



CS/COE 1501
Algorithm Implementation

7-Greedy Algorithms and Dynamic Programming

Fall 2018

Sherif Khattab

ksm73@pitt.edu

6307 Sennott Square

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Consider the change making problem

- What is the minimum number of coins needed to make up a given value k ?
- If you were working as a cashier, what would your algorithm be to solve this problem?

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
 - Nearest neighbor approach to travelling salesman

... But wait ...

- Nearest neighbor doesn't solve travelling salesman
 - Does not produce an optimal result
- 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
- Why is optimal substructure not enough?

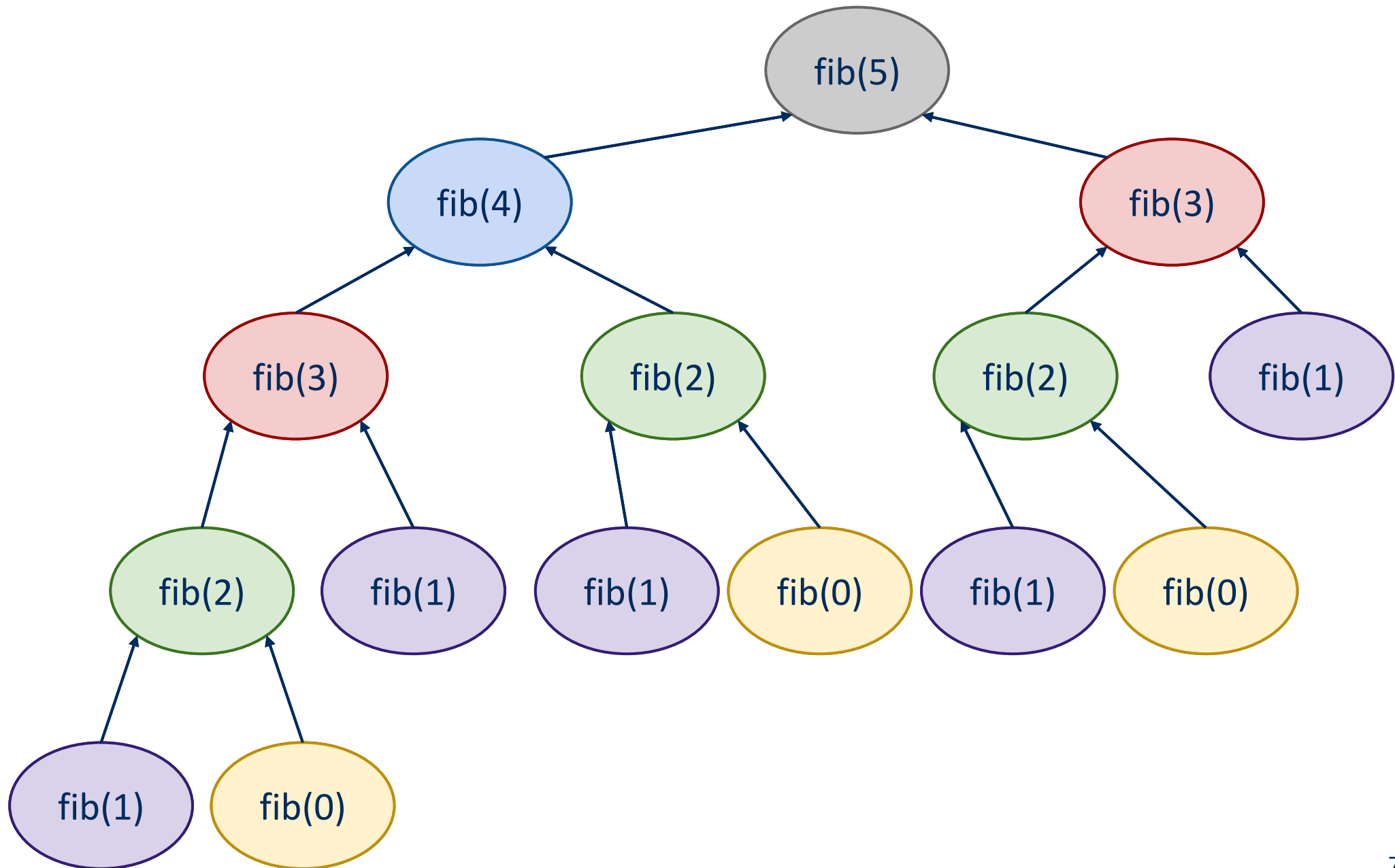
Finding all subproblems solutions can be inefficient

- Consider computing the Fibonacci sequence:

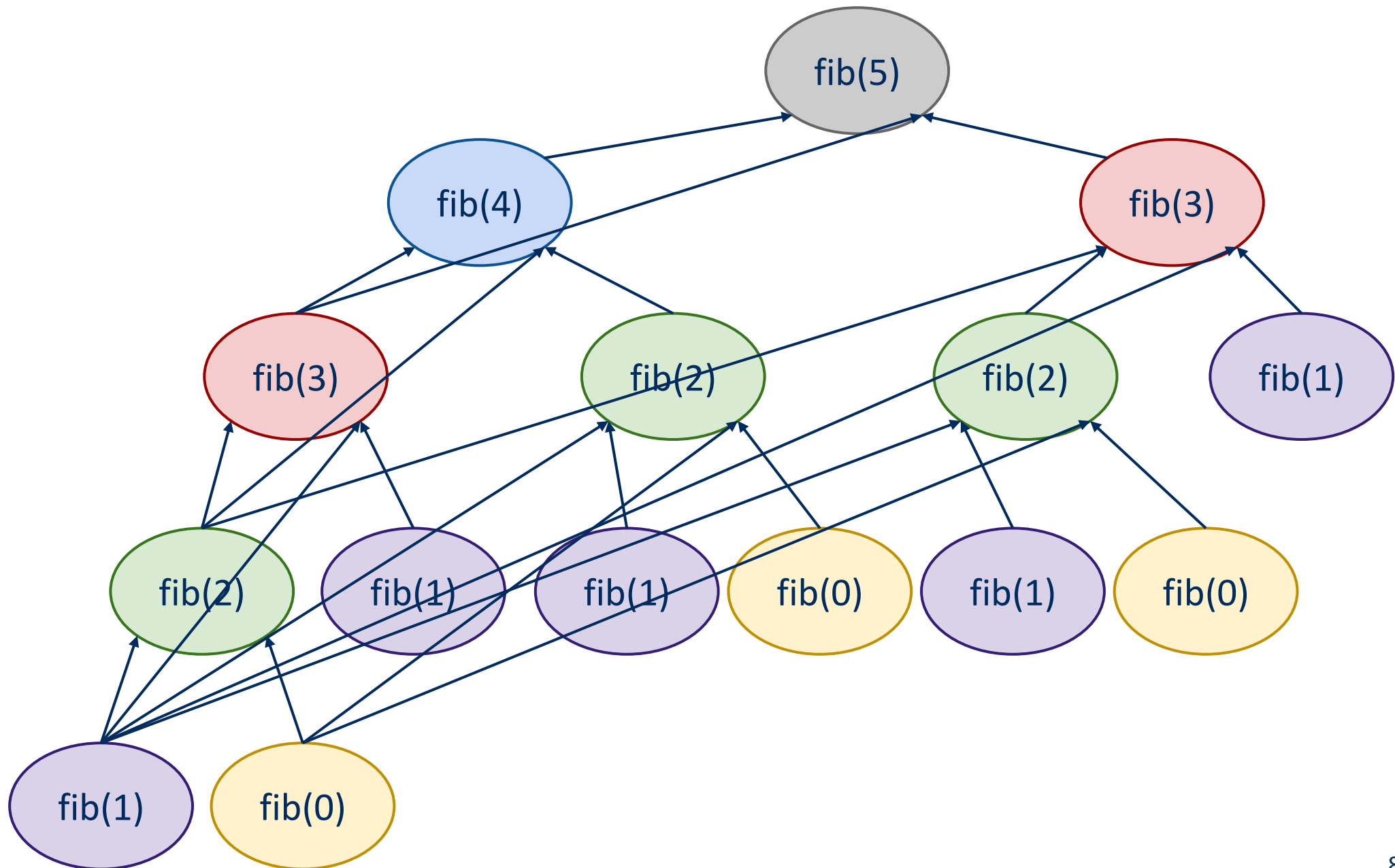
```
int fib(n) {  
    if (n == 0) { return 0 };  
    else if (n == 1) { return 1 };  
    else {  
        return fib(n - 1) + fib(n - 2);  
    }  
}
```

- What does the call tree for $n = 5$ look like?

fib(5)



How do we improve?



