8: Knapsack

Friday, April 2, 2021 11:51 AM

Knapsack Problem

Input:
- n items
  - value v_i
  - cost c_i (volume taken up)

  $\Big\} \in \mathbb{Z}^+$
- Capacity C

Output: A subset $S \subseteq \{1, 2, \dots, n\}$ that maximizes value

$$V(S) = \sum_{i \in S} v_i \quad \Leftarrow \text{objective function}$$

and satisfies

$$C(S) = \sum_{i \in S} c_i \leq C \quad \Leftarrow \text{constraints}$$

Ex: $C = 5$

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

} one of each item

← adding a second copy of item 4

What is optimal S:

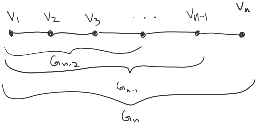
A): {1, 3} B) {1, 4} C) {1, 2, 3}, D) {2, 3}

↑

Designing a Dynamic Programming Alg.

0. Create series of increasingly smaller subproblems

   Recurrence object: Optimal output of each subproblem

- 
v_1
v_2
v_3
. . .
v_{n-1}
v_n
G_{n-2}
G_{n-1}
G_n

  S_i = MWIS of G_i

- n cents
  n-1 cents
  ⋮

  M_i = optimal coins to make i cents

1. Think about cases for "final elements" of recurrence object

- $n \in S_n$ or $n \notin S_n$
- last coin was 1, 4, or 6.

2. For each case, create recurrence in terms of smaller recurrence objects

- $S_i = \begin{cases} S_{i-1} \\ S_{i-2} \cup \{v_n\} \end{cases}$ & B.C.
- $M_i = \begin{cases} M_{i-1} + 1 \\ M_{i-4} + 1 \\ M_{i-6} + 1 \end{cases}$ fix & B.C.

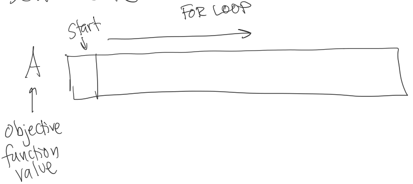
3. Convert into recurrence for objective function

- $W(S_n) = \max\{W(S_{i-1}), W(S_{i-2}) + w_n\}$ & B.C
- $C(n) = \min\{C(n-1), C(n-4), C(n-6)\} + 1$ & B.C

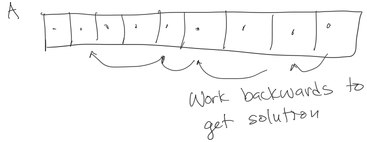
4. Pseudocode:

Store $w(S_n)$
$C(n)$

i.


FOR LOOP
start
A
objective function value

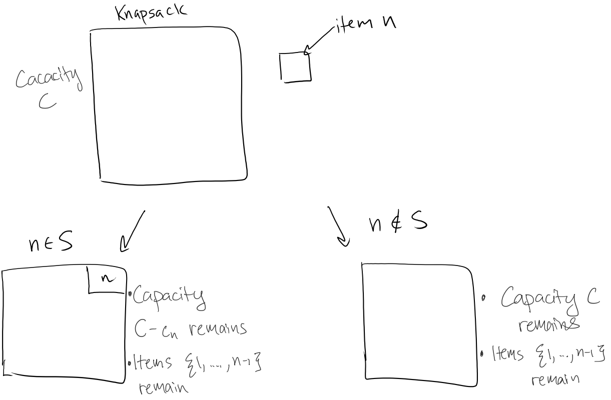
ii.


A
Work backwards to get solution

In this case, better to start with step 1, b/c not clear what the relevant subproblems are.

Cases for final choice? Ideas?

$n \in S_n$ vs $n \notin S_n$


Knapsack
Cacacity C
item n
$n \in S$
n
Capacity $C - c_n$ remains
Items $\{1, \dots, n-1\}$ remain
$n \notin S$
Capacity C remains
Items $\{1, \dots, n-1\}$ remain

Optimal solution should fill remaing space optimally!

