Karunya INSTITUTE OF TECHNOLOGY AND SCIENCES
(Declared as Deemed to be University under Sec. 3 of the UGC Act. 1956)
A CHRISTIAN MINORITY RESIDENTIAL INSTITUTION
MHRD Approved & NAAC Accredited
Karunya Nagar, Coimbatore - 641 114, Tamil Nadu, India.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ENGINEERING AND TECHNOLOGY

LABORATORY RECORD

2022-2023

ODD SEMESTER

Name: DARWIN RAJ A
Reg. No.: URK20CS1040

20CS2018Design and Analysis of AlgorithmsLAB

KARUNYA INSTITUTE OF TECHNOLOGY AND SCIENCES
(Declared as Deemed-to-be-University under Sec-3 of the UGC Act, 1956)
Karunya Nagar, Coimbatore - 641 114, India.

NOVEMBER 2022

Karunya INSTITUTE OF TECHNOLOGY AND SCIENCES
(Declared as Deemed to be University under Sec.3 of the UGC Act, 1956)
MoE, UGC & AICTE Approved
NAAC A++ Accredited

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ENGINEERING AND TECHNOLOGY

LABORATORY RECORD

Academic Year 2022-2023

Course Code

20CS2018L

Course Name

Design And Analysis Of Algorithm

Register No. URK20CS1040

It is hereby certified that this is the bonafide record of work done by **Mr. DARWIN RAJ A** during the odd semester of the academic year 2022-2023 and submitted for the University Practical Examination held on **09.11.2022**.

Faculty-in-charge **Program Coordinator**

Examiner

TABLE OF CONTENTS

Karunya INSTITUTE OF TECHNOLOGY AND SCIENCES
(Declared as Deemed to be University under Sec.3 of the UGC Act, 1956)
MoE, UGC & AICTE Approved
NAAC A++ Accredited

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ENGINEERING AND TECHNOLOGY

LABORATORY RECORD

Academic Year 2022-2023

Course Code

20CS2018L

Course Name

Design And Analysis Of Algorithm

Register No. URK20CS1040

It is hereby certified that this is the bonafide record of work done by **Mr. DARWIN RAJ A** during the odd semester of the academic year 2022-2023 and submitted for the University Practical Examination held on **09.11.2022**.

Faculty-in-charge **Program Coordinator**

Examiner

TABLE OF CONTENTS

Ex. No. 1 Date: 10-12-2021	**Fractional Knapsack Problem using Greedy Approach**

Video Link: https://youtu.be/Y_FIS9ss2Go

Aim: To solve the given Fractional Knapsack Problem using Greedy Approach

Procedure:

1. We have to make this problem approach in such a way that we get more profit with optimal solution
2. So for that there may be many approach either we may go by weights or either by values but by taking the ratio of the profit that we are getting per given weight will give us idea of which item should be included in the container and at what fraction of weight of the item should be included
3. Then after that we will get the ratio of the highest profit per given weight and that one item will be included along with its weight
4. Then With that fraction and with the weight we will get the weight of the items we are included and by the fraction and the value we will get the total value of the container

Algorithm:

```
//Input: Items are in the descending order of their value/weight ration
Greedy-Fractional-Knapsack (w[1..n], v[1..n], W) {
        for (i = 1 to n )
            do x[i] = 0 ;
        weight = 0 ; profit = 0.0;
        for (i = 1 to n ) {
            if weight + w[i] ≤ W then {
                x[i] = 1 ;
                weight = weight + w[i] ; profit = profit + v[i};
             }
             else {
                  x[i] = (W - weight) / w[i] ;
                  weight = W ; profit = profit + v[i]*x[i];
                  break ;
             }
        }
        return x
    }
```

