LAB NO. : 8

DATE

DYNAMIC PROGRAMMING

Lab Exercises :

1. Write a Program to implement 0/1 knapsack using dynamic programming technique and analyze the complexity.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

int knapsack (int c, int W[], int P[], int n) {
   int i, f;
   int k[2][c+1];
  for( i=0; i<=n; i++) {
     for(f=0; f<=c; f++) {
        if ( i==0 || f==0)
           k[i%2][f] = 0;
     else if (W[i-1] <= f) {
     k[i%2][f] = max( P[i-1] + k[(i-1)%2]
           [f-W[i-1]], k[(i-1)%2][f]);
   } else
        k[i%2][f] = k[(i-1)%2][f];
   }
}
```

DATE

```
    return k[n%2][c];
}

int main() {
    int n;
    cout << "Enter total no. of elements: ";
    cin >> n;
    cout << endl;

    int w[n], c, p[n];
    cout << "Enter Capacity: " ;
    cin >> c;
    cout << "Enter the weights: ";
    for( int i = 0; i<n; i++ ) {
        cin >> w[i]; }
    cout << "Enter the profits: ";
    for(int i=0; i<n; i++ ) {
     cin >> p[i]; }

    cout << knapsack( c,w,p,n );
}
```

DATE

O/p-

Enter total no. of elements : 3
Enter capacity : 10
Enter the weight : 3 5 7
Enter the profits : 20 10 5

⇒ 30

Time Complexity Analysis

The time complexity of the 0/1 knapsack problem using dynamic programming is O(nW), where n is the number of items and W is the maximum weight that the knapsack can carry. This is because we create a table of size (n+1) x (W+1) to store the solutions to all subproblems.

DATE

2. Write a program to find the optimal order of multiplication in a chain of matrices and analyze the complexity.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

int MatrixChainOrder(int p[], int i, int j) {
  if (i==j)
      return 0;
  int k;
  int mini = INT_MAX;
  int count1;
  for(k=i; k<j; k++) {
    count1 = MatrixChainOrder(p,i,k) +
  MatrixChainOrder(p,k+1,j) + p[i-1] * p[k] * p[j];
  mini = m(count1, mini);
}
return mini;
}

int main() {
  int n;
  cout << "Enter the no. of elements: ";
```

DATE

```
cin >> n;
int arr[n];
cout << "Enter the elements: ";
for(int i=0; i<n; i++)
    cin >> arr[i];
cout << "Minimum no. of multiplication is "
   << MatrixChainOrder(arr, 1, n-1);
return 0;
}
```

Output

```
Enter the number of elements: 5
Enter the num element: 40 20 30 10 30
Minimum number of multiplications is 26000.
```

Time Complexity Analysis

The time complexity of matrix chain multiplication using dynamic programming (DP) is $O(n^3)$ where n is the number of matrices in the chain.

DATE

3. Write a program to find the shortest path between every pair of vertices in a given graph using dynamic programming technique.

```
#include <bits/stdc++.h>
using namespace std;

void printSolution (int dist[][20], int V) {
  cout <<"The following matrix shows the
  shortest " distances between every
  pair of vertices \n";

for ( int i = 0; i < V; i++) {
    for ( int j = 0; j < V; j++ ) {
      if ( dist[i][j] == 999)
          cout << "INF"
            << " ";
      else
          cout << dist[i][j] << " ";
  }
  cout << endl;
}
}
```

DATE

```
void floydWarshall ( int dist[20][20], int V) {
 int i, j, k ;
  for( k=0; k < V; k++) {
     for( i=0; i<V; i++) {
        for( j=0; j<V; j++) {
   if( dist [i][j] >(dist [i][k]+ dist [k][j])
   && ( dist [k][j] != 999 && dist [i][k] != 999))
   dist [i][j] = dist [i][k] + dist [k][j] ;
 }
 }
 }
printSolution (dist, V);

int main () {
 int V ;
 cout << " Enter no. of vertices: " ;
 cin >> V;
int graph [20][20] ;
cout << " Enter the distances, Enter 999 if
  path does not exist " ;
for(int i=0; i<V; i++) {
    for( int j=0 ; j<V; j++) {
      cin >> graph [i][j]; }
}
```

