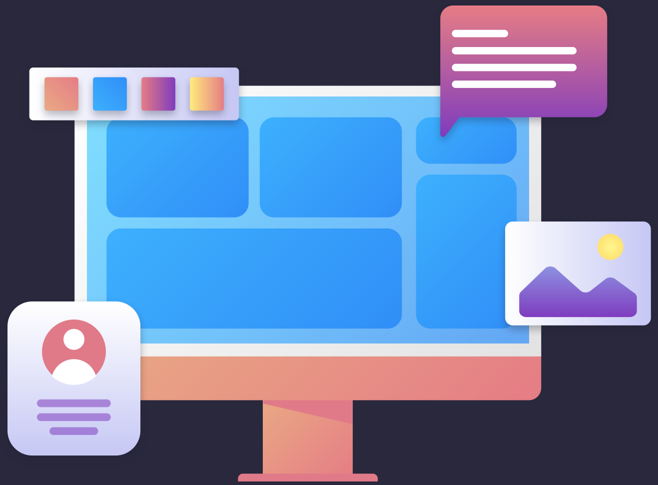
YOUR LOGO HERE

# Project 3: Unbounded Knapsack




SDDB Group 3

Chiang Qin Zhi

Donovan Goh Jun Heng

Gao Wen Jie


INDEX.HTML

YOUR LOGO HERE

# /TABLE OF CONTENTS


/01 Recursive Definition

/02 Subproblem Graph

/03 Bottom-up Dynamic Programming


INDEX.HTML

# /01

# Recursive Definition

Part (1)

# Q1. Unbounded Knapsack

We have a knapsack of capacity weight C (a positive integer) and n types of objects. Each object of the ith type has weight wi and profit pi (all wi and all pi are positive integers, i = 0, 1, …, n-1). There are unlimited supplies of each type of objects.

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

Let P(C) be the maximum profit that can be made by packing objects into the knapsack of capacity C.

Difference between 0/1 Knapsack:

0/1 Knapsack is a binary option to either exclude or include the object

# Q1. Recursive Definition

**Recurrence Definition**

Profit including the ith object

$$P(C) = \max_{0 \le i < n} \left(P(C), P(C - w_i) + p_i\right) \quad \text{for } C > 0, weight[i] < C$$

