Experiment No: 01

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

Objective:

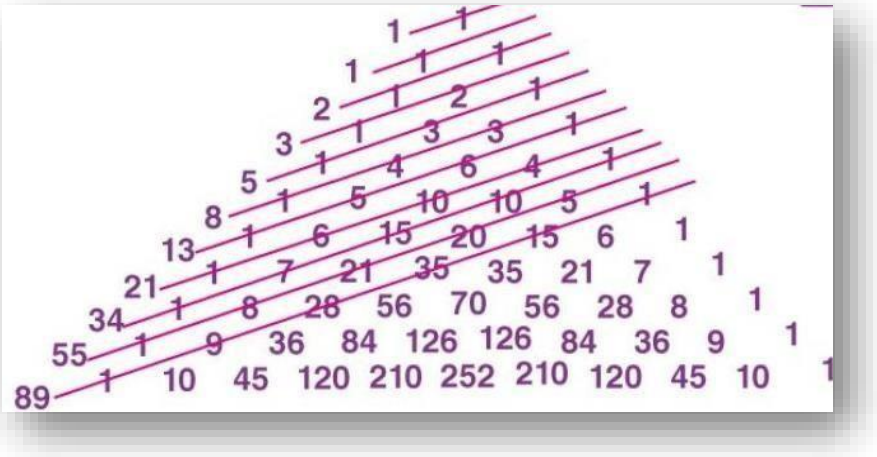
1. To understand the concept of Fibonacci Numbers.
2. To analyze the Time and space complexity.

Theory:

In Mathematics, the Fibonacci numbers are the numbers ordered in a distinct Fibonacci sequence. These numbers were introduced to represent the positive numbers in a sequence following a defined pattern. The list of the numbers in the Fibonacci series is represented by the recurrence relation: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞.

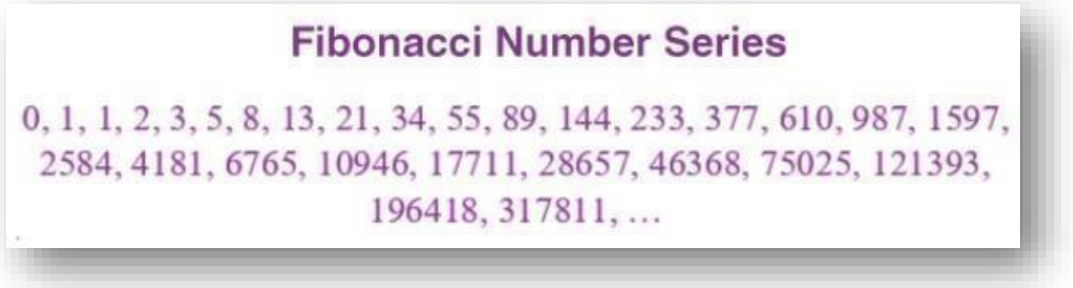
What is the Fibonacci Number?

A Fibonacci number is a series of numbers in which each Fibonacci number is obtained by adding the two preceding numbers. It means that the next number in the series is the addition of two previous numbers. Let the first two numbers in the series be taken as 0 and 1. By adding0 and 1, we get the third number as 1. Then by adding the second and the third number (i.e.) 1 and 1, we get the fourth number as 2, and similarly, the process goes on. Thus, we get the Fibonacci series as 0, 1, 1, 2, 3, 5, 8, Hence, the obtained series is called the Fibonacci number series.



Fibonacci Series List:

The list of numbers of the Fibonacci Sequence is given below. This list is formed by using the formula, which is mentioned in the above definition.



Fibonacci Numbers Formula:

The sequence of Fibonacci numbers can be defined as:

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

Where F_n is the nth term or number

...F_{n-1} is the $(n-1)^{th}$ term.

F_{n-2} is the $(n-2)^{th}$ term.

From the equation, we can summarize the definition as, the next number in the sequence, is the sum of the previous two numbers present in the sequence, starting from 0 and 1.

Fibonacci Number Properties:

The following are the properties of the Fibonacci numbers.

- In the Fibonacci series, take any three consecutive numbers and add those numbers. When you divide the result by 2, you will get the three numbers. For example, take 3 consecutive numbers such as 1, 2, 3. When you add these numbers, i.e. 1+ 2+ 3 = 6. When 6 is divided by 2, the result is3, which is 3.
- Take four consecutive numbers other than “0” in the Fibonacci series. Multiply the outer number and also multiply the inner number. You will get the difference “1” when you subtract these numbers. For example, take 4 consecutive numbers such as 2, 3, 5, 8. Multiply the outer numbers, i.e. 2(8), and multiply the inner number, i.e. 3(5). Now subtract these two numbers, i.e. 16-15 =1. Thus, the difference is 1.

Fibonacci Numbers Examples:

Question 1:

Write the first 6 Fibonacci numbers starting from 0 and 1.

Solution:

As we know, the formula for the Fibonacci sequence is;

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

Since the first term and second term are known to us, i.e. 0 and 1. Thus, $F_0 = 0$ and $F_1 = 1$ Hence, Third term, F2 = F0 + F1 = 0+1 = 1

Fourth term, F3 = F_2+F_1 = 1 + 1 = 2

Fifth term, F_4 = F_3+F_2 = 1+2 = 3 Sixth

term, F_5 = F_4+F_3 = 3 + 2 = 5

So, the first six terms of the Fibonacci sequence are 0,1,1,2,3,5.

i Recursive approach to calculate Fibonacci numbers

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation Fn = Fn-1 + Fn-2.

Recursive Algorithm:

Algorithm Fibonacci(n)

{if (n <= 1) return n;

else

return Fibonacci (n - 1) + Fibonacci (n - 2);}

Analysis of Time Complexity:

In the recursive Fibonacci program, the function calls itself twice for each input value n > 1, leading to an exponential growth in the number of function calls.

The time complexity of this recursive approach is approximately $O(2^n)$ because, for each level of recursion, it branches into two more recursive calls.

T(n) = T(n-1) + T(n-2) + c

= 2T(n-1) + c //from the approximation T(n-1) ~ T(n-2)

= 2*(2T(n-2) + c) + c

= 4T(n-2) + 3c

= 8T(n-3) + 7c

= 2 k * T(n - k) + (2 k - 1)*c

Let's find the value of k for which: n - k = 0 k = n T(n) =

2 n * T(0) + (2 n - 1)*c

= 2 n * (1 + c) - c T(n) = 2^n

Code:

```
#recursive  program to calculate fibonacci numbers
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
n = 10  # You can change this to any non-negative integer
result = fibonacci_recursive(n)
print(f"The {n}-th Fibonacci number is: {result}")
```

Output:

```
The 10-th Fibonacci number is: 55
```

ii Non-recursive approach to calculate Fibonacci numbers

It calculates Fibonacci numbers iteratively using a list to store intermediate results.

Algorithm:

Step 1: Define the Fibonacci function that takes an integer n as input.
Step 2: Check if n is less than or equal to 1. If true, return n as the result.
Step 3: Initialize a list fib of length (n + 1) with all elements initially set to 0.
Step 4: Set fib [1] to 1 because the Fibonacci of 1 is 1.

Step 5: Use a loop to iterate from i = 2 to n. Inside the loop:

- Calculate fib[i] by adding fib [i - 1] and fib [i - 2].

- This loop fills the fib list with Fibonacci numbers from 2 to n.

Step 6: Fib [n] contains the nth Fibonacci number after the loop finishes.

Step 7: Return fib[n] as the result.

