = train_test_split(df, test_size=0.2, random_state=42)
```

[43]:
```
# Develop the predictive model
X_train = train_data.drop('Close', axis=1)
y_train = train_data['Close']
X_test = test_data.drop('Close', axis=1)
y_test = test_data['Close']

model = LinearRegression()
model.fit(X_train, y_train)
```

[43]: LinearRegression()

[44]:
```
# Make predictions on the test data
predictions = model.predict(X_test)
```

[45]:
```
# Evaluate the model
mse = mean_squared_error(y_test, predictions)
print(f'Mean Squared Error: {mse:.2f}')
```

3

```
Mean Squared Error: 0.16
```

[46]:
```
future_data = pd.DataFrame({
    'Open': [100],
    'High': [120],
    'Low': [80],
    'Last': [110],  # Assuming this value based on domain knowledge
    'Total Trade Quantity': [10000],  # You need to provide a reasonable value
    'Turnover (Lacs)': [50.5]  # Similarly, include a value for turnover
})
```

[47]:
```
future_prediction = model.predict(future_data)
print(f'Predicted Stock Price: {future_prediction[0]:.2f}')
```

```
Predicted Stock Price: 108.71
```