chang@umbc.edu

Dynamic Programming

not a CS name

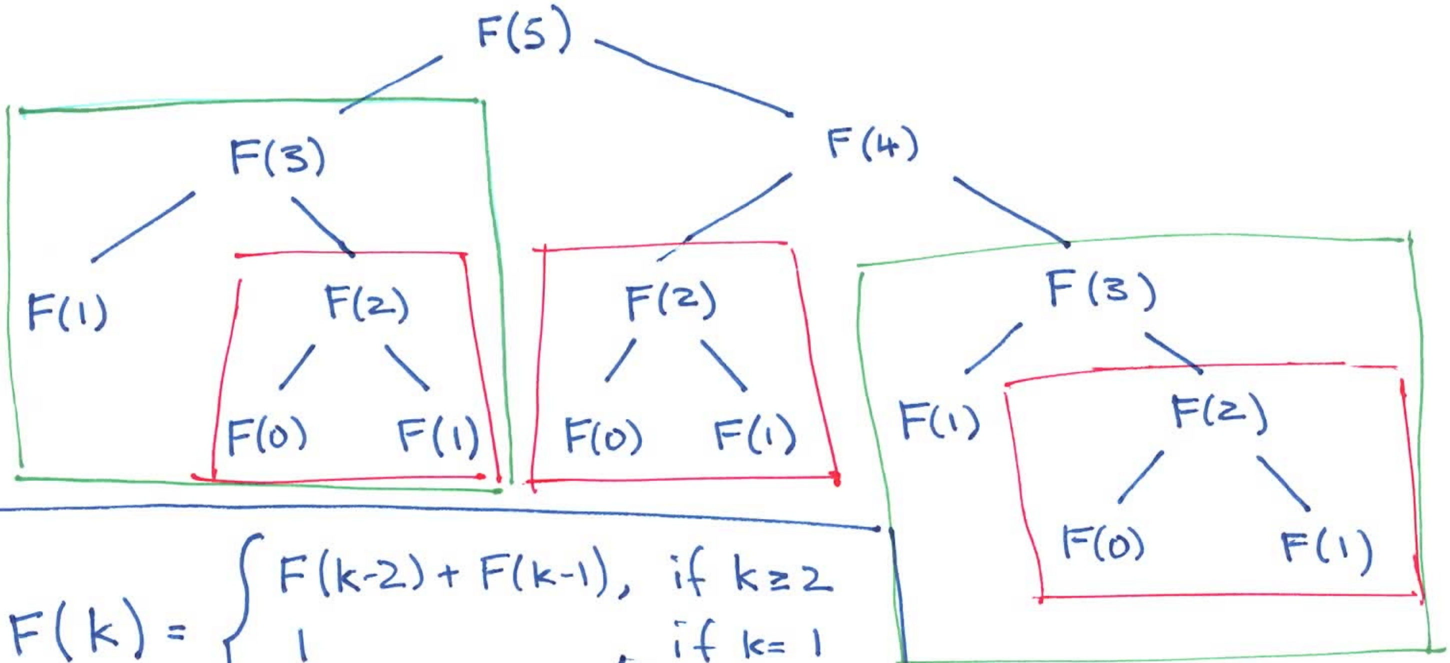
not as opposed to static.

nothing to do with coding

- a strategy for developing optimization algorithms
- interested in **best** answer
- THINK recursively, top down
- eliminate common subproblems
- final algorithms are iterative, bottom up

Fibonacci Numbers

World's worst possible example for recursive programs



$$F(k) = \begin{cases} F(k-2) + F(k-1), & \text{if } k \geq 2 \\ 1, & \text{if } k = 1 \\ 0, & \text{if } k = 0 \end{cases}$$

Memoization Table

Memo + iz~~e~~ + ation

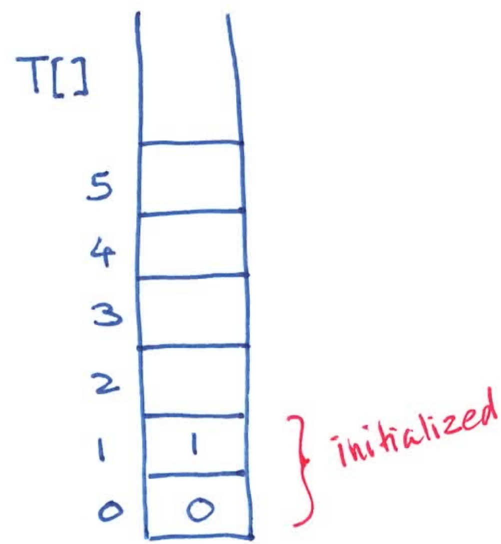
- avoids recomputing
- speeds up exponential time recursive algorithm
- linear-time Fibonacci function