Analysis of Time Complexity:

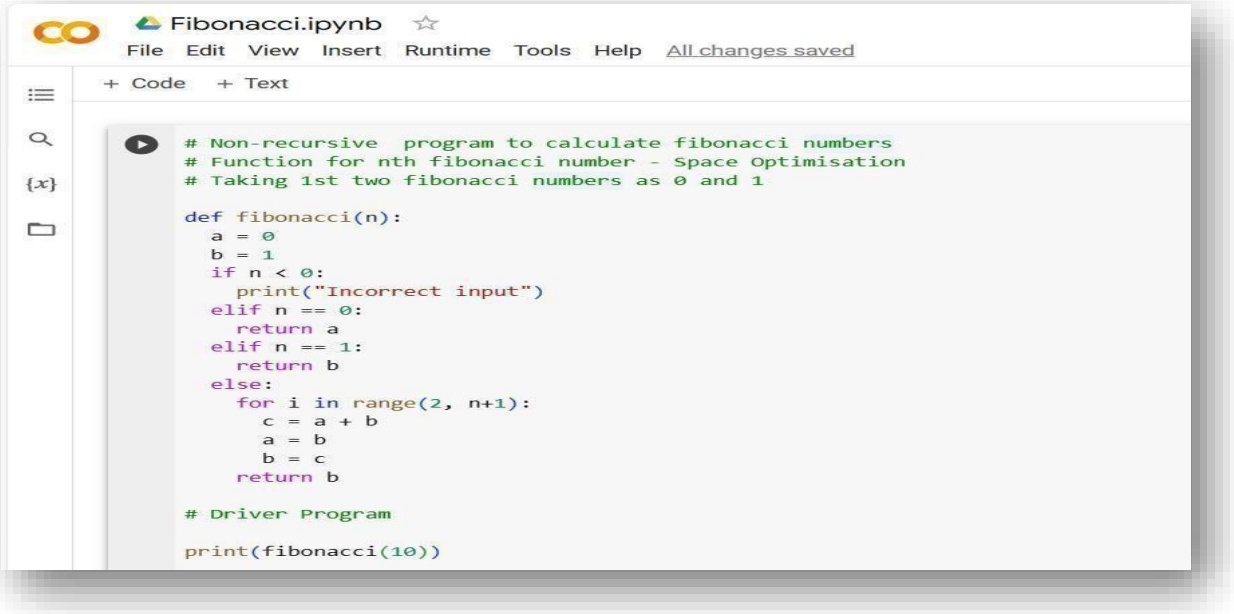
The loop in the Fibonacci function runs for n iterations, where n is the input number. We perform constant time operations (additions and assignments) inside the loop. Therefore, the time complexity of this non-recursive Fibonacci program is O(n).

Analysis of Space Complexity:

We create a list fib of size n + 1 to store all intermediate Fibonacci numbers.

Hence, the space complexity is O(n) because the size of the list grows linearly with the input n.

Code:



Output:

Conclusion:

We have studied Recursive and Non-Recursive ways to Calculate Fibonacci Numbers. the non-recursive approach is the most efficient and straightforward way to calculate Fibonacci numbers, making it suitable for most situations. The recursive approach is simple but becomes inefficient for larger inputs. The memorized recursive approach improves the time complexity and is a better choice when recursion is preferred, making it suitable for all values of n.

Experiment No: 02

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

Objective:

- Concept of Huffman Encoding
- Analysis of time complexity

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, which 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 the 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".

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.

Algorithm:

The method that is used to construct optimal prefix code is called **Huffman coding**.

This algorithm builds a tree in a bottom-up manner. We can denote this tree by **T**

Let, |c| be a number of leaves

|c| -1 are a number of operations required to merge the nodes. Q is the priority queue that can be used while constructing a binary heap.

