Dynamic Programming

Sydney Teo, Saniya Nangia, Namrah
Team 5

Problem Description

- Capacity = C
- No. of types of objects = n
- Each object has positive weight w_i and positive profit p_i ($i = 0, 1, \ldots, n-1$)
- Unlimited supplies of each type of object
    - There is no need to account for maximum number of each type of object that can be “included” in knapsack

Problem: Find largest total profit of any set of the objects that fits in the knapsack

Part 1

P(C) = maximum profit for knapsack of capacity C
Give a recursive definition of the function P(C)

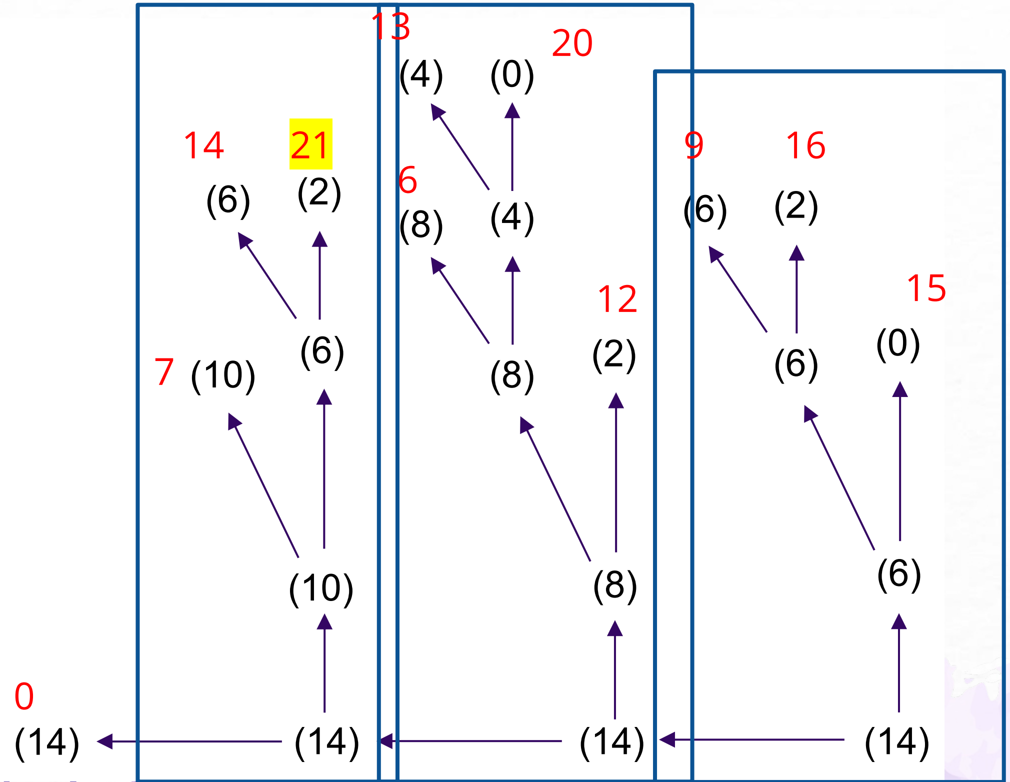
Recursive Function:
P(0) = 0 ⇒ number of objects will never be 0, so only one base case

$P(C) = \max(P(C), p_j + P(C - w_j))$
⇒ any object can be added to the knapsack; it does not have to the $(j-1)^{th}$ object

Time Complexity: O(Cn), where C is the capacity and n is the number of type of objects
Space Complexity: O(C)

Part 2

Draw the subproblem graph for P(14) where *n* is 3 with the weights and profits given below.



