VASAVI COLLEGE OF ENGINEERING

(AUTONOMOUS)
(Affiliated to Osmania University)
Hyderabad - 500 031.

DEPARTMENT OF : CSE

NAME OF THE LABORATORY : DAA

Name K.S.I.SIVANI Roll No. -052 Page No. 89

PRELAB QUESTIONS-8

1) What is the memorization? What are the supporting languages to implement high performance algorithms with dynamic programming?

→ Memorization refers to a technique used to optimize the performance of recursive algorithms by storing the results of expensive function calls and reusing them instead of recomputing.

→ Languages for implementing high performance algoruthms with dynamic programming are python, c++ Java etc.

2) Distinguish blw top-down and bottom up approaches used in dynamic programming.

→ Top-down approach starts from the original problem and recursively solves the sub problems.

→ Bottom up approach starts from the smallest subproblems & iteratively builds up the sol^ns to

to larger subproblems until the original problem is solved.

3) What is dynamic programming? What is the optimal substructure and optimal sol^n?

→ Dynamic programming is a technique used in computer science and mathematics to solve complex problems by breaking them down into overlapping sub-problems & solving them in an optimal manner.

→ Optimal substructure refers to the property of a problem where an optimal sol^n to the problem can be constructed from optimal solutions to its subproblems

→ Optimal sol^n in the context of dynamic programming refers to the best possible sol^n to the original problem.

4) Write algorithm to print optimal order sequence of matrix chain multiplication and derive the time complexity: Time complexity: $O(n^3)$.

```
MatrixChainOrder(p):
    n = length(p)-1;
    m=[[0]*n for _ in range(n)]
    s=[[0]*n for _ in range(n)]
    for l=2 to n:
        for i=1 to n-l+1:
            j= i+l-1
            m[i][j] = infinity;
            for k= i to j-1:
                q = m[j][k]+m[k+1][j]+p[i-1]
                ib q <m[i][j]:          *p[k]*p[j];
                    m[i][j]=q;
                    s[i][j]=k
PrintOptimalParenthesis(s,i,j):
    ib i==j:
        print("A"+i);
    else:
        print"(" PrintOptimalParenthesis(s,i, s[i][j])
                PrintOptimalParenthesis(s, s[i][j]+1, j)
        print ")"
```

5) List different problems solved with dynamic programming And specify the time complexity reduction with brute force sol^n v/s dynamic programming sol^n.

* 1) Knapsack problem
2) Longest Common Subsequence
3) Matrix Chain Multiplication
4) Travelling Salesman Problem
5) Coin Change Problem.

Dynamic programming can lead to significant time complexity ~~O(2^n)~~ reductions compared to brute force solutions often transforming exponential or factorial time complexity into polynomial or linear time complexity.

PRELAB PROGRAMS:8:

1) Implement Matrix Chain multiplication algorithm:
a) top-down approach solution with recursion:

```
def matrix_chain_multiplication(p):
    n = len(p) - 1
    memo = [[-1] * n for _ in range(n)]
```

```
return matrix_chain_multiply(p, 1, n, memo)

def matrix_chain_multiply(p, i, j, memo):
    if i == j:
        return 0
    if memo[i-1][j-1] != -1:
        return memo[i-1][j-1]
    memo[i-1][j-1] = float('inf')
    for k in range(i, j):
        cost = matrix_chain_multiply(p, i, k, memo)
             + matrix_chain_multiply(p, k+1, j, memo)
             + (p[i-1] * p[k] * p[j])
        memo[i-1][j-1] = min(memo[i-1][j-1], cost)
    return memo[i-1][j-1]
```

2) Implement Matrix Chain Multiplication algorithm using "iterative sol" with bottom up approach.

```
#include <stdio.h>
#include <limits.h>
void printOptimalParanthesis(int s[][100], int i, int j)
```

```
void matrixChainOrder( int p[], int n)
{ int m[100][100];
  int s[100][100];
  for (int i = 1;i<n; i++) { m[i][i] = 0;}
  for (int l=2; l<n; l++){
     for (int i = 1; i<n-l+1; i++)
     { int j = i+l-1;
       m[i][ j ] =INT_MAX;
       for (int k =i ; k<=j-1; k++)
       { int q =m[i][k]+ m[k+1][j]+p[i-1]*p[k]*p[j
         if (q<m[i][j])
         { m[i][j]=q; s[i][j]=k; }}
  }}
  printf("Optimal Paranthesization :");
  printOptimalParanthesis(s, 1,n-1);
  printf("\n Optimal Cost: %d \n",m[1][n-1]);}
void printOptimalParenthesis(int s[][100], int i, int j)
{ if (i==j)  printf("A%d", i);
  else { printf ("(");
```