DATE

```
floydWarshall(graph, V);
return 0;
}
```

Output :

```
Enter number of vertices: 4
Enter the distances, Enter 999 if path
doesn't exist:  0   5   999  10
                999 0   3    999
                999 999 0    1
                999 999 999  0
```

The following matrix shows the shortest distances between every pair of vertex.

```
  0   5   8   9
INF   0   3   4
INF INF   0   1
INF INF INF   0
```

Time

The t
algor
of v
algor
verti
to c
in o

DATE

Time Complexity Analysis:

The time complexity of the Floyd-Warshall algorithm is $O(n^3)$, where n is the number of vertices in the graph. This is because the algorithm needs to consider all possible pairs of vertices in order to and for each pair, it needs to consider all possible intermediate vertices in order to compute the shortest path between them.

LAB NO.: 9

DATE

BACKTRACKING :

LAB EXERCISES :

1. Write a program to find the Maximum clique of a graph using backtracking and analyze the complexity.

```
# include <bits/stdc++.h>
using namespace std;
const int MAX = 100;
int store[MAX], n;
int graph[MAX][MAX];
int d[MAX];

bool is_clique(int b) {
  for(int i = 1; i < b; i++) {
    for(int j = i+1; j < b; j++) {
      if(graph[store[i]][store[j]] == 0)
        return false; }
  }
  return true;
}
```

DATE

```
int maxCliques (int i, int l) {
    int max_ = 0;
for ( int j= i+1 ; j<=n ; j++) {
    store [l] = j ;
if ( is_clique ( l+1 )) {
    max_ = max ( max_ , l );
    max_ = max ( max_ , maxCliques (j , l+1 ));
 }
}
return max_ ;
}

int main () {
   int n ;
   cout << " Enter the number of nodes: " << endl;
   cin >> n ;
   int edges [n] [2];
   cout << " Enter edges relation: "
   for ( int i = 0 ; i<n ; i++ ) {
      for ( int j=0 ; j<2 ; j++) {
       cin >> edges [i] [j] ; }
    }
```

DATE

```
for (int i=0; i < size; i++) {
    graph [edges [i][0]][edges [i][1]] = 1;
    graph [edges [i][1]][edges [i][0] = 1;

    d [edges [i][0]] ++ ;
    d [edges [i][1]] ++ ;
}
cout << maxCliques (0, 1);
return 0;

}
```

Output:

```
Enter the number of nodes: 4
Enter edges relation: 1 2 2 3 3 1 4 2

= 3
```

Time Complexity Analysis

The time complexity of this algorithm is $O(2^n * n^2)$, where 'n' is the number of vertices in the graph. This is because 'max cliques' is a recursive function that calculates all possible subsets of the vertices

and calls
given the
The 'is_
it needs
graph.

{
[i]] = 1;
[0] = 1;

4
2 3 3 1 4 2

algorithm is
the number
his is because
e function that
ets of the vertices

DATE

and calls 'is_clique' function to check if the given the set of vertices form a clique or not. The 'is_clique' function takes $O(n^2)$ time as it needs to check for all the edges in the graph.

DATE

2. Implement Travelling Sales Person problem using backtracking technique.

```
#include <bits/stdc++.h>
using namespace std;
#define V 100

void tsp( int graph[][V], bool v[], int currpos,
int n, int count, int cost, int& ans, int verts)
{
if ( count == n && graph[currpos][0] )
 {
  ans = min ( ans, cost + graph[currpos][0] );
  return;

for( int i = 0; i < n; i++) {
  if( !v[i] && graph[currpos][i] )
  {
    v[i] = true;
   tsp[graph, v, i, n, count + 1, cost + graph[currpos][i]
   , ans, verts);
    v[i] = false;
  }
 }
};
```

```
int main() {
int verts;
cout << " N
cin >> vert
int graph [
cout << "

for( int i =
   for( int

       cin
   }
}
bool v[V]

for( int i=0
     v[i] =
   v[0] = t

int ans =
tsp ( graph
cout <<
```

oblem

], int currpos,
as , int verts)

os][0])

];

][i])