Profit excluding the ith object

**Base Case:**

$$P(C) = 0 \quad \text{when} \;\; C = 0 \text{ or } n = 0$$

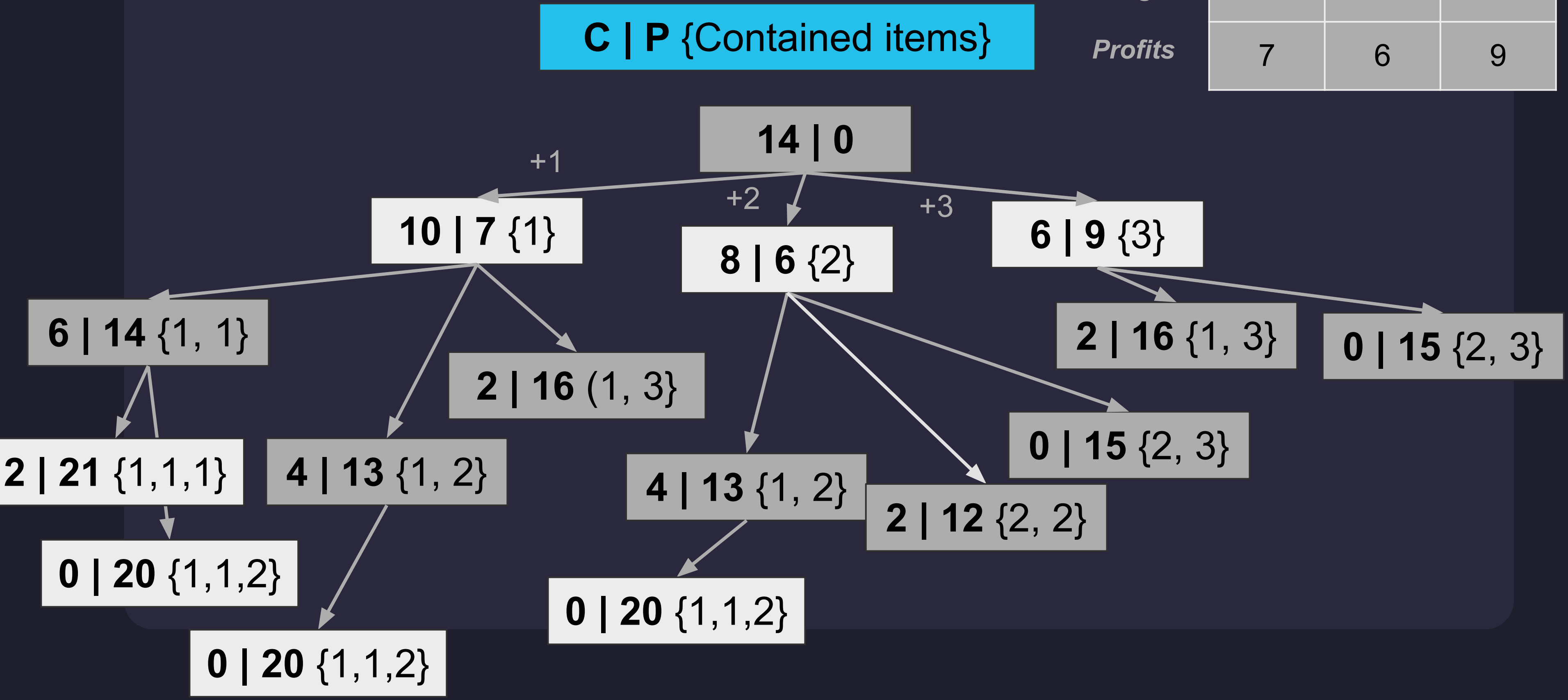
# /02

# Subproblem Graph

Part (2)