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]
```

***Steps to build Huffman Tree**:*

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

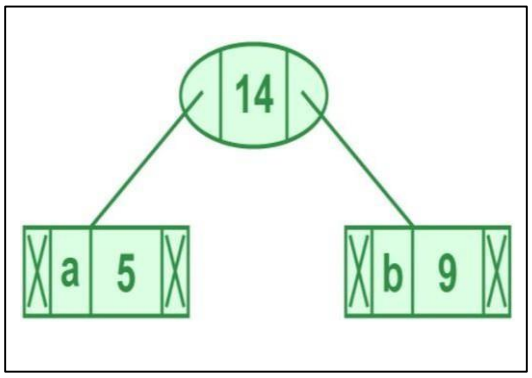
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of the frequency field is used to compare two nodes in the min heap. Initially, the least frequent character is at the root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two node's frequencies. Make the first extracted node as its left child and the other extracted node as its 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.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

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

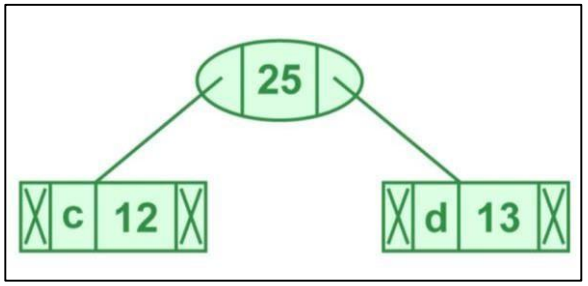
Step 2 Extract two minimum frequency nodes from the min heap. Add a new internal node with frequency 5 + 9 = 14.



character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Now min heap contains 5 nodes where 4 nodes are roots of trees with a single element each, and one heap node is the root of a tree with 3 elements.

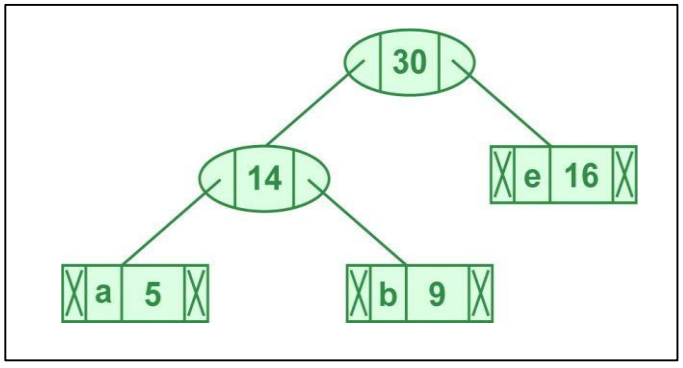
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.



character	Frequency
Internal Node	14
e	16
Internal Node	25

Now min heap contains 4 nodes where 2 nodes are roots of trees with a single element each, and two heap nodes are roots of trees with more than one node.

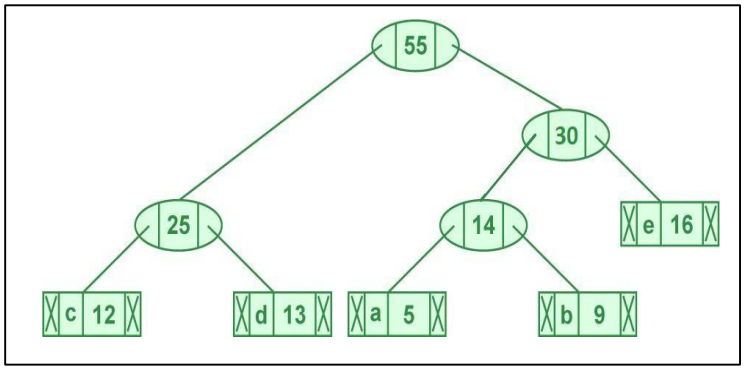
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.



character	Frequency
Internal Node	25
Internal Node	30
f	45

Now min heap contains 3 nodes.

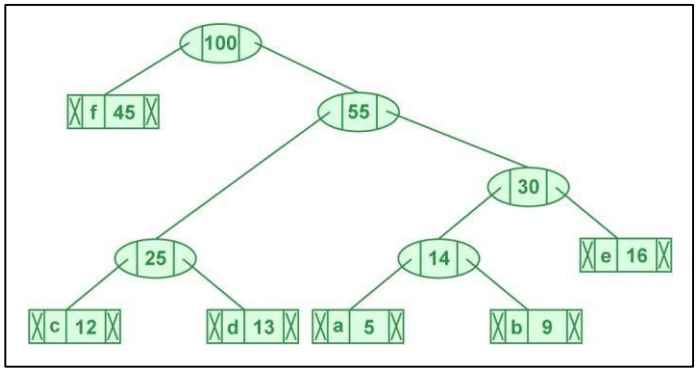
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



character	Frequency
f	45
Internal Node	55

Now min heap contains 2 nodes.

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



character	Frequency
Internal Node	100

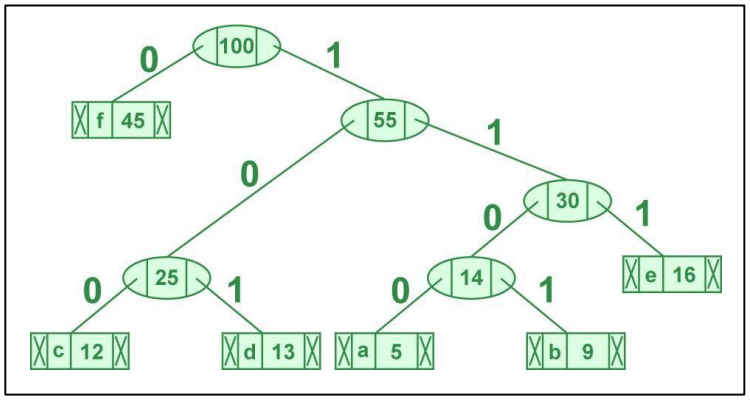
Now min heap contains only one node.

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

The codes are as follows:



character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

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.

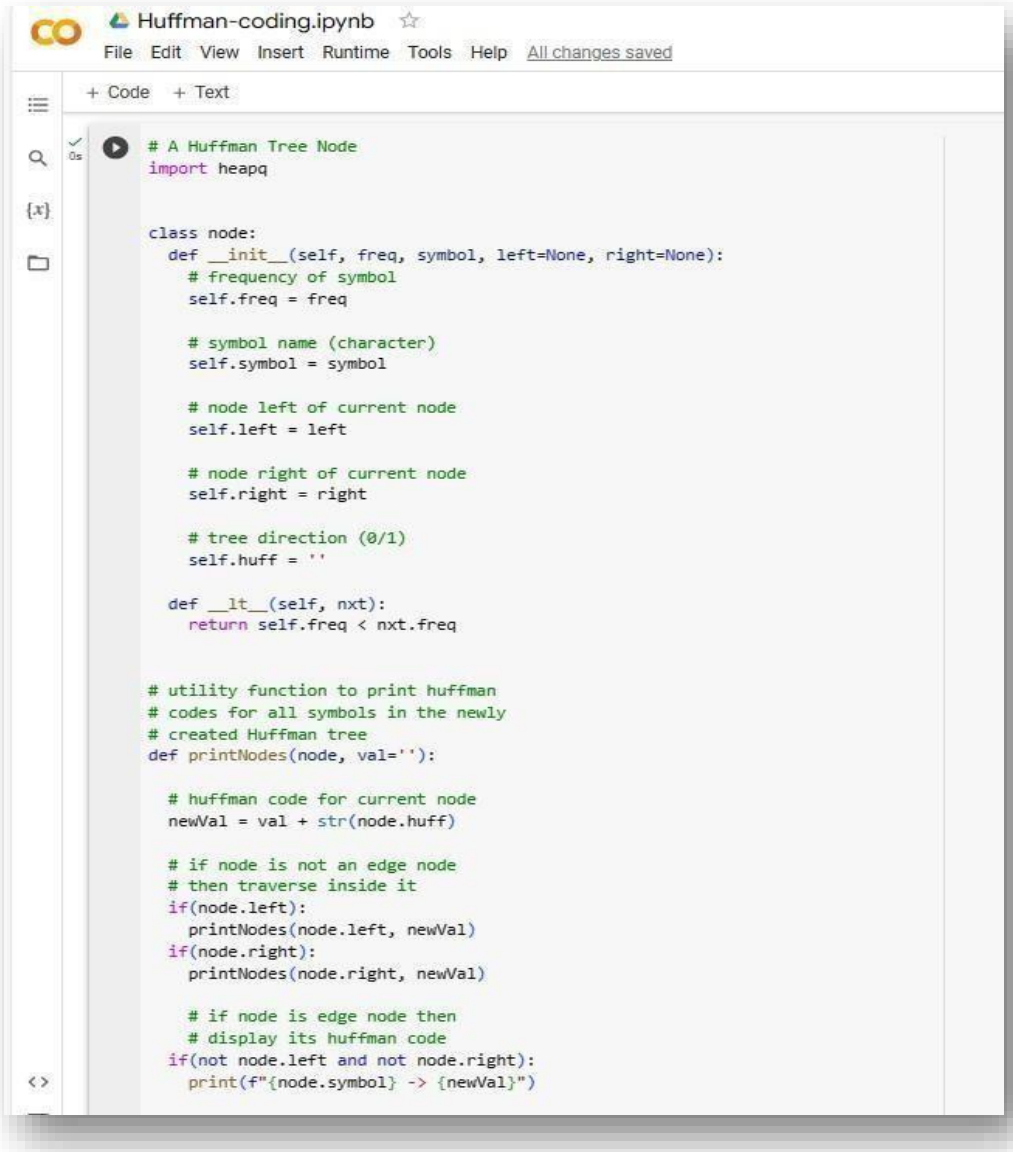
Space complexity:

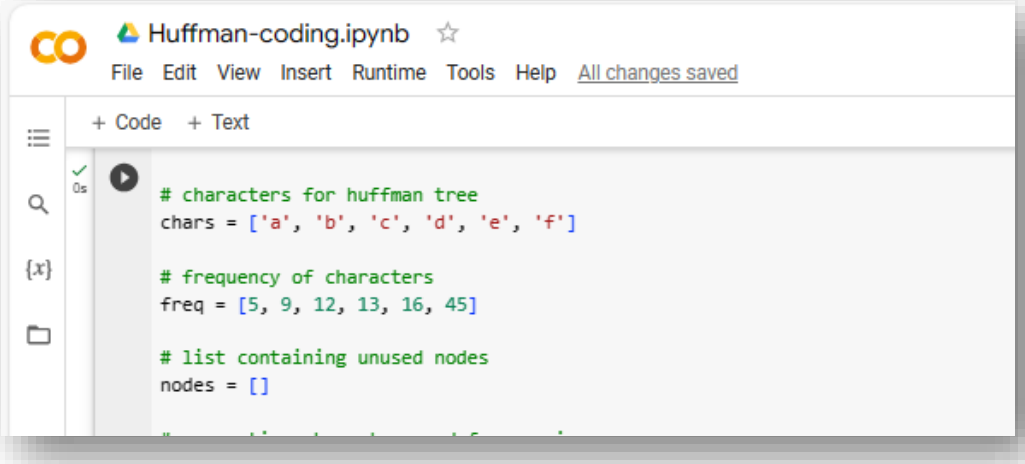
O(n) means that the amount of memory it uses increases proportionally with the size of the input. Memory usage grows linearly with input size, making it a straightforward and practical choice for many tasks.

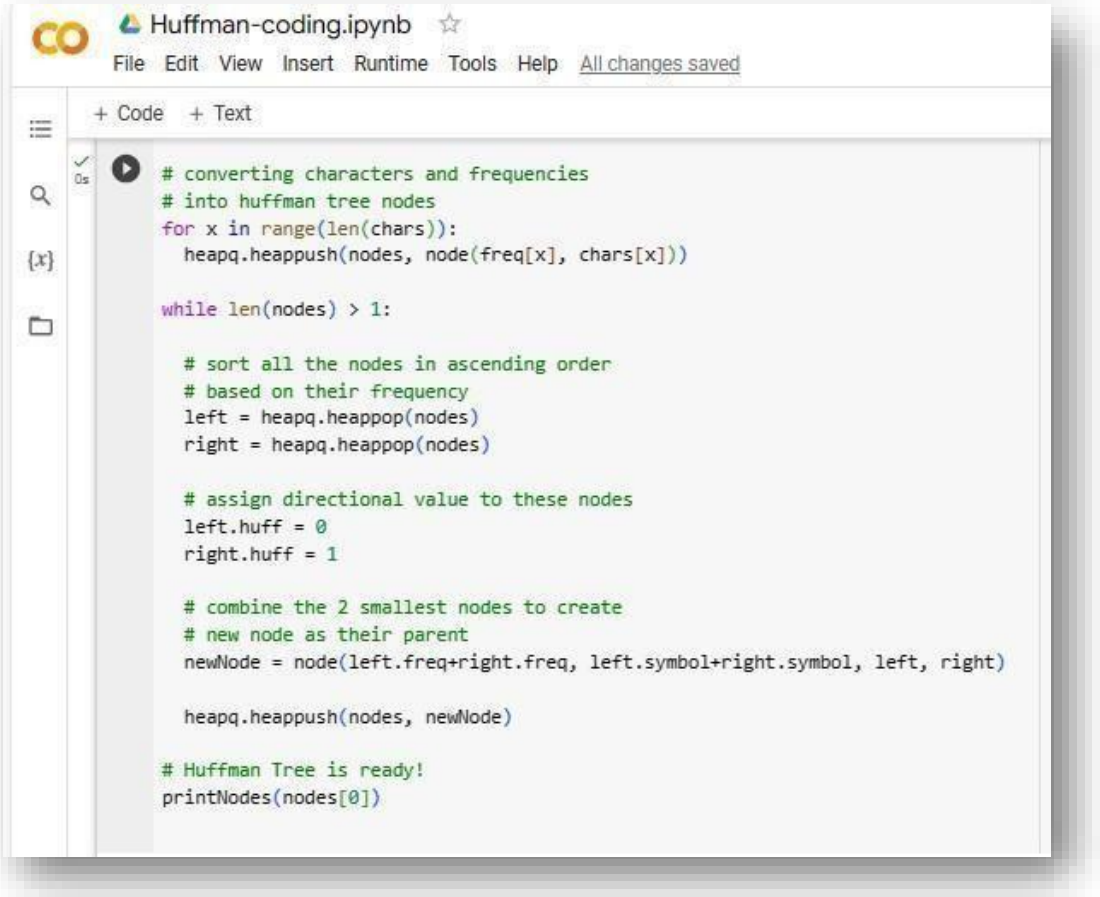
Applications of Huffman Coding:

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).

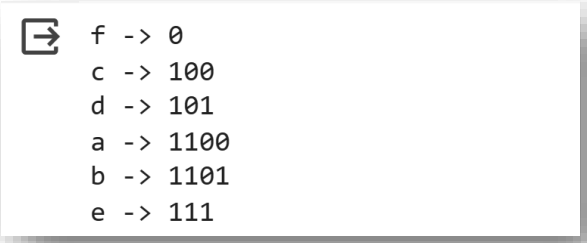
Code:


Huffman-coding.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
# A Huffman Tree Node
import heapq
class 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 __lt__(self, nxt):
return self.freq < nxt.freq
# utility function to print huffman
# codes for all symbols in the newly
# created Huffman tree
def printNodes(node, val=''):
# huffman code for current node
newVal = val + str(node.huff)
# if node is not an edge node
# then traverse inside it
if(node.left):
printNodes(node.left, newVal)
if(node.right):
printNodes(node.right, newVal)
# if node is edge node then
# display its huffman code
if(not node.left and not node.right):
print(f"{node.symbol} -> {newVal}")