DATE

```
int main() {
  int verts;
  cout << " Number of Cities" << endl;
  cin >> verts;
  int graph[V][V];
  cout << " Enter adj matrix of cities " << endl;

  for( int i=0; i< verts; i++) {
     for( int j=0; j< verts; j++) {

          cin >> graphs[i][j];
     }
  }
  bool v[V];

  for( int i=0; i< verts; i++) {
       v[i] = false;
    v[0] = true;

  int ans = INT_MAX;
  tsp( graph, v, 0; verts, 1, 0, ans, verts);
  cout <<
```

DATE

Output:

The time complexity of the backtracking approach for TSP is O(n!), where n is the number of cities. There are n! possible permutations of the cities to visit, and the backtracking algorithm will explore all of them in the worst case.

```
NUMBER OF CITIES
4
enter adj matrix of cities
0  10  15  20
10  0  35  25
15  35  0  30
20  25  30  0

⇒60
```

3

Find the opti
backtracking te
functions.

```
#include <iostr
using namespac

int knapsack (i
  int cw, int
{
  if (cw > c)
  if (cp > ma

for ( int i = n
  int ub = c
  if (ub < n

  cw += w[i
  cp += v[i]

if (cw <= c
  knapsack (w

  cw -= w[i]
  cp -= v[i]
}
```

DATE

3 Find the optimal packing of the knapsack using backtracking technique use proper bounding functions.

```
#include <iostream>
using namespace std;

int knapsack (int w[], int v[], int n, int c, int cp,
  int cw, int &maxp)
{
 if (cw > c) return 0;
 if (cp > maxp) maxp = cp;

 for ( int i = n ; i >= 1 ; i--) {
   int ub = cp + (c - cw) * v[i] / w[i] ;
   if (ub < maxp) return 0;

   cw += w[i] ;
   cp += v[i] ;

 if (cw <= c)
  knapsack (w, v, i-1, c, cp, cw, maxp);

  cw -= w[i] ;
  cp -= v[i] ;
}
```

DATE

```
    return maxp;
}

int main() {
    int n, c;
    cout << "Enter the no. of items: ";
    cin >> n;
    int w[n+1], v[n+1];
    cout << "Enter the weights and values of
    the items: \n";

    for(int i=1; i<=n; i++)
        cin >> w[i] >> v[i];
    cout << "Enter the capacity of the knapsack: ";
    cin >> c;

    int maxp = 0;
    knapsack(w, v, n, c, 0, 0, maxp);
    cout << "The maximum profit is: "
         << maxp << endl;

    return 0;
}
```

OUTPU

```
Enter th
Enter th
 2  10
 3  20
 4  30
 5  40

Enter the
The maxim
```

Time comp

The time
using ba
the num
either inc
Therefore,
of items

DATE

OUTPUT:

```
Enter the no. of items: 4
Enter the weight and values of the items:
2 10
3 20
4 30
5 40

Enter the capacity of the knapsack: 7
The maximum profit is: 50.
```

Time Complexity:

The time complexity of the knapsack algorithm using backtracking is $O(2^n)$, where n is the number of items. There are two choices either include the item in the knapsack or exclude it. Therefore, there are 2^n possible combinations of items that can be considered.

DATE

4. Implement container loading problem using backtracking technique.

```
# include <iostream>
using namespace std;

const int MAX_N = 100;
int num_boxes, capacity;
int boxes [MAX_N];

void backtracking (int cur_capacity, int cur_box)
{
if(cur_box == num_boxes || cur_capacity >= capacity)
{
 cout << " solution: " ;
for( int i=0; i < cur_box ; i++) {
    cout << boxes [i] << " " ;
 }
cout << endl;
return;

if ( cur_capacity + boxes [cur_box] <= capacity ){
  boxes [cur_box] = - boxes [cur_box];
  backtracking (cur_capacity + abs (boxes [cur_box]),
    cur_box + 1 );
```

```
  boxes [cur_
}
backtrackin
}

int main()
 cout << "
 cin >> ca

 cout << "
 cin >> nu

 cout << "
 for ( int i
    int box
    cin >>
    boxes [i
 }
 backtrackin
  return 0
}
```

```
    boxes[cur_box] = abs(boxes[cur_box]);
}
backtracking(cur_capacity, cur_box + 1);
}

int main() {
    cout << "Enter the capacity of the container: ";
    cin >> capacity;

    cout << "Enter the number of boxes: ";
    cin >> num_boxes;

    cout << "Enter the sizes of the boxes: ";
    for (int i = 0; i < num_boxes; i++) {
        int box_size;
        cin >> box_size;
        boxes[i] = box_size;
    }
    backtracking(0, 0);
    return 0;
}
```