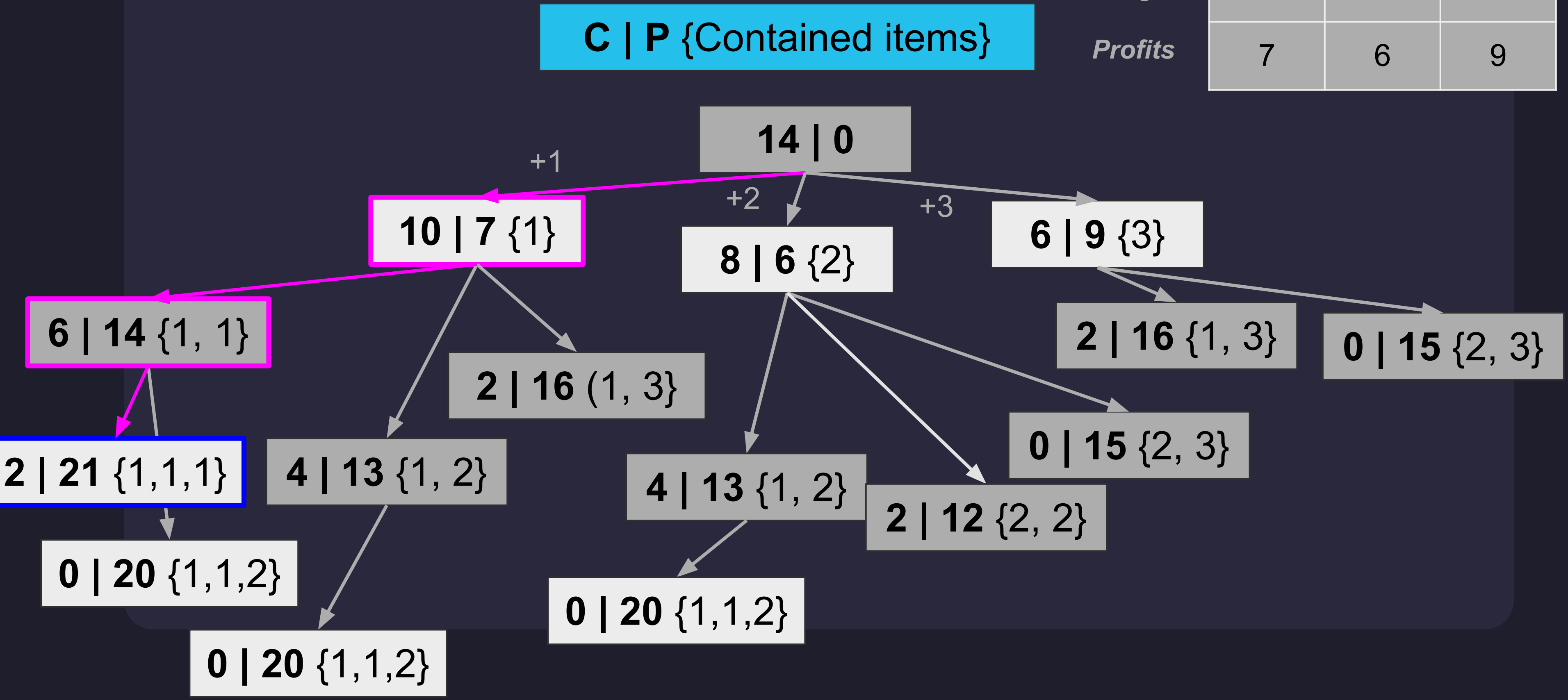
# Q2. Subproblem Graph

| | Item 1 | Item 2 | Item 3 |
|---|---|---|---|
| *Weights* | 4 | 6 | 8 |
| *Profits* | 7 | 6 | 9 |