Output:

Optimal Parenthesization ((A1A2)(A3A4)(A5A6))

Minimum No. of Multiplications: 2010

```
printOptimalParenthesis(s, i, s[i][j]);
printOptimalParenthesis(s, s[i][j]+1, j);
printf(")"); }}

int main()
{ int dimensions[] = {30,35,15,5,10,20,25};
  int n = sizeof(dimensions)/sizeof(dimensions[0]);
  matrixChainOrder(dimensions, n);
  return 0; }
```

LAB PROGRAMS: 8:

Huffmann Coding:

```
#<stdio.h>
#
```

VASAVI COLLEGE OF ENGINEERING

(AUTONOMOUS)
(Affiliated to Osmania University)
Hyderabad - 500 031.

DEPARTMENT OF : CSE

NAME OF THE LABORATORY : DAA

Name K.S.I.SIVANI Roll No. -052 Page No. 96

#inclu

LAB PROGRAMS - 8:

→ ## Huffmann Coding:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 100
struct MinHeapNode{
        char data;
        unsigned frequency;
        struct MinHeapNode * left, * right;};
struct MinHeap{
    unsigned size;
    unsigned capacity;
    struct MinHeapNode **array;};
struct MinHeapNode *newNode(char data,unsigned freq)
{ struct MinHeapNode * temp = (struct MinHeapNode)* malloc
                                (sizeof(struct MinHeapNode));
    temp → left = temp → right = NULL;
    temp → data = data;
```

```
    temp -> frequency = frequency;
    return temp; }
struct MinHeap *createMinHeap(unsigned capacity)
{   struct MinHeap * minHeap
        =(struct MinHeap *) malloc(sizeof(struct MinHeap));
    minHeap->size=0;
    minHeap->capacity = capacity;
    minHeap -> array =(struct MinHeapNode **) malloc
                    (minHeap-> capacity*
                        sizeof(struct MinHeapNode*));
    return minHeap; }

void swapMinHeapNode( struct MinHeapNode ** a,
                      struct MinHeapNode **b)
{ struct MinHeapNode *t = *a;
    *a= *b;  *b=t; }

void minHeapify (struct MinHeap *minHeap , int idx)
{  int smallest =idx;
   int left =2*idx+1;
```

```
int right = 2*idx +2;
if (left <minHeap→size && minHeap→array[left]→frequency
        <minHeap→array[smallest]→frequency)
    smallest = left;
if (right <minHeap→size && minHeap→array[right]→frequency
        <minHeap→array[smallest]→frequency)
    smallest = right;
if (smallest != idx)
    swapMinHeapNode(&minHeap→array[smallest],
                    &minHeap→array[idx]);
    minHeapify (minHeap, smallest);}}
int isSizeOne (struct MinHeap *minHeap)
{ return (minHeap→size == 1); }
struct MinHeapNode * extradMin(struct MinHeap *minHeap)
{ struct MinHeapNode *temp = minHeap→array[0];
  minHeap→array[0] = minHeap→array[minHeap→size-1];
```

```
--minHeap->size;
minHeapify(minHeap,0);
return temp;}
void insertMinHeap(struct MinHeap *minHeap,
                   struct MinHeapNode *minHeapNode)
{ ++minHeap->size;
  int i = minHeap->size-1;
  while(i && minHeapNode->frequency < minHeap->array[(i-1)/2]
                                        ->frequency)
  { minHeap->array[i] = minHeap->array[(i-1)/2];
    i = (i-1)/2;}
  minHeap->array[i] = minHeapNode;}
void buildMinHeap(Struct MinHeap *minHeap).
{int n = minHeap->size-1;
  int i;
 for(i=(n-1)/2;i>=0;--i).{minHeapify(minHeap,i);}}
void printArray(int arr[], int n)
{ int i;
  for(i=0;i<n;++i){printf("%d",arr[i]);}printf("\n");}
```

```
int isLeaf(struct MinHeapNode *root)
 { return !(root->left) && !(root->right); }

struct MinHeap * createAndBuildMinHeap(char data[],
                                       int frequency[], int size)