Huffman-coding.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
# characters for huffman tree
chars = ['a', 'b', 'c', 'd', 'e', 'f']
# frequency of characters
freq = [5, 9, 12, 13, 16, 45]
# list containing unused nodes
nodes = []



Output:



Conclusion:

We have studied Huffman encoding with a greedy approach is efficient and simple, ideal for static data with known frequencies. In contrast, dynamic Huffman encoding adapts to changing character frequencies, making it suitable for real-time data. The choice depends on data dynamics and application requirements. Greedy Huffman excels in simplicity and efficiency for static data, while dynamic Huffman suits situations with changing data but is more complex to implement.

Experiment No: 03

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

Objective:

- Concept of Knapsack Problem
- Concept of implementation of Fractional Knapsack using Greedy Strategy.

Theory:

The Fractional Knapsack problem is a classic optimization problem in computer science and mathematics. In this problem, you are given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to determine the most valuable combination of items to include in the knapsack without exceeding its weight capacity. Unlike the 0/1 Knapsack problem, in the Fractional Knapsack problem, you can take fractions of items, which makes it more flexible.

- **The theory behind the practical solution for the Fractional Knapsack problem using a greedy approach:**

1. **Value-to-Weight Ratio:** The key insight in solving this problem greedily is to calculate the value-to-weight ratio for each item. This ratio represents how much value you get per unit of weight. Sorting the items by this ratio in descending order ensures that you pick items with the highest ratio first.
2. **Greedy Choice Property:** The greedy choice property states that at each step, you should select the item with the highest value-to-weight ratio. This choice is greedy because it optimizes the immediate gain, selecting the item that contributes the most value for the least weight.
3. **Fractional Selection:** Unlike the 0/1 Knapsack problem, where items can either be taken completely or not at all, in the Fractional Knapsack problem, you can take fractions of items if it maximizes the total value while staying within the weight limit of the knapsack.
4. **Optimality:** The greedy algorithm is optimal for the Fractional Knapsack problem because it makes the best choice at each step. This is not always the case for other knapsack variants like the 0/1 Knapsack problem, where a greedy approach may not yield the optimal solution.

- **Fractional Knapsack Examples:**

Question 1: Given a knapsack with a capacity of 50 units and a set of items with their values and weights (Item1: 60 value, 10 weight; Item2: 100 value, 20 weight; Item3: 120 value, 30 weight), explain how the value-to-weight ratio is calculated for each item. Based on the provided values and the knapsack's capacity, describe the selection and addition of items to the knapsack by Fractional Knapsack problem using the greedy method.

Answer:

Step 1: Calculate Value-to-Weight Ratios

To calculate the value-to-weight ratio for each item, we divide the value of the item by its weight:

