Dynamic Programming

Chapter 15

What is Dynamic Programming

- Fancy talk
- A “method for efficiently solving problems that exhibit the characteristics of overlapping sub-problems and optimal substructure” (p. 203)
- **Optimal substructure**
  - A globally optimal solution can be found by combining optimal solutions to local sub-problems
  - Merge sort is globally optimal for sorting unordered lists
    - Split lists into increasingly smaller sublists
    - Merges increasingly larger sorted lists
- **Overlapping sub-problems**
  - Solving the same problem many times

Remember our friend Fibonacci Numbers?

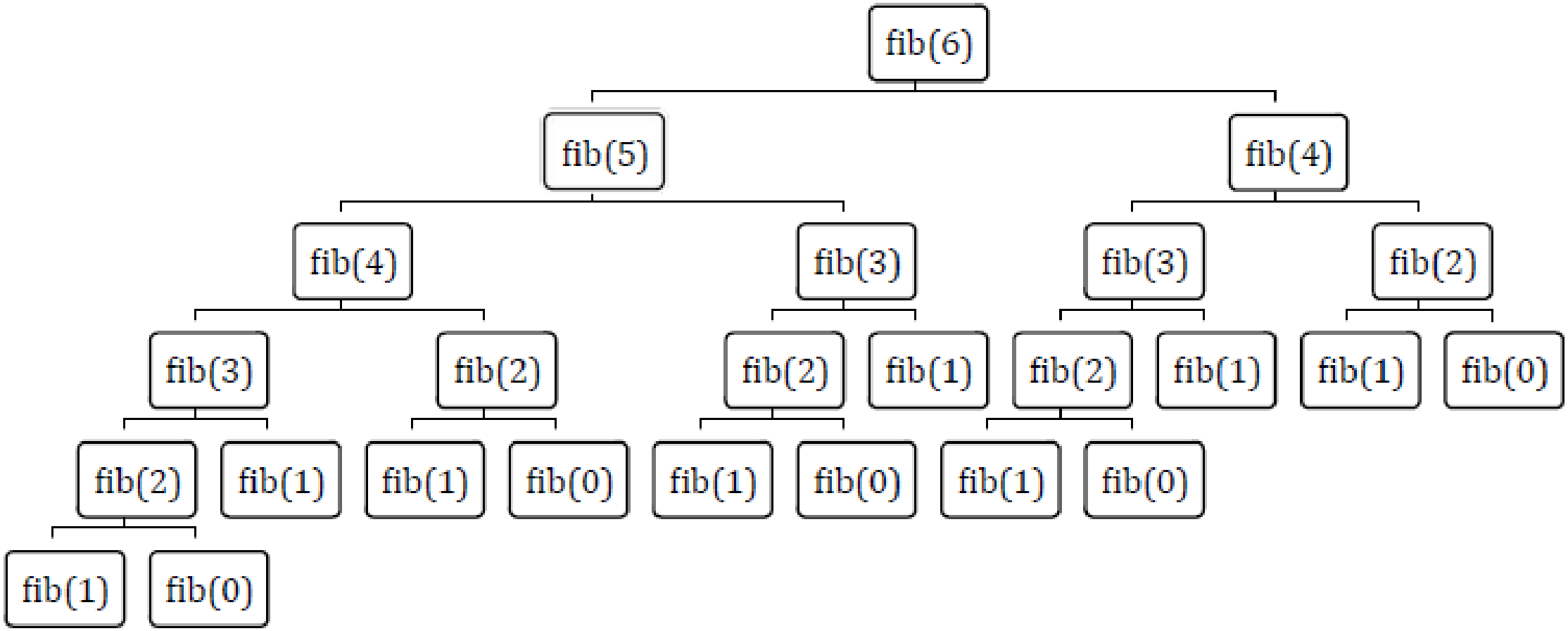
```
def fib(n):
    """Assumes n is an int >= 0
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- Lowly Conceptual Complexity
- High Computational Complexity
  - $O(fib(n))$

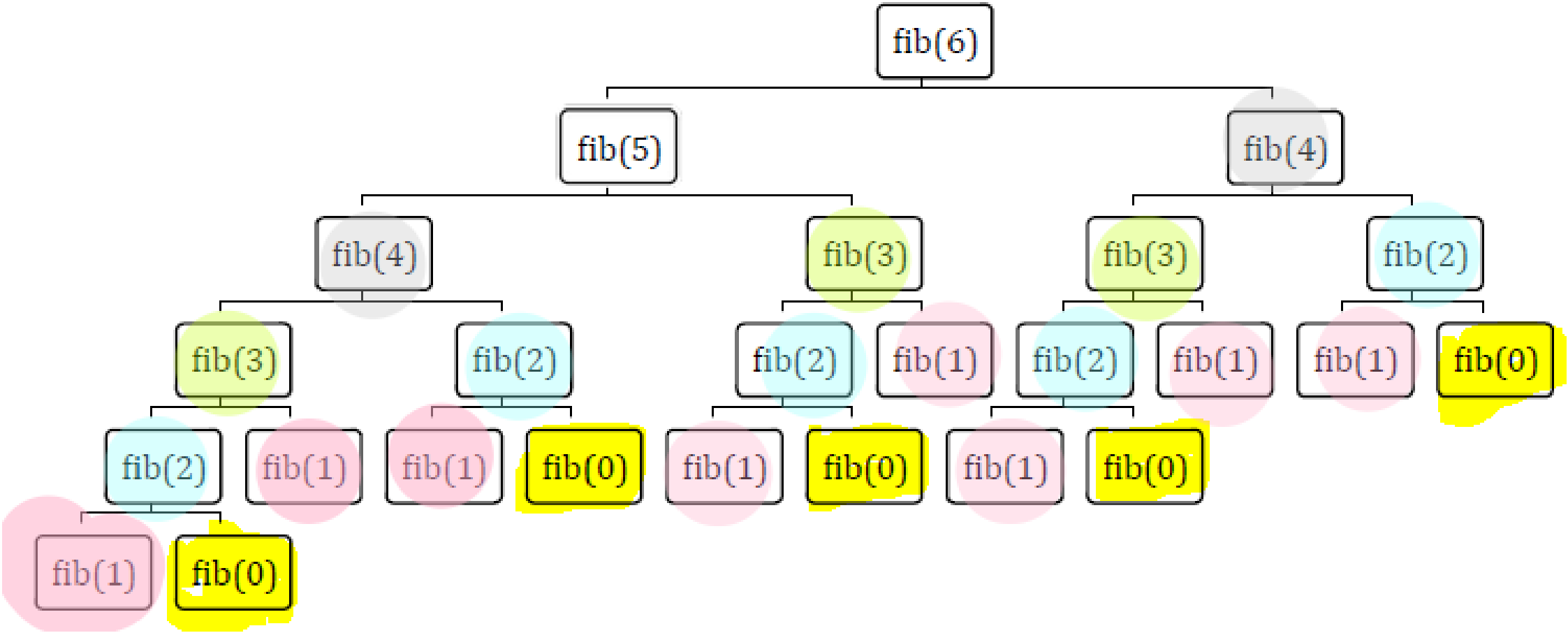
Fib(120)

- 8 Septillion
- 670 Sextillion
- 007 Quintillion
- 398 Quadrillion
- 507 Trillion
- 948 Billion
- 658 Million
- 051 Thousand
- 921

Fib(6) recursion visualized



Overlapping sub-problems



How many times?

- fib(6) X 1
- fib(5) X 1
- fib(4) X 2
- fib(3) X 3
- fib(2) X 5
- fib(1) X 8
- fib(0) X 5

- It is *almost* $\sum fib(n-i)$

Maybe we should **memorize**!

