Double Knapsack

Reasoning

The solution I found was to use a method very similar to the dynamic programming method we used previously, except instead of having a 2d array with item index, and weight, we could have a 3d array with item index, weight capacity of one bag, and weight capacity of the other. The dynamic solution would be as follows:

What: $S[j, v, w]$ the maximum cost of items up to item j where the weight in the first backpack is up to weight v and the weight in the second is up to weight w.

How: $S[j, v, w] =$

$$\begin{cases} 0 \text{ if } j = 0 \text{ or } (v = 0 \text{ and } w = 0) \\ \max\{S[j-1, v, w], c_j + S[j-1, v - w_j, w], c_j + S[j-1, v, w - w_j]\} \text{ if } j > 0, v \geq w_j \end{cases}$$

where w_j is the weight of the item and c_j is its cost.

Where: $S[n, W_1, W_2]$

Psudocode

```
def maxWeight(costs: array<int>, weights: array<int>, W_1, W_2, n):
    # Set the spots where j is zero to zero
    for v = 0 to W_1:
        for w=0 to W_2:
            S[0,v,w] = 0
    # Set the places where the total weight capacity is zero to zero
    for j = 0 to n:
        S[j,0,0] = 0;

    for j=1 to n:
        for v=0 to W_1:
            for w=0 to W_2:
                S[j,v,w]=S[j-1,v,w]
                if (weights[j] <= v and S[j-1, v-weights[j], w] + costs[j]>
S[j, v, w]):
                    S[j,v,w] =  S[j-1, v-weights[j], w] + costs[j];
                if (weights[j] <= w and S[j-1, v, w-weights[j]] + costs[j]
> S[j,v,w]):
                    S[j,v,w] =  S[j-1, v, w-weights[j]] + costs[j];
    # Return the reconstructed version of the solution with two output
arrays for each bag
    return reconstruction(costs, weights, W_1, W_2, n, S);
```

Proof

The base case for this algorithm is true, since without any items in the bag, the bag has no cost, so those places are all zero. Then we can set all the places where the total weight is zero to zero. After this we can move on to the steps which build off that. At any given $S[j, v, w]$ position, we know that it is the maximum possible cost of items up to item j, with weights fitting in knapsacks of size v and w, since it is either the preious maximum without the new item, or it is a previous maximum with the new item, whichever is larger, which will always be the largest possible cost, since the prior ones were the largest possible cost.

Running Time

Estimate

The running time is $\Theta(n \cdot W_1 \cdot W_2)$

Running Time Reasoning

```
def maxWeight(costs: array<int>, weights: array<int>, W_1, W_2, n):
    for v = 0 to W_1: # Runs W_1 times -> O(W_1 * W_2)
        for w=0 to W_2: # Runs W_2 times
            S[0,v,w]
    for j = 0 to n: # Runs n times
        S[j,0,0] = 0;

    for j=1 to n: # Runs n times -> O(n * W_1 * W_2)
        for v=0 to W_1: # Runs W_1 times -> O(W_1 * W_2)
            for w=0 to W_2: # Runs W_2 times -> O(W_2)
                S[j,v,w]=S[j-1,v,w]
                if (weights[j] <= v and S[j-1, v-weights[j], w] + costs[j]>
S[j, v, w]):
                    S[j,v,w] =  S[j-1, v-weights[j], w] + costs[j];
                if (weights[j] <= w and S[j-1, v, w-weights[j]] + costs[j]
> S[j,v,w]):
                    S[j,v,w] =  S[j-1, v, w-weights[j]] + costs[j];
    # Runs in O(n) time as proved below
    return reconstruction(costs, weights, W_1, W_2, n, S);
```

The total running time is $O(W_1 \cdot W_2) + O(n) + O(n \cdot W_1 \cdot W_2) + O(n) = O(n \cdot W_1 \cdot W_2)$

Reconstruction

For the reconstruction, we start from the solution element, which is $S[n, W_1, W_2]$, and trace the path that was taken backwards until we reach the start ($S[0, 0, 0]$). We do this by comparing the elements at j-1 w and v and if it is the same as the current element then we didn't take item j and we decrement j, otherwise, we can assume we took the current j element, and we can travel to $S[j-1, v-weight[j], w]$ or $S[j-1, v, w-weight[j]]$ (whichever one is equal to $S[j, v, w] - costs[j]$). This should give us one possible set of items that could be added to the backpacks to give us the maximum cost.

Reconstruction Psudocode

```
def reconstruction(costs: array<int>, weights: array<int>, W_1, W_2, n,
result: array<array<array<int>>>):
    j=n, v=W_1, w=W_2
    while (j>0,v>0,w>0):
        if S[j-1,v,w] == S[j,v,w]:
            j--;
        else:
            if S[j-1,v-weight[j],w] + costs[j] == S[j,v,w]:
                output.knapsack1.append(j)
                v-weight[j]
            else:
                output.knapsack2.append(j)
                w-weight[j]
    return output
```

Reconstruction Running Time

Reconstruction Estimate

The running time is $O(p \cdot q \cdot r)$

Reconstruction Reasoning

```
def reconstruction(costs: array<int>, weights: array<int>, W_1, W_2, n,
result: array<array<array<int>>>):
    j=n, v=W_1, w=W_2 # O(1)
    while (j>0,v>0,w>0): # Runs n times
        if S[j-1,v,w] == S[j,v,w]:
            j--;
        else:
            if S[j-1,v-weight[j],w] + costs[j] == S[j,v,w]:
                output.knapsack1.append(j)
                v-weight[j]
                j --;
            else:
                output.knapsack2.append(j)
                w-weight[j]
                j --;
    return output
```

This algorithm runs n times, since it starts at j=n and j is decremented each time the loop runs. This means it runs in $O(n)$ time.