Lecture 23: Dynamic Programming

CSE 373: Data Structures and Algorithms

Warm Up

Apply Bucket Sort and Radix Sort to the sequence

[7625, 5002, 6746, 7403, 2266, 5532, 2010]

Slido Event #3876460
https://app.sli.do/event/q3qkeoafKp3k9JMJkKTqNm

Bucket

1 scatter to buckets, perform insertion on each bucket, gather buckets

0	
1	
2	2266, 2010
3	
4	
5	5002, 5532
6	6746
7	7625, 7403
8	
9	

0	
1	
2	2010, 2266
3	
4	
5	5002, 5531
6	6746
7	7625, 7403
8	
9	

Radix

4 phases of scatter from list to buckets based on each digit place in the entries

0	2010
1	
2	5002, 5532
3	7403
4	
5	7625
6	6746, 2266
7	
8	
9	

0	5002, 7403
1	2010
2	7625
3	5532
4	6746
5	
6	2266
7	
8	
9	

0	5002, 2010
1	
2	2266
3	
4	7403
5	5532
6	7625
7	6746
8	
9	

0	
1	
2	2010, 2266
3	
4	
5	5002, 5532
6	6746
7	7403, 7625
8	
9	

Announcements

- EX 6 due Monday
- EX 7 releases Monday (last exercise)
- Final exam next Friday!
    - Practice final getting posted to website

Sorting Summary

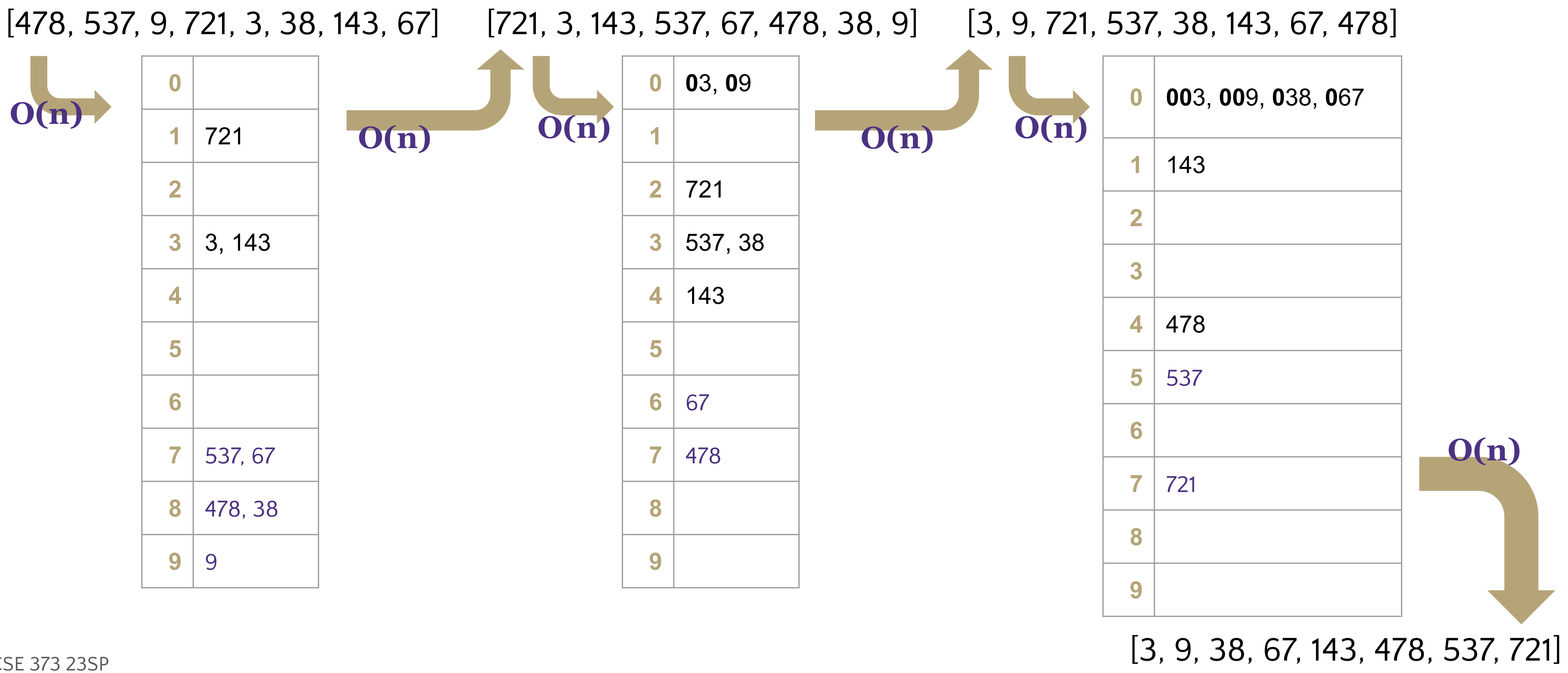
Radix Sort

- Radix = "the base of a number system"
  - We will use "10" as we are comfortable with 10 based systems
  - Could use any value, such as 128 for ASCII strings
- Idea
  - Bucket sort on one digit at a time
    - Only works on sequences of countable data: ints, doubles, stings
  - Number of buckets = radix
  - Start with least significant digit, do one pass of bucket sort per digit
- Fun fact: invented in 1890 as part of US census

Input:	[170, 45, 75, 90, 802, 24, 2, 66]
ones:	[170, 90, 802, 2, 24, 45, 75, 66]
tens:	[802, 2, 24, 45, 66, 170, 75, 90]
hundreds:	[2, 24, 45, 66, 75, 90, 170, 802]

Example Walk Through Video

Radix Sort



[478, 537, 9, 721, 3, 38, 143, 67]

0	
1	721
2	
3	3, 143
4	
5	
6	
7	537, 67
8	478, 38
9	9

[721, 3, 143, 537, 67, 478, 38, 9]

0	**0**3, **0**9
1	
2	721
3	537, 38
4	143
5	
6	67
7	478
8	
9	

[3, 9, 721, 537, 38, 143, 67, 478]

0	**00**3, **00**9, **0**38, **0**67
1	143
2	
3	
4	478
5	537
6	
7	721
8	
9	

[3, 9, 38, 67, 143, 478, 537, 721]

Sorting: Summary

	Best-Case	Worst-Case	Space	Stable
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	O(n)	$O(n^2)$	O(1)	Yes
Heap Sort	O(nlogn)	O(nlogn)	O(n)	No
In-Place Heap Sort	O(nlogn)	O(nlogn)	O(1)	No
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn) O(n)* optimized	Yes
Quick Sort	O(nlogn)	$O(n^2)$	O(n)	No
In-place Quick Sort	O(nlogn)	$O(n^2)$	O(1)	No
Bucket Sort	O(n)	$O(n^2)$	O(K+n)	Yes
Radix	O(n)	O(n)	O(n)	Yes

What does Java do?

