**G. Laboratory Experiments**

**Total: 30 Hours (P)**

**Part – 1**

**Task 6: Dynamic Programming – Knapsack algorithm**

**AIM**

To Implement 0/1 Knapsack problem using Dynamic Programming and find its complexity.

**ALGORITHM**

1. Consider the same cases as mentioned in the recursive approach.
2. In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as rows.
3. The state DP[i][j] will denote the maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place:
   1. Fill ‘wi’ in the given column.
   2. Do not fill ‘wi’ in the given column.
4. Now we have to take a maximum of these two possibilities, formally if we do not fill the ‘ith’ weight in the ‘jth’ column then the DP[i][j] state will be the same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row.
5. So we take the maximum of these two possibilities to fill the current state.

|  |
| --- |
| **Program:**  #include <stdio.h>  int max(int a, int b) { return (a > b) ? a : b; }  int knapSack(int W, int wt[], int val[], int n)  {    int i, w;      int K[n + 1][W + 1];       // Build table K[][] in bottom up manner      for (i = 0; i <= n; i++) {          for (w = 0; w <= W; w++) {              if (i == 0 || w == 0)                  K[i][w] = 0;              else if (wt[i - 1] <= w)                  K[i][w] = max(val[i - 1]                                    + K[i - 1][w - wt[i - 1]],                                K[i - 1][w]);              else                  K[i][w] = K[i - 1][w];          }      }       return K[n][W];  }   int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

**Output**

220

**Time Complexity:** O(N \* W). where ‘N’ is the number of elements and ‘W’ is capacity.   
**Auxiliary Space:** O(N \* W). The use of a 2-D array of size ‘N\*W’.

Below is the implementation of the same approach but with optimized space complexity:

**Result**

Thus Implementing 0/1 Knapsack problem using Dynamic Programming and find its complexity was learn successfully

**Task 7:** Greedy Technique – Prim’s algorithm

**AIM**

To Find Minimum Cost Spanning Tree of a undirected graph using Prim’s algorithm

**ALGORITHM**

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

**Program:**

|  |
| --- |
| 1. #include <stdio.h> 2. #include <limits.h> 4. #define V 5 6. int minKey(int key[], int mstSet[]) { 7. int min = INT\_MAX, min\_index; 8. int v; 9. for (v = 0; v < V; v++) 10. if (mstSet[v] == 0 && key[v] < min) 11. min = key[v], min\_index = v; 13. return min\_index; 14. } 16. int printMST(int parent[], int n, int graph[V][V]) { 17. int i; 18. printf("Edge Weight\n"); 19. for (i = 1; i < V; i++) 20. printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]); 21. } 23. void primMST(int graph[V][V]) { 24. int parent[V]; // Array to store constructed MST 25. int key[V], i, v, count; // Key values used to pick minimum weight edge in cut 26. int mstSet[V]; // To represent set of vertices not yet included in MST 28. // Initialize all keys as INFINITE 29. for (i = 0; i < V; i++) 30. key[i] = INT\_MAX, mstSet[i] = 0; 32. // Always include first 1st vertex in MST. 33. key[0] = 0; // Make key 0 so that this vertex is picked as first vertex 34. parent[0] = -1; // First node is always root of MST 36. // The MST will have V vertices 37. for (count = 0; count < V - 1; count++) { 38. int u = minKey(key, mstSet); 39. mstSet[u] = 1; 41. for (v = 0; v < V; v++) 43. if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) 44. parent[v] = u, key[v] = graph[u][v]; 45. } 47. // print the constructed MST 48. printMST(parent, V, graph); 49. } 51. int main() { 52. /\* Let us create the following graph 53. 2 3 54. (0)--(1)--(2) 55. | / \ | 56. 6| 8/ \5 |7 57. | / \ | 58. (3)-------(4) 59. 9 \*/ 60. int graph[V][V] = { { 0, 2, 0, 6, 0 }, { 2, 0, 3, 8, 5 }, 61. { 0, 3, 0, 0, 7 }, { 6, 8, 0, 0, 9 }, { 0, 5, 7, 9, 0 }, }; 63. primMST(graph); 65. return 0; 66. }   Output:  $ gcc PrimsMST.c  $ ./a.out    Edge Weight  0 - 1 2  1 - 2 3  0 - 3 6  1 - 4 5 |

**Result**

Thus Finding Minimum Cost Spanning Tree of a undirected graph using Prim’s algorithm was learn successfully