CS4102 Algorithms

Fall 2021 – Floryan and Horton

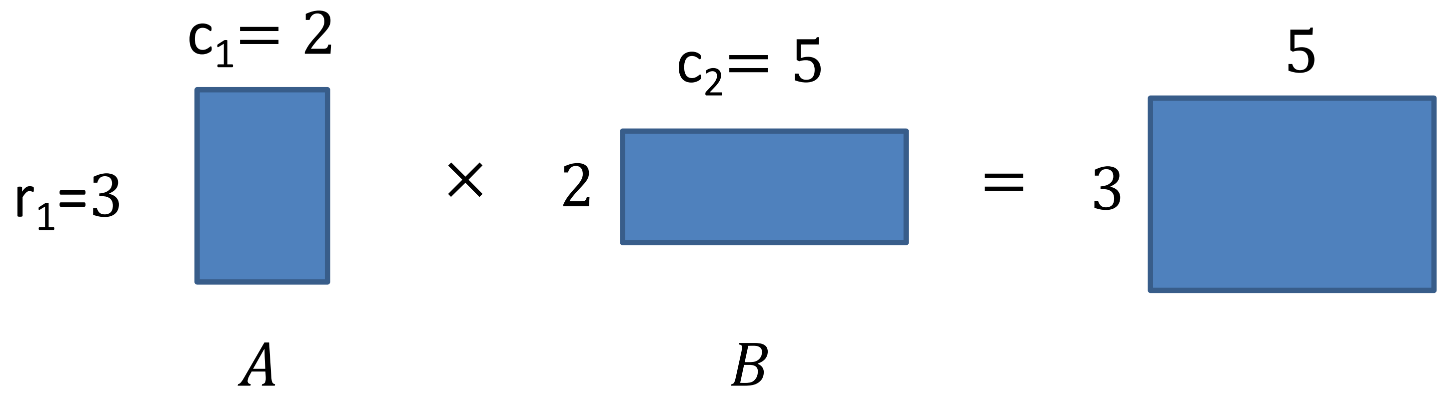
Module 8
Dynamic Programming

Dynamic Programming and Greedy Approach

- TOPICS:
  - Intro to Dynamic Programming
  - Memoization
  - Three DP Problems:
    - Log Cutting
    - 0/1 Knapsack
    - Coin Change
    - Weighted Activity Selection

Motivating Example

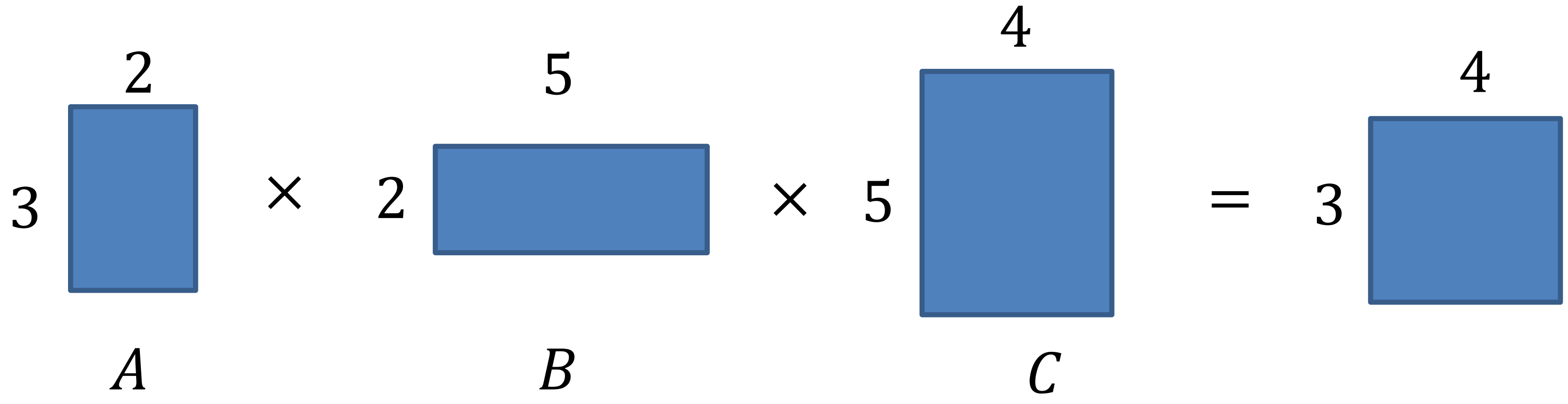
How many scalar multiplications are required to multiply matrices A and B in this example?



- $r_1 \cdot c_2$ elements in the result that we need to compute
- c_1 scalar multiplications per element in result
- Total cost: $r_1 \cdot c_1 \cdot c_2$
- So the answer is… $(3 \cdot 2 \cdot 5) = 30$

Trickier Question

What's the smallest number of scalar multiplications required to calculate the matrix product ABC in this example?



- For a pair of matrices, remember it's $r_1 \cdot c_1 \cdot c_2$
- Calculate this cost for multiplying one pair of matrices
- You need to multiply that result with the 3rd matrix, too, so there's a cost for that
- Total cost is the sum of these two costs
- So the answer is… $(3 \cdot 2 \cdot 5) + (3 \cdot 5 \cdot 4) = 90$

Nope! The answer is 64.
Think about how this might be!

CLRS Readings

- Chapter 15, Dynamic Programming
  - Section 15.1, Log/Rod cutting, optimal substructure property
    - Note: r_i in book is called Cut() or C[] in our slides. We do use their example.
  - Section 15.3, More on elements of DP, including optimal substructure property

Dynamic Programming and Greedy Approach

- Module 8 is on Dynamic Programming
  - Similar to ***Greedy Algorithms***
  - Solves problems that have ***optimal substructure***, but do NOT have a known greedy choice for optimal solutions
  - Instead, ***try every op***tion for the first "greedy" choice and see which one leads to optimal solution.
    - Will need some ***optimizations*** to make this efficient.