- Actually uses a combination of *3 different sorts*:
  - If objects: use Merge Sort* (stable!)
  - If primitives: use Dual Pivot Quick Sort
  - If "reasonably short" array of primitives: use Insertion Sort
    - Researchers say 48 elements

Key Takeaway: No single sorting algorithm is "the best"!

- Different sorts have different properties in different situations
- The "best sort" is one that is well-suited to your data

* They actually use Tim Sort, which is very similar to Merge Sort in theory, but has some minor details different

What Else is There?

Can we do better than n log n?

- For comparison sorts, **NO**. A proven lower bound!
    - Intuition: n elements to sort, no faster way to find “right place” than log n
- However, niche sorts can do better in specific situations!

Many cool niche sorts beyond the scope of 373!

Counting Sort (Wikipedia)

External Sorting Algorithms (Wikipedia) – For big data™

Intro to Dynamic Programming

Fibonacci Problem

Staircase Problem

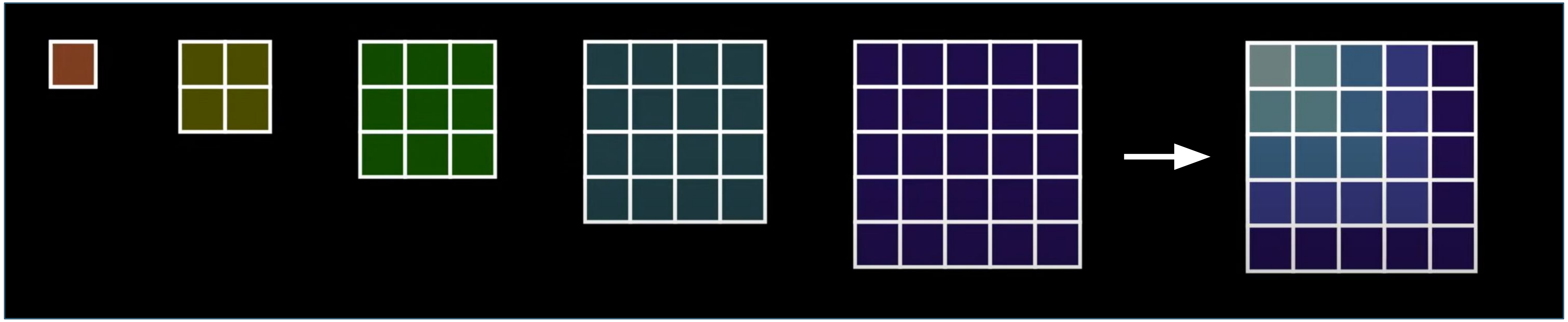
Box Problem

Knapsack Problem

What is Dynamic Programming

An algorithmic technique of optimizing a given algorithm by:

- Identifying the final solution as a summation of solutions to smaller sub problems
  - Building off of “divide and conquer”
- Intelligently ordering our solutions to the sub-problems to build up to the final solution



Dynamic Programming Techniques

1. Design “brute force” recursive solution
2. Take a recursive algorithm and find the overlapping subproblems, then cache the results for future recursive calls. (Memoize)
   a. Store subproblems of the main problem so we don’t have to re-compute them when we need them later on in solving the main problem
3. Bottom up approach

A little confusing? Don’t worry, you are not alone!

Intro to Dynamic Programming

Fibonacci Problem

Staircase Problem

Box Problem

Knapsack Problem

Question

Given a number `n`, print `n`-th Fibonacci Number.

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

How can we solve this problem in the most optimized way?

Clarify

1. Can `n` be a non-positive number?
   a. Depends, `n` can be 0, but not negative.
2. Can we use additional data structures?
   a. Yes, assume we want the fastest overall runtime.
3. What should be the result when `n = 0`?
   a. The result should be 0, before the first 1 in the sequence 1, 1, 2, ...,Fib(`n`)

Example

Edge Case 1:

```
n = 0; Fib = None
Output = 0
```

Edge Case 2:

```
n = 1; Fib = 1
Output = 1
```

Middle Case 1:

```
n = 2; Fib = 1, 1
Output = 1
```

Middle Case 2:

```
n = 9; Fib = 1, 1, 2, 3, 5, 8,
                 13, 21, 34
Output = 34
```

Approach

Brute Force:

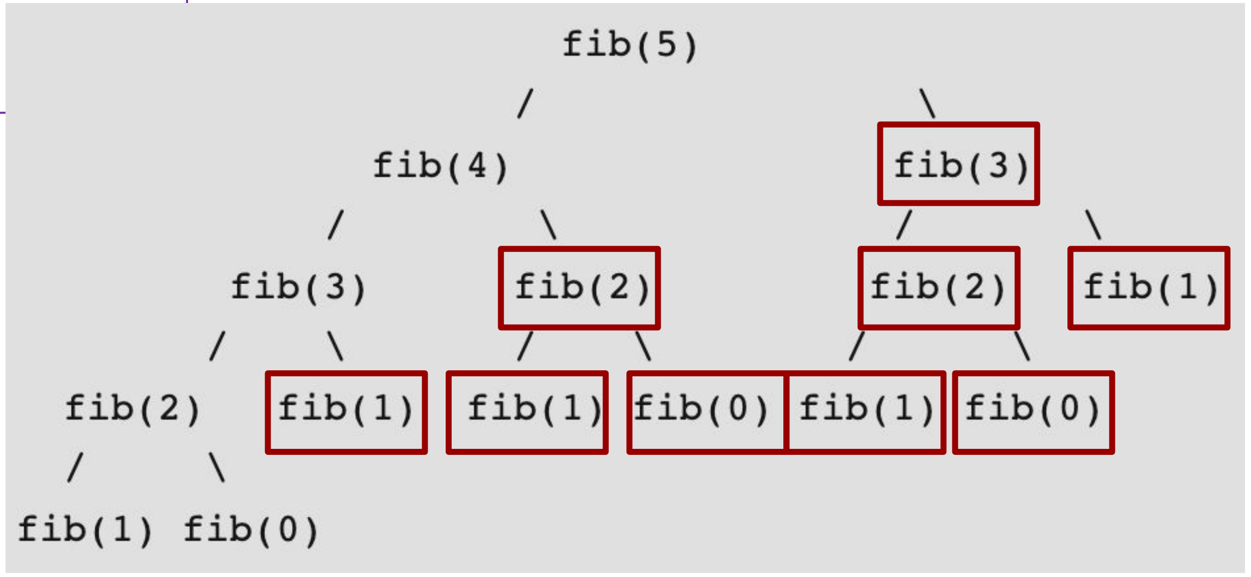
1. Use a recursive function to solve
2. Starting with `n` and descending down, recursively return the addition of the last and second to last numbers of our sequence
3. End our recursion when we hit our base case `n = 1`

Has $O(2^n)$ runtime with $O(n)$ space complexity…

There is a faster way using Dynamic Programming…

Recursive Solution

```
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```



Memoized Dynamic Programming Solution

```
public int fib(int n, int[] memo) {
   if (memo[n] != null) {
      return memo[n];
   } else if (n == 0 || n == 1) {
      return 1;
   } else {
      int result = fib(n-1) + fib(n-2);
      memo[n] = result;
      return result;
   }
}
```

* because of pass by reference for Arrays there is only ever one array and we are simply returning the reference to the updated Array, not remaking it with each recursive call

memo[] =

0	1	2	3	4	5
0	1	1	2	3	5

2n+1 recursive calls gives us $O(n)$ instead of $O(2^n)$

Detailed walkthrough of this solution

Bottom Up Dynamic Programming Solution

```
public int fib(int n) {
    int f[] = new int[n+1];
    f[1] = 1;
    f[2] = 1;
    for (int i = 3; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

Detailed walkthrough of this solution

Optimize

Optimized: Use Dynamic Programming to pre-compute Fib sequence up until `n` and return. Runs in O(N) runtime.