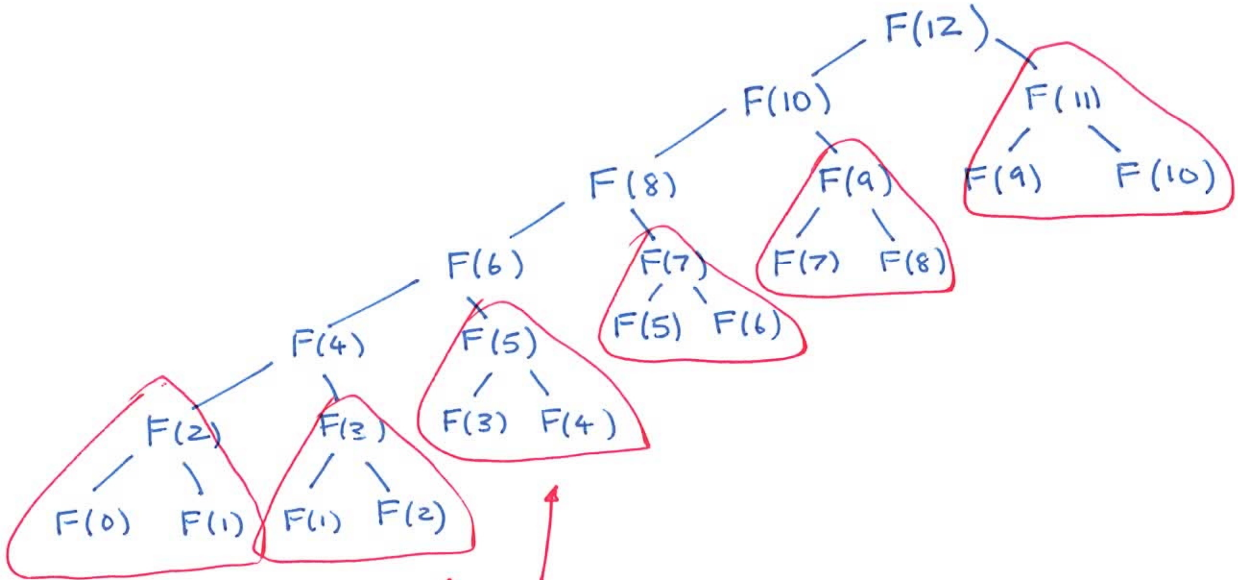
```
F(k) {
  if undef(T[k])
     T[k] = F(k-2) + F(k-1);

  return T[k];
}
```

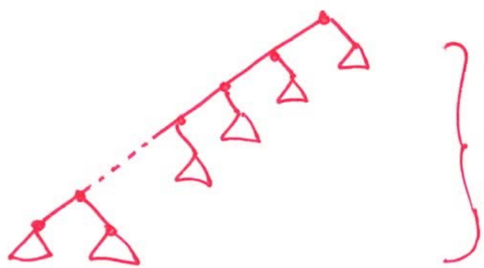
good use of global variable

Why is this linear time?





These subtrees take constant time.



$\Theta(n) \times$ constant time

Iterative bottom-up version:

just fill-in the memoization table

Hey, I thought you said no pseudo code

```
F(k) {
   T[0] = 0;
   T[1] = 1;
   for i=2 to k {
       T[i] = T[i-2] + T[i-1];
   }
   return T[k];
}
```

much easier to see that it's linear time.

faster too

Wait, we don't even need a table!

```
F(k) {
  Ti_minus_2 = 0 ;
  Ti_minus_1 = 1 ;

  for i=2 to k {
    Ti = Ti_minus_2 + Ti_minus_1 ;
    Ti_minus_2 = Ti_minus_1;
    Ti_minus_1 = Ti ;
  }

  return Ti ;
}
```

Knapsack Problem:

n items
item i has value v_i and weight w_i

Knapsack with capacity W

v_i, w_i & W integers.

Problem: find a subset S of the n items
such that

$$\sum_{i \in S} w_i \leq W$$

← don't exceed backpack's capacity

and $\sum_{i \in S} v_i$ is <u>maximized</u>

← yes, we want the BIGGEST.

