Assignment No : 01

Page No :
Date : / /

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

- Objectives : Students should be able to perform non-recursive & recursive programs to calculate fibonacci numbers and analyze their complexities.

- Prerequisite : 1. Basic of python or Java.
2. Concept of recursive & non-recursive fun^n.
3. Execution flow of calculate fibonacci numbers.
4. Basic of Time and space complexity.

- Theory :

What is Fibonacci series ?

The fibonacci series is the sequence of 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...

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

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

- Fibonacci Sequence Formula :

The fibonacci seq. of numbers "Fn" is defined using the recursive relation with the seed values F0 = 0 & F1 = 1 :

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

Page No :
Date : 1 1

Ex : Input : n = 2 Input : n = 9
Output : 1 Output : 34

Fn	0	1	2	3	4	5	6	7	8	9	....
Fibo. No.	0	1	1	2	3	5	8	13	21	34	....

- Method 1 (Use 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] & F[1] resp.
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 no. at position n in the sequence.

- Time & space complexity (Optimized method) :

Time complexity : T(N) i.e. linear
Space complexity : O(1)

- Time & space complexity of DP :

Time complexity : T(N)
Space complexity : O(N)

- Method 2 (Use Recursion) :

To evaluate F(n) for n > 1, we can 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) & F(n-2) to F(n-1)-1) and F((n-1)-2); and F((n-2)-1) and F((n-2)-2), respectively.

Our algo. for F(n) will have 2 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 two results.

- Time & space complexity:
  Time complexity : T(2^N) i.e. exponential
  Space complexity : O(N)

- App. of Fibonacci series:
  1. It is used in the grouping of numbers & used to study diff. other special mathematical sequences.
  2. It finds app. in coding.
  3. It is applied in numerous fields of science like quantum mechanics, cryptography etc.
  4. In finance market trading, Fibonacci retracement levels are widely used in tech. analysis.

- Conclusion : In this way we have explored concept of Fibonacci series using recursive & non-recursive method & also learned about the complexity.

Assignment No : 02

Page No :
Date : / /

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

- Objectives : Students should be able to understand & solve Huffman Encoding using greedy method.

- Prerequisite : 1. Basic of python or Java
  2. Concept of Greedy method
  3. Huffman Encoding concept

- 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 reverses 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.

- Advantages :
  - The algorithm is easier to describe.
  - This algorithm can perform better than other algorithms.

- Drawbacks :
  - The algorithm doesn't produce optimal solution.
  - For ex, suppose we want to find the longest path in the graph below from root to leaf.

- Greedy Algorithm :

1. To begin with, the sol^n set is empty.
2. At each step, an item is added to the sol^n set until a sol^n is reached.
3. If the sol^n set is feasible, the current item is kept.
4. Else, the item is rejected & never considered again.

- Huffman Encoding :

It is a technique of compressing data to reduce its size without losing any of the details. It is a famous Greedy Algorithm. It is used for the lossless compression of data. It uses variable length encoding. It assigns variable length code to all the characters. The code length of a character depends on how freq. it occurs in the given text. The character which occurs most freq. gets the smallest code. It is also known as Huffman Encoding.

- Prefix Rule :

1. Huffman coding implements a rule known as a prefix rule.
2. This is to prevent the ambiguities while decoding.
3. It ensures that the code assigned to any character is not prefix of the code assigned to any other character.

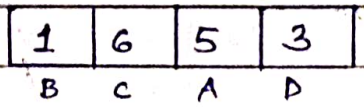
- Major steps :

1. Building a Huffman Tree from the input characters.
2. Assigning code to the characters by traversing the Huffman tree.

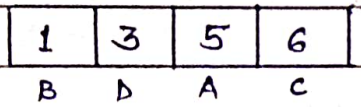
Page No :
Date : / /

- Steps :

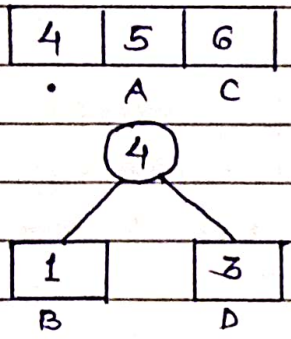
1. Calculate frequency of each character in the string.



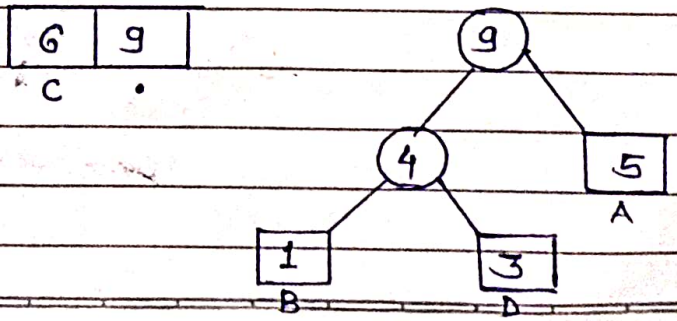
2. Sort the characters in increasing order of frequency. These are sorted in priority queue Q.