Value-to-Weight Ratio for Item1: 60 / 10 = 6.0

Value-to-Weight Ratio for Item2: 100 / 20 = 5.0

Value-to-Weight Ratio for Item3: 120 / 30 = 4.0

Step 2: Sorting by Ratio

We sort the items in descending order of their value-to-weight ratios:

Sorted Items: Item1 (6.0), Item2 (5.0), Item3 (4.0)

Step 3: Selection and Addition to the Knapsack

We'll now describe how the items are selected and added to the knapsack based on the greedy Fractional Knapsack algorithm:

Item 1 (Value 60, Weight 10): The knapsack's current capacity is 50 units, and Item 1 has a value-to-weight ratio of 6.0. Since Item 1's weight (10) is less than the remaining capacity (50 - 10 = 40 units), we add the entire item to the knapsack. The knapsack now contains Item1, and the total value becomes 60. The remaining capacity is reduced to 40 units.

Item2 (Value 100, Weight 20): Item 2 has a value-to-weight ratio of 5.0. Although its ratio is lower than Item 1's, there's still enough capacity in the knapsack (40 units) to add the entire Item 2. So, we add the whole Item 2 to the knapsack. The knapsack now contains both Item1 and Item2, and the total value becomes 60 + 100 = 160. The remaining capacity is reduced to 20 units.

Item3 (Value 120, Weight 30): Item3 has a value-to-weight ratio of 4.0. However, its weight (30) exceeds the remaining capacity (20 units) of the knapsack. Therefore, we cannot add Item3 in it

entirely.

At this point, the knapsack is full, and all items have been considered. The selected items in the knapsack are:

Item1 with a weight of 10 units

Item2 with a weight of 20 units

The maximum achievable total value is 160 units.

Algorithm:

Fractional Knapsack (Array W, Array V, int M)

for i ← 1 to size (V)

calculate cost[i] ← V[i] / W[i]

Sort-Descending (cost)

i ← 1

while (i <= size(V))

if W[i] <= M

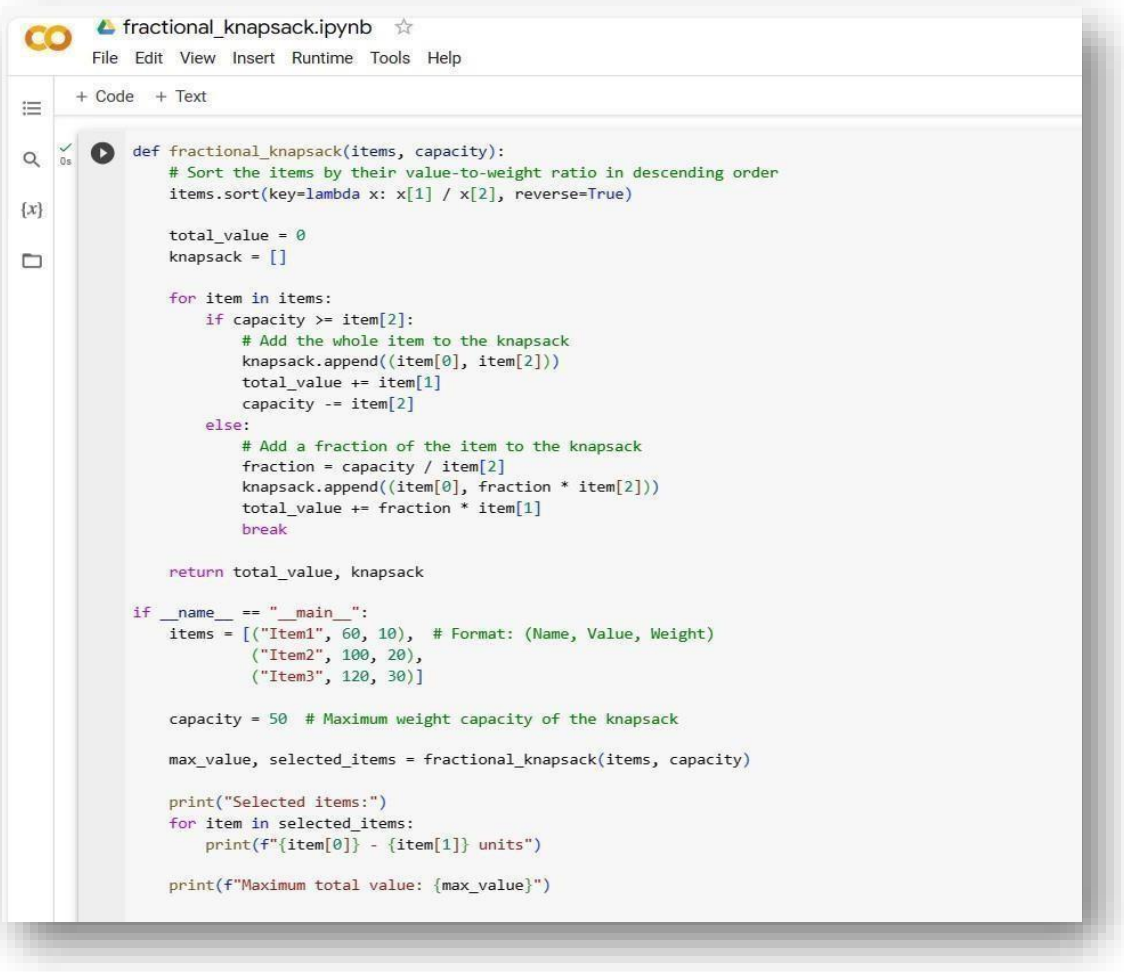
M ← M – W[i]

total ← total + V[i];

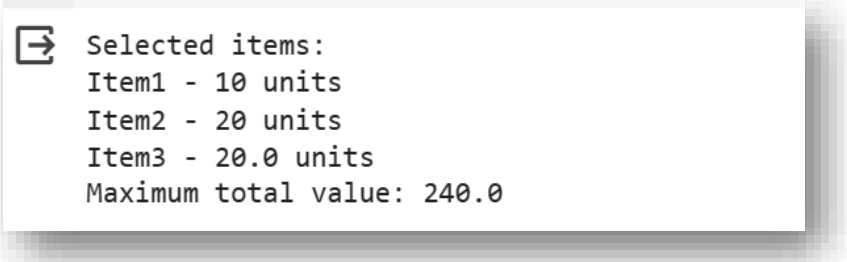
if W[i] > M

i ← i+1

Code:



Output:



Conclusion:

The Fractional Knapsack problem solution using a greedy approach is an efficient algorithm with a space complexity of O(n), where ‘n’ represents the number of items in the input. This space complexity is primarily driven by the need to store the input data and the selected items in the knapsack. The algorithm’s simplicity and low space requirements make it a practical choice for solving this optimization problem efficiently.

Experiment No: 04

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

Objective:

- The basic concept of 0/1 Knapsack Problem
- The basic concept of 0/1 Knapsack uses Dynamic Programming and Branch and Bound Approach.

Theory:

Knapsack problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Using a dynamic programming approach

Let *i* be the highest-numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - wi$ dollars and the value to the solution S is Vi plus the value of the sub-problem.

We can express this fact in the following formula: define c[i, w] to be the solution for items 1,2, …, i and the maximum weight w.

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences v = <v1, v2, …, vn> and w = <w1, w2, …, wn>

Algorithm

Dynamic-0-1-knapsack(v, w, n, W):

Initialize a 2D array c of size (n+1) x (W+1) to store the maximum values.

