CS 1501

Greedy Algorithms and Dynamic Programming

Change Making

What is the minimum number of coins needed to make up a given value k?

This is a *greedy algorithm*

- At each step, the algorithm makes the choice that seems to be best at the moment
- Have we seen greedy algorithms already this term?
    - Yes!
        - Building Huffman trees
        - Prim's algorithm
        - Kruskal's algorithm

… But wait …

- Does our change making algorithm solve the change making problem?
  - For US currency…
  - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?

So what changed about the problem?

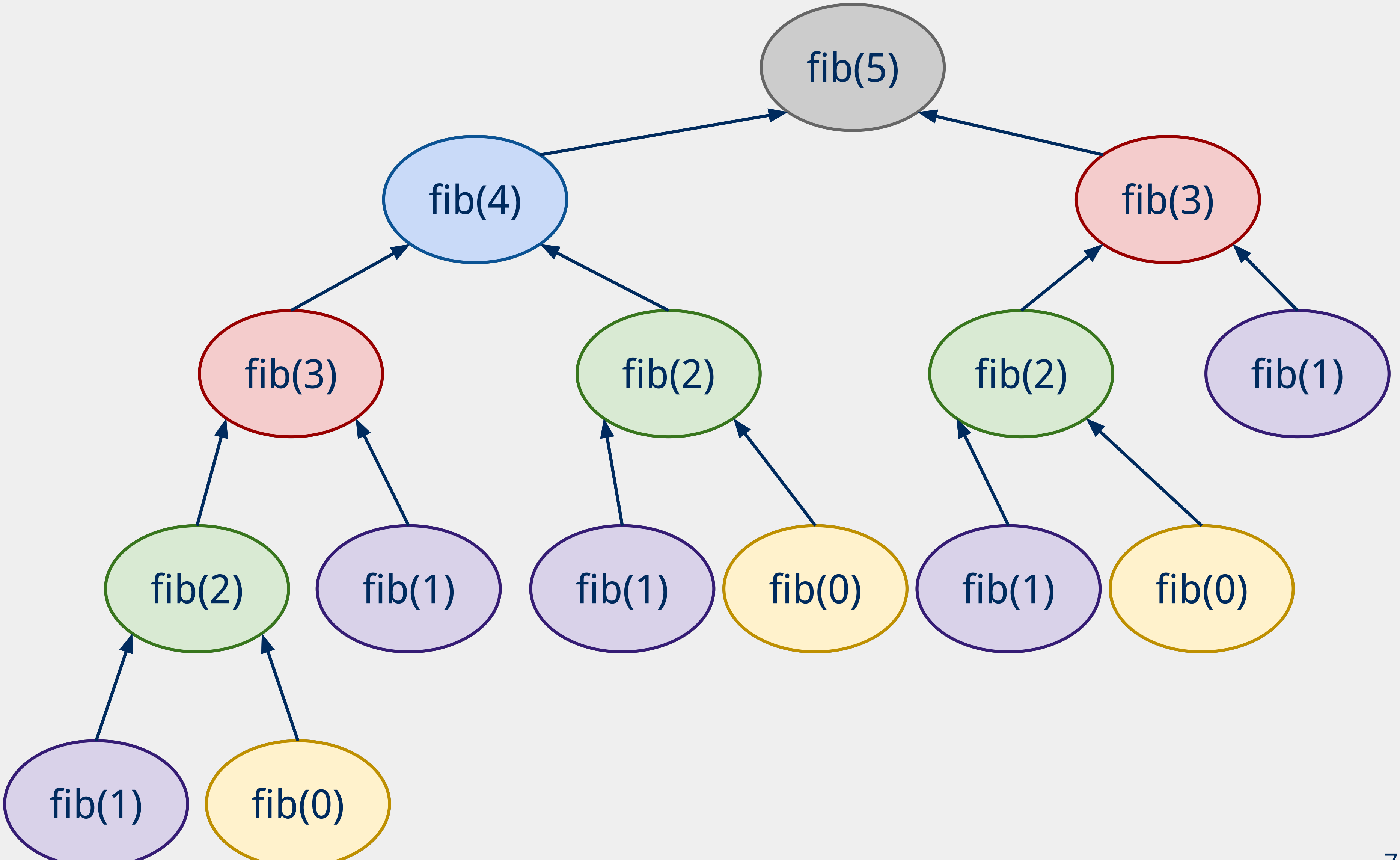
- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices

Finding all subproblems solutions can be inefficient

- Consider computing the Fibonacci sequence:
  - ```
    def fib(x):
        if x == 0:
            return 0
        elif x == 1:
            return 1
        else:
            return fib(x - 1) + fib(x - 2)
    ```

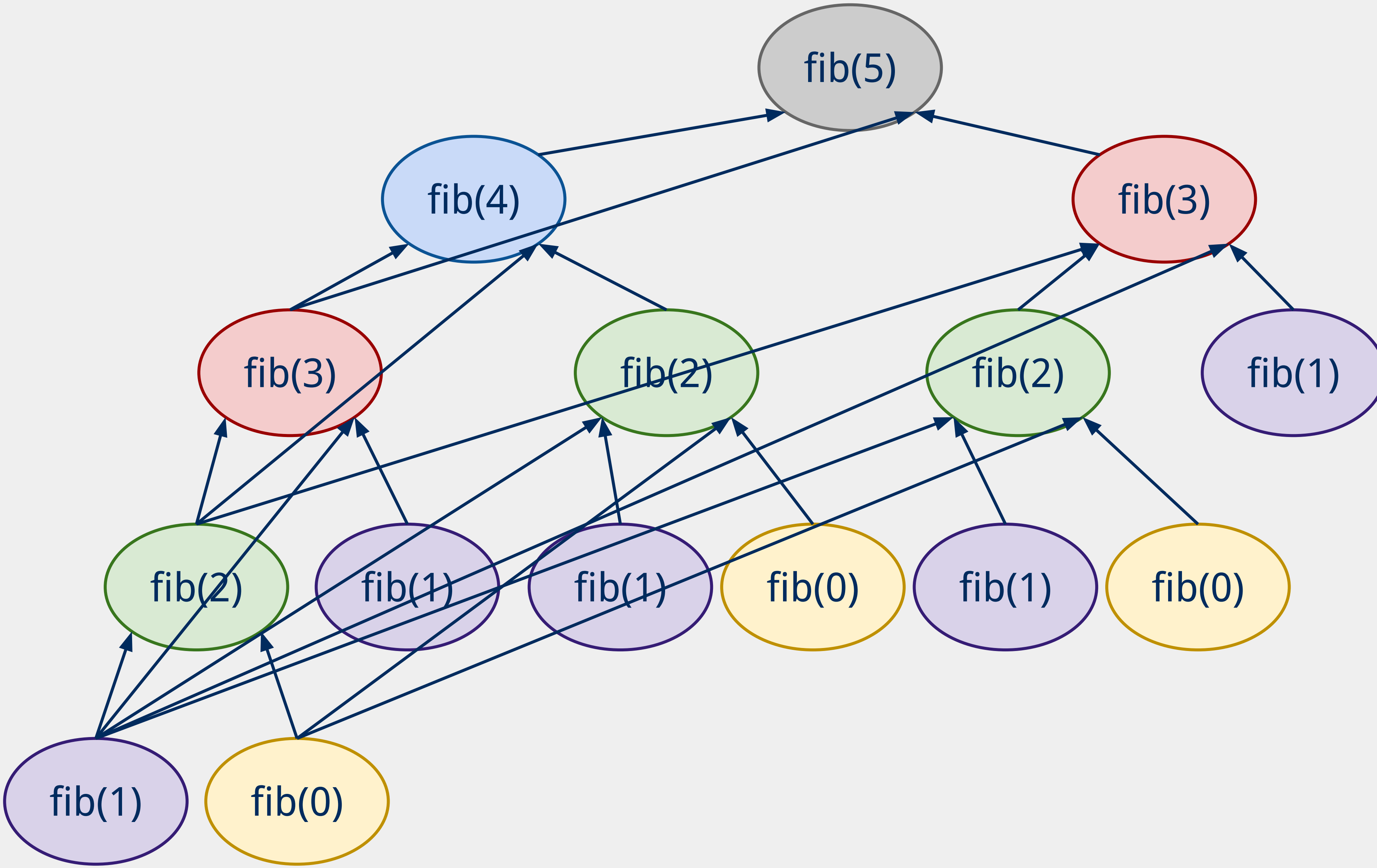
- What does the call tree for `x = 5` look like?
    ```

fib(5)



fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

How do we improve?



Memoization