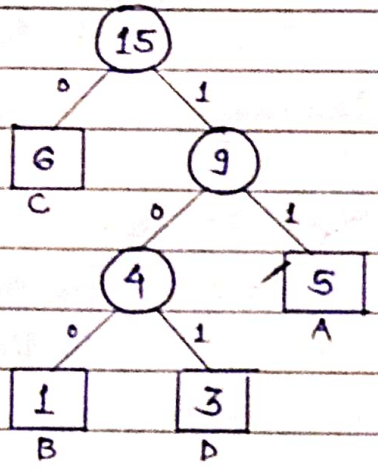
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum freq. to the left child of z and assign the second minimum freq. to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.



5. Remove these two minimum frequencies from Q and add the sum into the list of frqencies.
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all characters.



8. For each non-leaf node, assign 0 to the left edge & 1 to the right edge.



Character	Frequency	Code	Size
A	5	11	5 * 2 = 10
B	1	100	1 * 3 = 3
C	6	0	6 * 1 = 6
D	3	101	3 * 3 = 9
4 * 8 = 32bits	15 bits		28 bits

Without encoding, the total size of string was 120 bits.
After encoding the size is reduced to 32 + 15 + 28 = 75.

- Time complexity : $O(n \log n)$
- Conclusion : In this way we have explored concept of Huffman encoding using greedy method.

Assignment No : 03

- Title : Write a program to solve a Fractional knapsack problem using a greedy method.

- Objective : Students should be able to understand and solve fractional knapsack problems using greedy methods.

- Prerequisite : 1. Basic of python or Java
  2. Concept of Greedy method
  3. Fractional Knapsack problem.

- Theory :

1. Greedy Method.

2. Knapsack Problem :
   You are given the following -
   - A knapsack with limited weight capacity.
   - Few items each having some weight & value.

   The problem states that -
   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 Variants :
  1. Fractional knapsack problem
  2. 0/1 knapsack problem.

1. 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 is solved using greedy method in following steps :

1. For each item, compute its value / weight ratio.
2. Arrange all the items in decreasing order of their value/ weight ratio.
3. Start putting the items into the knapsack beginning from the item with the highest ratio.
Put as many items as you can into the knapsack.

- Problem :
For the given set of items & knapsack capacity = 60kg, find the optimal sol^n for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

n = 5
W = 60 kg
(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)
(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

Soln: Step 1: 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 2: Sort all items in decreasing order
6, 4, 3.6, 3.5, 3

Step 3: Start filling the knapsack by putting the items into it one by one.

Knapsack weight	Items in Knapsack	Cost
60	ϕ	0
55	I1	30
45	I1, I2	70
20	I1, I2, I3	160

Now,
- Knapsack weight left to be filled is 20kg but item-4 has a 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>

Page No :
Date : / /

Total cost of the knapsack
= 160 + (20/22) x 77
= 160 + 70
= 230 units

- Time complexity :
  - If the items are already arranged in the req. order, then while loop takes O(n) time.
  - Total time taken including the sort is O(nlogn).

- Conclusion : In this way we have explored concept of fractional knapsack using greedy method.

Assignment No : 04

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

- Objectives : Students should be able to understand & solve 0-1 knapsack problem using dynamic programming.

- Prerequisite : 1. Basic of Python or Java
  2. Concept of dynamic programming
  3. 0/1 Knapsack problem.

- Theory :

What is dynamic programming ?

- It is also used in optimization problems. Like divide and - conquer method. Dymanic prog. solves problems by combining the solutions of subproblems.
- It solves each sub-problem just once & 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 substructure.
- It also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly.

- Steps :

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal sol^n from the computed info.

- App. of dynamic prog. :

1. Matrix chain multiplication
2. Longest common subsequence
3. Travelling Salesman Problem.

- Knapsack problem :

You are given the following :
- A knapsack with limited weight capacity.
- Few items each having some weight and value.

- Knapsack problem Variants :

1. Fractional knapsack problem
2. 0/1 knapsack problem

- 0/1 Knapsack Problem :
- Items are indivisible here.
- We can not take a fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using a dynamic prog. approach.

- 0/1 knapsack problem using Greedy method :

Consider,
- Knapsack weight capacity = W
- No. of items each having some weight & value = n.