- Save the result of some key in a **memorization** table
- Check for that key
- If found – return the value
- Else compute the value for the key and add to the table

- Close to $O(n)$ for computational complexity
- Also $O(n)$ for memory

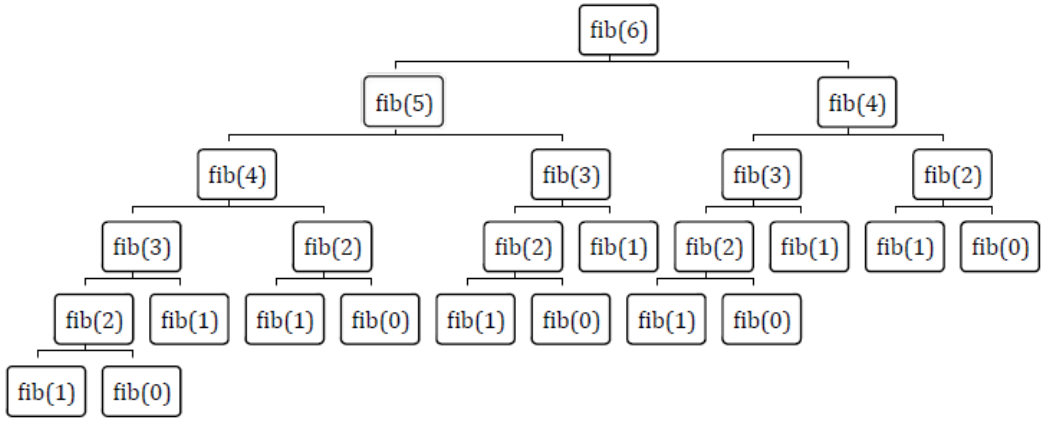
How about our knapsack problem?

- **Greedy** algorithm $O\left(n\ log(n)\right)$
  - Didn't guarantee an optimal solution
- **Exhaustive enumeration** (brute force) $O(2^n)$
  - Found the optimal solution
  - but who has that kind of time

- Can we find
  - Optimal substructure?
  - Overlapping sub problems

Rooted Binary Tree

- There is exactly one node with no parents. This is the **root**
- All other nodes have exactly one parent