DATE

OUTPUT:

```
Enter the capacity of the container: 50
Enter the number of boxes: 4
Enter the sizes of the boxes: 10 20 30 40

Possible loading configurations are:
10 20
10 30
10 40
10
20 30
20
30
40
```

Time Complexity:

The time complexity is $O(2^n)$, where the n is the number of items. This is because for each item, there are two possible choices: either include it in the container or exclude it.

LAB No. : 10

DATE

BRANCH and BOUND

Lab Exercises :

40

1. Write a program to implement container loading problem using Max-Heap branch and bound technique.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;
struct item {
  int wt, vol;
  double density; };
struct node {
  int level, value, wt, volume; };

struct compare node {
  bool operator()(const node &a, const node &b) {
    return a.value < b.value; }
int x, max_wt, int max_val;
vector <item> items;
```

DATE

```
int containerloading() {
 sort(items.begin(), items.end(), [](const item &a,
 const item &b) {
  return a.density > b.density ; }
}
priority_queue <Node, vector<node> > compare node> pq;

Node root = {0,0,0,0};
pq = push (root);
int max_value = 0;
while (!pq.empty()) {
 Node current. pq.top() ;
 pq.pop();

if((current.level) == n) {
max_value = max(max_value, current_value, contains);
 }
if(current.wt + items[current.level].volume
 <= max_volume ) {
  Node left {
  current_level+1, current.value, current.wt +
  items[current_level].wt = current.volume +
  items[current_level].volume };
 pq.push(left); }
```

DATE

```
return max_volume;
}
int main() {
  cin >> x >> max.wt >> max_vol ;
  items.resize(n);
  for(int i=0; i<n; i++)
    cin >> items[i].w >> items[i].volume;
  item[i].density = (double) items[i]
  (double) items[i].volume }
  int max_value = containerloading();
  cout << "Max value: " << max_value;
}
```

OUTPUT:

```
4 10 20
2 3 4 6 5 7 3 4
max value: 9
```

complexity:

worst case: $O(2^n)$:

best case: $O(n)$.

DATE

2. Implement TSP using Branch and Bound.

```
# include <iostream>
using namespace std;
# define V 4
int Tsp( int graph[][V], int s) {
  vector <int> vertex;
  for( int i=0; i<V; i++) {
    if ( i != s)
      vector. push_back(i);
  }
  int min_path = INT_MAX;
  int cur_pathwt = 0;
  int k = i;

  for( int i = 0; i< vertex.size() ; i++) {
    current. pathwt += graph [k] [vertex [i]];
    k = vertex [i];
  }
  current_pathwt += graph [k][s];
  min_path = min (min_path, curr_path);
  while (next. permutation (vertex. begin (),
    vertex. end()));
  return min_path;
}
```

DATE

```
int main() {
   int graph [][V];
    graph [][V] ={{ 0, 10, 15, 20 },
                  { 10, 0, 35, 25 },
                  { 15, 35, 0, 30 },
                  { 30, 25, 30, 0 }};

int i = 0;
cout << " Min cost: " << endl;
cout << TSP ( graph, s );
  return 0;
}
```

OUTPUT

min cost = 80

Complexity:

$O(n^2)$:

DATE

3. Max Clique of a graph

```
# include <stdio.h>
# include <iostream>
# define v 4

int maxcliquesize;
bool issafe (int v, bool graph [v] [v], int
   cliquesize, int clique [] ) {
  for (i=0; i< cliquesize ; i++) {
    if ( graph [v] [clique [c]] == false )
    return false; }
  return true;
}
void maxClique (bool graph [v] [v], int clique [],
  int index, int cliquesize ) {
  if ( ! index == v) {
    if ( ! uniquesize > maxcliquesize )
        maxclique = cliquesize;
  }
if (issafe (index, graph, cliquesize, clique) )
 clique [clique.size ] = index;
maxclique (graph, clique, index+1, cliquesize +1);
if (v-indx- cliquesize > maxcliquesize)
   maxcliqueutil (graph clique, index+1, cliquesize);
```