- Steps :

Step 1 : Draw a table say 'T' with (n+1) number of rows & (w+1) number of columns.
- Fill all the boxes of 0th row & 0th column with zeroes as :

	0	1	2	3	....	W
0	0	0	0	0	....	0
1	0					
2	0					
	....					
n	0					

Step 2: Start filling the table row wise top to bottom from left to right using the formula :

$$T(i,j) = \max\{T(i-1,j), value_i + T(i-1, j-weight_i)\}$$

Finding $T(1,1)$ -

We have,

- $i = 1$
- $j = 1$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get -

$T(1,1) = \max\{T(1-1,1), 3 + T(1-1,1-2)\}$

$T(1,1) = \max\{T(0,1), 3 + T(0,-1)\}$

$T(1,1) = T(0,1)$ { Ignore $T(0,-1)$ }

$T(1,1) = 0$

Finding $T(1,2)$ -

We have,

- $i = 1$
- $j = 2$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get -

$T(1,2) = \max\{T(1-1,2), 3 + T(1-1,2-2)\}$

$T(1,2) = \max\{T(0,2), 3 + T(0,0)\}$

$T(1,2) = \max\{0, 3+0\}$

$T(1,2) = 3$

Finding $T(1,3)$ -

We have,

- $i = 1$
- $j = 3$
- $(value)_i = (value)_1 = 3$
- $(weight)_j = (weight)_1 = 2$

Substituting the values, we get -

$T(1,3) = \max\{T(1-1,3), 3 + T(1-1,3-2)\}$

$T(1,3) = \max\{T(0,3), 3 + T(0,1)\}$

$T(1,3) = \max\{0, 3+0\}$

$T(1,3) = 3$

Similarly compute all the entries -

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7)

Max. value that can be put into the knapsack = 7

- Identifying items to be put into knapsack :
Following Step-04,
- We mark the rows labelled "1" and "2".
- Thus, items that must be put into the knapsack to obtain the max. value 7 are -
Item-1 and Item-2

- Time complexity : $\theta(nw)$

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

Assignment No : 05

Page No :
Date : / /

- Title : Design n-Queens matrix having first queen placed. Use backtracking to place remaining queen to generate the final n-queen's matrix.

- Objective : Students should be able to understand and solve n-Queen problem and understand basics of backtracking.

- Prerequisite : 1. Basic of python or Java.
  2. Concept of backtracking method
  3. N-Queen Problem.

- Theory :

What is Backtracking ?

- Backtracking is finding the sol^n of a problem whereby the solution depends on the previous steps taken.
- In backtracking, we first take a step & then we see if this step is correct or not. and if it doesn't then we just come back and change our first step.
- In general, it is accomplished by recursion.
- General steps are -

1. start with a sub-solution.
2. Check if this sub-solution will lead to the sol^n or not
3. If not, then come back & change the sub-solution and continue again.

- Applications of Backtracking :

1. N-Queen Problem
2. Sum of subsets problem
3. Graph coloring

- What is N-Queen Problems?

The N-Queen problem is a common example of the backtracking where the goal is to place n queens on an n x n chessboard so that no two queens attack each other. This means no 2 queens can be in the same row, column or diagonal. The problem is typically solved using backtracking algorithms.

Algorithm:

1. Start in the leftmost column.
2. If all queens are place return true
3. Try all rows in the current column.

a] If the queen can be placed safely in this row then mark this [row, column] as part of the sol^n & recursively check if placing queen here leads to a solution.

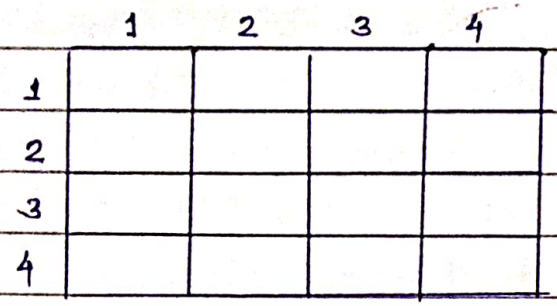
b] If placing queen in [row, column] leads to a sol^n then return true.

c] If placing queen doesn't lead to a sol^n then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

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

- 4-Queen Problem

Problem 1 : Given 4 X 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column or diagonal.



- We have to arrange four queens, Q_1, Q_2, Q_3, Q_4 in 4 X 4 chessboard. We will put with queen in i^{th} row. let us start with position (1,1). Q1 is the only queen, so there is no issue.
- We cannot place Q2 at positions (2,1) or (2,2). Position (2,3) is acceptable. the partial solution is <1, 3>.
- Next, Q3 cannot be placed in position (3,1) as Q1 attacks her. and it cannot be placed at (3,2) (3,3) or (3,4) as Q2 attaks her. There is no way to put Q3 in the third row. Hence the algo. backtracks and goes back to the previous sol^n and readjusts the position of queen Q2. Q2 is moved from positions (2,3) to (2,4). Partial solution is <1,4>.
- Now, Q3 can be placed at position (3,2). Partial sol^n is <1, 4, 3>.
- Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous sol^n and readjust the position of Q3. Q3 cannot be placed on (3,3) or (3,4). So the algo. backtracks even further.
- All possible choices for Q2 are already explored, hence the algorithm goes back to partial sol^n <1> and moves the queen Q1 from (1,1) to (1,2) and this process continues until a sol^n is found.

All possible soln for 4-queen are shown in fig (a) & fig (b) :

	1	2	3	4
1		Q_1		
2				Q_2
3	Q_3			
4			Q_4	

	1	2	3	4
1			Q_1	
2	Q_2			
3				Q_3
4		Q_4		

- Conclusion : In this way we have explored concept of backtracking method and solved n-queen problem using this method.