Dynamic Programming

By Jared, Jordan, and Ethan

Knapsack Problem

Since there are **unlimited supplies** of each item, we are looking at the Unbounded Knapsack Problem, rather than 0/1 Knapsack Problem.

- Each item can be chosen to be added to the knapsack an unlimited amount of times
- Goal: maximise profits!

Recursive Definition

Base Case:

$$\mathbf{P(0) = 0}$$

When Knapsack has capacity of 0, the largest total profit is 0

Recursive Case:

$$\mathbf{P(C) = \max_{i=0}^{n-1}[p_i + P(C - w_i)], \ for\ C \geq w_i}$$

$\boldsymbol{p_i}$: profit of the *i*th item
$\boldsymbol{w_i}$: weight of the *i*th item

Recursive Definition

$$\mathbf{P(C) = \max_{i=0}^{n-1}[p_i + P(C - w_i)],\ \ for\ C \geq w_i}$$

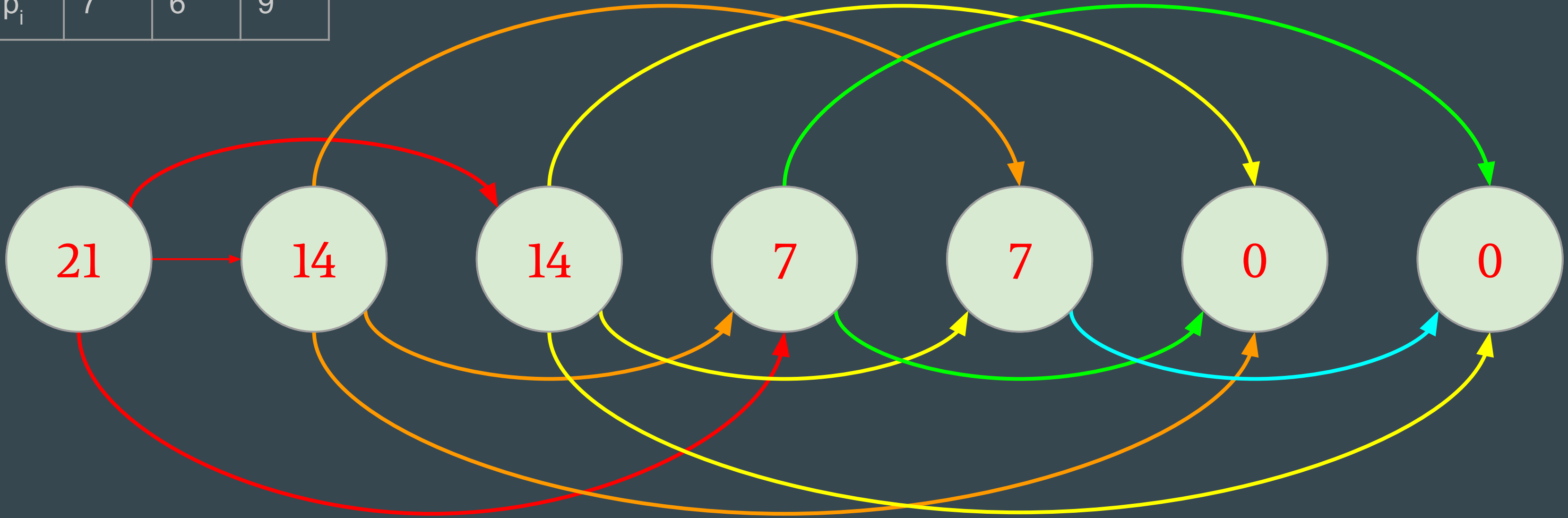
1. Iterate through each of the n items: For each item *i*, determine if it can fit within the current knapsack capacity *C*.

2. Check Capacity and Calculate Profit:
   a. If adding the item's weight, w_i, does not exceed *C*, calculate the profit of including this item, which is the profit p_i plus the maximum profit for the remaining capacity $(C - w_i)$.

3. Update Maximum Profit:
   a. For each item, choose whether to include or exclude it to maximize the profit.
   b. Repeat this check for each item type, using the recursive formula to ensure that each combination of items optimally fills the knapsack.

Subproblem Graph

i	0	1	2
w_i	4	6	8
p_i	7	6	9



(3) Give a dynamic programming algorithm to compute the maximum profit using the bottom up approach.

Profit of each object

Number of objects

int knapsack(int [ ]w, int [ ]p, int C, int n)

Weight of each object

Capacity of knapsack

```
profit[C+1][n+1]
for c = 0 to n
      profit[0][c] = 0;
for r = 1 to C
      profit[r][0] = 0;
```

profit

0	0	0	0	0
0				
0				
0				
0				

Capacity of knapsack

Index of object

```
for row = 1 to C
    for col = 1 to n
```

Populate profit from the bottom-up

```
        profit[row][col] = profit[row][col-1];
        if (w[col] <= row)
            profit[row][col] = MAX( profit[row-w[col-1]][col] + p[col],
                                    profit[row][col] )
```

If you can fit another object, and if it will increase the profit, include the object

(4) Code your algorithm in a programming language

4) Implementation in Python

Why Python?

- Reads like pseudo code, easier to implement

```python
def knapsack(w, p, C, n):
    # Initialize a 2D array with (C+1) rows and (n+1) columns, filled with zeros
    profit = [[0 for _ in range(n + 1)] for _ in range(C + 1)]

    for r in range(1, C + 1):
        for c in range(1, n + 1):
            # Start with the maximum profit excluding this item
            profit[r][c] = profit[r][c - 1]
            if w[c - 1] <= r:
                # Allow multiple instances of the same item by adding it repeatedly
                profit[r][c] = max(profit[r][c], profit[r - w[c - 1]][c] + p[c - 1])
```

4a) P(C=14)

Choose 3
of object 0

	0	1	2
w_i	4	6	8
p_i	7	6	9

4x3 = 12
7x3 = 21

4a) P(C=14)

	0	1	2
w_i	4	6	8
p_i	7	6	9

3+1 rows

14 + 1 columns

```
Table:
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 7, 7, 7]
[0, 7, 7, 7]
[0, 7, 7, 7]
[0, 7, 7, 7]
[0, 14, 14, 14]
[0, 14, 14, 14]
[0, 14, 14, 14]
[0, 14, 14, 14]
[0, 21, 21, 21]
[0, 21, 21, 21]
[0, 21, 21, 21]
Indices of items in the knapsack: [0, 0, 0]
Maximum profit: 21
```

4b) P(C=14)

	0	1	2
w_i	5	6	8
p_i	7	6	9

$5 + 8 = 13$

$7 + 9 = 16$

4b) P(C=14)

	0	1	2
w_i	5	6	8
p_i	7	6	9

3+1 rows

14 + 1 columns

```
Table:
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 7, 7, 7]
[0, 7, 7, 7]
[0, 7, 7, 7]
[0, 7, 7, 9]
[0, 7, 7, 9]
[0, 14, 14, 14]
[0, 14, 14, 14]
[0, 14, 14, 14]
[0, 14, 14, 16]
[0, 14, 14, 16]
Indices of items in the knapsack: [0, 2]
Maximum profit: 16
```

~ The End ~