Optimization Problems

- Both DP and Greedy solve ***optimization problems**:*
  Find the best solution among all ***feasible*** solutions
- An example you know: *Find the shortest path in a weighted graph G from s to v*
  - Form of the solution: a path (and sum of its edge-weights)
- Feasible solutions must meet problem constraints
  - Example: All edges in solution are in graph G and form a simple path from *s* to *v*
- We can get a score for each feasible solution on some criteria:
  We call this the ***objective function***
  - Example: the sum of the edge weights in path
- One (or more) feasible solutions that scores highest (by the objective function) is the ***optimal solution(s)***

Memoization

Remember Fibonacci numbers?

- Formula: F(n) = F(n-1) + F(n-2)
- Recursive code:

```
long fib(int n) {
    assert(n >= 0);
    if ( n == 0 ) return 0;
    if ( n == 1 ) return 1;
    return fib(n-1) + fib(n-2);
}
```

- What's the problem?
  – Repeatedly solves the same subproblems
  – "Obscenely" exponential

Top-down using Memoization

- Before talking about bottom-up dynamic programming using tables, top-down approach uses general technique of **Memoization**
  - AKA using a *memory function*
- Simple idea:
  - Calculate and store solutions to subproblems
  - Before solving it (again), look to see if you've remembered it

Memoization

- Use a Table abstract data type
  - Lookup key: whatever identifies a subproblem
  - Value stored: the solution
- Could be an array/vector or 2D table(s)
  - E.g. for Fibonacci, store **fib(n)** using index **n**
  - Need to initialize the array
- Could use a map / hash-table

Memoization and Fibonacci

- Before recursive code below called, must initialize results[] so all values are -1

```
long fib_mem(int n, long results[]) {
    if ( results[n] != -1 )
        return results[n];  // return stored value
    long val;
    if ( n == 0 || n ==1 ) val = n; // odd but right
    else
        val = fib_mem(n-1, results)
              + fib_mem(n-2, results);
    results[n] = val; // store calculated value
    return val;
}
```

Observations on fib_mem()

- Same elegant top-down, recursive approach based on definition
  - Without repeated subproblems
- Memory function: a function that remembers
  - Save time by using extra space
- Can show this runs in $\Theta(n)$

Dynamic Programming and Log Cutting

Dynamic programming

- Old “bad” name (see Wikipedia or textbook)
- Useful when the solution can be recursively described in terms of solutions to sub-problems (*optimal substructure*)
  - But *greedy choice property* doesn’t hold for the problem
- Algorithm finds solutions to sub-problems and stores them in memory for later use
- More efficient than *brute-force methods* or recursive approaches that solve the same sub-problems over and over again

Optimal Substructure Property

- Definition
  - If S is an optimal solution to a problem, then the components of S are optimal solutions to sub-problems
- Examples:
  - True for coin-changing
  - True for single-source shortest path
  - Not true for longest-simple-path
  - True for knapsack

Dynamic Programming

- Works "bottom-up"
  - Finds solutions to small sub-problems first
  - Stores them
  - Combines them somehow to find a solution to a slightly larger sub-problem
- Comparison to greedy approach
  - Also requires optimal substructure
  - But greedy makes choice first, then solves
  - Greedy looks only at the current situation, not at a past 'history'
- DP is good when sub-problems overlap, when they're not independent
  - No need to repeat the calculation to solve them
  - Dynamic programming has stored them, so doesn't repeat the calculation

Process for Dynamic Programming

1. Recognize what the sub-problems are

2. Identify the recursive structure of the problem in terms of its sub-problems
   - At the top level, what is the “last thing” done?
   - This helps you see a recursive solution for any generic sub-problem in terms of smaller sub-problems

3. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time

4. Develop an algorithm that loops through data structure solving each sub-problem one at a time
   - Bottom-up: from smallest sub-problems, to next largest, …, to complete problem

Problems Solved with Dyn. Prog.

- Log cutting (first example, uses list data structure)
- 0/1 knapsack problem
- Coin changing with “non-standard” coin selection
- Longest common subsequence
- Multiplying a sequence of matrices
  - Can do in various orders: (AB)C vs. A(BC)
  - Pick order that does fewest number of scalar multiplications

And ones we might not get to:

- All-pairs shortest paths (Floyd’s algorithm)
- Constructing optimal binary search trees

Log Cutting

Given a log of length n, and
a list (of length n) of prices P ($P[i]$ is the price of a cut of size i)
Find the best way to cut the log to maximize our profit.
(Imagine we can sell each piece of the log at price $P[i]$)

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

Select a list of lengths $\ell_1, \ldots, \ell_k$ such that:
$\sum \ell_i = n$
to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively. (Using memorization – we'll do later!)
     - "Bottom Up": Iteratively solve smallest to largest

