DAA - HW - 12

Name :- Vallepalli Venkata Rama krishna

ID :- 1002203515

CHAPTER - 15

1) what is the difference b/w divide & Conquer & Dynamic Programming

Ans):-

The Divide & Conquer and dynamic Programming are two commonly used techniques in Algorithm design, both aimed at solving Problems by breaking them down into smaller subproblems.

Divide & conquer:-

Divide & Conquer involves the breaking down a Problem into smaller more manageable subProblems, solving These subProblems Independently & then combining their Soluting to Form the solution to the original Problem

--> divide the Problem into smaller subProblems, Conquer the subProblems by solving them recursively & combine the solutions of subProblems to Form the solution of original Problem.

Dynamic Programming:-

Dynamic Programming breaks down a Problem into smaller overlapping subProblems & solve each subProblem only once, storing the

Solution to each subproblem in a table to avoid redundant computations.

-> It Typically involves solving a Problem by solving smaller subProblems & storing their solutions in a table.

-> Dynamic Programming Relies on identifying & solves overlapping subproblems, divide & Conquer typically solves non-overlapping subproblems

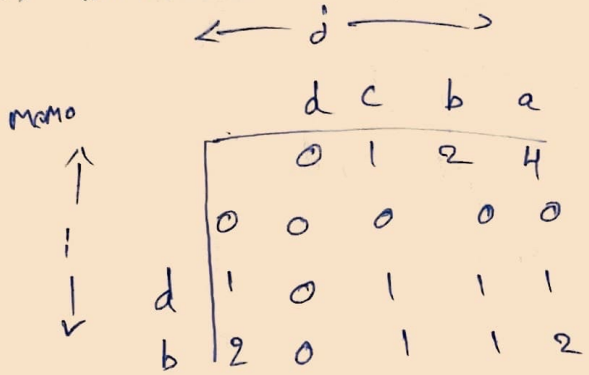
2) a) int Max (int a, int b)

```
{
  if a > b
    return a
  else
    return b
```

```
int knapSack (int w ; int wt<3 ; int val int n );
if (n==0 || w==0)
  return 0
if (wt[n-1]> w)
  return knapsack (w, wt, val, n-1)
else

return max (Val (n-1) + Knapsack( w - wt[n-1], wt, val, n-1),
  knapsack (w, wt, val, n-1));
```

3) Bottom UP approach



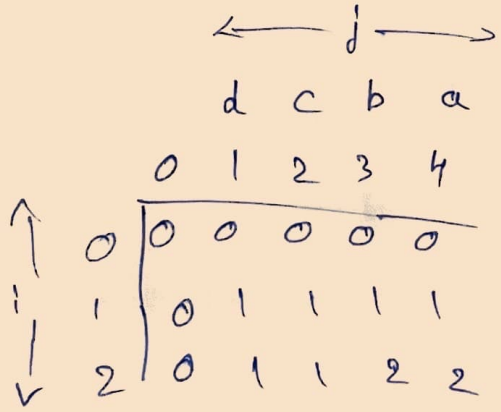

length of LSM = 2
which is
memo [2] [4]

4) ~~K = OB~~

X = (d, b) y = (dcba)

c = ?

P. 394




5)

Given,

$$Fac(n) = \begin{cases} 1 & \text{if } n = 1 \\ nFac(n-1) & \text{if } n > 1 \end{cases}$$

a) is This a good DP algorithm? why or why not.

Ans):-

No, This not an ideal Problem For dynamic Programming because it lacks overlapping subproblems. The given Formula For Factorial

is More Aligned with the Divide & Conquer approach rather than DP.

-> Calculating Factorials recursively doesn't involves reusing the identical subproblem solution Each recursive call computes a new value & doesn't need the previous result again.

b) Native recursive algorithm.

```
def recurr(n):
    if n == 0 || n == 1
        return 1
    else
        return n + recurr(n-1)
```

c) Top-Down Approach

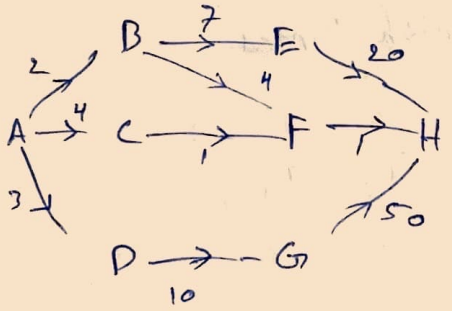
```
def recurr-top Down (n, menu = { })
    if n == 0 in Memo
        return Memo [n]
    if n == 1
        return 1
    Memo [n] = n * recurr-top Down (n-1, Memo)
    return Memo(n).
```

d) Bottom-up approach.

```
def recurr_bottomup(n).
    if n==0 || n==1.
        return 1
    Fact_table = [0] * (n+1)
    Fact_table [1] = 1
    For i in range [2,n+1]
        Factorial_table [i] = i + Fact_table (i-1).
    return Fact_table (n).
```

## chapter-16

1)




To Find the shortest part, short with A.

| | | |
|---|---|---|
| A | B | 2 |
| A | C | H |
| A | D | 3 |

→ shortest Path

From B

| | | |
|---|---|---|
| B | E | E7 |
| B | F | 4 |

→ shortest Path

From F.

F —— H 1. -> shortest Path.