```
for w from 0 to W:
    c[0][w] = 0
for i from 1 to n:
    c[i][0] = 0
```

```
for i from 1 to n:
    for w from 1 to W:
        if w_i ≤ w:
            c[i][w] = max(v_i + c[i-1][w-w_i], c[i-1][w])
        else:
            c[i][w] = c[i-1][w]
Initialize an empty list, result, to store the selected items
i = n
current_w = W
while i > 0 and current_w > 0:
    if c[i][current_w] != c[i-1][current_w]:
        # Item i is part of the solution
        result.append(i)
        current_w -= w_i
    i -= 1
return c[n][W]
```

Branch and Bound Approach:

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

Let us consider below 0/1 Knapsack problem below to understand Branch and Bound. Consider an example where $n = 4$, and the values are given by {10, 12, 12, 18}and the weights given by {2, 4, 6, 9}. The maximum weight is given by $W = 15$. Here, the solution to the problem will include the first, third, and fourth objects. In solving this problem, we shall use the Least Cost- Branch and Bound method, since this shall help us eliminate exploring certain branches of the tree. We shall also be using the fixed-size solution here. Another thing to be noted here is that this problem is a maximization problem, whereas the Branch and Bound method is for minimization problems. Hence, the values will be multiplied by -1 so that this problem gets converted into a minimization problem.

Now, consider the 0/1 knapsack problem with *n* objects and total weight W. We define the upper bound(U) to be the summation of vixi (where vi denotes the value of that object, and xi is a binary value, which indicates whether the object is to be included or not), such that the total weights of the included objects is less than *W*. The initial value of U is calculated at the initial position, where objects are added in order until the initial position is filled.

We define the cost function to be the summation of vifi, such that the total value is the maximum that can be obtained which is less than or equal to *W*. Here fi indicates the fraction of the object that is to be included. Although we use fractions here, it is not included in the final solution.

Here, the procedure to solve the problem is as follows:

- Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
- If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than *U*, then replace the value of *U* with this value. Then, kill all unexplored nodes which have a cost function greater than this value.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