Some common attempts:

- Try all subsets ← too slow, there are 2^n subsets

- Take most valuable item ← could also be heaviest

- Take item with highest value:weight ratio

W=8

	w	v
1	2	2
2	2	2
3	2	2
4	2	2
5	4	3
6	5	1
7	5	1

most valuable →

W=8

	w	v	v/w
1	2	4	2.0
2	2	4	2.0
3	2	4	2.0
4	5	11	2.2
5	2	4	2.0
6	7	1	1.42
7	7	1	1.42

← highest v/w

Recipe for dynamic programming:

1. What are the choices?
2. What subproblems result from a choice?
3. Rigorously define the subproblem.
4. Does the solution to the subproblem identify the optimum choice?

   no? then reformulate

5. How many subproblems are there?

   too many

6. How much time does it take to solve a single subproblem, if ITS subproblems have been solved already?

   too slow

7. Total running time = 5 × 6
8. Reconstruct the solution from sequence of choices.

Some advice:

1. Think recursively.
2. Don't unwind the recursion.
3. Subproblem should return a numeric value used to find optimum choice.
4. Don't be too clever, try all possible choices.
5. Use lots of global variables, use parameters only for values that change.
6. Don't pass intermediate solutions into a recursive call.
7. Optimum choice cannot depend only on the attributes of current item.
8. Look for overlapping subproblems.

Knapsack

① What are the choices?

Should I take item #1? or leave it behind?

② What subproblems result from a choice?

Take item #1: Capacity reduced by w_1
Fewer items to consider

Don't take item #1: Same capacity as before.
Fewer items to consider

3. Most important step: define the subproblem.

Get this right, you are done!

Solution of the subproblems, must tell me whether to take item #1 or leave it behind.

WRONG subproblem: Should I take item #2?

returns a numeric value

OPT_KS (i, C) =

store values v_i's in a global array

the highest possible total value obtained by choosing items from item #i thru item #n such that the weight of the chosen items does not exceed C.

store weights w_i's in a global array

Note: we don't know how to compute OPT_KS yet, but the important thing is that we can use OPT_KS to identify the optimum choices.

Recursively define OPT_KS(,)

$$\text{OPT\_KS}(i, C) = \max\Big(\overbrace{\text{OPT\_KS}(i+1, C - w_i) + v_i}^{\text{take item \#}i},\ \overbrace{\text{OPT\_KS}(i+1, C)}^{\text{don't take item \#}i} \Big)$$

decreased capacity

added value

Base cases:

if $C < 0$, $\text{OPT\_KS}(i, C) = -\infty$

if $C = 0$, $\text{OPT\_KS}(i, C) = 0$

if $i = n+1$, $\text{OPT\_KS}(i, C) = 0$

Note:
Be strict with types!
# of parameters must match!

4. ... identify the optimum choice?

If we take item #1, then the best we can do is:

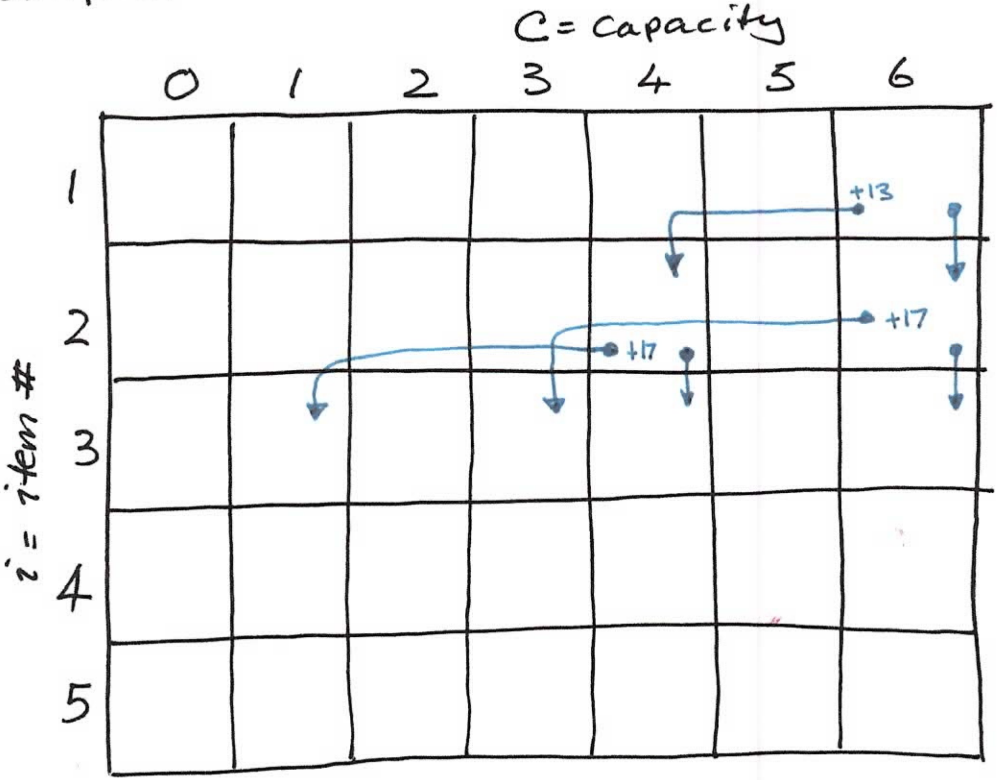
$$\underbrace{v_1}_{\text{value of item \#1}} + OPT\_KS(2, \underbrace{K}_{\text{capacity}} - \underbrace{w_1}_{\text{weight of item \#1}})$$