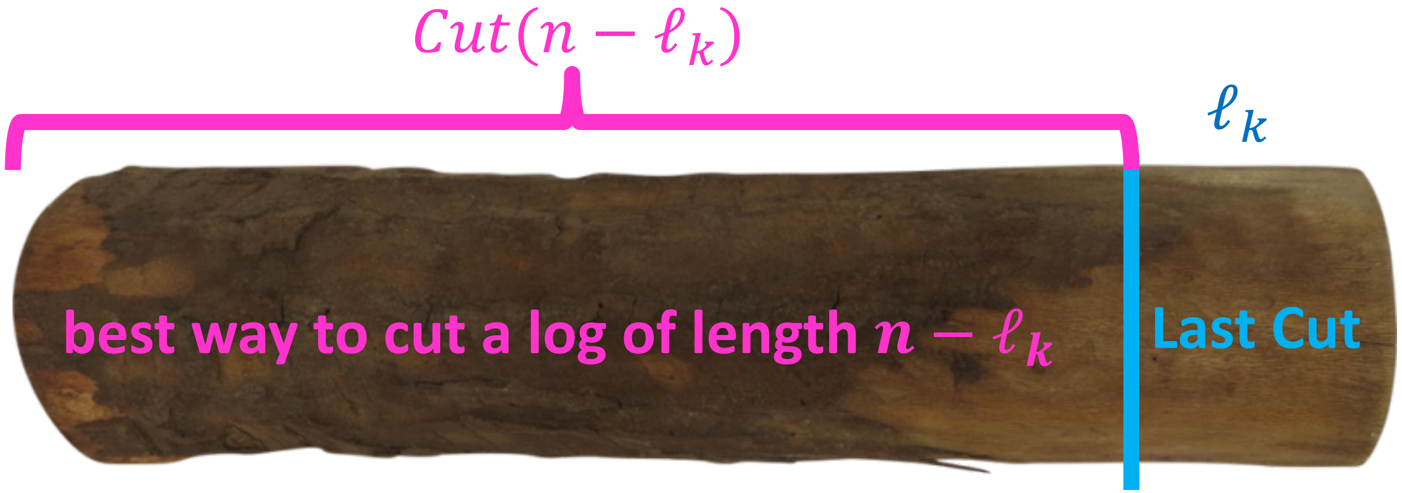
1. Identify Recursive Structure

$P[i]$ = value of a cut of length i

$Cut(n)$ = value of best way to cut a log of length n

$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \dots \\ Cut(0) + P[n] \end{cases}$$

So for a given value of *n*, to find *Cut(n)*, we need sub-problem solutions for *Cut(n-1)* down to *Cut(0)*.



What's the problem with a top-down recursive approach?

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(0) = 0$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

0

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(1) = Cut(0) + P[1]$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(2) = \max \begin{cases} Cut(1) + P[1] \\ Cut(0) + P[2] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

2

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(3) = \max \begin{cases} Cut(2) + P[1] \\ Cut(1) + P[2] \\ Cut(0) + P[3] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

3

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(4) = \max \begin{cases} Cut(3) + P[1] \\ Cut(2) + P[2] \\ Cut(1) + P[3] \\ Cut(0) + P[4] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

4

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

Log Cutting Pseudocode

```
Initialize Memory C
Cut(n):
    C[0] = 0
    for i=1 to n:  // log size
        best = 0
        for j = 1 to i: // last cut
            best = max(best, C[i-j] + P[j])
        C[i] = best
    return C[n]
```

Run Time: $O(n^2)$

How to find the cuts?

- This procedure told us the profit, but not the cuts themselves
- Idea: remember the choice that you made, then backtrack

Remember the choice made

```
Initialize Memory C, Choices
Cut(n):
    C[0] = 0
    for i=1 to n:
        best = 0
        for j = 1 to i:
            if best < C[i-j] + P[j]:
                best = C[i-j] + P[j]
                Choices[i]=j
        C[i] = best
    return C[n]
```

Gives the size of the last cut

Reconstruct the Cuts

- Backtrack through the choices

Choices:	0	1	1	2	4	3	4	1	2	4	3
Length:	0	1	2	3	4	5	6	7	8	9	10
		1	2				6	7			

Example to demo Choices[] only. Profit of 20 is not optimal!

Backtracking Pseudocode

```
i = n
while i > 0:
        print Choices[i]
        i = i – Choices[i]
```

Our Example: Getting Optimal Solution

i	0	1	2	3	4	5	6	7	8	9	10
C[i]	0	1	5	8	10	13	17	18	22	25	30
Choices[i]	0	1	2	3	2	2	6	1	2	3	10

- If n were 5
  - Best score is 13
  - Cut at Choices[n]=2, then cut at Choices[n-Choices[n]]= Choices[5-2]= Choices[3]=3
- If n were 7
  - Best score is 18
  - Cut at 1, then cut at 6

Weighted Activity Selection

Weighted Interval Scheduling

- Recall Interval Scheduling:
  - Given a list of intervals pick a *schedule* of non-overlapping intervals that maximizes the number chosen
    - i.e. each one has the same value

- Weighted interval scheduling is similar, but…
  - Each interval has a different value

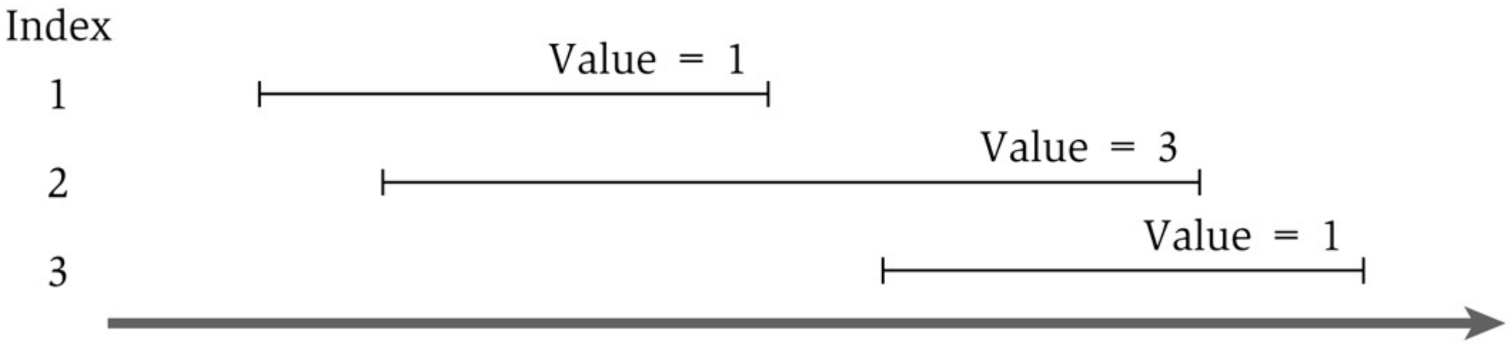
Greedy solution to interval scheduling

- The algorithm:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities

- Intuition is even more simple:
  - Always pick next activity that finishes earliest

Greedy solution to the weighted version

- What would the greedy algorithm pick for this example?
- And is that answer optimal?