Optimized no additional data structure: We compute the value of our current term with a fixed number of elements. O(1)

Note: When you are computing a value in a sequence in an interview, think about using DP if applicable.

Implement

Create 3 variables to hold our second last value, our last value, and our current value with respect to our current term in the sequence when iterating.

Iterate in a for-loop until we hit term `n` and add our last and second last values and set them equal to our current value.

When we exit the for-loop, we will have computed the Fibonacci value at `n`.

Implement – Java Code

```
static int fib(int n) {
    int a = 0, b = 1, c;
    if (n == 0)
        return a;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Test

Test with Middle Case 2:

```
n = 9
Fib = 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Resulting variable values after for loop ends:

```
a = 21, b = 34, c = 34
Return b = 34
```

Question

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

How can we solve this problem in the most optimized way?

Clarify

1. Can `n` be a non-positive number?
   a. No, `n` must be equal to or greater than 1.
2. Can we use additional data structures?
   a. Yes, assume we want the fastest overall runtime.
3. Can we climb the same number of steps in a row?
   a. Yes, you can climb in any combination of 1 or 2 steps.

Example

Edge Case 1:

```
n = 1;
1) Take one step forward.
Output = 1
```

Edge Case 2:

```
n = 2;
1) 1 + 1 step
2) 2 steps
Output = 2
```

Middle Case:

```
n = 3;
1) 1 + 1 + 1 steps
2) 1 + 2 steps
3) 2 + 1 steps
Output = 3
```

Approach

Brute Force:

1. Use a recursive function to solve
2. Starting with `n` and descending down, recursively return the sum of the combinations it took to get to the last and second last steps from our current step
3. End our recursion when we hit our base cases `n = 1, n = 0`

Has $O(2^n)$ runtime with $O(n)$ space complexity…unless…dynamic programming…

Optimize

Optimized: Use Dynamic Programming to pre-compute the combinations of steps it takes to get to `n` and return. Runs in O(N) runtime.

Optimized no additional data structure: We compute the value of our current step with a fixed number of elements (our last and second last step it took to get to our current step). O(1)

Note: We are computing a value in a sequence, just like the Fibonacci problem…

Implement

Create 3 variables to hold the number of combinations it took to get to our second last step, our last step, and our current step with respect to our current step in the sequence when iterating.

Iterate in a for-loop until we hit term `n` and add our last and second last steps' combinations and set them equal to the number of combinations to get to our current step.

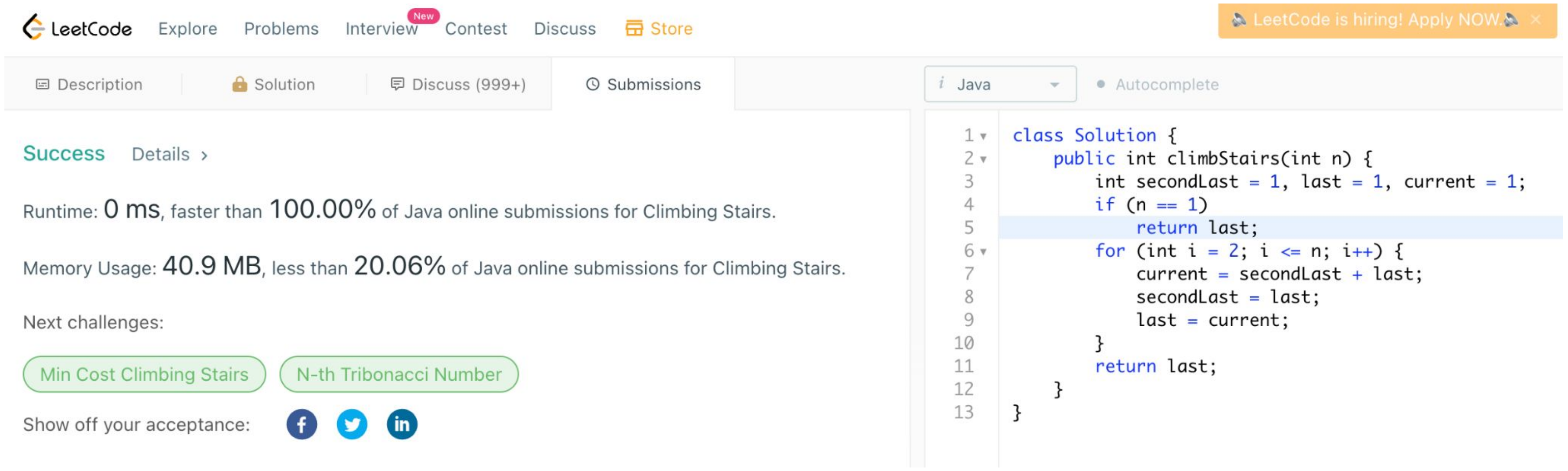
When we exit the for-loop, we will have the number of combinations to get to step `n`.

Implement – Java Code

```
public int climbStairs(int n) {
    int secondLast = 1, last = 1, current = 1;
    if (n == 1)
        return last;
    for (int i = 2; i <= n; i++) {
        current = secondLast + last;
        secondLast = last;
        last = current;
    }
    return last;
}
```

Fun Fact

The previous solution runs faster than 100% of leetcode solutions for the problem…which is the first time I have ever gotten a 100%…if you haven't been convinced of the power of DP yet…you should be now…



Success Details ›

Runtime: 0 ms, faster than 100.00% of Java online submissions for Climbing Stairs.

Memory Usage: 40.9 MB, less than 20.06% of Java online submissions for Climbing Stairs.