If we don't take item #1, then the best we can do is

$$OPT\_KS(\underbrace{2}_{\text{only consider items \#2 thru \#n}}, \underbrace{K}_{\text{still the same capacity}})$$

Knapsack example:

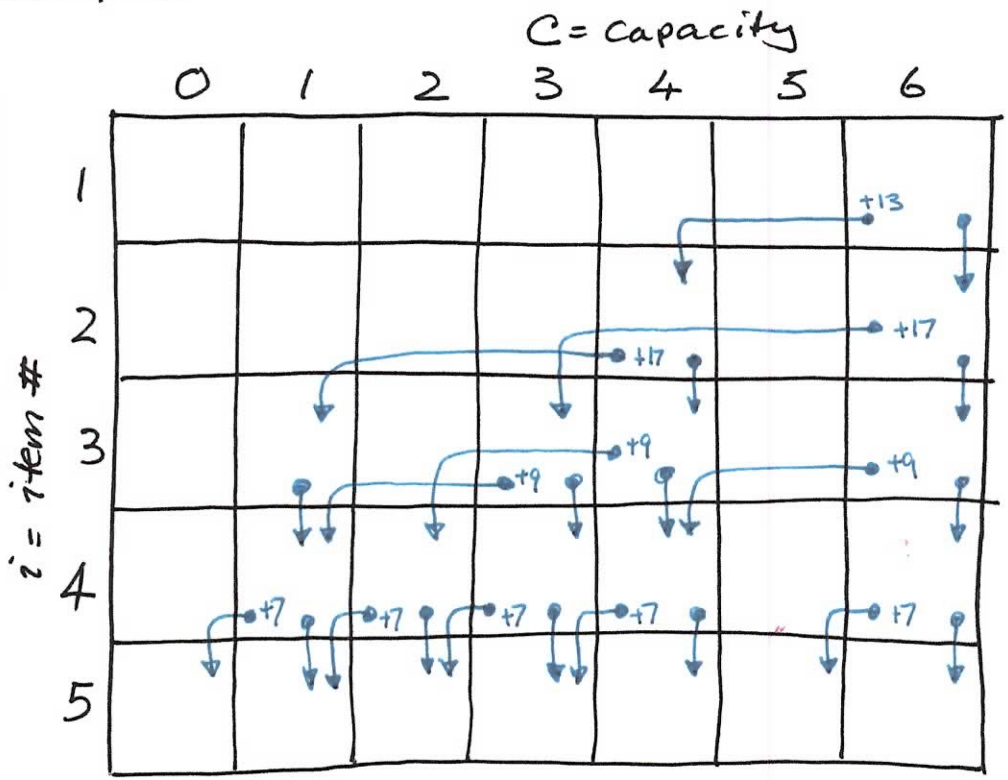
item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23



first few dependencies

Knapsack example:

item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23



all the dependencies

Note overlapping subproblems!

Knapsack example:

item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23

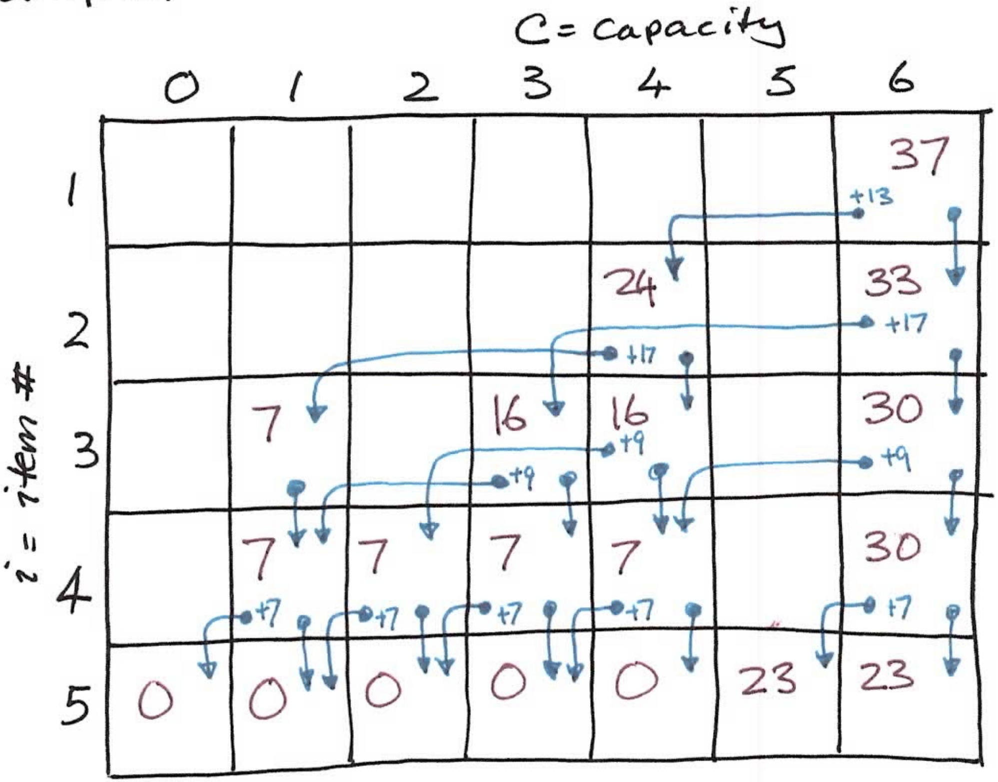
bottom 2rows of OPT_KS values

C = capacity

i = item #	0	1	2	3	4	5	6
1							
2							
3							
4		7	7	7	7		30
5	0	0	0	0	0	23	23

Knapsack example:

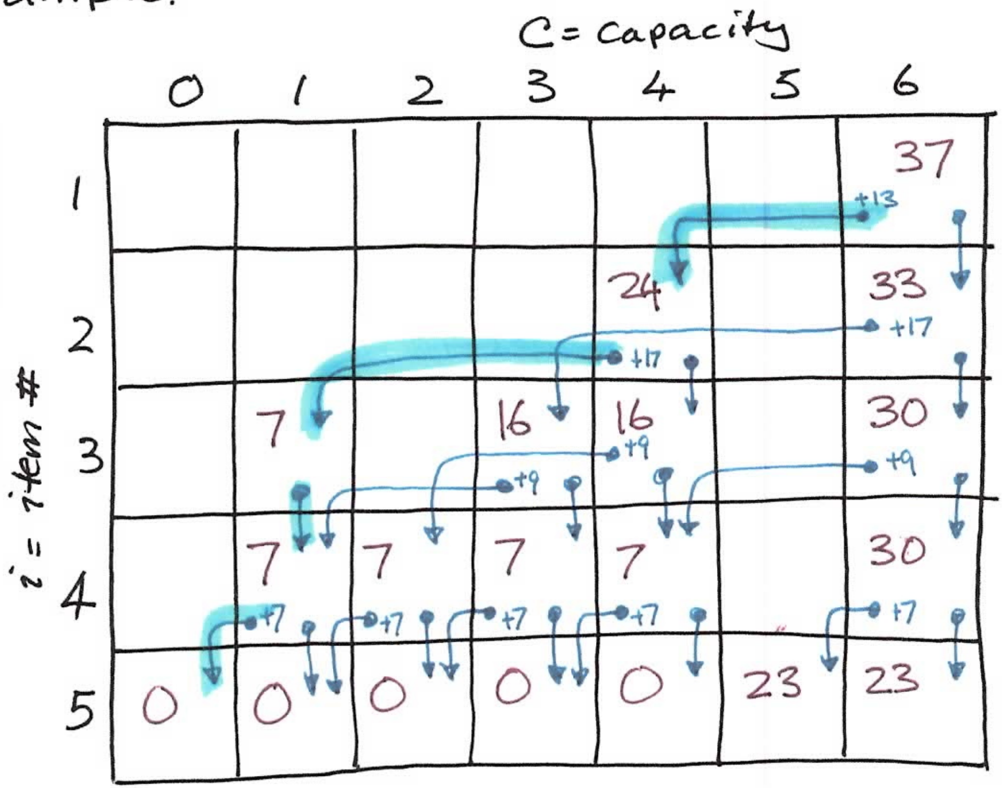
item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23



all the OPT_KS values

Knapsack example:

item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23



Recovering the solution:
take items 1, 2 & 4

Knapsack example:

item	weight	value
1	2	13
2	3	17
3	2	9
4	1	7
5	5	23

C = capacity

i = item #

i \ C	0	1	2	3	4	5	6
1	0	7 ↓	13 ←	20 ←	24 ↓	30 ←	37 ←
2	0	7 ↓	9 ↓	17 ←	24 ←	26 ←	33 ←
3	0	7 ↓	9 ←	16 ←	16 ←	23 ↓	30 ↓
4	0	7 ←	7 ←	7 ←	7 ←	23 ↓	30 ←
5	0	0	0	0	0	23	23