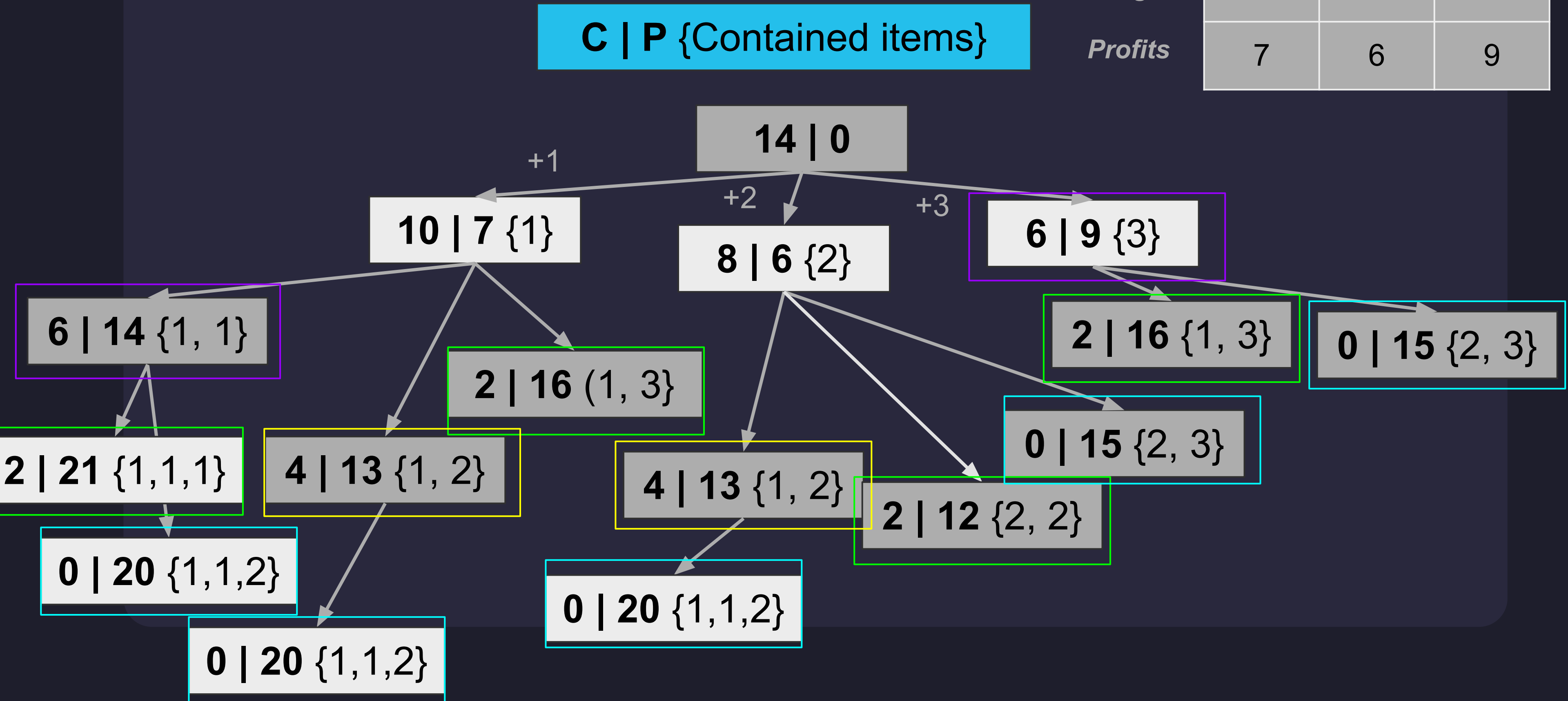
# Q2. Subproblem Graph

| | Item 1 | Item 2 | Item 3 |
|---|---|---|---|
| *Weights* | 4 | 6 | 8 |
| *Profits* | 7 | 6 | 9 |



# Q2. Subproblem Graph

| | Item 1 | Item 2 | Item 3 |
|---|---|---|---|
| *Weights* | 4 | 6 | 8 |
| *Profits* | 7 | 6 | 9 |