Next challenges:

Min Cost Climbing Stairs | N-th Tribonacci Number

Show off your acceptance:

```java
class Solution {
    public int climbStairs(int n) {
        int secondLast = 1, last = 1, current = 1;
        if (n == 1)
            return last;
        for (int i = 2; i <= n; i++) {
            current = secondLast + last;
            secondLast = last;
            last = current;
        }
        return last;
    }
}
```

Test

Test with Middle Case:

```
n = 3
3 different ways to get to step 3
```

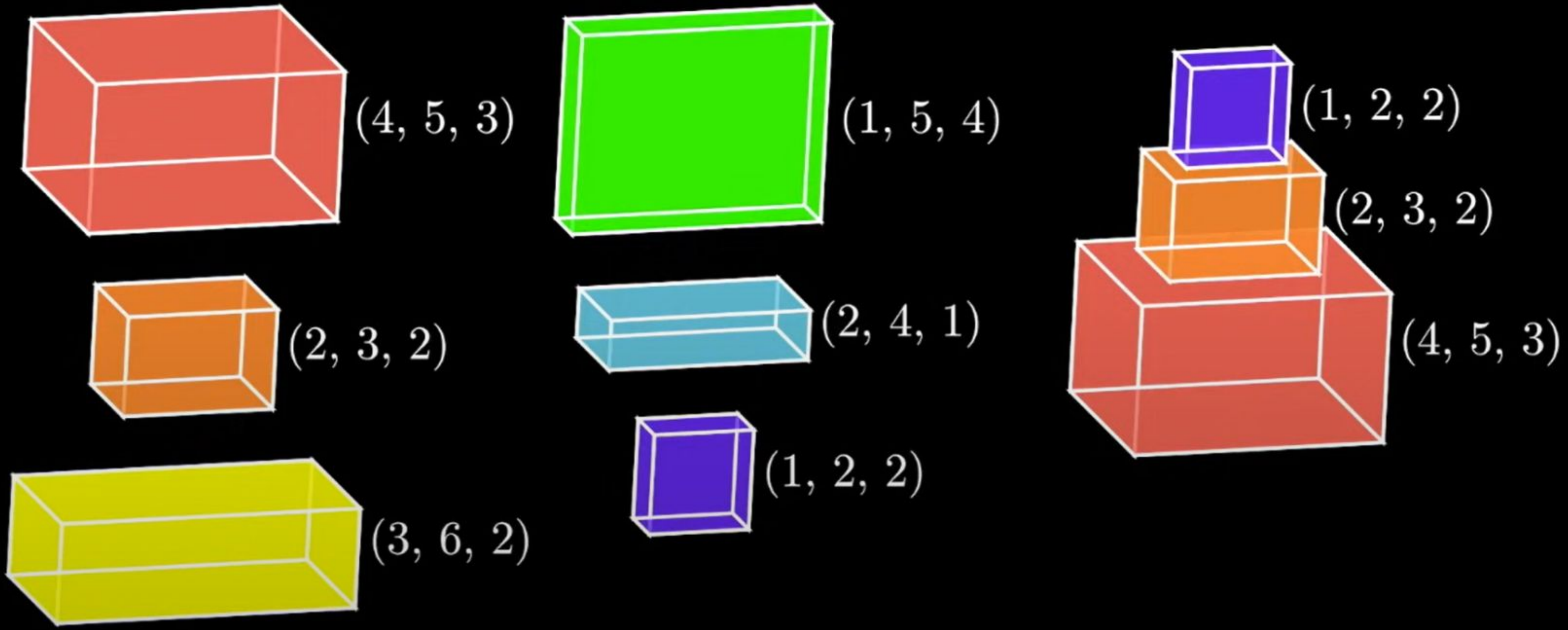
Resulting variable values after for loop ends:

```
secondLast = 2, last = 3, current = 3
```

Return **`last = 3`**

Given n boxes $[(L_1, W_1, H_1), (L_2, W_2, H_2), \ldots, (L_n, W_n, H_n)]$ where box i has length L_i, width W_i, and height H_i, find the height of the tallest possible stack. Box (L_i, W_i, H_i) can be on top of box (L_j, W_j, H_j) if $L_i < L_j, W_i < W_j$.

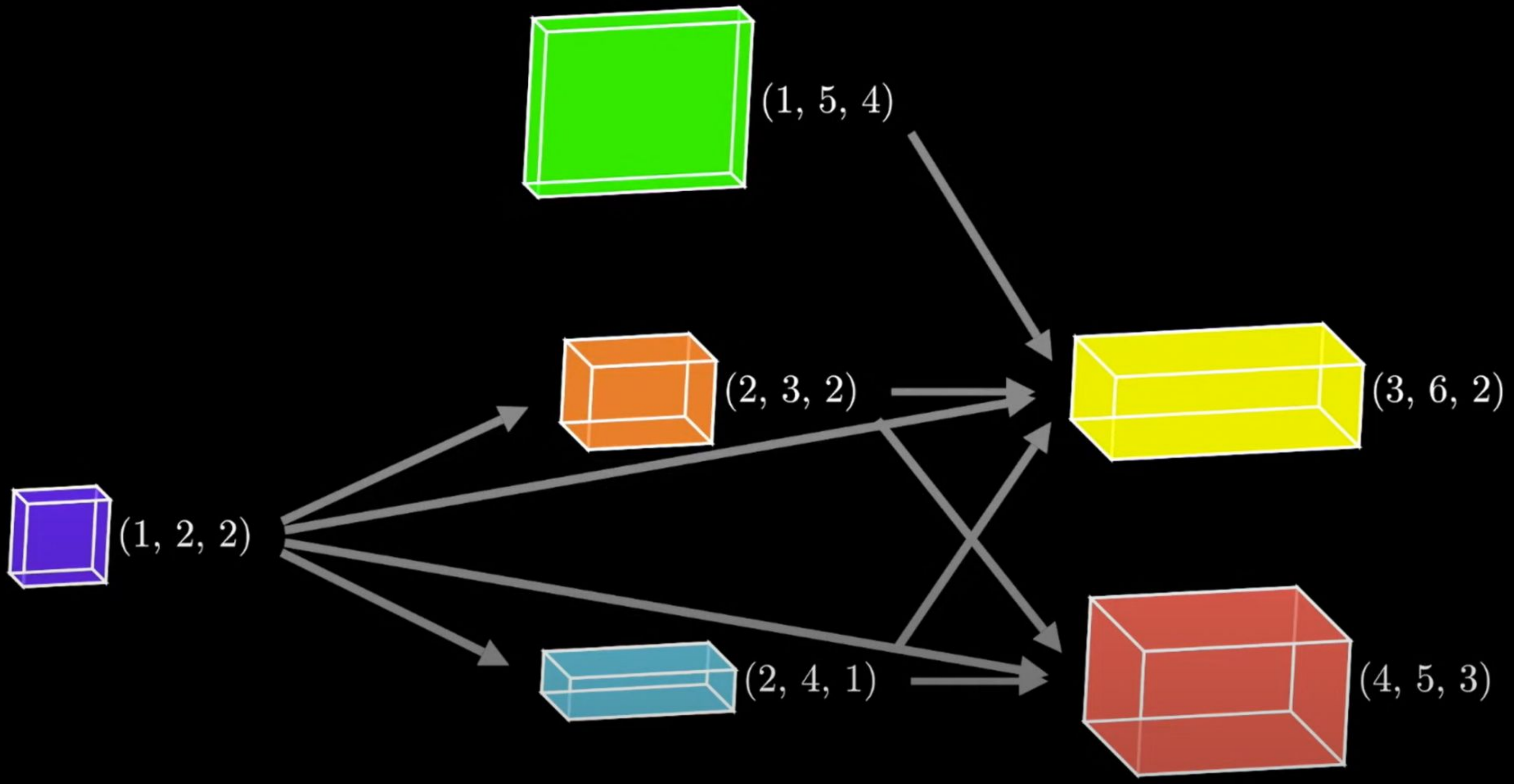
$$[(4,5,3),(2,3,2),(3,6,2),(1,5,4),(2,4,1),(1,2,2)]$$