{ struct MinHeap *minHeap = createMinHeap(size);
  for(int i=0; i<size; ++i)
  {minHeap-> array[i] = newNode(data[i], frequency[i]); }

  minHeap->size = size;
  buildMinHeap(minHeap); return minHeap; }

struct MinHeapNode *HuffmannTree(char data[],
                                 int frequency[], int size)
{ struct MinHeapNode *left, *right, *top;
  struct MinHeap * minHeap = createAnd BuildMinHeap
                             (data, frequency, size);
  while(!isSizeOne(minHeap))
  { left = extractMin(minHeap);
    right = extractMin(minHeap);
top = newNode('$', left->frequency + right->frequency);
```

```
    top->left = left;
    top->right = right;
    insertMinHeap(minHeap, top);}
  return extractMin(minHeap);}
void printCodes(struct MinHeapNode *root, int arr[],
                                          int top)
{ if (root->left)
  { arr[top]=0;
    printCodes(root->left, arr, top+1);}
  if(root->right)
  { arr[top]=1;
    printCodes(root->right, arr, top+1);}
  if(isLeaf(root))
  { printf("%c:", root->data);
    printArray(arr, top);}}
void HuffmannCodes(char data[], int frequency[], int size)
{ struct MinHeapNode *root = buildHuffmanTree(data,
                                          frequency, size)
  int arr[MAX_TREE_HT], top=0;
  printCodes(root, arr, top);}
```

```
int main()
{ char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
  int frequency[] = {5, 9, 12, 13, 16, 45};
  int size = sizeof(data)/sizeof(data[0]);
  printf("Huffmann Codes:\n");
  HuffmannCodes(data, frequency, size);
  return 0; }.
```

Output:

```
Huffmann codes:
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

2) Coin Change Problem: using dynamic programming

```
#include <stdio.h>
#include <limits.h>
int minCoins(int coins[], int n, int amount)
{ int dp[amount+1];
   dp[0]=0;
  for(int i=1; i<=amount; i++)
  { dp[i]=INT_MAX; }
 for(int i=1; i<= amount; i++)
 { for(int j=0; j<n; j++){
    if (coins[j]<=i)
     { int subRes = dp[i-coins[j]];
       if (subRes != INT_MAX && subRes+1 < dp[i])
          { dp[i] = subRes+1; }
     }}
  }
  return dp[amount];}

int main()
 { int coins[] = {1,2,5,6};
   int n = sizeof(coins)/ sizeof(coins[0]);
```

```
int amount = 11;
int minCoinCount = minCoins(coins, n, amount);
printf("Minimum no. of coins required to make
%d : %d\n", amount, minCoinCount);
return 0;}
```

O/P:

Minimum no. of coins required to make 11 : 2

3) Subset Problem using dynamic programming.

```
#include <stdio.h>
#include <stdbool.h>

bod subsetSum(int set[], int n, int sum){
    bool subset[n+1][sum+1];
for(int i=0;i<=n;i++)
  { subset[i][0] =true; }
for(int i=1; i<=sum;i++)
  {  subset[0][i] =false; }
for (int i= 1; i<=n; i++)
{ for(int j = 1; j<=sum; j++)
```

```
    if (j < set[i-1])
      { subset[i][j] = subset[i-1][j];
    else
      { subset[i][j] = subset[i-1][j]
                    || subset[i-1][j-set[i-1]];}
  } return subset[n][sum];}
int main()
{ int set[] = {3,34,4,12,5,2};
  int sum = 7;
  int n = sizeof(set)/sizeof(set[0]);
  if (subsetSum(set, n, sum))
   {printf("Subset with sum %d exists\n", sum);}
  else
   {printf("Subset with sum %d does not exist\n", sum);}
  return 0; }
```

O/P

Subset with sum 7 exists

20/12/23

VASAVI COLLEGE OF ENGINEERING

(AUTONOMOUS)
(Affiliated to Osmania University)
Hyderabad - 500 031.

DEPARTMENT OF : CSE

NAME OF THE LABORATORY : DAA

Name K.S.I. SIVANI Roll No. - 052 Page No. 106

PRELAB QUESTIONS-9:

1) What is Longest common subsequence problem?

Ans Longest Common Subsequence is a classic problem solved using dynamic programming. This problem deals with finding a longest common subsequence that is common to 2 (or) more given sequences. A subsequence is a sequence that can be derived from another subsequence by deleting some or no elements without changing the order of remaining elements.

2) What is the LCS for the strings given below?