```
F = [-1 for i in range(x + 1)]
F[0] = 0
F[1] = 1


def memo_fib(y):
    if F[y] == -1:
        F[y] = memo_fib(y-1) + memo_fib(y-2)
    return F[y]

memo_fib(x)
```

Note that we can also do this bottom-up

```
def dp_fib(x):
    F = [-1 for i in range(x + 1)]
    F[0] = 0
    F[1] = 1
    for i in range(2, x + 1):
        F[i] = F[i-1] + F[i-2]
    return F[x]
```

Can we improve this bottom-up approach?

```
def final_fib(x):
    prev2 = 0
    prev1 = 1
    for i in range(2, x + 1):
        new = prev1 + prev2
        prev2 = prev1
        prev1 = new
    return prev1
```

Where can we apply dynamic programming?

- To problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times

Our dynamic programming approach

1. Build a recursive solution
2. Use the recursive approach to design a memoization data structure
3. Populate the memoization data structure bottom up

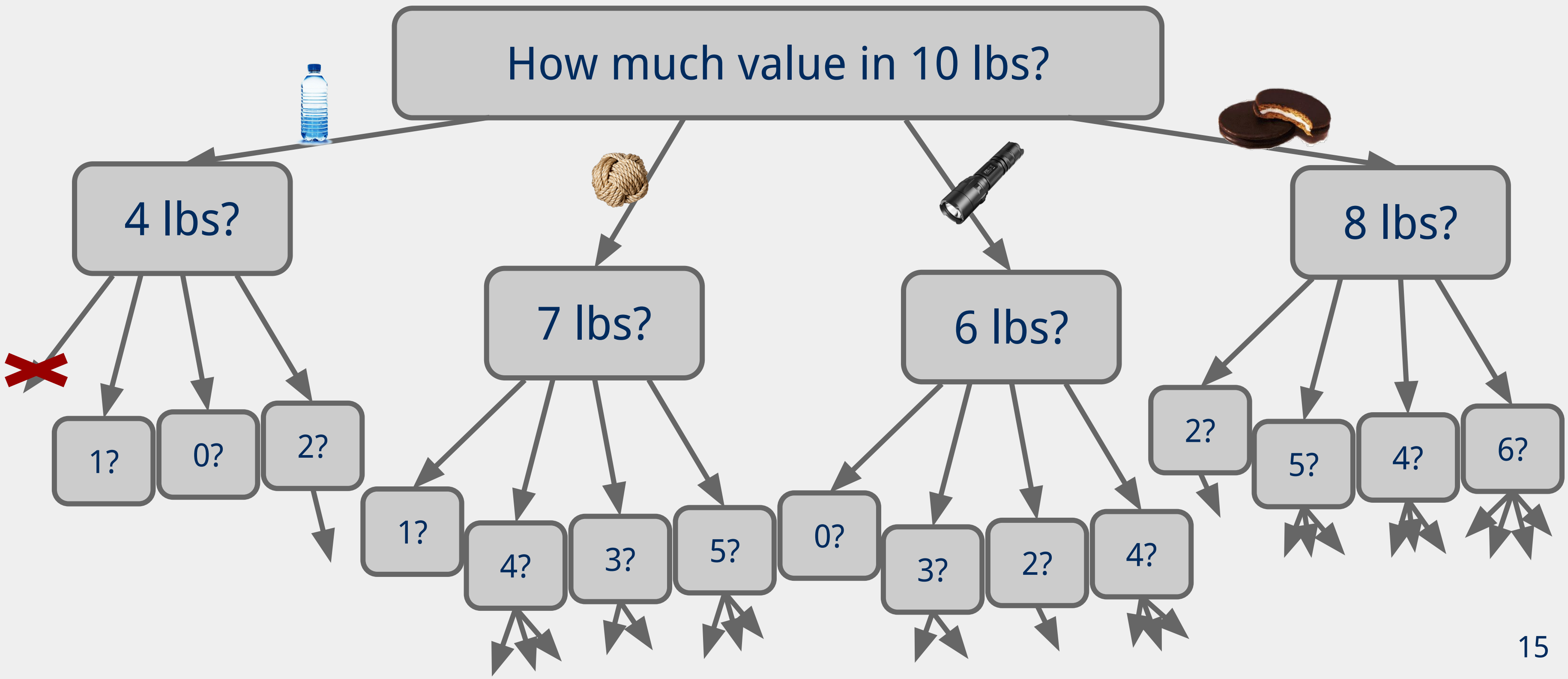
The Unbounded Knapsack Problem

Given a knapsack that can hold a weight limit L, and a set of n types items that each has a weight (w_i) and value (v_i), what is the maximum value that can fit in the knapsack with unbounded copies of each item?

Recursive example

weight:	6	3	4	2
value:	30	14	16	9

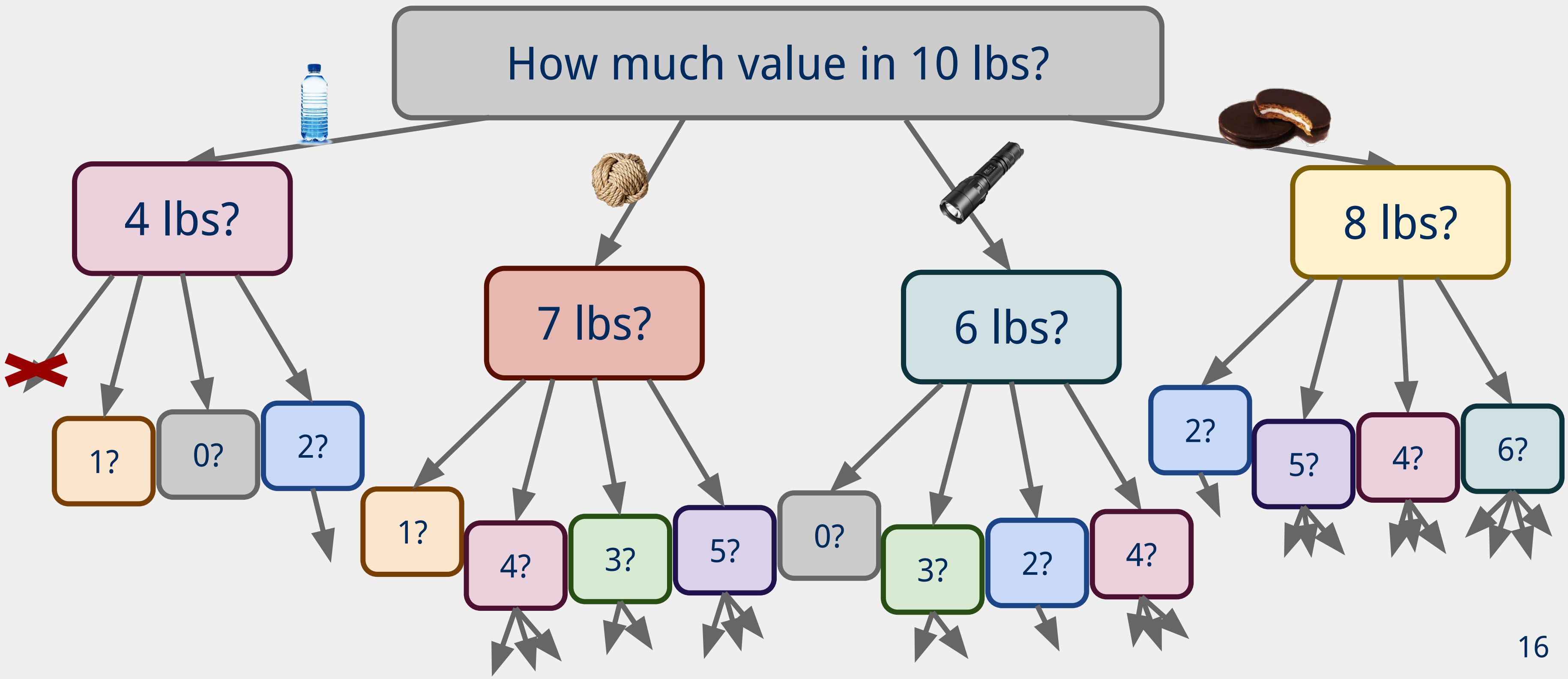
10 lb.
capacity



Recursive example

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Bottom-up example

				
weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

What would have happened with a greedy approach?