height $= 7$

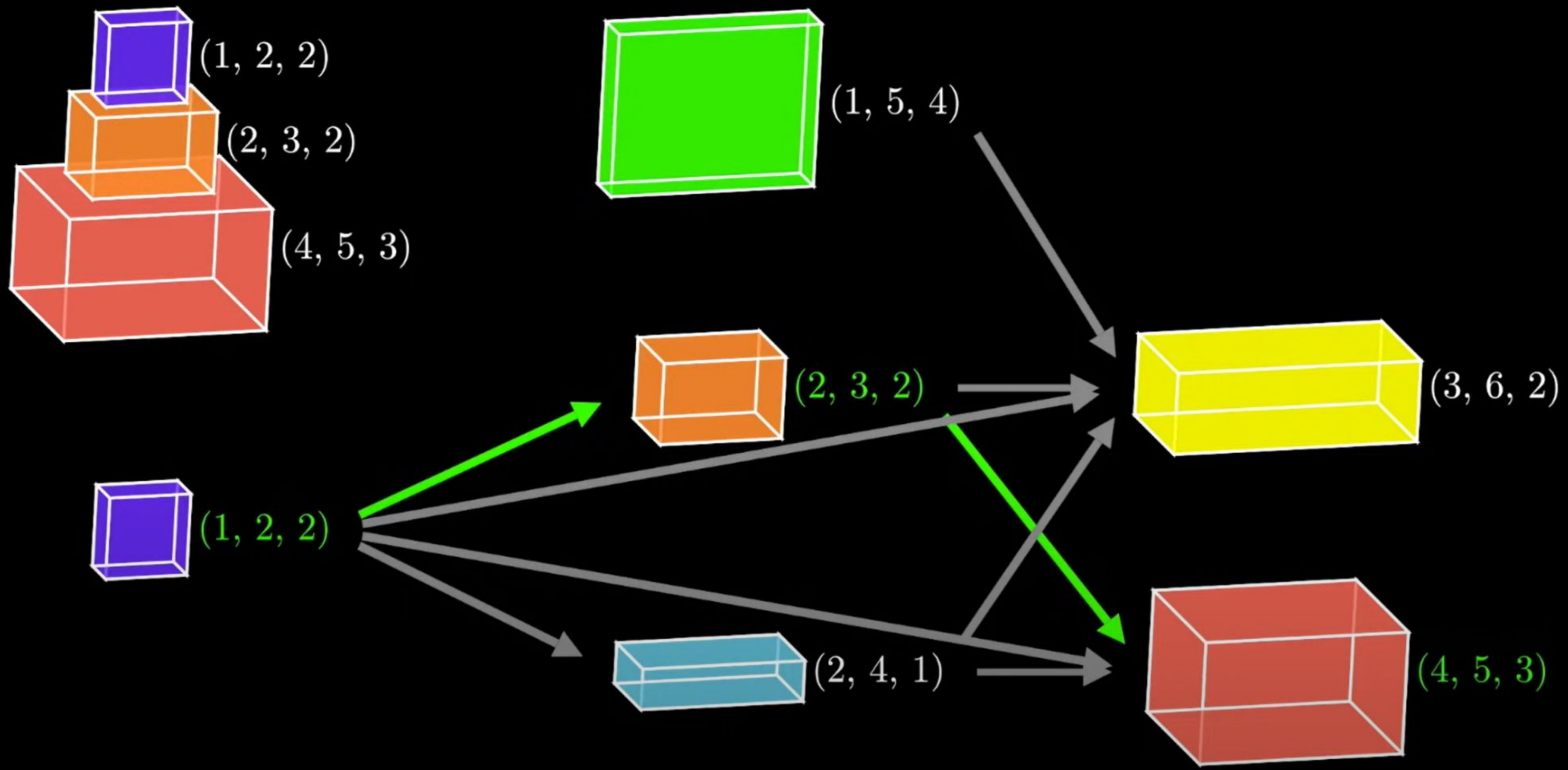
1. Use a DAG to visualize which box can stack on top of which

Box (L_i, W_i, H_i) can be on top of box (L_j, W_j, H_j) if $L_i < L_j, W_i < W_j$.



1. Visualize Examples

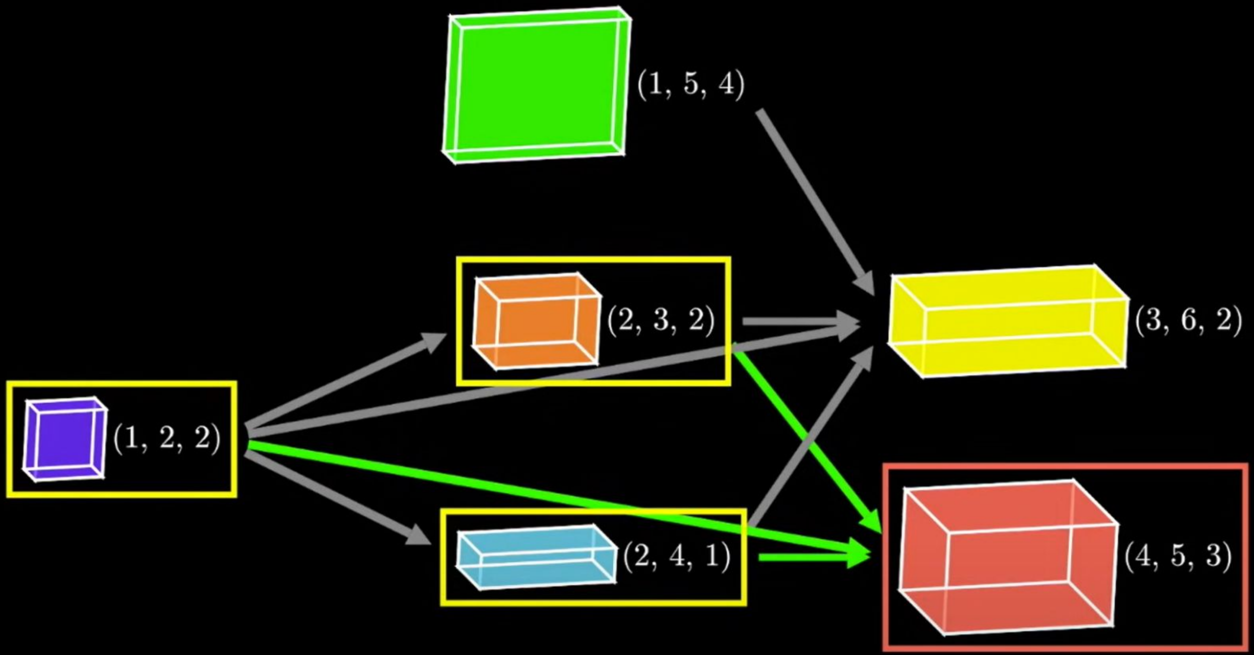
2. Identify sub problems: Paths in the DAG represent valid stacks



