**A1- Fibonacci numbers.**

**Non-Recursive Approach**

In [4]:

**def** fibonacci\_iterative(n):

**if** n **<=** 1:

**return** n

a, b **=** 0, 1

**for** i **in** range(2, n **+** 1):

a, b **=** b, a **+** b

**return** b

*# Example usage:*

n **=** int(input("Enter number: "))

result **=** fibonacci\_iterative(n)

print(f"The {n}th Fibonacci number is {result}.")

**Non-Recursive Approach with Execution Time**

In [11]:

**import** time

start **=** time**.**time()

**def** fibonacci\_iterative(n):

**if** n **<=** 1:

**return** n

a, b **=** 0, 1

**for** i **in** range(2, n **+** 1):

a, b **=** b, a **+** b

**return** b

*# Example usage:*

n **=** int(input("Enter number: "))

result **=** fibonacci\_iterative(n)

print(f"The {n}th Fibonacci number is {result}.")

end **=** time**.**time()

print("Execution time is: {}ms"**.**format((end**-**start)**\***10**\*\***3))

**Recursive Approach**

In [3]:

**def** fibonacci\_recursive(n):

**if** n **<=** 1:

**return** n

**return** fibonacci\_recursive(n **-** 1) **+** fibonacci\_recursive(n **-** 2)

*# Example usage:*

n **=** int(input("Enter number: "))

result **=** fibonacci\_recursive(n)

print(f"The {n}th Fibonacci number is {result}.")

**Recursive Approach with Execution Time**

In [1]:

**import** time

start **=** time**.**time()

**def** fibonacci\_recursive(n):

**if** n **<=** 1:

**return** n

**return** fibonacci\_recursive(n **-** 1) **+** fibonacci\_recursive(n **-** 2)

*# Example usage:*

n **=** int(input("Enter number: "))

result **=** fibonacci\_recursive(n)

print(f"The {n}th Fibonacci number is {result}.")

end **=** time**.**time()

print("Execution time is: {}ms"**.**format((end**-**start)**\***10**\*\***3))

**A2-**

class Node:

"""A Huffman Tree Node"""

def \_\_init\_\_(self, freq\_, symbol\_, left\_=None, right\_=None):

# frequency of symbol

self.freq = freq\_

# symbol name (character)

self.symbol = symbol\_

# node left of current node

self.left = left\_

# node right of current node

self.right = right\_

# tree direction (0/1)

self.huff = ""

def print\_nodes(node, val=""):

"""Utility function to print huffman codes """

# huffman code for current node

new\_val = val + str(node.huff)

# if node is not an edge node then traverse inside it

if node.left:

print\_nodes(node.left, new\_val)

if node.right:

print\_nodes(node.right, new\_val)

# if node is edge node then display its huffman code

if not node.left and not node.right:

print(f"{node.symbol} -> {new\_val}")

chars = ["a", "b", "c", "d", "e", "f"]

freq = [5, 9, 12, 13, 16, 45]

nodes = [Node(freq[x], chars[x]) for x in range(len(chars))]

while len(nodes) > 1:

# sort all the nodes in ascending order based on their frequency

nodes = sorted(nodes, key=lambda x: x.freq)

# pick 2 smallest nodes

left = nodes[0]

right = nodes[1]

# assign directional value to these nodes

left.huff = 0

right.huff = 1

# combine the 2 smallest nodes to create new node as their parent

newNode = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

# remove the 2 nodes and add their parent as new node among others

nodes.remove(left)

nodes.remove(right)

nodes.append(newNode)

print("Characters :", f'[{", ".join(chars)}]')

print("Frequency :", freq, "\n\nHuffman Encoding:")

print\_nodes(nodes[0])

**A3-**

**import** time

start **=** time**.**time()

**def** fractional\_knapsack(items, capacity):

algo\_start **=** time**.**time()

*# Calculate the value-to-weight ratio for each item*

item\_value\_ratio **=** [(item[1] **/** item[0], item) **for** item **in** items]