- While including an object, one needs to check whether adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

In this manner, we shall find a value of U at the end which eliminates all other possibilities. The path to this node will determine the solution to this problem.

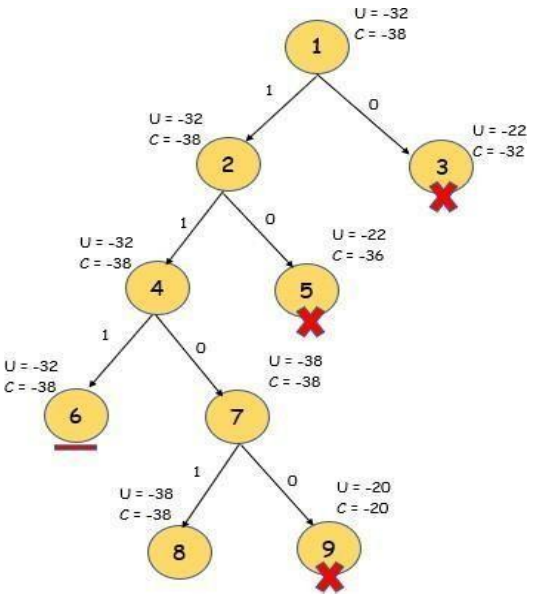
Time and Space Complexity:

Even though this method is more efficient than the other solutions to this problem, its worst-case time complexity is still given by O(2n), in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to be explored, and hence its best-case time complexity is given by O(n). Since this method requires the creation of the state space tree, its space complexity will also be exponential.

Solving an Example: Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initial upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

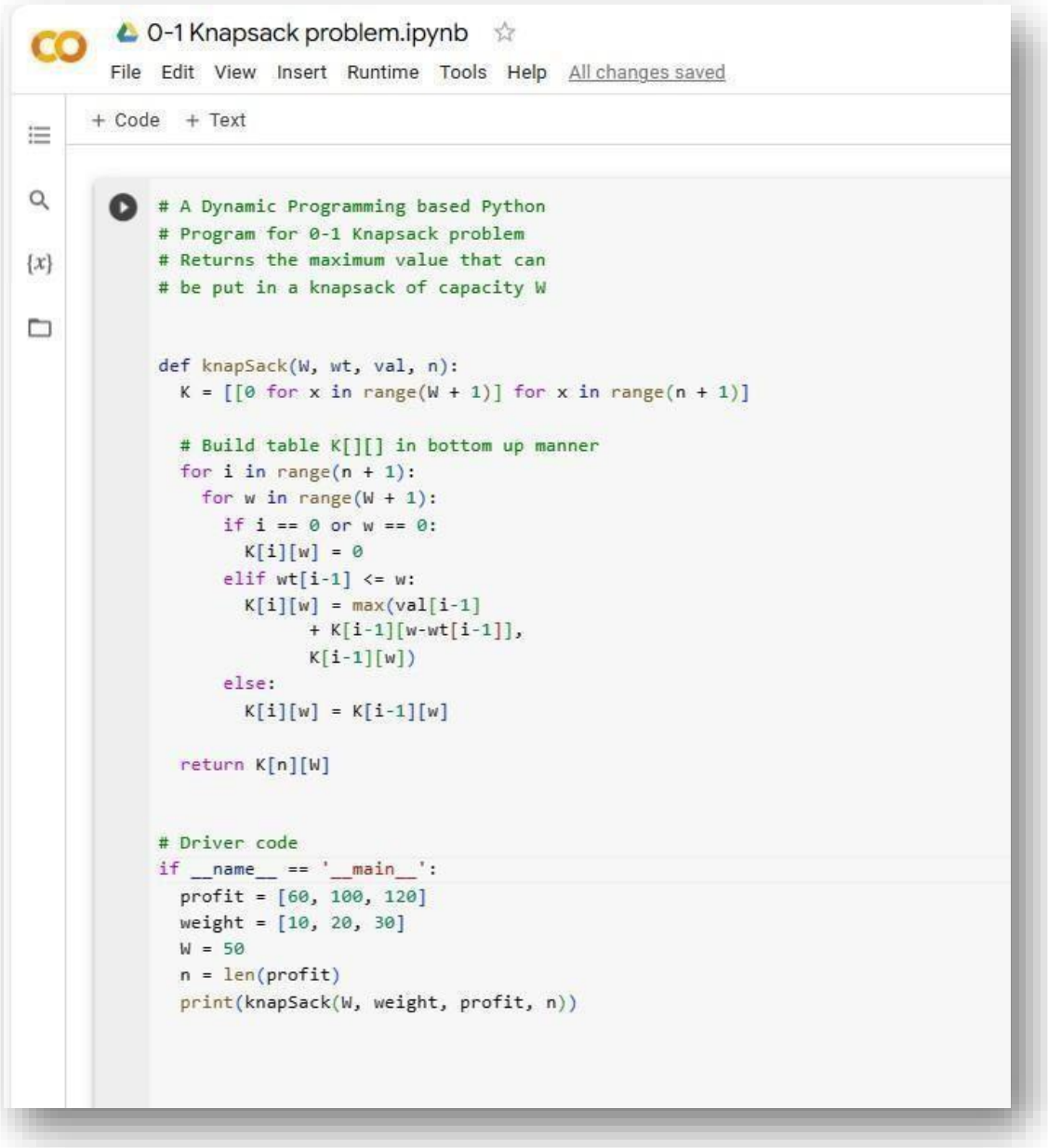
After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state

space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):

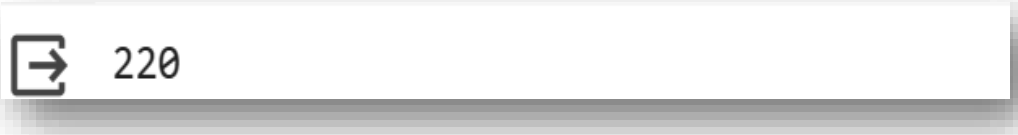


Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

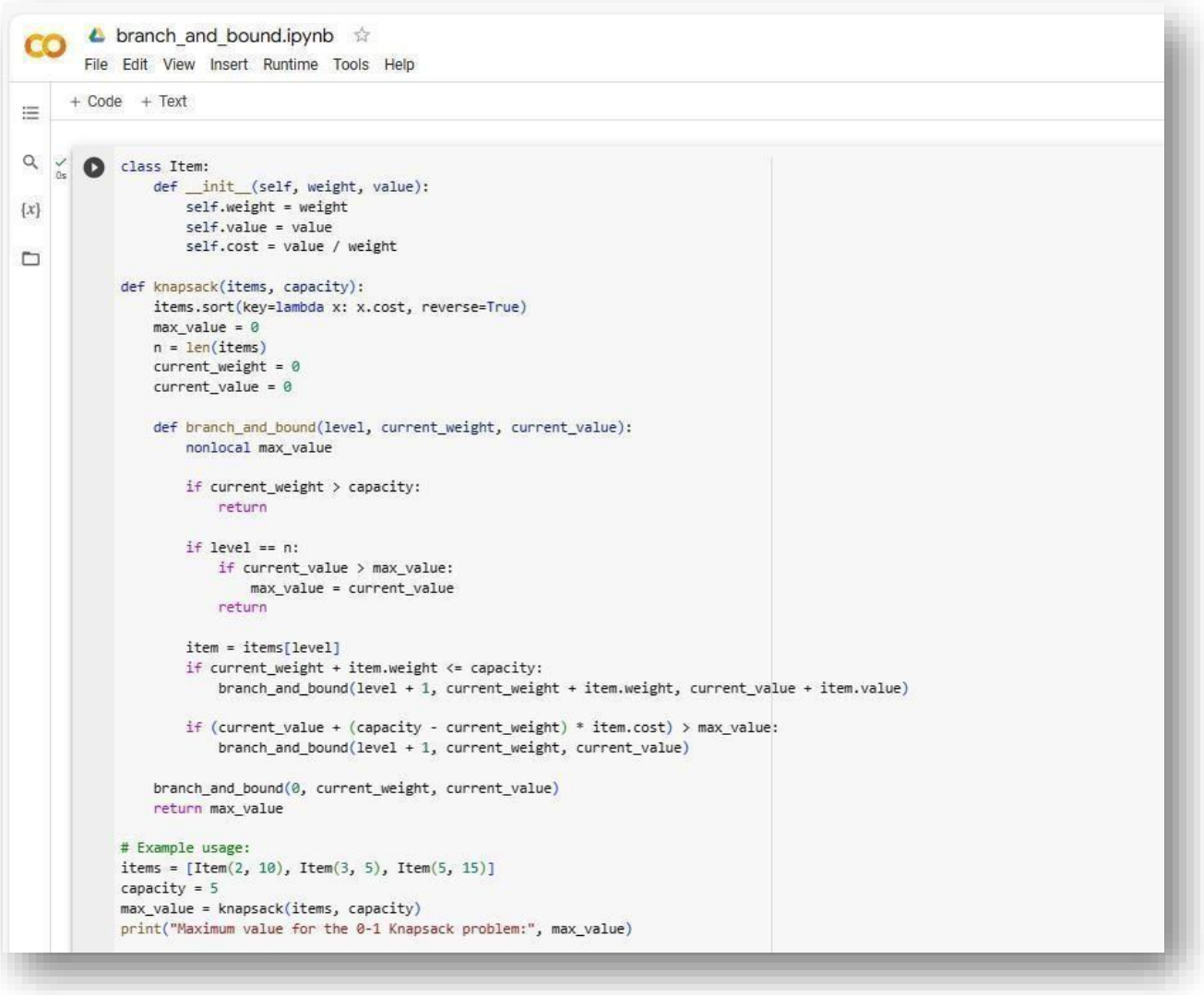
Dynamic Approach Code:



Output:



Branch and Bound Approach Code:



Output:

Conclusion:

We can solve the 0/1 knapsack problem by using Dynamic Programming and Branch and Bound Approach. However, the choice between them depends on the specific problem instance and constraints. Dynamic programming is generally more straightforward to implement and works well for small to moderate-sized instances, while branch and bound is better suited for larger instances with a vast search space.

Experiment No: 05

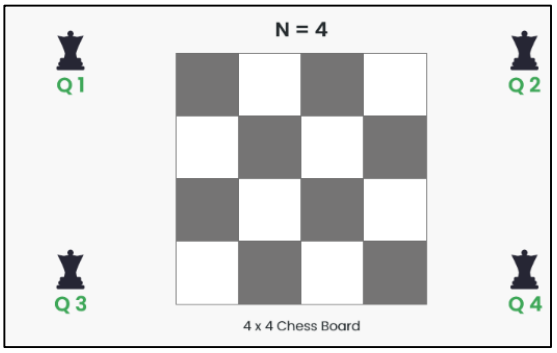
Aim: Design n-Queens matrix having the first Queen placed. Use backtracking to place the remaining Queens to generate the final n-queens matrix.

Objective:

- The Basic Concepts of N Queens Problem
- N queens using Backtracking

Theory:

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 problems.

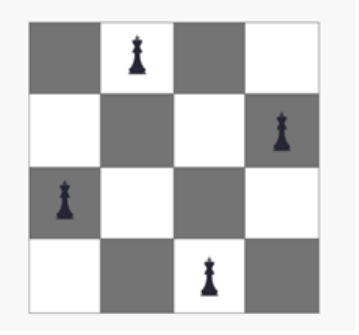
The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.

{0, 1, 0, 0}

{0, 0, 0, 1}

{1, 0, 0, 0}

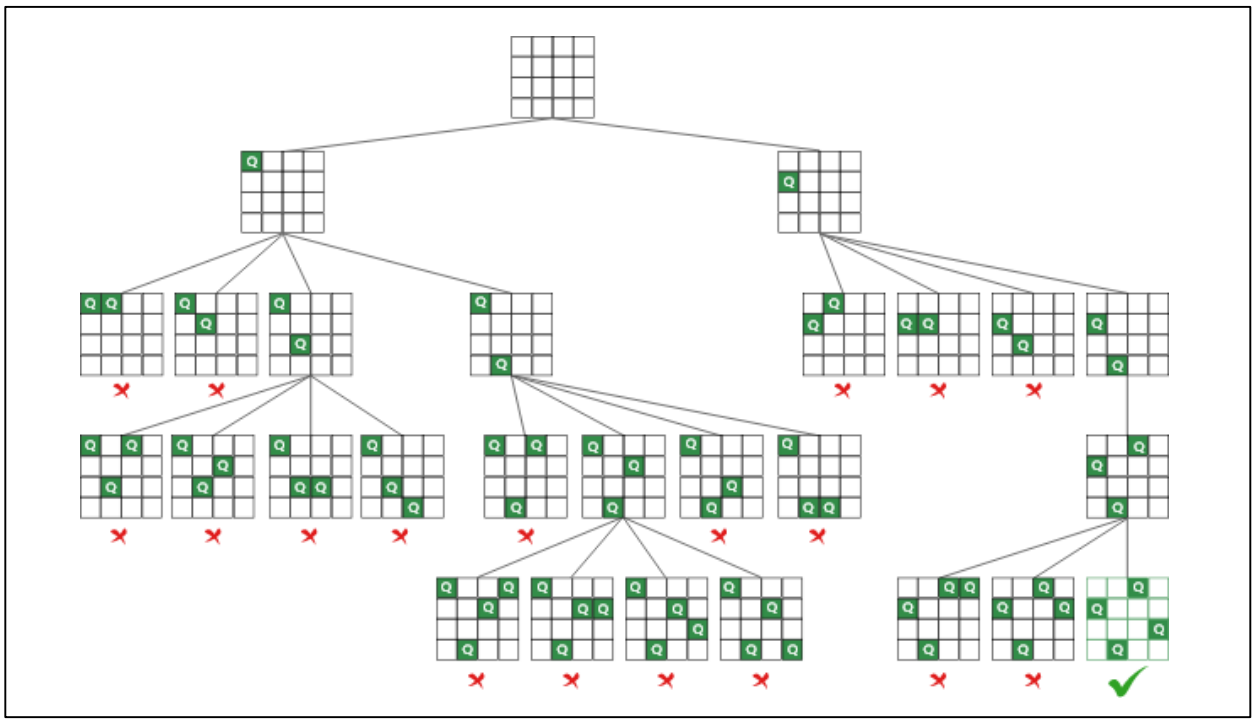
{0, 0, 1, 0}



Backtracking Algorithm

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:



Algorithm:

1. Start from the leftmost column.