3. Find how subproblems build to larger solution

Subproblem: $\text{height}[(L_i, W_i, H_i)]$

Largest height of stack with box (L_i, W_i, H_i) at the bottom

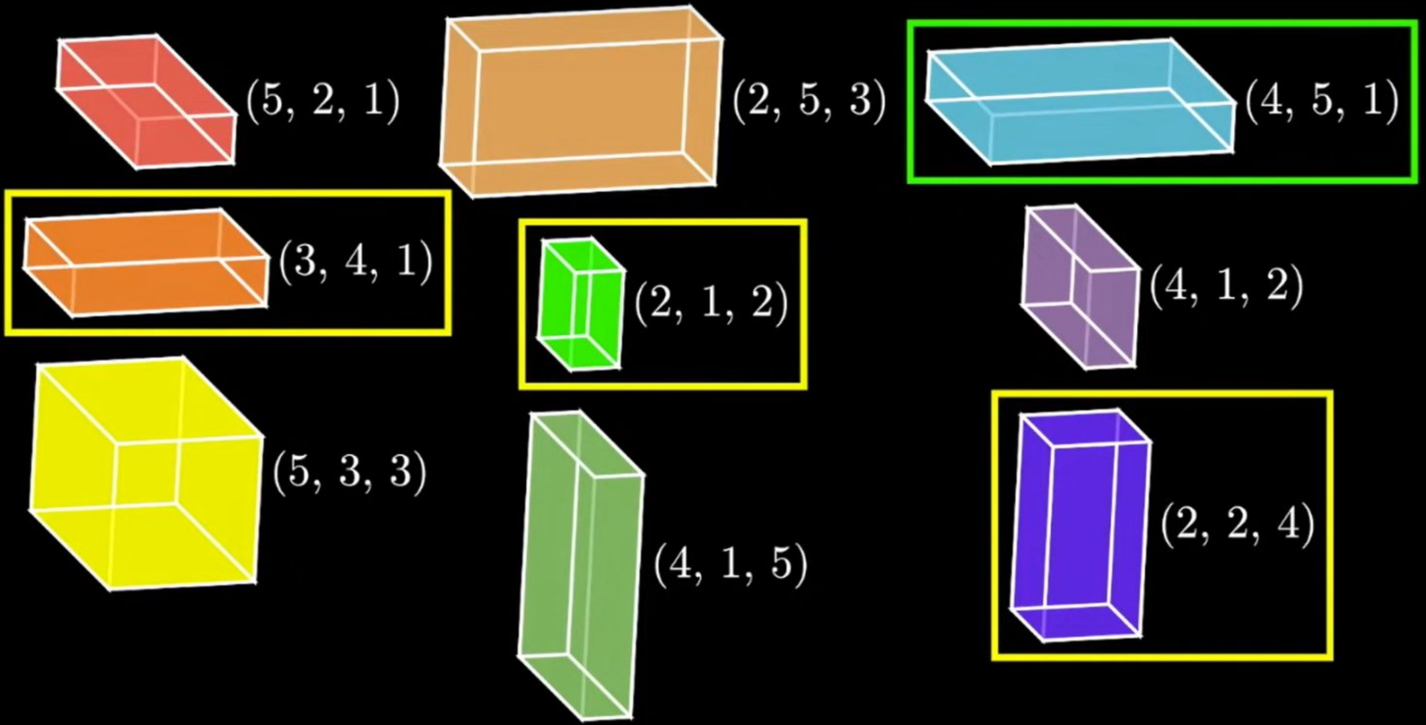


What subproblems are needed to solve $\text{height}[(4,5,3)]$?

$\text{height}[(2,3,2)] = 4 \quad \text{height}[(2,4,1)] = 3 \quad \text{height}[(1,2,2)] = 2$

3. Find relationships among subproblems

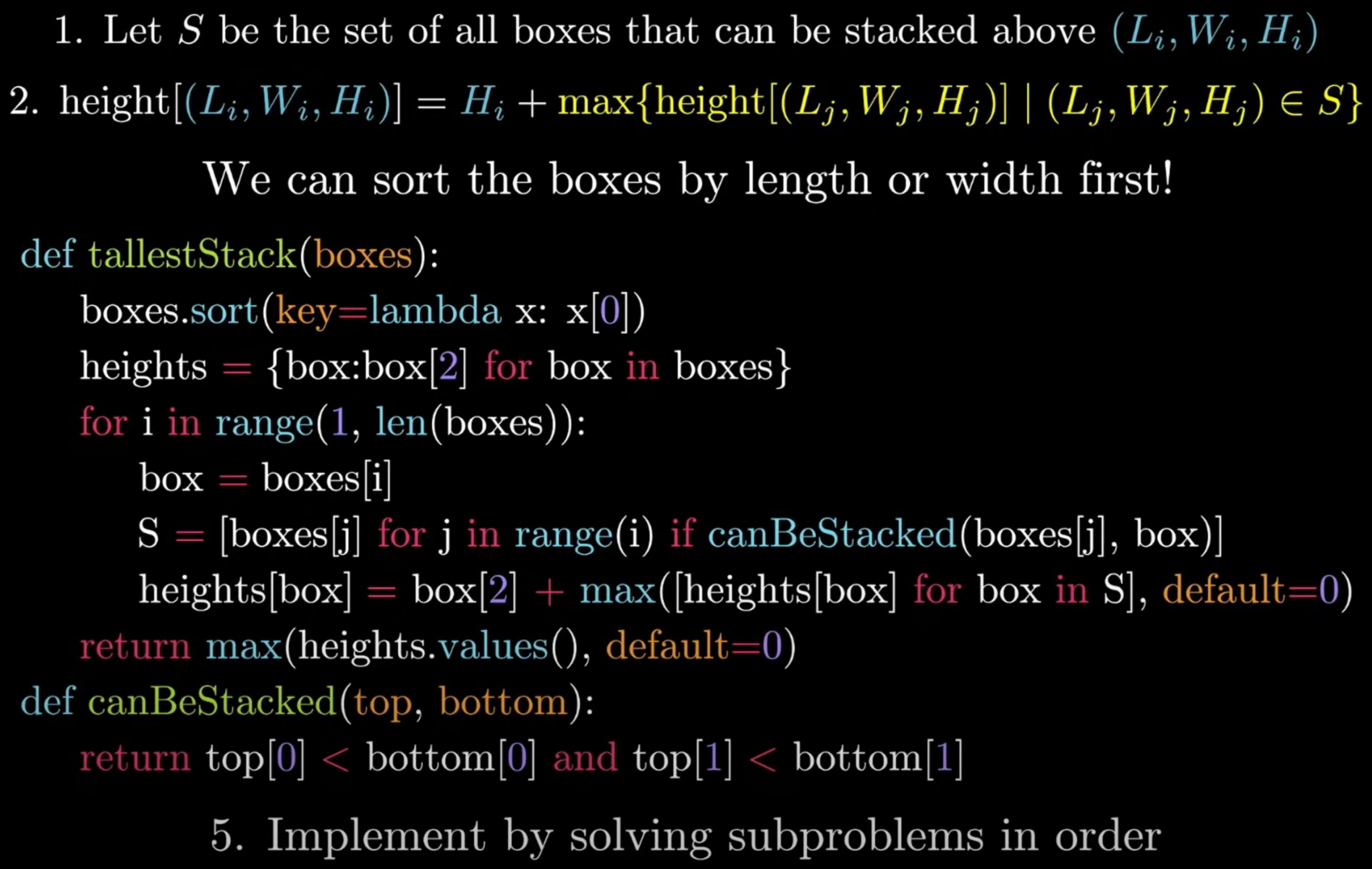
4. Generalize relationship between subproblems and final solution