- Each node has no more than two children
- A child node with no children is called a **leaf**



Decision Tree

- Each node represents a decision
  - In our case True or False
  - The left child represents the True choice
  - The right child represents the False choice
- Generate left-first, depth-first
  - For each node if decision is True we generation the left node
    - Repeat until false
    - Return to the parent
    - Generate the right node
  - Eventually all possible solutions are generated

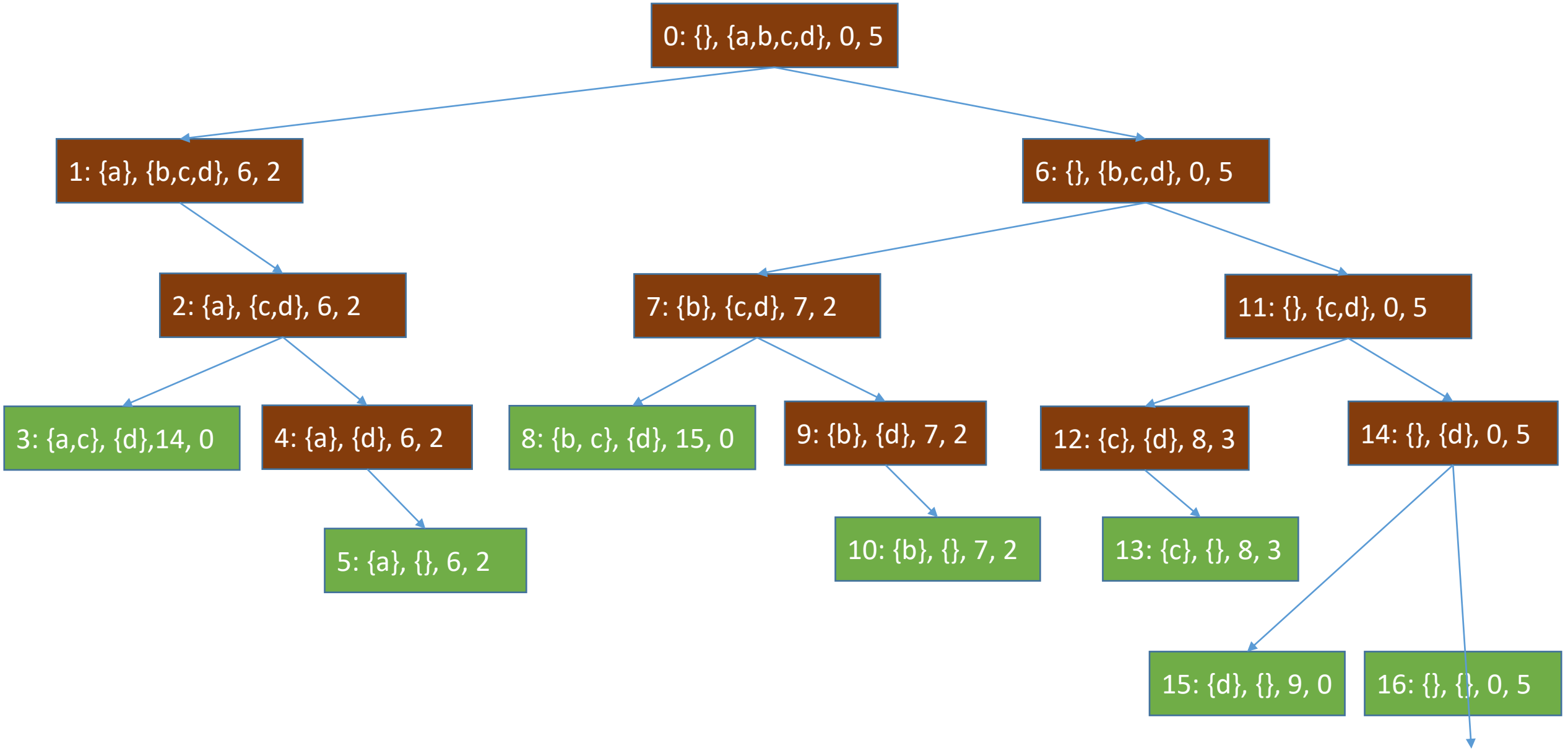
Building a decision tree for the knapsack problem

- What is the decision?
  - Given space available, can I take the next thing?
- What do I need to know?
  - Items I have already taken