Table filled bottom up

↓ = don't take

← = take

path to actual solution

like they say on Myth busters

5. How many subproblems are there? ever?

Count the range of the parameters to OPT_KS()

OPT_KS(i , c)

ranges from 1 to n

ranges from 0 to K

# of possible subproblems = $n \cdot K$

6. Time to solve a single subproblem.

$$\mathrm{OPT\_KS}(i, C) =$$

$$\max\Big(\underbrace{\mathrm{OPT\_KS}(i+1, C-w_i)}_{\text{subproblem \#1}} + v_i\;,\; \underbrace{\mathrm{OPT\_KS}(i+1, C)}_{\text{subproblem \#2}}\Big)$$

$\Theta(1)$ time.

assume these have been solved and stored in a 2D memoization table

7. Total running time = $5 \times 6 = n \cdot K \cdot \Theta(1) = \Theta(nK)$.

hopefully, K is not too large.

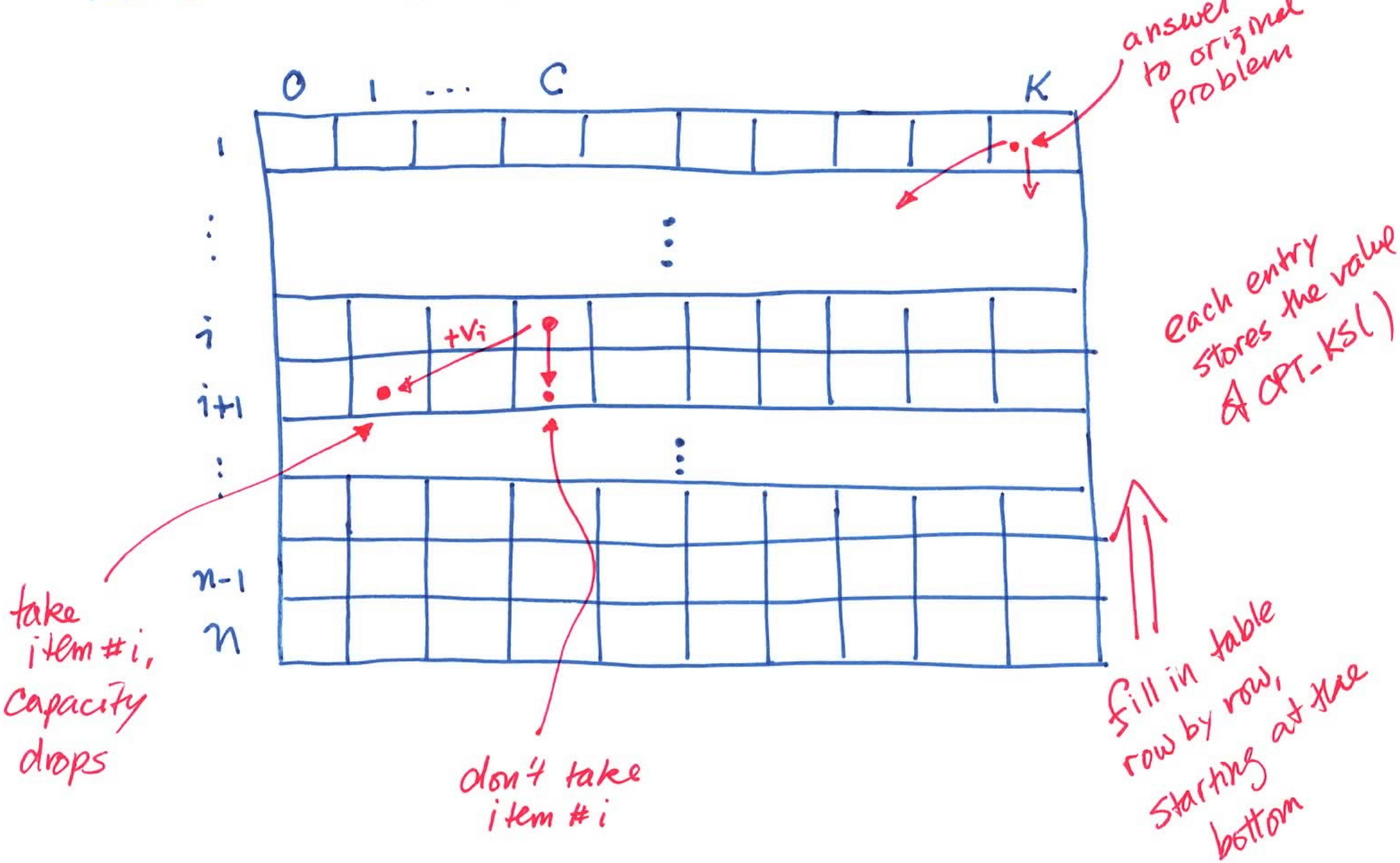
8. Reconstruct the solution...