How to solve height$[(L_i, W_i, H_i)]$ in general?

1. Let S be the set of all boxes that can be stacked above (L_i, W_i, H_i)
2. $\text{height}[(L_i, W_i, H_i)] = H_i + \max\{\text{height}[(L_j, W_j, H_j)] \mid (L_j, W_j, H_j) \in S\}$

5. Implement solving subproblems in correct order



1. Let S be the set of all boxes that can be stacked above (L_i, W_i, H_i)

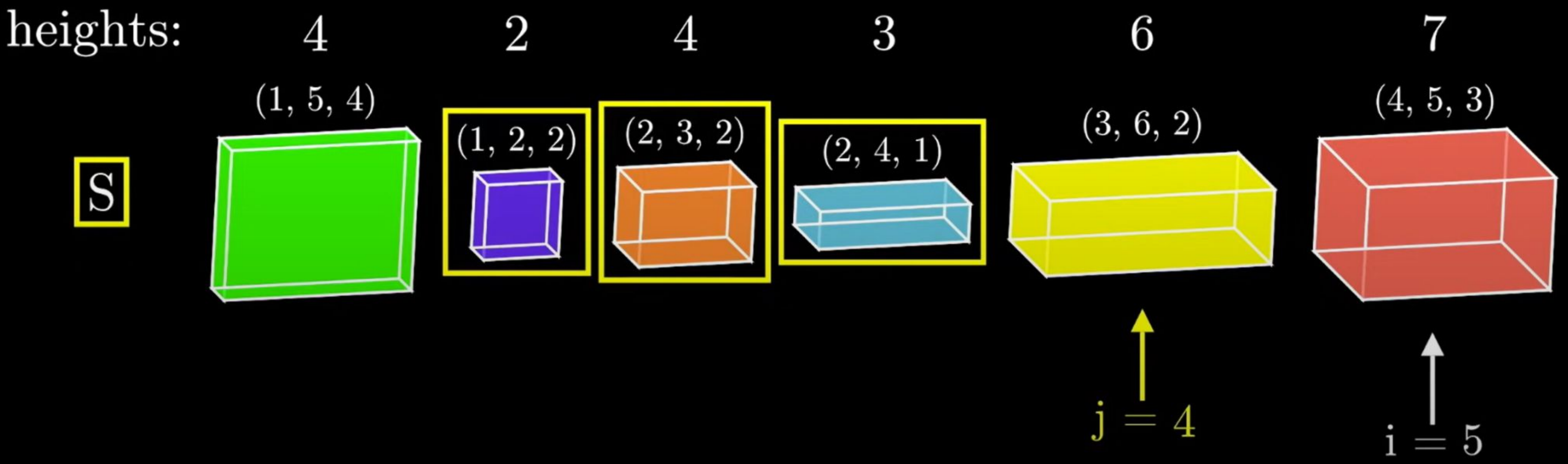
2. $\text{height}[(L_i, W_i, H_i)] = H_i + \max\{\text{height}[(L_j, W_j, H_j)] \mid (L_j, W_j, H_j) \in S\}$

We can sort the boxes by length or width first!

```
def tallestStack(boxes):
    boxes.sort(key=lambda x: x[0])
    heights = {box:box[2] for box in boxes}
    for i in range(1, len(boxes)):
        box = boxes[i]
        S = [boxes[j] for j in range(i) if canBeStacked(boxes[j], box)]
        heights[box] = box[2] + max([heights[box] for box in S], default=0)
    return max(heights.values(), default=0)
def canBeStacked(top, bottom):
    return top[0] < bottom[0] and top[1] < bottom[1]
```

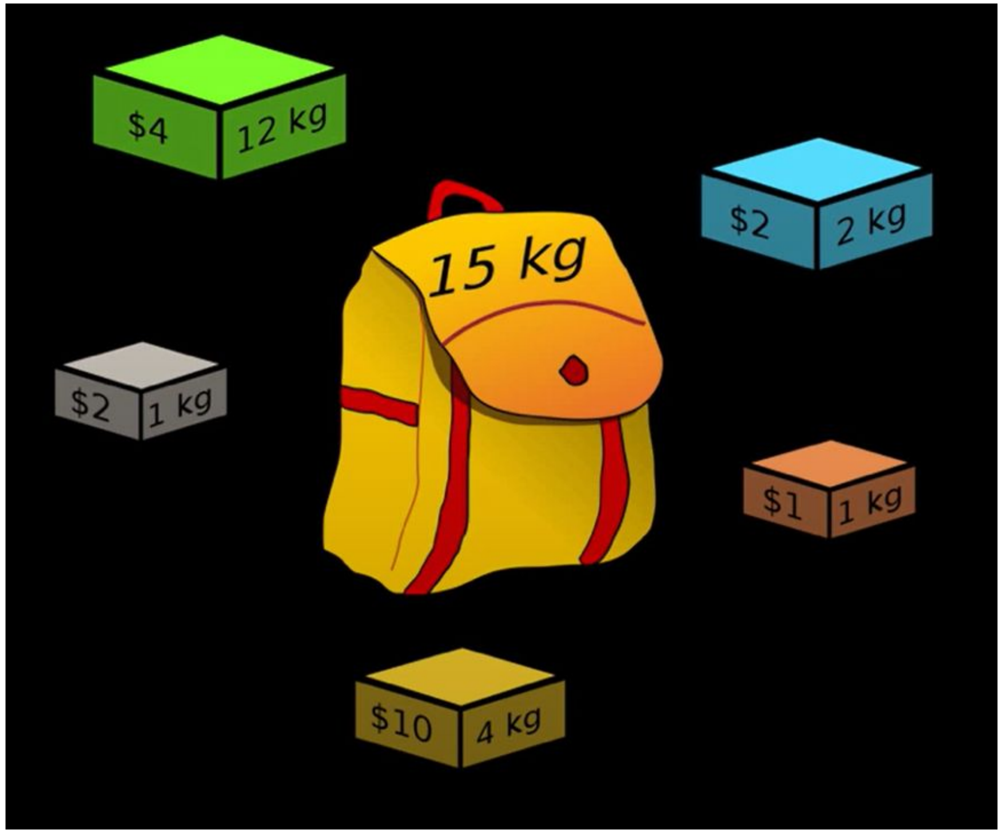
5. Implement by solving subproblems in order

```
def tallestStack(boxes):
    boxes.sort(key=lambda x: x[0])
    heights = {box:box[2] for box in boxes}
    for i in range(1, len(boxes)):
        box = boxes[i]
        S = [boxes[j] for j in range(i) if canBeStacked(boxes[j], box)]
        heights[box] = box[2] + max([heights[box] for box in S], default=0)
    return max(heights.values(), default=0)
def canBeStacked(top, bottom):
    return top[0] < bottom[0] and top[1] < bottom[1]
```