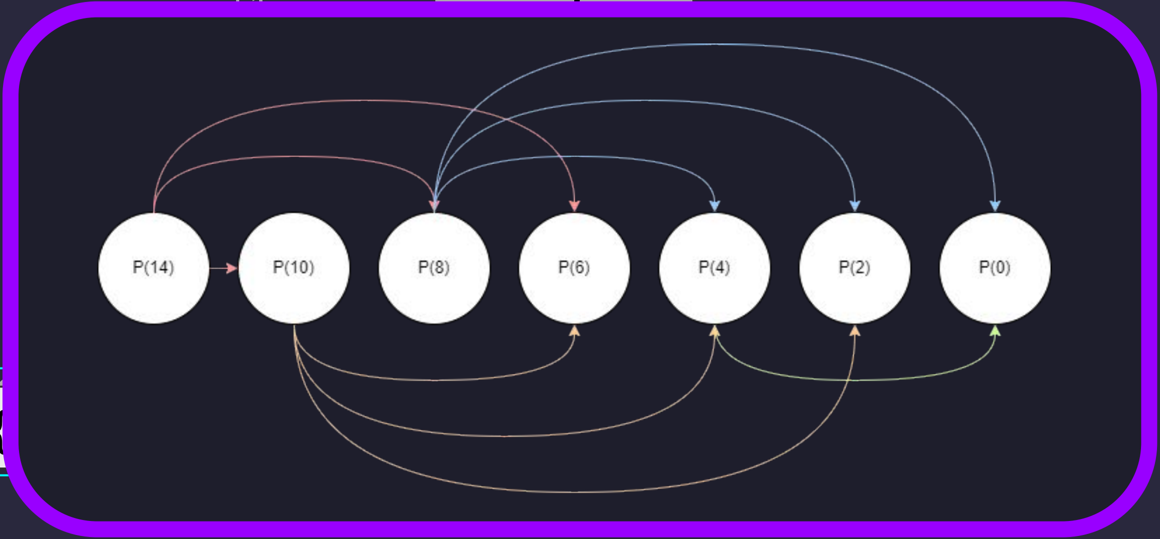
# Q2. Subproblem Graph

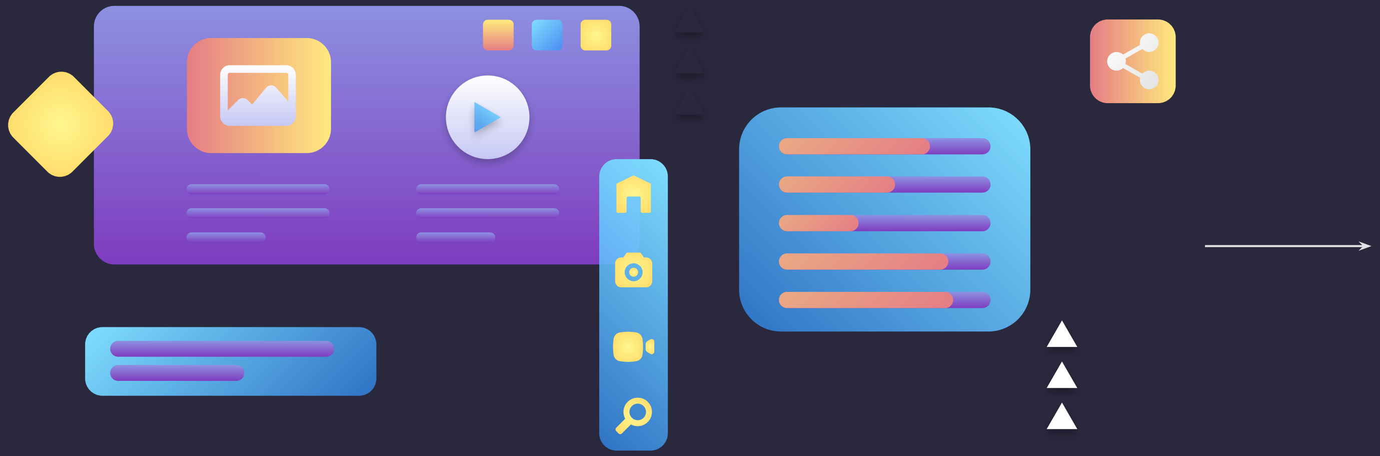
| | Item 1 | Item 2 | Item 3 |
|---|---|---|---|
| **Weights** | 4 | 6 | 8 |
| **Profits** | 7 | 6 | 9 |

**C | P** {Contained items}

**14 | 0**

**0 | 20 {**





# /03 Bottom-up DP

Part (3) & (4)

# Q3. Bottom-up DP Algorithm

**Bottom-up Approach:** Iteratively build up solution for increasing capacities, from 0 up to C. Each smaller subproblem is solved first, and its solution is used to solve larger subproblems.

| | **Capacity** | | |
|---|---|---|---|
| | *0* | . . . | *C* |
| ***Max Profits*** | | | |

# Q3. Bottom-up DP Algorithm

## Algorithm Steps

1. Create an array `knapsack` of size C + 1 to store the maximum profit for **capacities 0 to C**. Initialize all elements of `knapsack` to 0.

**Capacity**

| | *0* | *. . .* | *C* |
|---|---|---|---|
| ***Max Profits*** | | | |

# Q3. Bottom-up DP Algorithm

## Algorithm Steps

2a. For each capacity c from **1 to C**, compute the **maximum profit**.

2b. Maximum profit: for each item i from 0 to n-1, check if the item can fit into the knapsack of current capacity c (i.e., $w_i \le c$). If it can, calculate the profit of including this item, which is **$p_i$ + knapsack[c - $w_i$]** (the profit of the item + the max profit for the remaining capacity). Update `knapsack[c]` to the maximum of its current value and this new profit. Repeat for each capacity c.

4. After filling the `knapsack` array, the maximum profit for the knapsack of capacity C will be stored in `**knapsack[C]**`.