S1 = abbcdbbacbcd S2 = cabcd.

		a	b	b	c	d	b	b	a	c	b	c	d
	0	1	2	3	4	5	6	7	8	9	10	11
c	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	1	1	1	1	1
b	2	0	1	1	1	1	1	1	1	1	2	2	2
c	3	0	1	1	2	2	2	2	2	2	2	3	3
d	4	0	1	1	1	3	3	3	3	3	3	3	4

Longest common subsequence : cabcd

3) Analyze the time complexity of LCS problem:

Time complexity of longest common subsequence using dynamic programming is O(mn); where m,n are length of input sequences.

Creating LCS array: O(mn)

Filling the LCS array: O(mn)

Backtracking: O(min(m,n)).

4) What is Travelling Sales person problem?

Ans: Travelling Salesperson is a computational problem in computer science which seeks to find the shortest possible route for a salesman whoneeds to visit a set of cities & return to the starting city, visiting each city exactly once. which is solved using dynamic programming.

5) Analyse the time complexity of TSP.

Time complexity of TSP using dynamic programming is $O(n^2 * 2^n)$.

PRELAB PROGRAMS-9:

1) Travelling Sales Person Problem:

```
#include <stdio.h>
#include <limits.h>
# define V 4
int min (int a, int b){ return (a<b)? a:b; }
int tsp(int graph [][V], int mask, int pos, int
                                        dp[ ][V])
{ if (mask == (1<<V)-1) {return graph[pos][0];}
  if(dp[mask][pos] != -1){ return dp[mask][pos];}
  int minCost = INT_MAX;
  for(int city=0; city<V; city++)
  { if((mask & (1<<city))==0)
    {int newMask = mask|(1<<city);
     int cost = graph[pos][city]+tsp(graph, newMask,
                                        city, dp);
     minCost = min(minCost, cost);}}
```

```
    dp[mask][pos]=minCost;
    return minCost;}
int main()
{ int graph[v][v]={{0,10,15,20},{10,0,35,25},
                   {15,35,0,30},{20,25,30,0}};

int dp[1<<v][v];
 for(int i=0;i(1<<v);i++)
 {for(int j=0;j<v;j++) {dp[i][j]=-1;}}
   int mask=1;
   int pos=0;
   int minCost = tsp(graph,mask,pos,dp);
   printf("Minimum Cost: %d\n", minCost);
   return 0;}
```

O/P:

Minimum Cost : 80

2) ~~Subset Sum Problem:~~

#

2) Longest Subsequence Problem:

```
#include <stdio.h>
#include <string.h>
int max(int a, int b) {return (a>b)?a:b;}
void lcs(char *X, char *Y, int m, int n)
 { int L[m+1][n+1]; int i,j;
   for(i=0; i<=m; i++)
   { for(j=0; j<=n; j++)
     { if (i==0 || j==0) {L[i][j]=0;}
       else if (X[i-1] == Y[i-1])
         { L[i][j] = L[i-1][j-1] + 1; }
       else
       { L[i][j] = max(L[i-1][j], L[i][j-1]); }}
   int index = L[m][n];
   char lcs[index+1];
   lcs[index] = '\0';
```

```
i=m; j=n;
while(i>0 && j>0)
{ if (x[i-1] == Y[j-1])
  { lcs[index-1] = x[i-1];
    i--; j--; index--; }
  else if (L[i-1][j] > L[i][j-1]) {i--;}
  else {j--;}
  printf("LCS : %s\n", lcs);}

int main()
{ char x[] = "AGGTAB";
  char Y[] = "GXTXAYB";
  int m = strlen(x); int n = strlen(Y);

  lcs(x, Y, m, n);
  return 0;}
```

O/P: LCS : GTAB.

LAB PROGRAM 9:

→ Longest increasing subsequence:

```
#include <stdio.h>
#include <stdlib.h>
int max(int a, int b)
{ return (a>b) ? a:b;}

int main()
{ int n,m,i,j;
 printf("Enter the no. of integers: ");
 scanf("%d", &n);
  int a[n],t[n];
 for(i=0; i<n;i++) { t[i]=1;}
 for(i=0 ; i<n;i++) {scanf("%d",&a[i]);}
 for(i=1;i<n ;i++)
  { for(j=0; j<=i-1;j++)
     { if(a[j] <a[i])
        {t[i]=max(t[i],t[j]+1) ;}}}}
 m=0;
 for(i=0;i<n;i++) { if (m<t[i]){m=t[i];}}
```

```
printf("Length of the longest increasing subsequence: %d
return 0;
}
```

O/P:

Enter the no. of integers: 7

3
4
-1
0
6
2
3

Length of the longest increasing subsequence: 4.