- We can see that the greedy algorithm does not work for the weighted version

Step 1

- Define the sub-problem
- This problem has optimal substructure, so let's only consider intervals up to a certain point.
- Let Opt(j) be the solution to this problem when only considering intervals 1 through j
  - How should we order the intervals? Does it matter? We will see soon that it does.
- Note that Opt(0) = 0

Step 2

- Define solution to problem in terms of sub-problems

- Base Case:
  - Opt(0) = 0

- Opt(j) = ?

Step 2

- Opt(j) = ?

- Two cases:
  - Interval j is not in the optimal solution
    - Opt(j) = Opt(j-1) //same solution, because j interval doesn't matter

  - Interval j is in the optimal solution
    - Opt(j) = Vj + Opt(intervals compatible with j)
    - Intervals compatible with j? Yikes? How do we calculate that?

Calculating Opt(j)

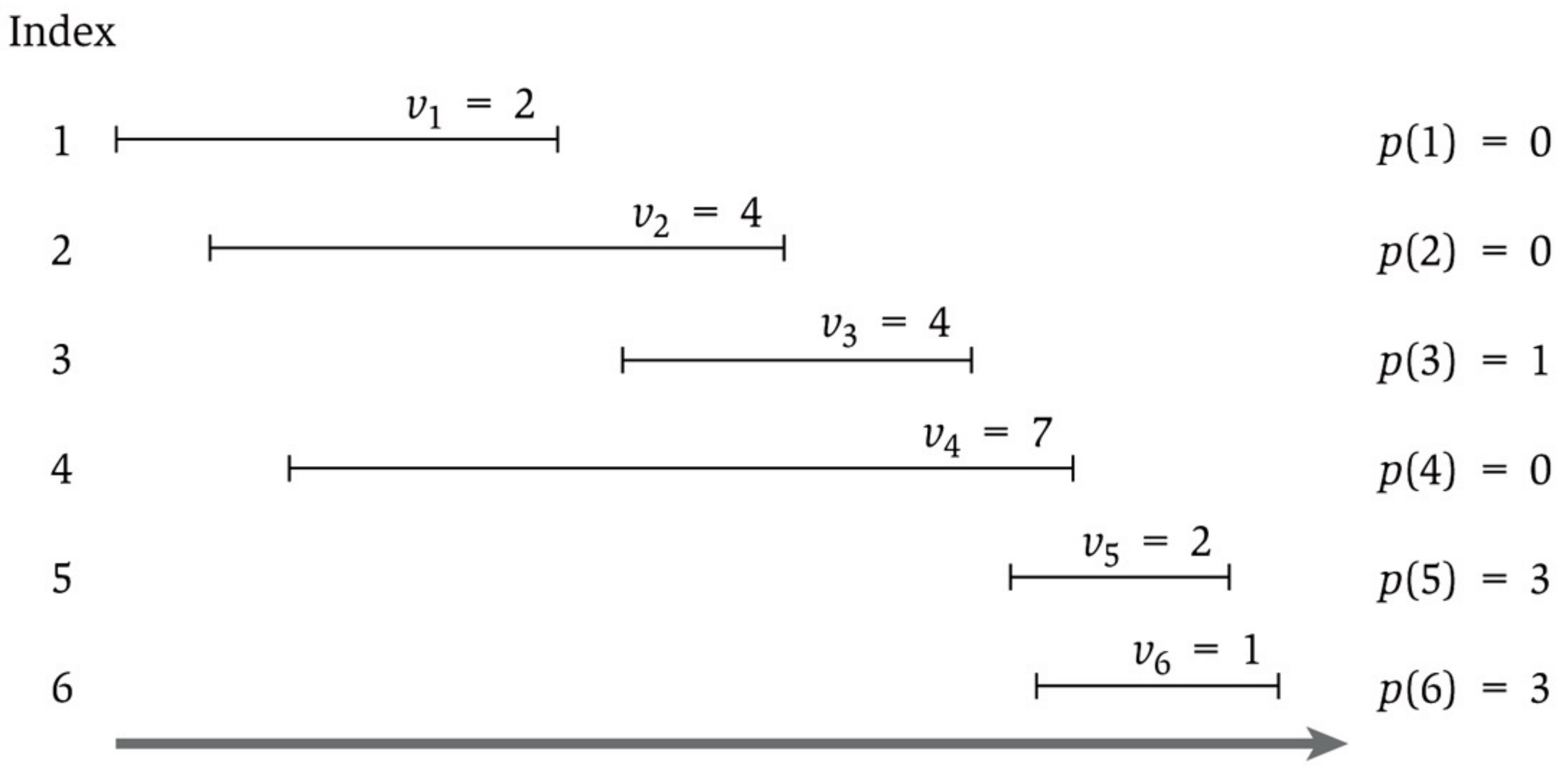
- Sort all intervals by their finish time
  - And number them sequentially

- We define interval i is less than interval j if i finishes before j (i.e. is before it in the sort)

- Define p(j) to be the highest numbered interval i<j such that i and j are disjoint

- Define OPT(j) to be the value of an optimal solution for intervals 1 through j only

Showing p(j)



Step 2

- Opt(j) = ?