Path is A → B → F → H
2 4 1

Cost 2+4+1
= 7

2) solve the Following Activity Problem.

| Activity | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ |
|---|---|---|---|---|---|---|---|---|
| sort | 1 | 0 | 1 | 4 | 2 | 5 | 3 | 4 |
| Finish | 3 | 4 | 2 | 6 | 9 | 8 | 5 | 5 |

Greedy Approach

1=> sort the Activites based on their Finish Times.

| Activity | $a_3$ | $a_1$ | $a_2$ | $a_7$ | $a_8$ | $a_4$ | $a_6$ | $a_5$ |
|---|---|---|---|---|---|---|---|---|
| sort | 1 | 1 | 0 | 3 | 4 | 4 | 5 | 2 |
| Finish | 2 | 3 | 4 | 5 | 5 | 6 | 8 | 9 |

2) select the Finish Activity.

$a_3$ => 1 -> 2

3) Next select the Activity that has start time >= 2.

$a_7$ -> 3 -> 5

Repeat

$a_6 \rightarrow s \rightarrow 8$

Activities selected

| | $a_3$ | $a_7$ | $a_6$ |
|---|---|---|---|
| start | 1 | 3 | 5 |
| Finish | 2 | 5 | 8 |

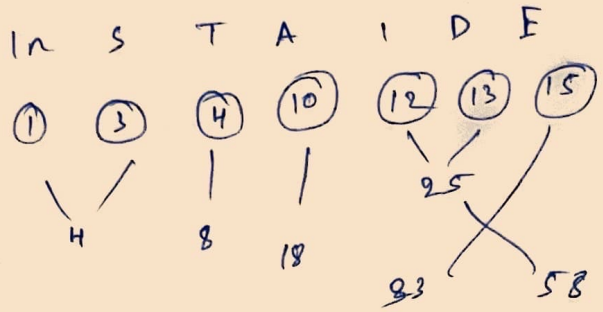
3) use huffman coding to compress the message

given,

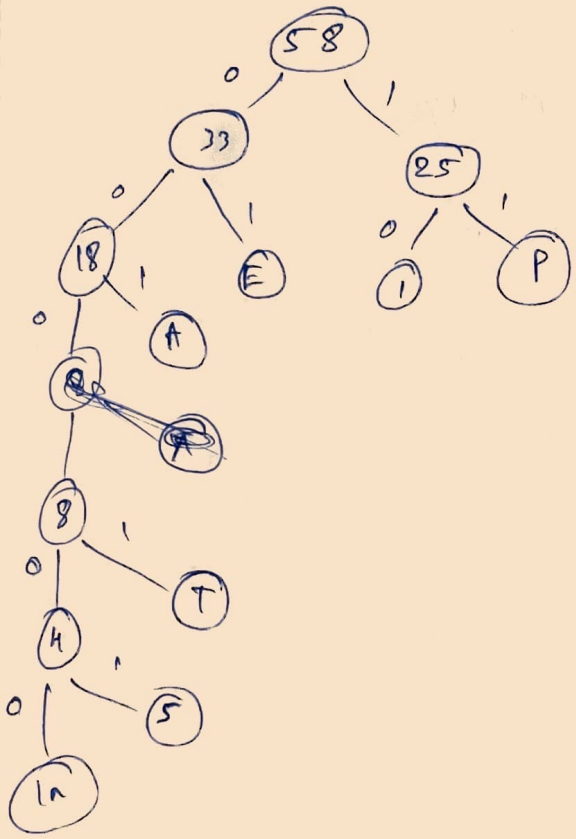
| char | count | code |
|---|---|---|
| A | 10 | |
| E | 15 | |
| I | 12 | |
| S | 3 | |
| T | 4 | |
| P | 13 | |
| In | 1 | |

ascending order

| char | count | code |
|---|---|---|
| In | 1 | 000 |
| S | 3 | 001 |
| T | 4 | 010 |
| A | 10 | 011 |
| I | 12 | 100 |
| P | 13 | 101 |
| T | 15 | 110 |




1+3+4+10+12+13+15 = 58




In -> 0 0 0 0 0

S -> 0 0 0 0 1

T -> 0 0 0 1

A -> 0 0 1

E -> 0 0 1

I -> 0 1 0

P -> 0 1 1

CHAPTER - 17

1) Inserting n elements using.

① Aggregate Method:-

The Table doubles its size when its when its is Full.

=> The size of original Array is 5, After inserting it double its size to 10.

=> In General After 12 doublings the size is $2^k$.

Pseudo code:-

```
capacity = 1
For i = 1 to n.
    if table is will.
        new table = create new table
            with size 2x current size.
        copy elements from oldtable to.
            new table
        Table = new table.
    insert element : into table
```

Let, $k = \log(n+1) - 1$

Total cost = $O(n)\,k$

$= O(n \log n)$

Cost Per insertion = $O(\log n)$

Runtime Per insertion = $O(\log n)$

Total Time is $O(n) * \log(n+1)$

2) Accounting Method.

Charge 2*M units For each insertion when the table doubles in size From n to 2M Credit in units.

=> Total credit = $M + 2M + 4M + \ldots$

$n/2 * M = O(n)$.

Pseudo Code!-

Initialize table with Capacity = 1

For i=1 to n

IF table is Full:

new_table = create new table

with size * Current size

Copy elements From old table to new table.

Table = new table

insert element i into table

initialize Changes = 0

initialize credits = 0

For $i=1$ to $n$:

Changes $t=2$

IF table doubled in size

From $M$ to $2M$

Credits $t=M$

Total changes $= 2 * n = O(n)$

Total Credits $= M + 2M + \ldots\ldots$

$= n/2 * M = O(n)$

Cost per iteration = total $/ n$

$= O(n) / n$

$= O(1)$

Runtime. Per insertion $= O(1)$

Total time $= O(n)$