  - What's left
  - How much room do I have left *
  - How much is it all worth *
- When are we done?
  - When I have no items left or knapsack is full

Stuff

- Consider a capacity of 5 (somethings?)

Name	Value	Weight
a	6	3
b	7	3
c	8	2
d	9	5


0: {}, {a,b,c,d}, 0, 5
1: {a}, {b,c,d}, 6, 2
6: {}, {b,c,d}, 0, 5
2: {a}, {c,d}, 6, 2
7: {b}, {c,d}, 7, 2
11: {}, {c,d}, 0, 5
3: {a,c}, {d},14, 0
4: {a}, {d}, 6, 2
8: {b, c}, {d}, 15, 0
9: {b}, {d}, 7, 2
12: {c}, {d}, 8, 3
14: {}, {d}, 0, 5
5: {a}, {}, 6, 2
10: {b}, {}, 7, 2
13: {c}, {}, 8, 3
15: {d}, {}, 9, 0
16: {}, {}, 0, 5

Codey, Code, Code

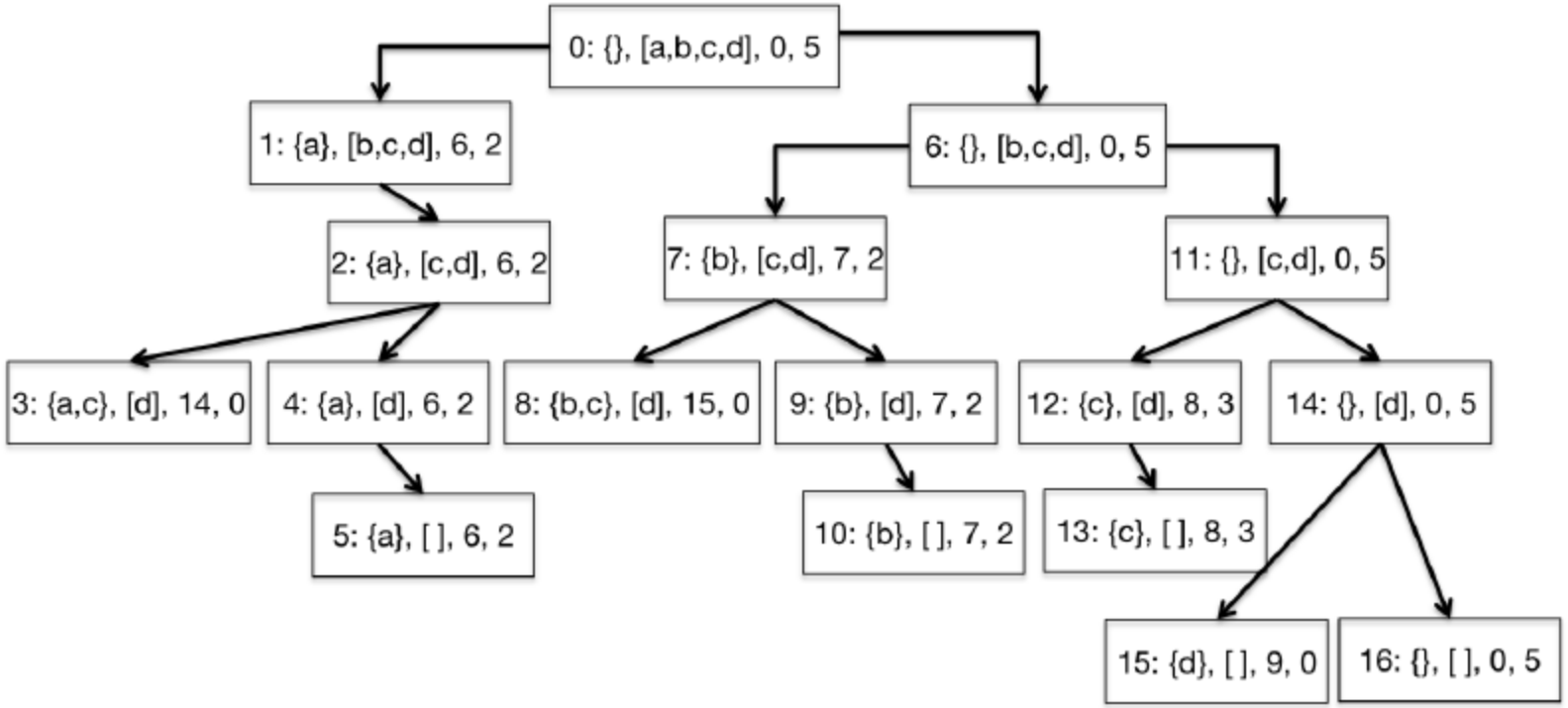
- Since we are building our tree depth-first that implies **recursion**

How deep (tall) is your tree?