# Q3. Bottom-up DP Algorithm

**Pseudocode**

```
function unboundedKnapsack(C, weights, profits, n):
    knapsack = array of size C + 1, initialized to 0

    for c from 1 to C:
        for i from 0 to n-1:
            if weights[i] ⩽ c:
                knapsack[c] = max(knapsack[c], profits[i] + knapsack[c -
weights[i]])

    return knapsack[C]
```

# Q4. Python Implementation

```python
def unboundedKnapsack (capacity, table):
    if (capacity == 0):
        return 0

    n = len(table[0])
    weights = table[0]
    profits = table[1]
    knapsack = [0 for _ in range(capacity + 1)]

    # Iterate over all capacities from 1 to 'capacity'
    for i in range(1, capacity + 1):
        # Check each item to see if it can fit in the current capacity 'i'
        for j in range(n):
            if weights[j] <= i:
                knapsack[i] = max(knapsack[i], knapsack[i - weights[j]] + profits[j])

    return knapsack[capacity]
```

# Q4a. Results

| Item | 0 | 1 | 2 |
|---|---|---|---|
| $w_i$ | 4 | 6 | 8 |
| $p_i$ | 7 | 6 | 9 |

**Capacity** = 14, **P(14)** = <u>21</u>

# Q4a. Results

```
# Part A
tableA = [[4, 6, 8], [7, 6, 9]] # [[weights], [profits]]
print("(Part A) The maximum profit is: ", unboundedKnapsack(14, tableA))
```

✓ 0.0s

```
Capacity: 1, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 2, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 3, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 4, Knapsack Array: [0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 5, Knapsack Array: [0, 0, 0, 0, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 6, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 7, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0]
Capacity: 8, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 0, 0, 0, 0, 0, 0]
Capacity: 9, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 0, 0, 0, 0, 0]
Capacity: 10, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 0, 0, 0, 0]
Capacity: 11, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 14, 0, 0, 0]
Capacity: 12, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 14, 21, 0, 0]
Capacity: 13, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 14, 21, 21, 0]
Capacity: 14, Knapsack Array: [0, 0, 0, 0, 7, 7, 7, 7, 14, 14, 14, 14, 21, 21, 21]
(Part A) The maximum profit is:  21
```

# Q4b. Results

| Item | 0 | 1 | 2 |
|---|---|---|---|
| $w_i$ | 4 | 6 | 8 |
| $p_i$ | 7 | 6 | 9 |

**Capacity** = 14, **P(14)** = <u>16</u>

# Q4b. Results

```
# Part B
tableB = [[5, 6, 8], [7, 6, 9]] # [[weights], [profits]]
print("(Part B) The maximum profit is: ", unboundedKnapsack(14, tableB))
```

✓ 0.0s

```
Capacity: 1, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 2, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 3, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 4, Knapsack Array: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 5, Knapsack Array: [0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 6, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0]
Capacity: 7, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0]
Capacity: 8, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 0, 0, 0, 0, 0, 0]
Capacity: 9, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 0, 0, 0, 0, 0]
Capacity: 10, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 0, 0, 0, 0]
Capacity: 11, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 14, 0, 0, 0]
Capacity: 12, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 14, 14, 0, 0]
Capacity: 13, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 14, 14, 16, 0]
Capacity: 14, Knapsack Array: [0, 0, 0, 0, 0, 7, 7, 7, 9, 9, 14, 14, 14, 16, 16]
(Part B) The maximum profit is:  16
```

# Q4. Algorithm Complexity

**Time Complexity:** O(n * Capacity)
**Space Complexity:** O(Capacity)

# <Thank You>

# Q3. Bottom-up DP Algorithm

**Bottom-up Approach:** Iteratively build up solution for increasing capacities, from 0 up to C. Each smaller subproblem is solved first, and its solution is used to solve larger subproblems.

**Capacity**

| | | 0 | . . . | C |
|---|---|---|---|---|
| | 0 | | | |
| **Items** | ⋮ | | | |
| | n | | | |