- Try adding as many copies of highest value per pound item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moonpie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb knapsack?
  - 44
    - Bogus!

Bottom-up implementation

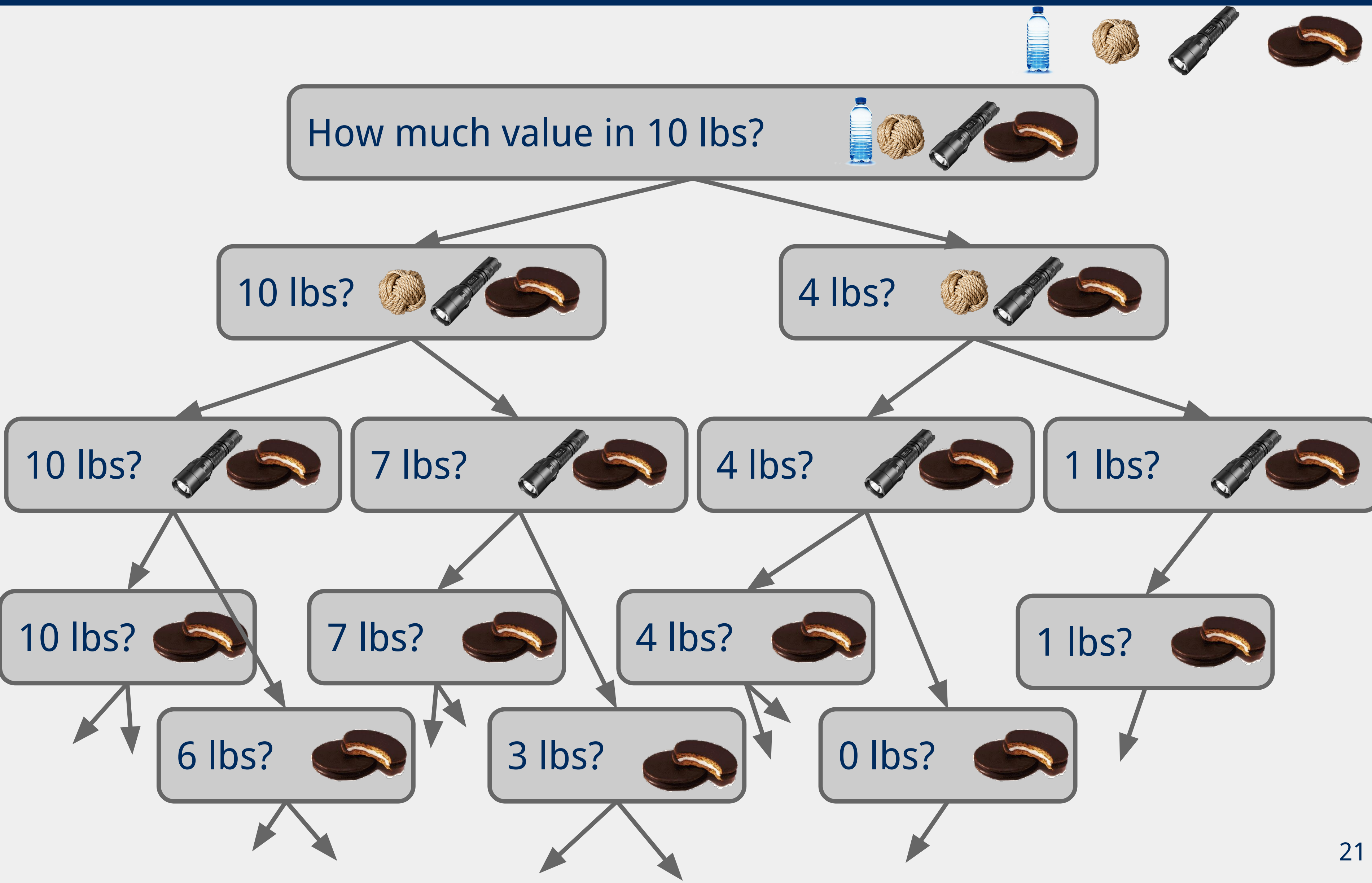
```
def unbound_knapsack(wt, val, L, n):
    K = [0 for i in range(L + 1)]
    for l in range(1, L + 1):
        max = 0
        for i in range(n):
            if (wt[i] <= l) and (val[i] + K[l-wt[i]] > max):
                max = val[i] + K[l - wt[i]]
        K[l] = max
```

The 0/1 Knapsack Problem

Given a knapsack that can hold a weight limit L, and a set of n items that each has a weight (w_i) and value (v_i), what is the maximum value that can fit in the knapsack with only a single copy of each item available?

0/1 Recursive example

weight:	6	3	4	2
value:	30	14	16	9



Recursive solution