- The tree can possibly have n + 1 levels (generation)
- Each generation can possibly have 2^{i-1} nodes
- Total possible nodes $\sum 2^i$
- Or $2^n - 1$possible nodes

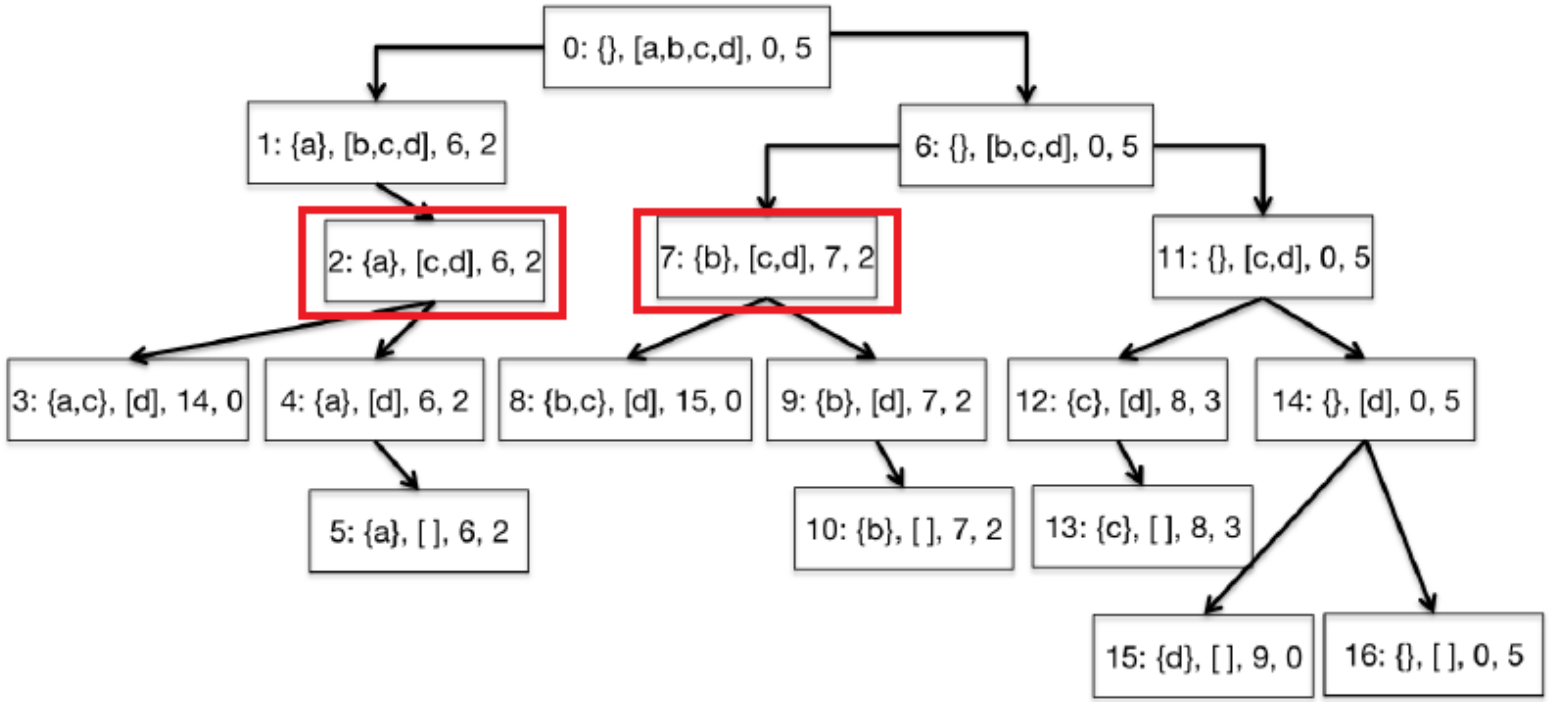
Can we optimize this silly thing?

- The **rooted binary tree** is the **optimal substructure**
- What about an **overlapping subproblem**?



Hint, hint

- If we think back to our implementation what were the key elements?
  - Items to Consider
    - Since we always work from the beginning of the list this can be a count
  - Available Weight



Dynamic Programming vs Divide-and-Conquer

- Both based on breaking a problem down into subproblems
- Divide-and-Conquer
  - Decompose into increasingly small subproblems
  - Not dependent, primarily, on problem structure
- Dynamic Programming
  - Subproblem is only slightly smaller than the original
  - Number of subproblems to solve should be significantly fewer than total number of subproblems