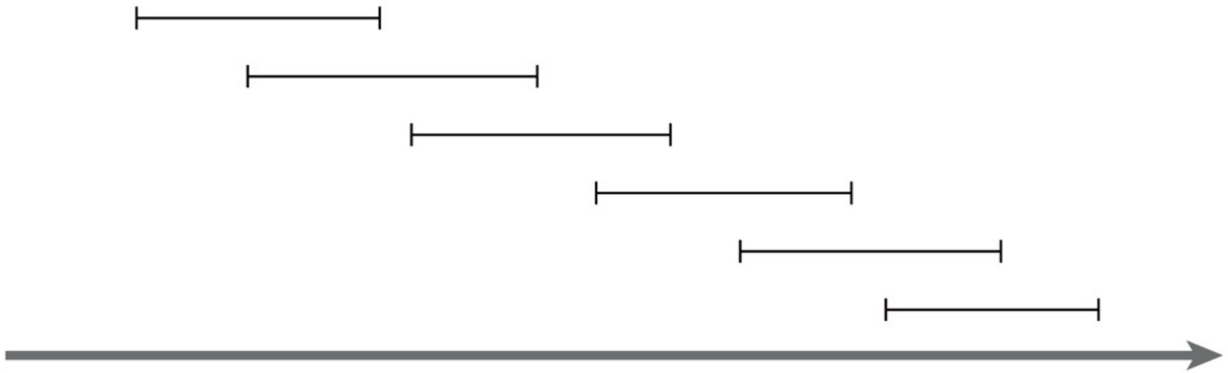
- Two cases:
  - Interval j is not in the optimal solution
    - Opt(j) = Opt(j-1) //same solution, because j interval doesn't matter

  - Interval j is in the optimal solution
    - Opt(j) = Vj + Opt(p(j))

  - So…we have
    - Opt(j) = Max(Opt(j-1), Vj + Opt(p(j)))

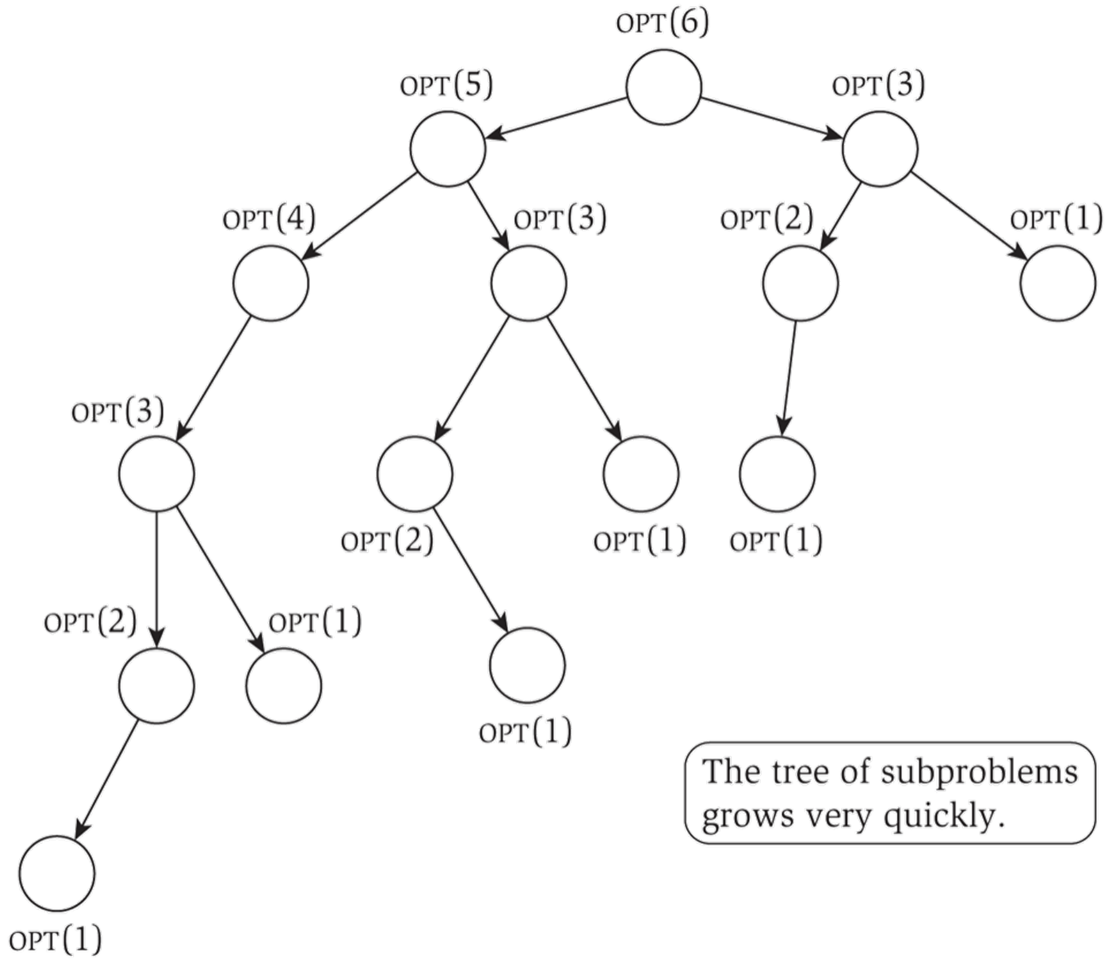
Recursive solution

- OPT(j) = max(v_j + OPT(p(j)), OPT(j-1))
  - And OPT(0) = 0
- This is similar in running time to the Fibonacci sequence
  - And similarly exponential
- Consider a simple example:



That example will take exponential time

- Notice that the sub-problems are being re-computed each time

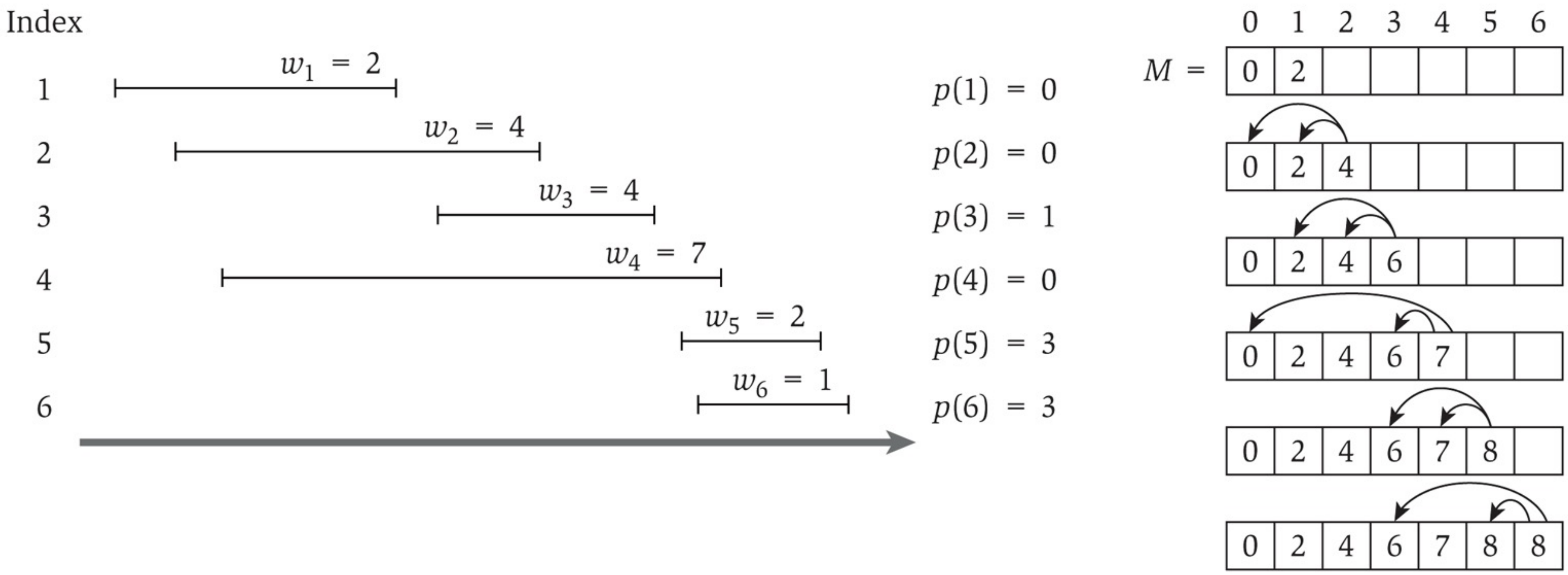


Step 3!

- Formulate the data structure to look up subproblems.
- Pretty simple, define M[n]
- M[j] stores the solution to Opt(j)

So we add memoization…

- This runs in linear time



Computing the intervals

- The solution only gives us the final value
  - Computing a sub-array each step would make it quadratic running time
- To determine the intervals:
  - If $v_j + M[p(j)] \geq M[j-1]$
    - Then j is part of the solution, and consider p(j)
  - Else
    - Then j is NOT part of the solution, and consider j-1

0/1 Knapsack Problem

Reminder: Knapsack Problems

- Pages 425-427 in textbook
- **Description:** Thief robbing a store finds n items, each with a profit amount p_i and a weight w_i
  - Wants to steal as valuable a load as possible
  - But can only carry total weight C in their knapsack
  - Which items should they take to maximize profit?
- Form of the solution: an x_i value for each item, showing if (or how much) of that item is taken
- Inputs are: C, n, the p_i and w_i values

Two Types of Knapsack Problem

- 0/1 knapsack problem
  - Each item is discrete: must choose all of it or none of it. So each x_i is 0 or 1
  - Greedy approach does not produce optimal solutions
  - But dynamic programming does

- Fractional knapsack problem (AKA continuous knapsack)
  - Can pick up fractions of each item. So each x_i is a value between 0 or 1
  - A greedy algorithm finds the optimal solution

A Bit More Terminology

- Problems solvable by both Dynamic Programming and the Greedy approach have the **optimal substructure property:**
  - An optimal solution to a problem contains within it optimal solutions to subproblems
  - This allows us to build a solution one step at a time, because we can solve increasingly smaller problems with confidence
- Dynamic Programming not a good solution for problems that have the **greedy-choice property:**
  - We can assemble a globally-optimal solution for the current by making a locally-optimal choice, without considering results from subproblems

0/1 knapsack

n = 3, C = 4

Item	Value	Weight	Ratio
1	3	1	3
2	5	2	2.5
3	6	3	2

Let's try this same greedy solution with the 0/1 version

- New example inputs →

1. Item 1 first. So x_1 is 1.
   Capacity used is 1 of 4. Profit so far is 3.
2. Item 2 next. There's room for it! So x_2 is 1. Capacity used is 3 of 4.
   Profit so far is 3 + 5 = 8.
3. Item 3 would be next, but its weight is 3 and knapsack only has 1 unit left!
   So x_3 is 0. **Total profit is 8. x_i = (1, 1, 0)**

But picking items 1 and 3 will fit in knapsack, with total value of 9

- Thus, the greedy solution does not produce an optimal solution to the 0/1 knapsack algorithm
- Greedy choice left unused room, but we can't take a fraction of an item
- The 0/1 knapsack problem doesn't have the *greedy choice property*

Reminders about Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Strategy:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - “Bottom Up”: Iteratively solve smallest to largest
     - “Top Down”: Solve each recursively. (We won’t do this for 0/1 knapsack.)

Dynamic programming solution to 0/1

We need to:

- Identify a recursive definition of how a larger solution is built from optimal results for smaller sub-problems.

For 0/1 knapsack, what a <u>sub-problem</u> solution look like?
What can be "smaller"?