```
def zero_one_knapsack(wt, val, L, n):

    if n == 0 or L == 0

        return 0

    if wt[n - 1] > L:

        return knapSack(wt, val, L, n-1)

    else:

        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1)
                   )
```

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4						

The 0/1 knapsack dynamic programming solution

```
def knapSack(wt, val, L, n):
    K = []
    for i in range(n + 1):
        K.append([0 for x in range(L + 1)])
        for l in range(L + 1):
            if i==0 or l==0:
                K[i][l] = 0
            elif wt[i-1] > l:
                K[i][l] = K[i-1][l]
            else:
                K[i][l] = max( val[i-1] + K[i-1][l-wt[i-1]],
                               K[i-1][l])
    return K[n][L]
```

The Longest Common Subsequence Problem

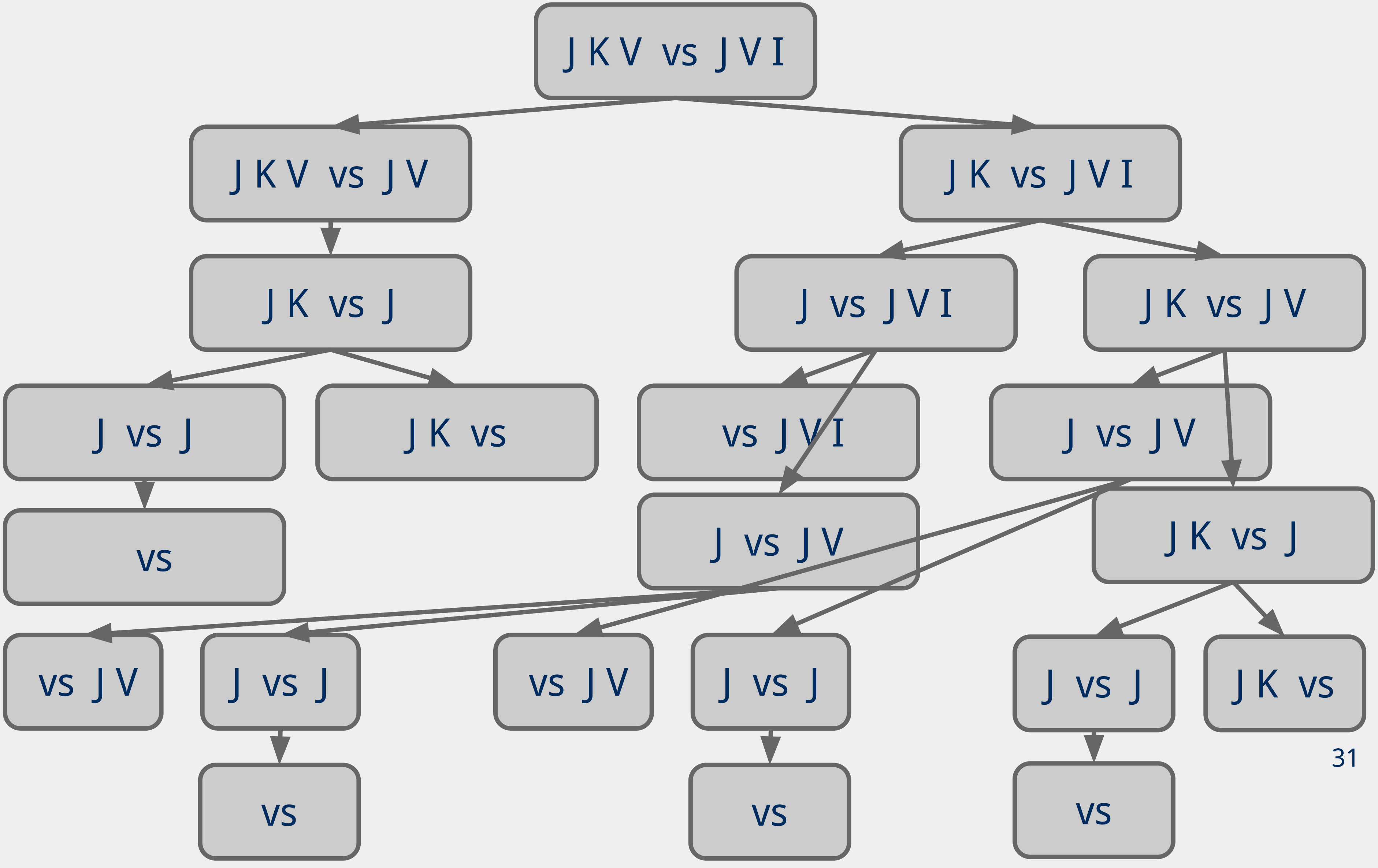
Given two sequences, return the longest common subsequence

LCS Example

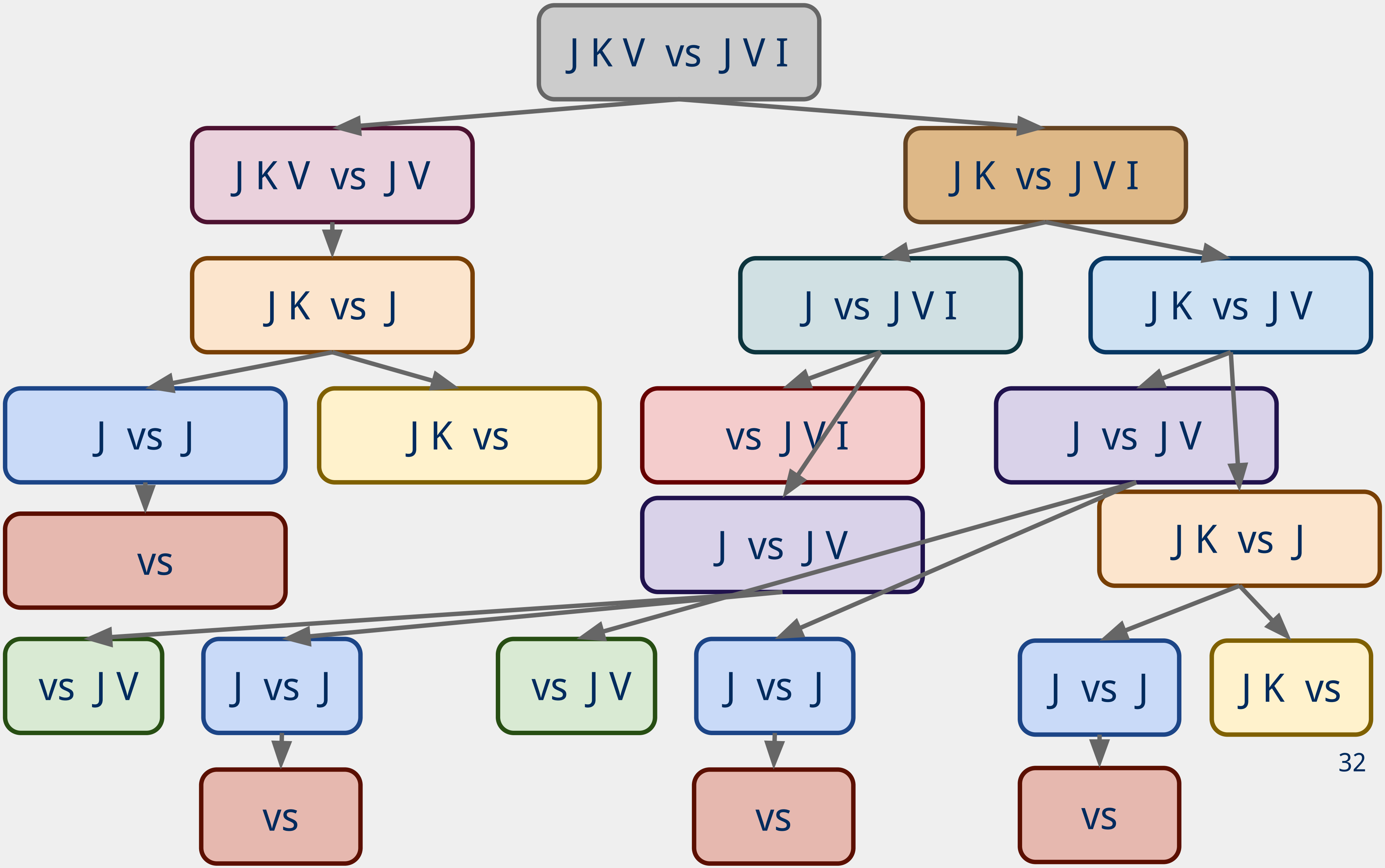
- A **Q** S R **J** K **V** B **I**

  Q B W F **J** **V** **I** T U
- We'll consider a relaxation of the problem and only look for the *length* of the longest common subsequence

LCS recursive example



LCS recursive example



LCS recursive solution

```
def LCSLength(x, y, m, n):
    if m == 0 or n == 0:
        return 0
    if x[m - 1] == y[n - 1]:
        return 1 + LCSLength(x, y, m - 1, n - 1)
    else:
        return max(LCSLength(x, y, m, n - 1),
                   LCSLength(x, y, m - 1, n)
                  )
```

LCS dynamic programming example

`x = A Q S R J B I`

`y = Q B I J T U T`

i\j	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

LCS dynamic programming solution

```
def LCSLength(x, y):
    m = []
    for i in range(len(x) + 1):
        m.append([0 for k in range(len(y) + 1)])
        for j in range(len(y) + 1):
            if i == 0 or j == 0:
                m[i][j] = 0
            if x[i] == y[j]:
                m[i][j] = m[i-1][j-1] + 1
            else:
                m[i][j] = max(m[i][j-1], m[i-1][j])
    return m[len(x)][len(y)]
```

The Change Making Problem

Consider a currency with n different denominations of coins d_1, d_2, …, d_n. What is the minimum number of coins needed to make up a given value k?