Source Code:

```
def knapsack (W,weights,values):
  profitratio = [v/w for v,w in zip (values,weights)]
  n=len(weights)
  a= list(range(n))
  a.sort(key = lambda i : profitratio[i], reverse = True)
  max_value = 0
  sum=0
  fraction = [0]*n
  for i in a:
    if weights[i] <=W:
      max_value += values[i]
    W -=weights[i]
    fraction[i]=1
    sum=sum+weights[i]
    else:
      fraction[i]=W/weights[i]
      max_value +=values[i]*fraction[i]
      sum=sum+weights[i]
  print(fraction)
  print("The Total Weight value of the selected items as per values and fraction of which they have selected:",s
um)
  return max_value
weights=[10,30,40,20]
values=[60,100,120,30]
W=50
print("The given weights",weights)
print("The Given Values:",values)
print("The given weight of the container:",n)
knapsack(W,weights,values)
print("The Maximum Profit for the given objects is :",knapsack(W,weights,values)
```

Output:

```
The given weights [10, 30, 40, 20]
The Given Values: [60, 100, 120, 30]
The given weight of the container: 100
[1, 1, 0.25, 0.5]
The Total Weight value of the selected items as per values and fraction of which they have
selected: 100
[1, 1, 0.25, 0.5]
The Total Weight value of the selected items as per values and fraction of which they have
selected: 100
The Maximum Profit for the given objects is : 205.0
```

Result:

Here we have to solve the given problem using greedy method and we have to take the ratio of profit per weight and we have to fill those in the container and those whole weights must not exceed above the given container weight ,hence we have bounded to the given constraints and solved the problem and got the amount of profit.

Ex.No:2 Date: 17.12.2021	Greedy algorithm for scheduling tasks with deadlines

Video Link: https://youtu.be/jfZA-boRhug

Aim:

To implement greedy algorithm for scheduling tasks with deadlines.

Algorithm:

1. Sort all jobs in decreasing order of profit.
2. Iterate on jobs in decreasing order of profit.For each job , do the following :
3. Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in this slot and mark this slot filled.
4. If no such i exists, then ignore the job.

Program:

```
def printJobScheduling(arr, t):

    n = len(arr)
    for i in range(n):
        for j in range(n - 1 - i):
            if arr[j][2] < arr[j + 1][2]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    result = [False] * t
    job = ['-1'] * t
    for i in range(len(arr)):
        for j in range(min(t - 1, arr[i][1] - 1), -1, -1):
            if result[j] is False:
                result[j] = True
                job[j] = arr[i][0]
                break
    print(job)

arr = [['a', 2, 100],   # Job Array
       ['b', 1, 19],
       ['c', 2, 27],
       ['d', 4, 25],
       ['e', 3, 15],
       ['f', 1, 120],
       ['g', 6, 150]]

print(" Maximum profit sequence of jobs is :")
printJobScheduling(arr, 5)
```

Output:

```
C:\Users\gchan\PycharmProjects\chan\venv\Scripts\python.exe
 Maximum profit sequence of jobs is :
['f', 'a', 'e', 'd', 'g']

Process finished with exit code 0
```

\

Result:

Implementing greedy algorithm for scheduling tasks with deadlines is solved.

Date: -07-01-2022	0/1-KNAPSNACK PROBLEM USING DYNAMIC PROGRAMMING
EXP.NO:3	Video Url: https://youtu.be/xNpPlTTRf7o

Aim: To Perform the 0/1 knap snack problem using Dynamic programming.

Objective: To find solution for 0/1 knapsack problem using dynamic programming technique.

Procedure:

1. We have to draw the tabular form for the better understanding and we have place the weights and profits in columns and rows respectively
2. Then we have to fill the table according to the formula for the knapsack problem
3. **V(i , j) = max { V (i-1 , j) , valuei + V(i-1 , j – weighti) }**
4. Then we have to analysis the tabular form to get the maximum profit and then we have to see which objects can be filled
5. Similar we have to code using python where we have to use the arrays to make a tabular form and we have to fill the table and return the maximum profit value
6. We have to define the function and then we have pass the aruguments.
7. Then we have to build the table using two arrays
8. Then if the row and the column value is zero then fill all rows and columns at particular with zero
9. Otherwise fill according to the formula and the return the value of the last cell as the maximum profit

Algorithm:

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W
c[0, w] = 0
fori = 1 to n {
c[i, 0] = 0
for w = 1 to W  {
ifwᵢ ≤ w
if vᵢ + c[i-1, w-wᵢ]
c[i, w] = vᵢ + c[i-1, w-wᵢ]
else c[i, w] = c[i-1, w]
else
c[i, w] = c[i-1, w]
   }

}
```

Program:

```
def knapSack(Weight, wt, values, n):
    K = [[0 for x in range(Weight + 1)] for x in range(n + 1)]
    for i in range(n + 1):
      for w in range(Weight + 1):
        if i == 0 or w == 0:
          K[i][w] = 0
        elif wt[i-1] <= w:
          K[i][w] = max(values[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
        else:
          K[i][w] = K[i-1][w]
    return K[n][Weight]
values=[100,20,60,40]
wt=[3,2,4,1]
Weight=5
n = len(values)
print("Given values are:",values)
print("Given weights are :",wt)
print("The Maximum capcity of a Knapsnack is:",Weight)
print("The Maximum profit is:",knapSack(Weight, wt, values, n))
```

Output:

```
Given values are: [100, 20, 60, 40]
Given weights are : [3, 2, 4, 1]
The Maximum capacity of a Knapsack is: 5
The Maximum profit is: 140
```

Result:

Here we have to done the problem using dynamic programming and implemented 0/1 knapsnack problem and we have analysed the tabular form and got the maximum profit as the output.

EXP.NO:4	Implement dynamic programming solution for longest common subsequence problem
Date:21-02-2022	YouTube Link: https://youtu.be/h-8aqvJq4j4

Aim: To Implement Dynamic programming solution for longest common subsequence problem

Objective: To write a program to compute the longest common subsequence of two strings

Procedure:

1. First we have to take two strings as input to find the longest common string of them
2. Then we have create a table structure where first row and the first column is zero
3. Then if both the letters are matched then we have to add 1 to the diagonal element
4. If they are not common then we have to fill that cell with the maximum of the of the diagonal elements and then add them
5. Then we have to print the longest subsequence of the elements and its length

Algorithm:

Algorithm:

```
Algorithm: LCS-Length-Table-Formulation (X, Y)
m := length(X)
n := length(Y)
for i = (1 to m)    C[i, 0] := 0
for j = (1 to n )   C[0, j] := 0
for i = 1 to m
  for j = 1 to n {
    if x[i] = y[j] {
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'      }
    else {
      if C[i -1, j] ≥ C[i, j -1] {
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := 'U'        }
      else {
        C[i, j] := C[i, j - 1]
        B[i, j] := 'L'        }
    }
  }
return C and B
```

```
Algorithm: Print-LCS (B, X, i, j)
if (i = 0 and j = 0)     return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
  Print(x_i)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else print-LCS(B, X, i, j-1)
```

Source Code:

```
def lcs_algo(S1, S2, m, n):
    L = [[0 for x in range(n+1)] for x in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif S1[i-1] == S2[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])
    index = L[m][n]
    lcs_algo = [""] * (index+1)
    lcs_algo[index] = ""
    i = m
    j = n
    while i > 0 and j > 0:
      if S1[i-1] == S2[j-1]:
        lcs_algo[index-1] = S1[i-1]
        i -= 1
        j -= 1
        index -= 1
      elif L[i-1][j] > L[i][j-1]:
            i -= 1
      else:
        j -= 1
    print("S1 : " + S1 + "\nS2 : " + S2)
    print("Longest Common Sequence: " + "".join(lcs_algo))


S1 = input("Enter First String:")
S2 = input("Enter Secound String")
m = len(S1)
n = len(S2)
lcs_algo(S1, S2, m, n)
```

Output:

```
Enter First String:stone
Enter Secound Stringlongest
S1 : stone
S2 : longest
Longest Common Sequence: one
```

Result: Here we have found the longest common subsequence string which has the maximum length and we have return that string.

Date:04-02-2022	Finding optimal path for travelling salesman problem using dynamic programming
EX.NO:5	Video URl: https://youtu.be/GScv3wqFs5c

Aim: Implement dynamic programming solution to find optimal path for travelling salesman problem.

Objective: To write a program to find optimal path for traveling salesman problem using dynamic programming.

Procedure:

1. First we have to take a graph of matrix 4x4
2. Then we have setup the starting point
3. We have create a vertex list so that we can append all the possible values except the taken initial value
4. Then we have take an function called permutation that will give the all the combination ways of the given vertex
5. Then we have to add the values by see the matrix to get the optimal path
6. After adding values by seeing in the matrix we will get the optimal path by taking the minimum path
7. The that path is called optimal path

Algorithm:

Algorithm: Traveling-Salesman-Problem

$C(\{1\}, 1) = 0$

for $s = 2$ to n do

  for all subsets $S \in \{1, 2, 3, \dots, n\}$ of size s and containing 1

    $C(S, 1) = \infty$

  for all $j \in S$ and $j \neq 1$

    $C(S, j) = \min \{C(S - \{j\}, i) + d(i, j)$ for $i \in S$ and $i \neq j\}$

Return $\min_j C(\{1, 2, 3, \dots, n\}, j) + d(j, i)$

Source Code:

```
from sys import maxsize
from itertools import permutations
V=4

def sales_man(garph ,s):
  vertex=[]
  for i in range(4):
    if i != s:
      vertex.append(i)
  min_path = maxsize
  next_permutation = permutations(vertex)
  for i in next_permutation:
    current_pathweight = 0
    k=s
    for j in i:
      current_pathweight += graph[k][j]
      k=j
    current_pathweight += graph[k][s]
    min_path = min(min_path, current_pathweight)
    if (min_path == current_pathweight):
      pos=i
  return min_path

graph = [[0,10,15,20],[5,0,9,10],[6,13,0,12],[8,8,9,0]]
s=0
print("The Taken Graph is:",graph)
print("The starting point is:",s)
print("The Number of Vertex is:",V)
print("The best path that a sales man should take is:",sales_man(graph,s))
```

Output:

```
The Taken Graph is: [[0, 10, 15, 20], [5, 0, 9, 10], [6, 13, 0, 12], [8,
8, 9, 0]]
The starting point is: 0
The Number of Vertex is: 4
The best path that a sales man should take is: 35
```

Result:

Here we have written an algorithm that takes graph as input and then return the minimum path by taking the starting point and the vertex as in constraints after traversing to each and every path the algorithm finds the optimal path for the travelling then return the path .

Ex. No: 6	Implement the Prim's algorithm for finding Minimum Spanning Tree in a graph
Date: 11-02-2022	

Video link: https://youtu.be/6jRP7b50Dq8

Aim:

Toimplement the Prim's algorithm for finding Minimum Spanning Tree in a graph.

Algorithm:

Step 1: Implement the Prim's algorithm.
Step 2: Initialize the variables initially.
Step 3: Include the boolean expression.
Step 4: Use while loop.

Sourcecode:

```
N = 6
M = [[0,4,3,0,0,0],
    [4,0,1,2,0,0],
    [3,1,0,4,0,0],
    [0,2,4,0,2,0],
    [0,0,0,2,0,6],
    [0,0,0,0,0,0]]
s_node = [0, 0, 0, 0, 0, 0]
no_edge = 0
s_node[0] = True
print("Minimum Spanning Tree: \n")

while (no_edge < N - 1):
  min = 9999999
  a = 0
  b = 0
  for m in range(N):
    if s_node[m]:
      for n in range(N):
        if ((not s_node[n]) and M[m][n]):

          # not in selected and there is an edge
          if min > M[m][n]:
            min = M[m][n]
            a = m
            b = n

  print("Edge", str(a) + "-" + str(b) + " : Key " + str(M[a][b]))
  s_node[b] = True
  no_edge += 1
```

Output:

```
Minimum Spanning Tree:

Edge 0-2 : Key 3
Edge 2-1 : Key 1
Edge 1-3 : Key 2
Edge 3-4 : Key 2
Edge 4-5 : Key 6
```

Result:

The implementation of the Prim's algorithm for Minimum Spanning Tree in a graph is found.

EX.NO:7	**Kruskal algorithm**
DATE:18-03-2022	**Youtube Url: https://youtu.be/17A_12Z3kho**

Aim : Implement Kruskal algorithm for finding out the minimum cost spanning tree

Objective: To write a program to implement Kruskals's algorithm for finding MST in graph.

Procedure:

1. Sort all the edges of the graph from low weight to high.
2. Take the edge of the lowest weight and add it to the required spanning tree. If adding this edge creates a cycle in the graph, then reject this edge.
3. Repeat this process until all the vertices are covered with the edges.

Algorithm:

MST- KRUSKAL (G, w)

1. A ← Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A

Source Code:

```
class Graph:
    def __init_(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])


    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e     =     e     +     1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))
```

```
g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)
g.kruskal_algo()
```

Output:

```
The Kurskal Algorithm Sorting:
Start - Destination : Weight
1 - 2: 2
2 - 5: 2
2 - 3: 3
3 - 4: 3
0 - 1: 4
```

Result:

Here we have sorted the graph with using kurskal algorithm and found the minimum weighted and optimal path from the source and the destination

Ex.No.8	Implement single source shortest path algorithm
04/03/2022	

Video URL:

https://youtu.be/JGklMb6HLKk

Aim:

To Implement single source shortest path algorithm.

Algorithm:

Source Code:

```
import sys

class Graph():
    def __init_(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def printSolution(self, dist):
        print("Vertex \tDistance from source")
        for node in range(self.V):
            print(node, "\t", dist[node])
    def minDistance(self, dist, sspa):
        min = sys.maxsize
        for u in range(self.V):
            if dist[u] < min and sspa[u] == False:
                min = dist[u]
```

```
                min_index = u
        return min_index
    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sspa = [False] * self.V
        for cout in range(self.V):
            x = self.minDistance(dist, sspa)
            sspa[x] = True
            for y in range(self.V):
                if self.graph[x][y] > 0 and sspa[y] == False and \
                dist[y] > dist[x] + self.graph[x][y]:
                        dist[y] = dist[x] + self.graph[x][y]
        self.printSolution(dist)
g = Graph(9)
g.graph = [[0, 1, 0, 0, 0, 0, 0, 6, 0],
        [1, 0, 6, 0, 0, 0, 0, 10, 0],
        [0, 6, 0, 5, 0, 1, 0, 0, 3],
        [0, 0, 5, 0, 8, 13, 0, 0, 0],
        [0, 0, 0, 8, 0, 9, 0, 0, 0],
        [0, 0, 1, 13, 9, 0, 3, 0, 0],
        [0, 0, 0, 0, 0, 3, 0, 4, 5],
        [6, 10, 0, 0, 0, 0, 4, 0, 8],
        [0, 0, 5, 0, 0, 0, 7, 6, 0]
        ];

g.dijkstra(0);
```

Output:

```
Vertex  Distance from source
0        0
1        1
2        7
3        12
4        17
5        8
6        10
7        6
8        10
```

Result:

TheImplementation of single source shortest path algorithm is executed successfully.

Ex.No:9	Implementationof0/1KnapsackusingBranchand Bound
Date:11-03-2022	

Video link : https://youtu.be/dfcZWwttI6w

Aim:

Toimplement0/1Knapsack usingBranch andBound

Algorithm:

STEP 1: The number of objects, maximum capacity, profits, weights and the profit weight ratio isgiven.

STEP2: Aclasspriorityqueueis createdalongwith therequiredfunctions.

STEP3: Aclassnamednodeiscreated withlevel, profit,weightand anarray nameditems.

STEP 4: The cost is checked with the upper bound and if there is any requirement the upperboundvalue is updated and all the nodes arechecked.

Program:

```
n = 4
W = 16
p = [40, 30, 50, 10]
w = [2, 5, 10, 5]
p_per_weight = [20, 6, 5, 2]


class Priority_Queue:
    def __init_(self):
        self.pqueue = []
        self.length = 0

    def insert(self, node):
        for i in self.pqueue:
            get_bound(i)
        i = 0
        while i < len(self.pqueue):
            if self.pqueue[i].bound > node.bound:
                break
            i+=1
        self.pqueue.insert(i,node)
        self.length += 1
```