  - Smaller capacity for the knapsack
  - Fewer items

Some assumptions and observations

- Given a set S of the objects and a capacity C
  - We assume the optimal solution is O, a subset of S
  - For example, the items in O could be the bolded ones:
    S = { $\mathbf{s_1}$, s_2, $\mathbf{s_3}$, …, s_{k-1}, $\mathbf{s_k}$, …, s_n }
  - Note that the last item s_n may or may not be in the solution O

- Let's use subscripts on O_k and S_k when we're talking about the first *k* items

- BTW, we'll assume C and all w_i are integer values
  - And, most books etc. use "W" for what we're calling C

Recursive Structure

What's a recursive definition of how a solution of **size n** is built from optimal results for smaller sub-problems? $S = \{ s_1, s_2, s_3, \ldots, s_{n-1}, s_n \}$

- Let's say $s_n \notin O_n$ (last item **is not** in optimal solution for S_n):
  - Last item didn't add anything to best solution for smaller subproblem
  - We need optimal solution O_{n-1} for the following smaller subproblem S_{n-1}:
    n-1 items using <u>same</u> knapsack capacity C

- Let's say $s_n \in O$ (last item **is** in optimal solution for S_n):
  - Last item contributed w_i to total weight we're carrying
  - We need optimal solution O_{n-1} for the following smaller subproblem S_{n-1} :
    n-1 items using <u>reduced</u> capacity $C\text{-}w_n$

(Note that "getting smaller" decreases number of items and also maybe capacity.)

First Step: Getting Things Started

- For sub-problems, what variables change in size?
  - Maybe C (the capacity) and definitely k (number of items to steal)
- Define what we're calculating: call it **Knap(k, w)**
  - Note: we'll use "w" for the changing capacity value in Knap(), but keep "C" as the overall total capacity for the entire problem. (Sorry if confusing!)
- Whether we do recursion of work bottom-up, we need to know the smallest cases
- Some small or boundary cases:
  - No knapsack capacity (w=0), can't add an item, so Knap(k, 0) = 0
  - Nothing to steal (k=0), so Knap(0, w) = 0

Three cases to calculate Knap(k, w)

- Three cases for calculating Knap(k, w):
  1. There is sufficient capacity to add item s_k to the knapsack, and that creates an optimal solution for k items
  2. There is sufficient capacity to add item s_k to the knapsack, and that does **NOT** create an optimal solution for k items
  3. There is insufficient capacity to add item s_k to the knapsack

- Case 3 is easy to determine; we'll have to compute whether 1 or 2 is optimal
  – How do we know which is optimal? Compute both, pick larger value!

Case 1: Sufficient capacity and Optimal

- There is sufficient capacity to add item s_k to the knapsack, and that creates an optimal solution for k items

- Thus, our solution for the first k items is when we add item s_k to the optimal solution for the first k-1 items

- But by adding item s_k to the knapsack, we have reduced capacity
  - In particular, we only have **$w-w_k$** for to steal the first **k-1** items

- So the value for **$Knap(k, w) = v_k + Knap(k-1, w-w_k)$**

Case 2: Sufficient Capacity but Non-optimal

- There is sufficient capacity to add item s_k to the knapsack, and that does **NOT** create an optimal solution for k items

- Thus, our solution for the first k items is when we do NOT add item s_k to the solution for the first k-1 items
  - Since we are **not** adding item s_k to the knapsack, the solution is the optimal solution to steal the first **k-1** items with the **same capacity**
  - So **Knap(k, w) = Knap(k-1, w)**

Case 3: Insufficient Capacity

- There is insufficient capacity to add item s_k to the knapsack
  - This is because $w-w_k < 0$ (i.e. $w < w_k$)
- Then **Knap(k, w) = Knap(k-1, w)**
  - Since we can't add item s_k to the knapsack, the solution is the same as the first k-1 items with the same capacity
  - Note that this formula is the same as case 2

Putting It All Together

- Recursively define solutions to sub-problems
- Base Case

  Knap(k,0) = 0

  Knap(0,w) = 0
- Recursive Case

  Subproblems are smaller!

  Knap(k, w) = max(Knap(k-1, w), Knap(k-1, w-w_k) + v_k)

  No room for s_k or not part optimal solution

  s_k is part of optimal solution

Reminders about Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Strategy:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - “Bottom Up”: Iteratively solve smallest to largest
     - “Top Down”: Solve each recursively. (We won’t do this for 0/1 knapsack.)

Lookup Table

- We want a data-structure that allows us to lookup a sub-problem value in O(1) time
- Knap(k, w) has two parameters, so two-dimensional array works great.
- Make an array called V[k, w]
  - Store solution to Knap(k, w) at position V[k, w]

Determining the cases

- To determine between cases 1 and 2
  - Simply compute both values, and take the higher

```
if (w-w_k< 0) // not room for item k
      V[k, w] = V[k-1, w] // best result for k-1 items
else {
      val_with_kth = v_k + V[k-1, w-w_k] // Case 1 above
      val_for_k-1 = V[k-1, w] // Case 2 above
      V[k, w] = max( val_with_kth, val_for_k-1 )
}
```

Put Values in Table

- Write a loop that fills in the table one cell at a time
- The table fills in one row at a time, moving rightwards and downwards

V[k,w]	w = 0	w = 1	w = 2	…	w = C
k = 0	0	0	0	0	0
k = 1	0				
k = 2	0				
…	0				
k = n	0				

Pseudo-code

```
Knapsack(v, w, C) {
   for (w = 0 to C) V[0, w] = 0
   for (k = 0 to n) V[k, 0] = 0
   for (k = 1 to n) {          // loop over all rows
         for (w = 1 to C) {   // loop over all columns
           if (w-w_k <  0)    // not room for item k
              V[k, w] = V[k-1, w] // best result for k-1 items
           else {
              val_with_kth = v_k + V[k-1, w-w_k] // Case 1 above
              val_for_k-1 = V[k-1, w]          // Case 2 above
              V[k, w] = max( val_with_kth, val_for_k-1 )
           }
         }
   }
   return V[n,C]
}
```

But our solution is only the value!