*# Sort items by value-to-weight ratio in descending order*

item\_value\_ratio**.**sort(reverse**=True**)

total\_value **=** 0 *# Total value of selected items*

knapsack **=** [] *# Items selected for the knapsack*

**for** value\_per\_weight, item **in** item\_value\_ratio:

**if** capacity **>=** item[0]: *# If the item can fit entirely*

knapsack**.**append((item, 1)) *# Add the whole item*

total\_value **+=** item[1]

capacity **-=** item[0]

**else**: *# If the item can only fit partially*

fraction **=** capacity **/** item[0]

knapsack**.**append((item, fraction)) *# Add a fraction of the item*

total\_value **+=** item[1] **\*** fraction

**break** *# The knapsack is now full*

algo\_end **=** time**.**time()

algo\_exec **=** (algo\_end**-**algo\_start)**\***10**\*\***3

**return** total\_value, knapsack, algo\_exec

*# Example usage:*

*# (X, Y): Where X = Item Weight, Y = Item Profit*

items **=** []

n **=** int(input("How many items to add: "))

**for** i **in** range(n):

print("\nEnter details for item-{}"**.**format(i**+**1))

item\_w **=** int(input("Enter item weight: "))

item\_p **=** int(input("Enter item profit: "))

items**.**append((item\_w, item\_p))

print("\nItems are: {}"**.**format(items))

capacity **=** int(input("Enter capacity: "))

max\_value, selected\_items, algo\_execution\_time **=** fractional\_knapsack(items, capacity)

print("\nMaximum value: {}"**.**format(max\_value))

print("Selected items:")

**for** item, fraction **in** selected\_items:

print(" {}: Fraction = {}"**.**format(item,round(fraction,2)))

end **=** time**.**time()

print("\nAlgorithm Execution time is: {}ms"**.**format(algo\_execution\_time))

print("Program Execution time is: {}ms"**.**format((end**-**start)**\***10**\*\***3))

OUTPUT:-

How many items to add: 3

Enter details for item-1

Enter item weight: 2

Enter item profit: 10

Enter details for item-2

Enter item weight: 3

Enter item profit: 17

Enter details for item-3

Enter item weight: 2

Enter item profit: 5

Items are: [(2, 10), (3, 17), (2, 5)]

Enter capacity: 3

Maximum value: 17.0

Selected items:

(3, 17): Fraction = 1

(2, 10): Fraction = 0.0

Algorithm Execution time is: 0.0ms

Program Execution time is: 14130.146026611328ms

**A5-**

**import** time

start **=** time**.**time()

**def** print\_solution(board):

**for** row **in** board:

print(" "**.**join(row))

print("\n")

**def** is\_safe(board, row, col, n):

*# Check if no Queen can attack in the same column*

**for** i **in** range(row):

**if** board[i][col] **==** 'Q':

**return** **False**

*# Check if no Queen can attack in the left diagonal*

**for** i, j **in** zip(range(row, **-**1, **-**1), range(col, **-**1, **-**1)):

**if** board[i][j] **==** 'Q':

**return** **False**

*# Check if no Queen can attack in the right diagonal*

**for** i, j **in** zip(range(row, **-**1, **-**1), range(col, n)):

**if** board[i][j] **==** 'Q':

**return** **False**

**return** **True**

**def** solve\_n\_queens(board, row, n):

**if** row **==** n:

*# All Queens are placed successfully, print the solution*

print\_solution(board)

**return**

**for** col **in** range(n):

**if** is\_safe(board, row, col, n):

board[row][col] **=** 'Q'

solve\_n\_queens(board, row **+** 1, n)

board[row][col] **=** '.' *# Backtrack*

**def** n\_queens(n):

**if** n **<=** 0:

**return** []

board **=** [['.' **for** \_ **in** range(n)] **for** \_ **in** range(n)]

solve\_n\_queens(board, 0, n)

n **=** int(input("Enter number of Queens: "))

n\_queens(n)

end **=** time**.**time()

print("\nExecution time is: {}ms"**.**format((end**-**start)**\***10**\*\***3))

**A4-**

**import** time

start **=** time**.**time()

**def** knapsack\_dynamic\_programming(values, weights, capacity):

algo\_start **=** time**.**time()

n **=** len(values)

*# Create a table to store the maximum values for different capacities*

dp **=** [[0] **\*** (capacity **+** 1) **for** \_ **in** range(n **+** 1)]