```
int maxclique (bo
 maxcliquesize =
 int clique [v]
 clique [0] = 0
 maxclique util (
 return maxcliqu
}

int main () {
 bool graph [v]

 cout << maxclig
 return 0;
}

OUTPUT
maxclique = {

Time complexity:
O(2^n)
```

DATE

```
int maxclique (bool graph [v][v]);
  maxcliquesize = 1;
  int clique [v];
  clique [0] = 0;
  maxclique util (graph, clique, 1, 1);
  return maxcliquesize;
}

int main () {
  bool graph [v] [v] = { { 0, 1, 1, 0, 0 },
                         { 1, 0, 1, 1, 0 },
                         { 1, 1, 0, 1, 1 },
                         { 0, 1, 1, 0, 1 },
                         { 0, 0, 1, 1, 0 } }
  cout << maxclique (graph);
  return 0;
}
```

OUTPUT

maxclique = { 1, 2, 3, 4 }

Time complexity:

$O(2^n)$

LAB 11

DATE

APPROXIMATE ALGORITHMS

Lab Exercises

1. Program to find vertex cover of Minimum size in given directed graph.

```
# include <iostream>
# include <vector>
# include <string>
using namespace std;
# define max 100
vector <int> adj [max]
bool visited [max];
int vertex_cover [max];

void dfs (int u) {
visited [u] = true;
    for ( int v: adj [u] ) {
     if ( ! visited [v] ) {
         dfs [v] ;
    vertex_ cover [u] = 1;
    vertex_cover [v] = 1; 444
```

DATE

```
int main () {
int n, m;
for (i=0; i<m; i++) {
 int u, v;
 adj[u].push_back[v];
 adj[v].push_back[v]; }
 memset (visited, false, sizeof (visited));
 memset (vertex_cover, 0, sizeof (vertex_cover));
 for( int i=1; i<=n; i++ ) {
  if ( ! visited [i] ) {
     dfs [i] ; } }
 cout << " vertex cover " ;
 for ( int i=1; i<=n; i++ ) {
    if ( vertex_cover [i] ) {
      cout << i ; }
  }
 return 0;
}
```

DATE

OUTPUT:

6 7
1 2
1 3
2 3
2 4
3 4
4 5
4 6

Vertex cover : 4

Time Complexity:
$O(n^3)$.

DATE

2. Program implementing TSP problems

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
const int INF;
int n;
double adj[20][20];

int findminkey(double key, bool mst_set[]) {
    double min_val = INF;
    int min_idx = -1;
    for(i=0; i<n; i++) {
      if(mst_set[i] && key[i] < min_val) {
        min_val = key[i];
        min_idx = i; }}
    return min_idx; }}
double tsp_mst() ; {
    double mst.cost = 0.0;
    vector<double> key(n, INF);
    vector<bool> mst_set(n, false);
    key[0] = 0.0;
```

DATE

```
for ( i=0; i<n-1 ; i++)
  int u = find. minkey ( key. data (), mst_set,
  data ());
 mst_set [u] = true;
 for ( int v= 0; v<n; v++)
   if ( adj [u][v] && mst [v] && adj [v][u]<
          key [v] )
    key [v] = adj [u] [v]; }
}
for ( int i = 1; i< n; i++) {
    min_cost += adj [i] [parent [i] ];
  return mst_cost;
}

int main () {
  for ( int i=0; i<n; i++) {
    for ( int j=0; j<n; j++) {
      cin >> adj [i][j] ; }
  }
 double mst_cost = tspmst();
  cout << mst_cost;
  return 0;
}
```

, mst_set,

adj [v][u]<

];

DATE

OUTPUT

0	10	20	30	40
10	0	25	35	15
20	25	0	45	30
30	35	45	0	50
40	13	30	50	0

mst cost = 80
tsp cost = 160

Time Complexity:
$O(n^2)$.