	0	1	2
w_i	4	6	8
p_i	7	6	9

Part 3

```
int knapsack(int[] w, int[] p, int C)
        profit = []

    for r = 0 to C:
        profit[0] = 0

    for r = 0 to C:
        for c = 1 to len(w):
            if w[c] <= r:
                profit[r] = max(P[r], p[c] + P(r - w[c]))
```

Part 4 - Code Implementation

```
def knapsack(wi, pi, C):
    dp = []

    for r in range(C+1):
        dp.append(0)

    for r in range(C+1):
        for c in range(len(wi)):
            if(wi[c] <= r):
                if(dp[r] < dp[r-wi[c]] + pi[c]):
                    dp[r] = dp[r-wi[c]] + pi[c]

    print("Running Result of P({}): {}".format(C, dp))
    printKnapsack(C, pi, wi, dp)
    return dp[C]
```

- Iterate over all elements available for each knapsack capacity and determine if it can be used to achieve a greater profit
- Use 1D array dp[C+1]
- dp[i] stores maximum profit value using all items and knapsack capacity i
- dp[i] = max(dp[i], dp[i-wi[j]] + pi[j])

Part 4 - Code Implementation

```
def printKnapsack(C, pi, wi, dp):
    if C == 0:
        return
    n = len(wi)
    ans = 0
    chosenItem = -1
    for j in range(n):
        if (C - wi[j] >= 0):
            newAns = dp[C - wi[j]] + pi[j]
            if newAns > ans:
                ans = newAns
                chosenItem = j

    if chosenItem == -1:
        return

    global knapsackContents
    knapsackContents.append(wi[chosenItem])
    printKnapsack(C - wi[chosenItem], pi, wi, dp)
```

- Backtrack to print selected weights in knapsack that lead to maximum profit
- Iterates over all available elements and sees which element leads to maximum profit for selected capacity
- Recursively does the same for capacities C-wi[chosenItem] to find all selected weights

Part 4 - Code Implementation

```
wi = [4, 6, 8]
pi = [7, 6, 9]
C = 14
knapsackContents = []
print("\nwi:", wi)
print("pi:", pi)
print("Max Profit:", unboundedKnapsack(wi, pi, C))
print("Knapsack Contents:", knapsackContents)

wi = [5, 6, 8]
pi = [7, 6, 9]
knapsackContents = []
print("\nwi:", wi)
print("pi:", pi)
print("Max Profit:", unboundedKnapsack(wi, pi, C))
print("Knapsack Contents:", knapsackContents)
```

- Driver code

Part 4 - Running Results

```
wi: [4, 6, 8]
pi: [7, 6, 9]
Running Result of P(14): [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 14, 21, 21, 21]
Max Profit: 21
Knapsack Contents: [4, 4, 4]

wi: [5, 6, 8]
pi: [7, 6, 9]
Running Result of P(14): [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 14, 14, 16, 16]
Max Profit: 16
Knapsack Contents: [5, 8]
```

Alternative Code Implementation

```
def unboundedKnapsack(wi, pi, C):
    print("Running Result of P("+str(C)+"): ")
    n = len(pi)
    dp = [[-1]*(C+1) for i in range(n)]

    for i in range(n):
        dp[i][0] = 0

    for i in range(n):
        for j in range(1, C + 1):
            include, exclude = 0, 0
            if wi[i] <= j:
                include = pi[i] + dp[i][j - wi[i]]
            if i > 0:
                exclude = dp[i - 1][j]
            dp[i][j] = max(include, exclude)
    for i in dp:
        print('  '.join(map(str, i)))
    printKnapsack(dp, wi, C)
    return dp[n - 1][C]
```

- Less space optimized version of first implementation
- Uses 2D array dp[n][C+1]
- dp[i][j] stores maximum profit value using items [0,...,(i-1)] and knapsack capacity j
- dp[i][j] = max(dp[i-1][j], pi[i] + dp[i][j - wi[i]])

Alternative Code Implementation

```python
def printKnapsack(dp, wi, C):
    global knapsackContents
    i = len(wi) - 1

    while i >= 0 and C >= 0:
        if (i > 0 and dp[i][C] != dp[i - 1][C]) or (i == 0 and C >= wi[i]):
            knapsackContents.append(wi[i])
            C -= wi[i]
        else:
            i -= 1
```

- Backtrack to print selected weights in knapsack that lead to maximum profit
- Iterates over all available elements and sees which element leads to maximum profit for selected capacity

Thank You