*# Fill the table using dynamic programming*

**for** i **in** range(n **+** 1):

**for** w **in** range(capacity **+** 1):

**if** i **==** 0 **or** w **==** 0:

dp[i][w] **=** 0

**elif** weights[i **-** 1] **<=** w:

dp[i][w] **=** max(values[i **-** 1] **+** dp[i **-** 1][w **-** weights[i **-** 1]], dp[i **-** 1][w])

**else**:

dp[i][w] **=** dp[i **-** 1][w]

*# Backtrack to find the items included in the knapsack*

items **=** []

i, j **=** n, capacity

**while** i **>** 0 **and** j **>** 0:

**if** dp[i][j] **!=** dp[i **-** 1][j]:

items**.**append(i **-** 1)

j **-=** weights[i **-** 1]

i **-=** 1

items**.**reverse()

algo\_end **=** time**.**time()

algo\_exec **=** (algo\_end**-**algo\_start)**\***10**\*\***3

**return** dp[n][capacity], items, algo\_exec

*# Example usage*

n **=** int(input("How many items to add: "))

values **=** []

weights **=** []

**for** i **in** range(n):

print("\nEnter details for item-{}"**.**format(i))

item\_v **=** int(input("Enter item value: "))

values**.**append(item\_v)

item\_w **=** int(input("Enter item weight: "))

weights**.**append(item\_w)

print("\nItems are:\nValues = {}\nWeights = {}"**.**format(values, weights))

capacity **=** int(input("Enter capacity: "))

max\_value, selected\_items, algo\_execution\_time **=** knapsack\_dynamic\_programming(values, weights, capacity)

print("\nMaximum Value: {}"**.**format(max\_value))

print("Selected Items: {}"**.**format(selected\_items))

end **=** time**.**time()

print("\nAlgorithm execution time is: {}"**.**format(algo\_execution\_time))

print("Execution time is: {}ms"**.**format((end**-**start)**\***10**\*\***3))

How many items to add: 3

Enter details for item-0

Enter item value: 150

Enter item weight: 25

Enter details for item-1

Enter item value: 112

Enter item weight: 29

Enter details for item-2

Enter item value: 95

Enter item weight: 20

Items are: Values = [150, 112, 95]

Weights = [25, 29, 20]

Enter capacity: 50

Maximum Value: 245

Selected Items: [0, 2]

Algorithm execution time is: 0.9999275207519531

Execution time is: 24930.551528930664ms

The time complexity of the provided code is O(n \* capacity), and the space complexity is also O(n \* capacity).

**A6-**

import random

import time

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[0]

left = [x for x in arr[1:] if x <= pivot]

right = [x for x in arr[1:] if x > pivot]

return quick\_sort(left) + [pivot] + quick\_sort(right)

def randomized\_quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot\_index = random.randint(0, len(arr) - 1)

pivot = arr[pivot\_index]

left = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x <= pivot]

right = [x for x in arr[:pivot\_index] + arr[pivot\_index+1:] if x > pivot]

return randomized\_quick\_sort(left) + [pivot] + randomized\_quick\_sort(right)

def analyze\_quick\_sort(arr, variant="deterministic"):

start\_time = time.time()

if variant == "deterministic":

sorted\_arr = quick\_sort(arr)

elif variant == "randomized":

sorted\_arr = randomized\_quick\_sort(arr)

else:

print("Invalid variant specified")

return

end\_time = time.time()

execution\_time = end\_time - start\_time

return sorted\_arr, execution\_time

# Test the analysis with a sample array

arr = []

n = int(input("How many elements: "))

for i in range(n):

element = int(input("Enter Element: "))

arr.append(element)

print("Given array is: ",arr)

sorted\_arr\_det, time\_det = analyze\_quick\_sort(arr, variant="deterministic")

sorted\_arr\_rand, time\_rand = analyze\_quick\_sort(arr, variant="randomized")

print("\nDeterministic Quick Sort:")

print("Sorted Array:", sorted\_arr\_det)

print("Execution Time: {:.6f} seconds".format(time\_det))

print("\nRandomized Quick Sort:")

print("Sorted Array:", sorted\_arr\_rand)

print("Execution Time: {:.6f} seconds".format(time\_rand))