- Value V[n, C] is the optimal value
- To find which items were chosen, we can trace backward through the table starting at V[n, C]
  - If V[k, w] = V[k-1, w], then **s_k is not an item in the knapsack** (this was from cases 2 and 3). Look at V[k-1, w] next.
  - Otherwise, **s_k is an item in the knapsack**, and we look at V[k-1, w-w_k] next (this was from case 1)

Coin Change
with non-traditional coin sets

Making Change

- The problem:
  - Give back the right amount of change, and…
  - Return the fewest number of coins!
- Inputs: the dollar-amount to return
  - Also, the set of possible coins. (Do we have half-dollars? That affects the answer we give.)
- Output: a set of coins

- Note this problem statement is simply a transformation
  - Given input, generate output with certain properties
  - No statement about how to do it.
- Can you describe the algorithm you use?

Greedy algorithm

- Given coin cent amounts of 10, 6, 5, and 1
- Compute the coins needed for 12 cents
  - The greedy algorithm picks {10, 1, 1}
  - But {6, 6} is more optimal (fewer coins)

Definitions

- We define an array denom which holds the denominations of the coins such that:
  - denom[1] > denom[2] > … > denom[n] = 1
  - In other words, we sort the coin denominations in decreasing order, ending with a penny
- We are obtaining change for an amount A
- Consider the i,j problem:
  - The available denominations are denom[i] through denom[n], where i ≥ 1 (i.e. the smaller n-i+1 coins)
    - Note: when i is large, you're working with fewer types of coins, and when i=1 you're working with your complete set
  - The amount we are looking for is j, where j ≤ A (i.e. the remaining amount of money)

The i,j problem

- Consider the i,j problem: (Remember, i is which coins, and j is the amount)
  - The available denominations are denom[i] through denom[n], where i ≥ 1 (i.e. the smaller n-i+1 coins)
  - The amount we are looking for is j, where j ≤ A (i.e. the remaining amount of money)
- Given coins of denominations 10, 6, and 1, here's the table showing how to create change up to 12 cents: **Our answer!**

j (the amount)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	
1	0	1	2	3	4	5	1	2	3	4	1	2	2	Can use 1, 6 & 10
2	0	1	2	3	4	5	1	2	3	4	5	6	2	Can use 1 & 6
3	0	1	2	3	4	5	6	7	8	9	10	11	12	Can use 1

Solving the problem

- How to solve the i,j problem (Remember, i is which coins, and j is the amount)
  - If denom[i] > j, then not possible to include this coin
    - Then the solution is the same as the (i+1),j problem (same amount, but with one fewer of the coin-options)
    - In the table, that's the cell right below the current cell.
    - Is this making the problem simpler?
  - Maybe the best answer <u>does</u> use a coin of denomination i
    - Then the solution is 1 more than the i,(j-denom[i]) problem
    - j changes to j-denom[i] because we subtract off the value of the coin used
    - i doesn't change because there could be multiple coins of denomination i used in the solution
  - Maybe the best answer <u>does NOT</u> use a coin of denomination i
    - Then the solution is the same as the (i+1),j problem
    - In the table, that's the cell right below the current cell

The formulaic solution

- The solution becomes:

$$C[i][j] = \begin{cases} C[i+1][j] & \text{if } denom[i] > j \\ \min\left(C[i+1][j], 1 + C[i][j - denom[i]]\right) & \text{if } denom[i] \le j \end{cases}$$

- Where C[i][0] = 0 for all values of i

- If we have a penny, then C[n][j] = j
  - This is required to get all amounts, so we assume a penny is the smallest denomination

Recursive solution

- The solution:

$$C[i][j] = \begin{cases} C[i+1][j] & \text{if } denom[i] > j \\ \min\left(C[i+1][j], 1 + C[i][j - denom[i]]\right) & \text{if } denom[i] \le j \end{cases}$$

- Note that a given problem (C[i][j]) is expressed in terms of sub-problems

- We can write a solution now using memorization with a top-down solution (recursive calls), or a bottom-up approach (build a table)

The bottom-up algorithm

```
dynamic_coin_change1 (denom, A, C) {
    n = denom.last
    for j = 0 to A
        C[n][j] = j
    for i = n-1 down to 1
        for j = 0 to A
            if ( denom[j] > j ||
                C[i+1][j] < 1 + C[i][j-denom[i]] )
                    C[i][j] = C[i+1][j]
            else
                    C[i][j] = 1 + C[i][j-denom[i]]
}
```

Time complexity?

Constant time to file each cell in the table. So $\Theta(n \cdot A)$ where n is the number of coins and A is the amount

But how to get the coins chosen?

- It's easy to trace back through the values
- Or, we could keep a *used* Boolean array
  - If used[i][j] is true, then the solution for i,j does use a coin of denom[i] for amount j
  - If false, it does not

j

i

	0	1	2	3	4	5	6	7	8	9	10	11	12	
1	F	F	F	F	F	F	F	F	F	F	T	T	F	Can use 1, 6 & 10
2	F	F	F	F	F	F	T	T	T	T	T	T	T	Can use 1 & 6
3	F	T	T	T	T	T	T	T	T	T	T	T	T	Can use 1

Recording the answers

```
dynamic_coin_change2 (denom, A, C, used) {
    n = denom.last
    for j = 0 to A
        C[n][j] = j
        used[n][j] = true
    for i = n-1 downto 1
        for j = 0 to A
            if ( denom[j] > j ||
                C[i+1][j] < 1+C[i][j-denom[i]] )
                    C[i][j] = C[i+1][j]
                    used[i][j] = false
            else
                    C[i][j] = 1 + C[i][j-denom[i]]
                    used[i][j] = true
}
```

Obtaining the coin set

```
optimal_coins_set (i, j, denom, used) {
  if ( j == 0 )
      return
  if ( used[i][j] )
      println ("Use coin of denomination " + denom[i])
      optimal_coins_set (i, j-denom[i], denom, used)
  else
       optimal_coins_set (i+1, j, denom, used)
}
```