Goals:
- Finish Knapsack DP Algorithm
- Motivate NP definition

Announcements/Questions:
- Monday @7 on Zoom: Tapia Panel
- Last week with these groups :(
- Other base cases?
- "0" cols and rows, pros/cons?
- Why work backwards? Decrement C?

Recurrence Relation for $S_{i,r}$

$$S_{i,r} = \begin{cases} S_{i-1,r} & \text{if } i \notin S_{i,r} \\ S_{i-1,r-c_i} \cup \{i\} & \text{if } i \in S_{i,r} \end{cases}$$

$\forall r \; S_{0,r} = \emptyset$ Base Case

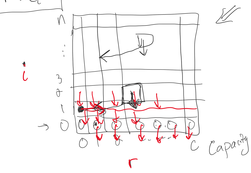
Recurrence Relation for Objective Function

$$V(S_{i,r}) = \max \{ V(S_{i-1,r}), V(S_{i-1,r-c_i}) + v_i \}$$

$\forall r, V(S_{0,r}) = 0$

Only an option if $r \geq c_i$

$A \Rightarrow A[2,3] = V(S_{2,3})$



Pseudocode

Input: $n+1$ arrays V (values) and c (costs)
Output: Optimal set to put in knapsack
Initialize A as $(n+1) \times (C+1)$ array
//Base Case

```
For r = 0 to C
    A[0,r] = 0          ← Base Case
For i = 1 to n:
    For r = 0 to C:
        if r ≥ c_i:
            A[i,r] = max{A[i-1,r], A[i-1, r-c_i] + v_i}
        else:
            A[i,r] = A[i-1,r]
```

Q: What is the runtime of DP knapsack algorithm?

A) $O(n)$ B) $O(C)$ C) $O(n \cdot C)$ D) $O(nC)$ ✓

```
S ← ∅
i ← n        in ex n = 4
r ← C        in ex C = 5
while (i > 0, r > 0)        might need if statement to avoid index-out-of-bounds errors
    if (A[i,r] = A[i-1,r]):
        i = i-1
    else
        S = S ∪ {i}
        r = r - c_i
        i = i-1
return S
```

O(n)

$A[i,r] = A[i-1, r-c_i] + v_i$

Ex: C = 5

Item	Value	Cost
1	6	2
2	5	1
3	8	3
4	7	4

c_i

A $A[i,r]$

items i \ r	0	1	2	3	4	5
4	0	5	6	11	13	14
3	0	5	6	11	13	14
2	0	5	6	11	11	11
1	0	0	6	6	6	6
0	0	0	0	0	0	0

r capacity

i	r
~~4~~	~~5~~
~~3~~	~~5~~
~~2~~	~~2~~
~~1~~	~~2~~
0	0

S = 3, 1

Q: What is the runtime of DP knapsack algorithm?

A) $O(n)$ B) $O(C)$ C) $O(n \cdot c)$ D) $O(nC)$

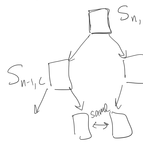
Knapsack: Brute Force $O(n \cdot 2^n)$

$n \cdot C$

Do we really need to fill out whole array?

Another approach to dynamic programming: memoization

Recursive Alg. + Hash Table



Before making a recursive call, check if already solved in hash table. Anytime you solve a new recursive subproblem you store in hash table.

Pros: Faster b/c only solve the subproblems you need.

Cons: More complex code to maintain hash table