0/1 Knapsack Problem

Given a set of objects which have both a value and a weight (v_i, w_i) what is the maximum value we can obtain by selecting a subset of these objects such that the sum of the weights does not exceed the knapsacks given capacity?



Problem Walk Through Video

Brute Force Recursive Solution

```
public int knapsack(int n, int cap, int[] w, int[] v) {
   if (n == 0 || cap == 0) {
      result = 0;
   } else if (w[n] > cap) {
      result = knapsack(n-1, cap, w, v);
   } else {
      int temp1 = knapsack(n-1, cap, w, v);
      int temp2 = v[n] + knapsack(n-1, cap - w[n-1], w, v);
      result = Math.max(temp1, temp2);
      return result;
   }
}
```

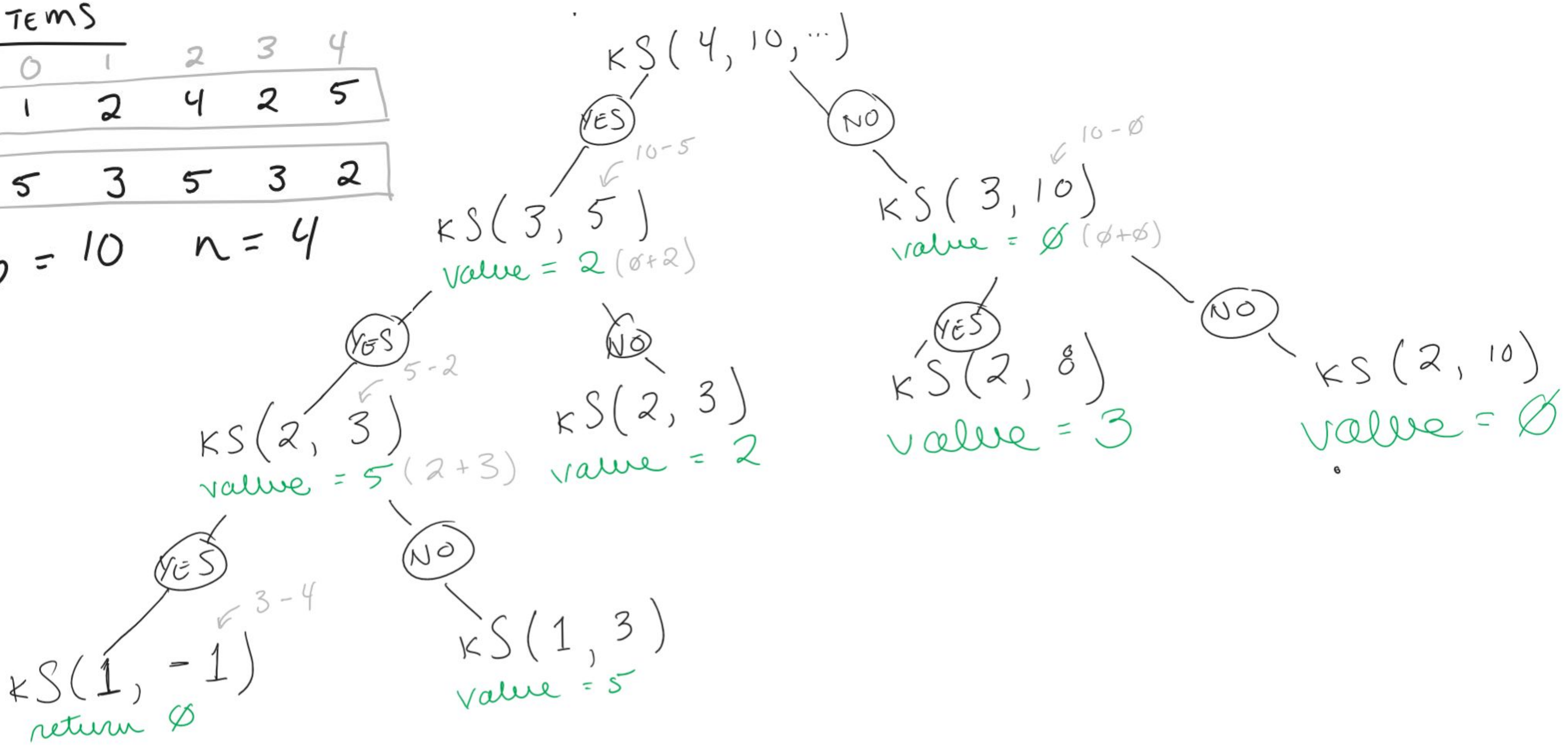
$O(2^n)$

Recursive Solution is Exponential

ITEMS

	0	1	2	3	4
w	1	2	4	2	5
v	5	3	5	3	2

cap = 10 n = 4



Memoized DP Solution

```
public int knapsack(int n, int cap, int[] w, int[] v, int[][] memo) {
   int result = memo[n][cap];
   if (memo[n][cap] == NULL) {
      if (n == 0 || cap == 0) {
         result = 0;
      } else if (w[n] > c) {
         result = knapsack(n-1, cap);
      } else {
         int temp1 = knapsack(n-1, cap);
         int temp2 = v[n] + knapsack(n-1, cap - w[n-1]);
         result = Math.max(temp1, temp2);
         memo[n][cap] = result;
      }
   }
   return result;
}
```

O(n)

More Examples of Dynamic Programming Problems

- Count the different ways to move through a 6x9 grid
- Given a set of coins, how can we make 27 cents using the smallest number of coins?
- Given a set of strings, what are the possible ways to construct the string "potentpot"
- Knapsack problem

Helpful walk through videos:

5 Simple Steps for Solving Dynamic Programming Problems

Questions?

That’s all!