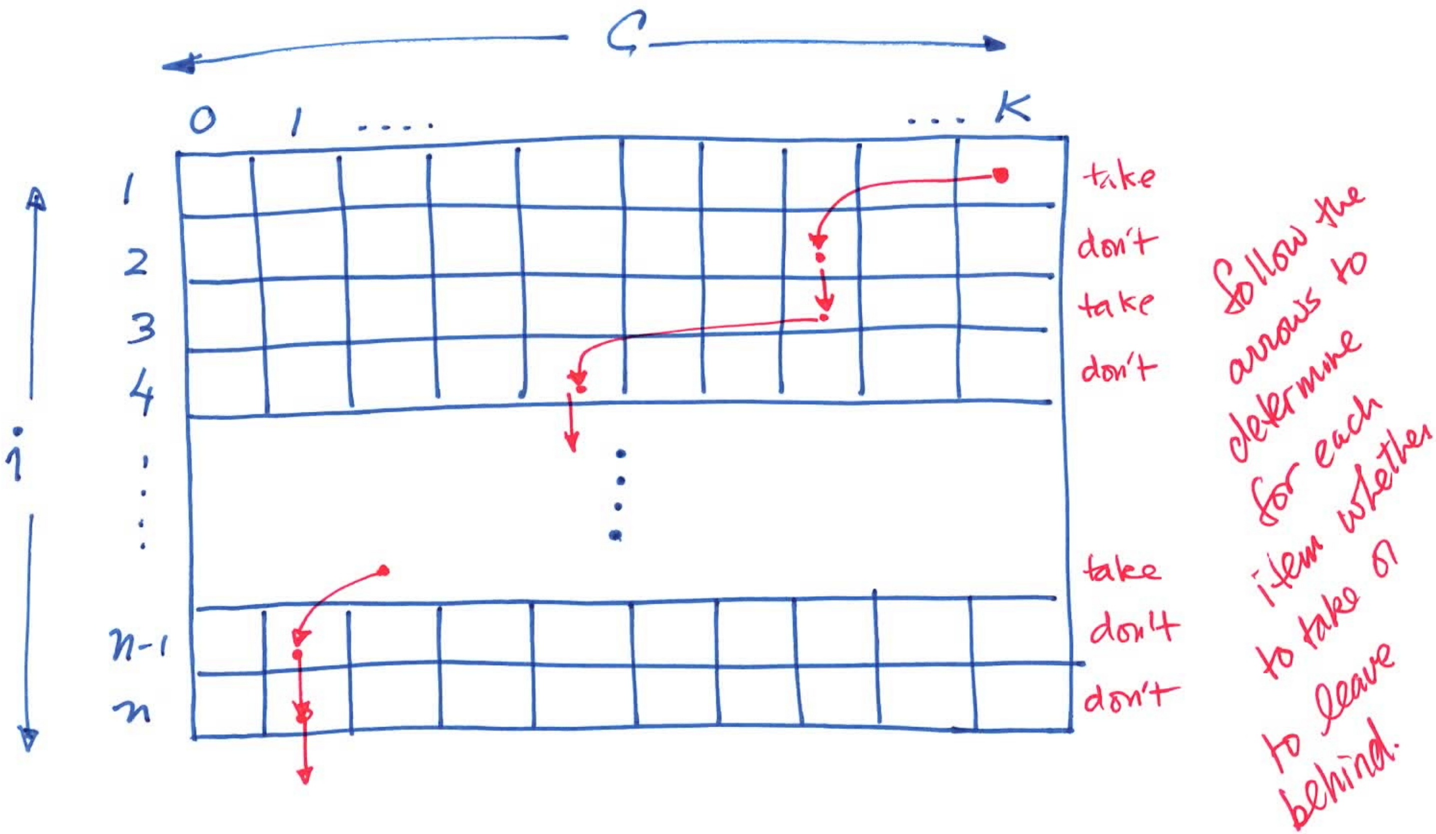
OPT_KS(1, K) is the "solution" to the original problem, but it is just a number.

Q: How can we tell whether to include item #i ?