2. If all queens are placed, return true (a solution is found).
3. Try all rows in the current column. For each row, do the following:
   a. If the queen can be placed safely in this row, mark this [row, column] as part of the solution.
   b. Recursively check if placing the queen here leads to a solution.
   c. If placing the queen in [row, column] leads to a solution, return true.
   d. If placing the queen doesn't lead to a solution, unmark this [row, column]

(Backtrack) and go to step '3a' to try other rows.

4. If all rows have been tried and nothing worked, return false to trigger backtracking.

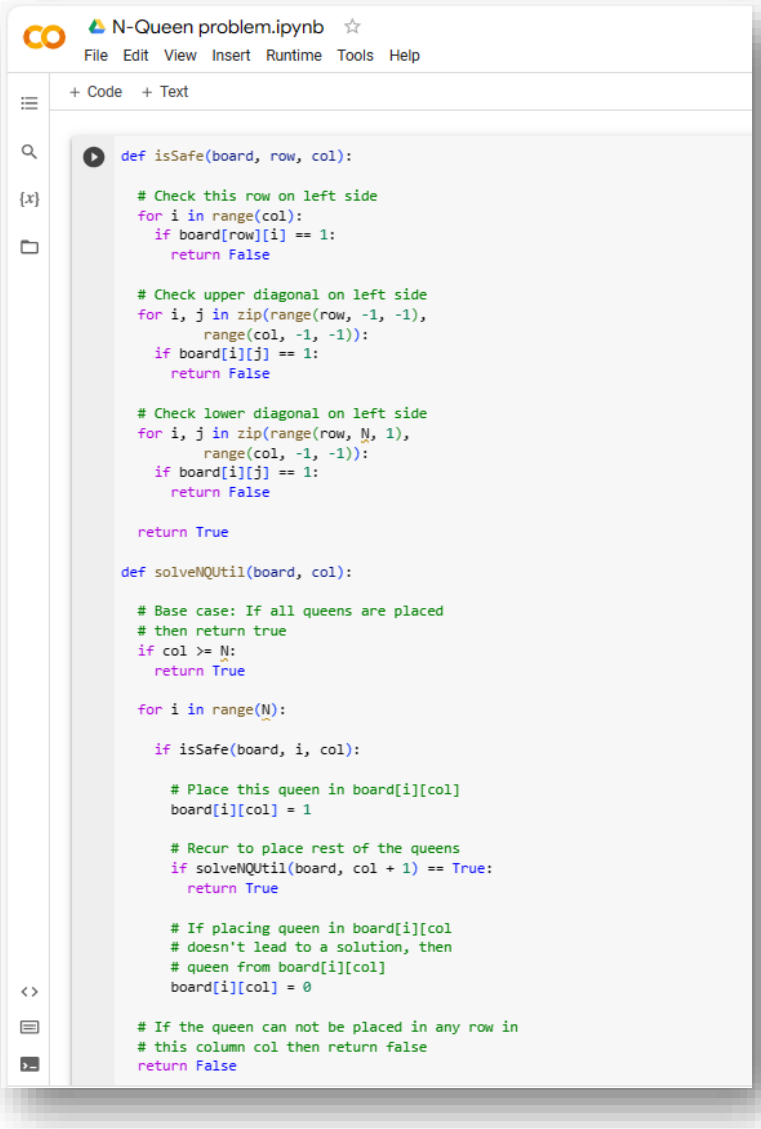
Time Complexity:

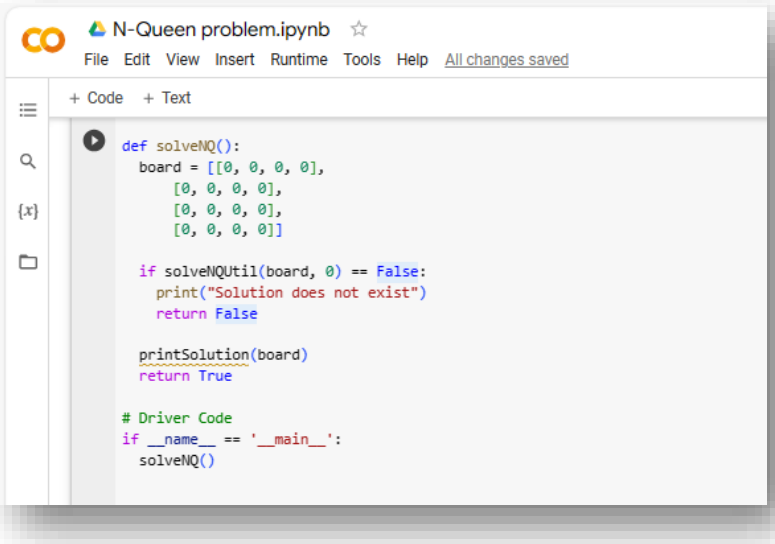
O(n!), where nN is the number of queens (size of the chessboard).

Space Complexity:

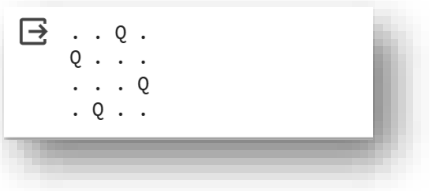
$O(n^2)$ for the chessboard representation and O(n) for the recursive call stack.

Code:


N-Queen problem.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
def isSafe(board, row, col):
# Check this row on left side
for i in range(col):
if board[row][i] == 1:
return False
Check upper diagonal on left side
for i, j in zip(range(row, -1, -1),
range(col, -1, -1)):
if board[i][j] == 1:
return False
Check lower diagonal on left side
for i, j in zip(range(row, N, 1),
range(col, -1, -1)):
if board[i][j] == 1:
return False
return True
def solveNQUtil(board, col):
# Base case: If all queens are placed
# then return true
if col >= N:
return True
for i in range(N):
if isSafe(board, i, col):
# Place this queen in board[i][col]
board[i][col] = 1
# Recur to place rest of the queens
if solveNQUtil(board, col + 1) == True:
return True
# If placing queen in board[i][col
# doesn't lead to a solution, then
# queen from board[i][col]
board[i][col] = 0
# If the queen can not be placed in any row in
# this column col then return false
return False


N-Queen problem.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
def solveNQ():
board = [[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]]
if solveNQUtil(board, 0) == False:
print("Solution does not exist")
return False
printSolution(board)
return True
# Driver Code
if __name__ == '__main__':
solveNQ()

Output:



Conclusion:

We have studied the N-Queen Problem using the Backtracking Approach. This approach provides a systematic and efficient way to solve the n-Queens problem, a well-known problem in computer science and combinatorial optimization.