Algorithm-9

—— 0-1 Knapsack

A. ***Problem Description***

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

B. ***Description of algorithm***

```
/*
The set of items to take can be deduced from the table,
starting at c[n. w] and tracing backwards where the optimal
values came from. Ifc[i, w] = c[i−1, w] item i is not part
of the solution, and we are continue tracing with c[i−1, w].
Otherwise item i is part of the solution, and we continue
tracing with c[i−1, w−W].
*/

Dynamic-0-1-knapsack (v, w, n, W)
    FOR w = 0 TO W
        DO  c[0, w] = 0
    FOR i=1 to n
        DO c[i, 0] = 0
```

```
FOR w=1 TO W
    DO IFf wi ≤ w
        THEN IF  vi + c[i-1, w-wi]
            THEN 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]
```

C. ***Time Complexity T=θ(nw)***

This dynamic-0-1-kanpsack algorithm takes θ(nw) times, broken up as follows: θ(nw) times to fill the c-table, which has (n +1).(w+1) entries, each requiring θ(1) time to compute. O(n) time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

D. Code[Python]

```python
#!/usr/bin/python
# Filename: Knapsack.py

def min(a, b):
  if a < b:
    return a
  else:
    return b

def max(a, b):
```

```
  if a > b:
    return a
  else:
    return b

def Knapsack(v, w, c, n, m):
  jMax = max(w[n] + 1, c)
  for j in range(0, jMax + 1):
    m[n][j] = 0
  for j in range(w[n], c + 1):
    m[n][j] = v[n]
  for i in range(n - 1, 1, -1):
    jMax = min(w[i] - 1, c)
    for j in range(0, jMax + 1):
      m[i][j] = m[i + 1][j]
    for j in range(w[i], c + 1):
      m[i][j] = max(m[i + 1][j], m[i + 1][j - w[i]] + v[i])

  m[1][c] = m[2][c]
  if c >= w[1]:
    m[1][c] = max(m[1][c], m[2][c - w[1]] + v[1])

def Traceback(m, w, c, n, x):
  for i in range(1, n):
    if m[i][c] == m[i + 1][c]:
      x[i] = 0
    else:
      x[i] = 1
      c = c - w[i]
  x[n] = 1 if m[n][c] != 0 else 0
```