Ans: Record the choices in the memoization table.

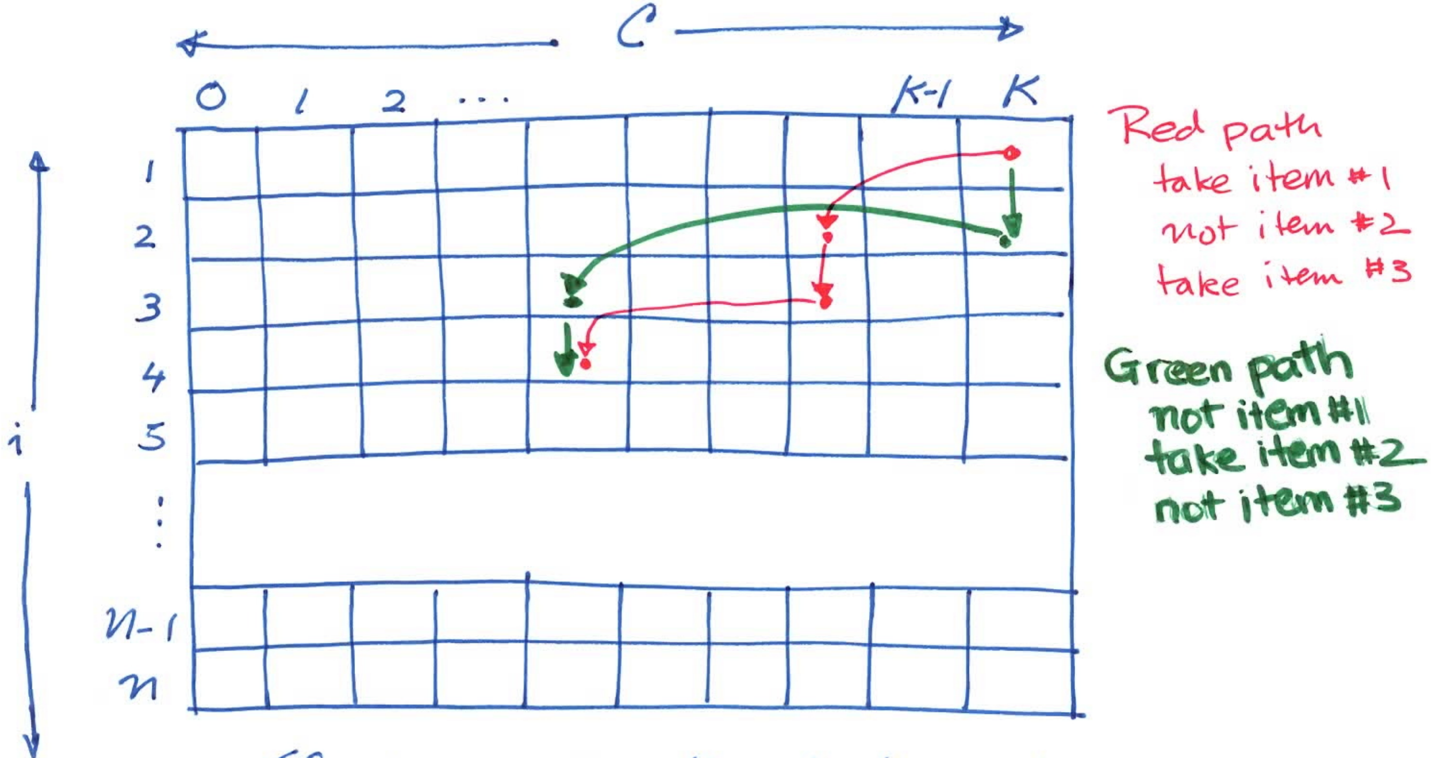
Two-dimensional memoization table





Reconstructing the "actual" solution

Overlapping subproblems



If $w_2 = w_1 + w_3$, then both path end up calling OPT_KS(4, $K - w_2$)

Code?

```
for i = n to 1 {
    for c = 1 to K {
        ⋮        // compute T[i,c]
    }
}
```

we fill in all entries even though many are not needed

constant time

Total running time $\Theta(nK)$

Final code is usually iterative & bottom up.

Proof of correctness?

We tried all possible choices
and picked the largest.

Q.E.D.

Some advice:

1. Think recursively.
2. Don't unwind the recursion.
3. Subproblem should return a numeric value used to find optimum choice.
4. Don't be too clever, try all possible choices.
5. Use lots of global variables, use parameters only for values that change.
6. Don't pass intermediate solutions into a recursive call.
7. Optimum choice cannot depend only on the attributes of current item.
8. Look for overlapping subproblems.