```
    def print_pqueue(self):
        for i in list(range(len(self.pqueue))):
            print ("pqueue",i, "=", self.pqueue[i].bound)
    def remove(self):
        try:
            result = self.pqueue.pop()
            self.length -= 1
        except:
            print("Priority queue is empty, cannot pop from empty list.")
        else:
            return result

class Node:
    def __init__(self, level, profit, weight):
        self.level = level
        self.profit = profit
        self.weight = weight
        self.items = []


def get_bound(node):
    if node.weight >= W:
        return 0
    else:
        result = node.profit
        j = node.level + 1
        totweight = node.weight
        while j <= n-1 and totweight + w[j] <= W:
            totweight = totweight + w[j]
            result = result + p[j]
            j+=1
        k = j
   if k<=n-1:
            result = result + (W - totweight) * p_per_weight[k]
        return result


nodes_generated = 0
pq = Priority_Queue()

v = Node(-1, 0, 0)
nodes_generated+=1
maxprofit = 0
v.bound = get_bound(v)


pq.insert(v)

while pq.length != 0:

    v = pq.remove()
    if v.bound > maxprofit:
        u = Node(0, 0, 0)
        nodes_generated+=1
```

```
        u.level = v.level + 1
        u.profit = v.profit + p[u.level]
        u.weight = v.weight + w[u.level]
        #take v's list and add u's list
        u.items = v.items.copy()
        u.items.append(u.level) # adds next item
        if u.weight <= W and u.profit > maxprofit:
            #update maxprofit
        u.profit = v.profit + p[u.level]
        u.weight = v.weight + w[u.level]
        #take v's list and add u's list
        u.items = v.items.copy()
        u.items.append(u.level) # adds next item
        if u.weight <= W and u.profit > maxprofit:
            #update maxprofit
            maxprofit = u.profit
            bestitems = u.items
        u.bound = get_bound(u)
        if u.bound > maxprofit:
            pq.insert(u)
        u2 = Node(u.level, v.profit, v.weight)
        nodes_generated+=1
        u2.bound = get_bound(u2)
        u2.items = v.items.copy()
        if u2.bound > maxprofit:
            pq.insert(u2)



print("\nEND maxprofit = ", maxprofit, "nodes generated = ", nodes_generated)
print("bestitems = ", bestitems)
```

Output:

```
END maxprofit =  90 nodes generated =  11
bestitems =  [0, 2]
```

Result:

Therefore,implementationof 0/1Knapsackusingbranch andbound hasbeendonesuccessfully.

Ex. No: 10	IMPLEMENT N – QUEENS PROBLEM USING BACKTRACKING
Date:18-03-2022	

Video link : https://youtu.be/_wx9XFFBqhY

Aim:

To implementN-queens problem using backtracking.

Algorithm:

STEP 1: Check the columns to find any attacking queens.

STEP 2: The range of row and column for the upper diagonal is taken and check the position of attacking queens.

STEP 3: The range of lower diagonal is taken and we check the position of attacking queens.

STEP 4: If all queens are placed it returns true

Program:

```
n=4
def solution(board):
  for i in range(n):
    for j in range(n):
      print(board[i][j],end=" ")
    print()

def check(board,row,column):
  for i in range(column):
    if board[row][i]==1:
      return False
  for i,j in zip(range(row,-1,-1),range(column,-1,-1)):
    if board[i][j]==1:
      return False
  for i,j in zip(range(row,n,1),range(column,-1,-1)):
    if board[i][j]==1:
      return False
  return True

def placing(board,column):
  if column>=n:
    return True
```

```
    for i in range(n):
      if check(board,i,column):
        board[i][column]=1
        if placing(board,column+1)==True:
            return True
        board[i][column]=0
    return False
  def NQUEENS():
    board=[[0, 0, 0, 0],
           [0, 0, 0, 0],
           [0 ,0, 0, 0],
           [0, 0, 0, 0]]
    if placing(board,0)==False:
      print("No solution")
      return False
    solution(board)
    return True
```

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
True
```

Result:

The Program is implemented by N-queens problem using backtracking method.