Relevant subproblem: Knapsack(i, r)

$\uparrow$ items 1 to i allowed

capacity r (room)

Recurrence Object: $S_{i,r}$ = optimal set of items if can only include items $\{1, \dots, i\}$, capacity r

What is $S_{2,4}$

A) $\{1, 1\}$

B) $\{1, 2\}$

C) $\{2, 3\}$

D) $\{4\}$

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

Recurrence Relation for $S_{i,r}$

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

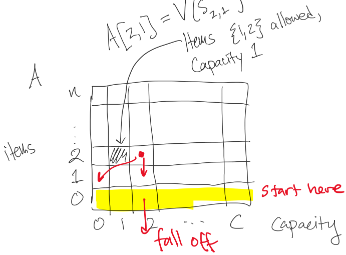
$S_{0,r} = \emptyset \quad \forall r$ (0 items allowed)

Come back to base case

Recurrence Relation for Objective Function

$$V(S_{i,r}) = \max\{\underbrace{V(S_{i-1,r})}_{k^{th}\text{ item not in set}}, \underbrace{V(S_{i-1,r-c_i})}_{k^{th}\text{ item in set}} + v_i\}$$

$$V_{0,r} = 0$$


A[2,1] = V(S_{2,1})
Items {1,2} allowed,
Capacity 1
A
items
n
2
1
0
0
1
2
C
Capacity
Start here
fall off

Pseudocode

Input: n×1 arrays v (values) and c (costs)

Output: Optimal set to put in knapsack

```
• Initialize A as (n+1) × (C+1) array
• For r = 0 to C:
  | A[0,r] ← 0
• For i = 1 to n            // items
  | For r = 0 to C:         //capacity
  |   | If r - c[i] > 0:    //if room for ith item
  |   |     A[i,r] = max{ A[i-1,r], A[i-1, r-c[i]] + v[i] }
  |   | Else:
  |   |     A[i,r] = A[i-1, r]
```

Always look to lower row!

(fill in base case)

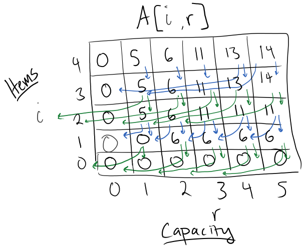
```
S = ∅
i = n
r = C
while i > 0 and r > 0:      // while items to put in & capacity available
|   if A[i,r] = A[i-1, r]:
```

```
    i--
else:
    S = S + i
    r = r - c[i]
    i--
```

Test

Ex: $C = 5$

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



A[i,r]

Items i \ Capacity 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

Replace code with work backwards code

```
• For i = 1 to n                // items
      For r = 0 to C:           //capacity
          If r - c[i] > 0:      //if room for ith item
              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) $O(n+C)$ D) $O(nC)$

Creating A: $O(nC)$ ← longer!

Creating S from A: $O(n)$

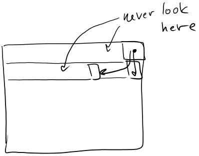
Total: $O(nC)$

Compare to brute force: $O(n2^n)$

- test all possible sets
- test value, validity of each set

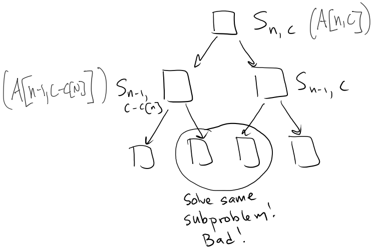
Do we really need to fill out whole array?

⇓


never look here

Another approach to dynamic programming: memoization

Recursive Algorithm


$S_{n,c}$ $(A[n,C])$
$(A[n-1,C-c[n]])$ $S_{n-1,C-c[n]}$
$S_{n-1,c}$
Solve same subproblem! Bad!

Before making recursive calls, check in hash table if already solved. Store new solutions in hash table.

Pros: Uses less time if many elements of A are not used in backtracking step.

Cons: More complex to maintain hash table