Memoization

```
int[] F = new int[n+1];
    F[0] = 0;
    F[1] = 1;
    for(int i = 2; i <= n; i++) { F[i] = -1 };

int dp_fib(x) {
    if (F[x] == -1) {
        F[x] = dp_fib(x-1) + dp_fib(x-2);
    }
    return F[x];
}
```

Note that we can also do this bottom-up

```
int bottomup_fib(n) {  
    if (n == 0)  
        return 0;  
  
    int[] F = new int[n+1];  
    F[0] = 0;  
    F[1] = 1;  
    for(int i = 2; i <= n; i++) {  
        F[i] = F[i-1] + F[i-2];  
    }  
    return F[n];  
}
```

Can we improve this bottom-up approach?

```
int improve_bottomup_fib(n) {  
    int prev = 0;  
    int cur = 1;  
    int new;  
    for (int i = 0; i < n; i++) {  
        new = prev + cur;  
        prev = cur;  
        cur = new;  
    }  
    return cur;  
}
```

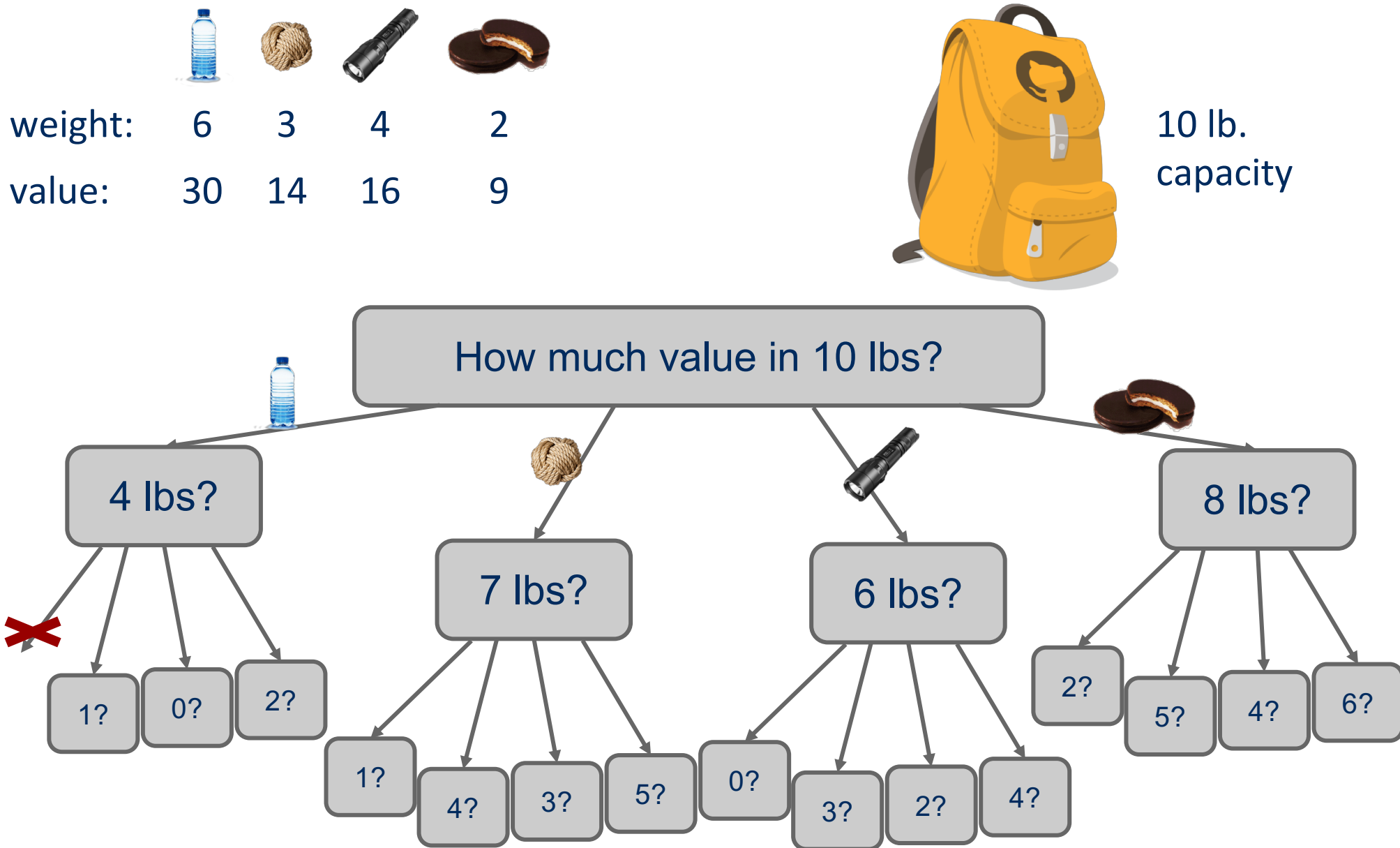
Where can we apply dynamic programming?

- 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

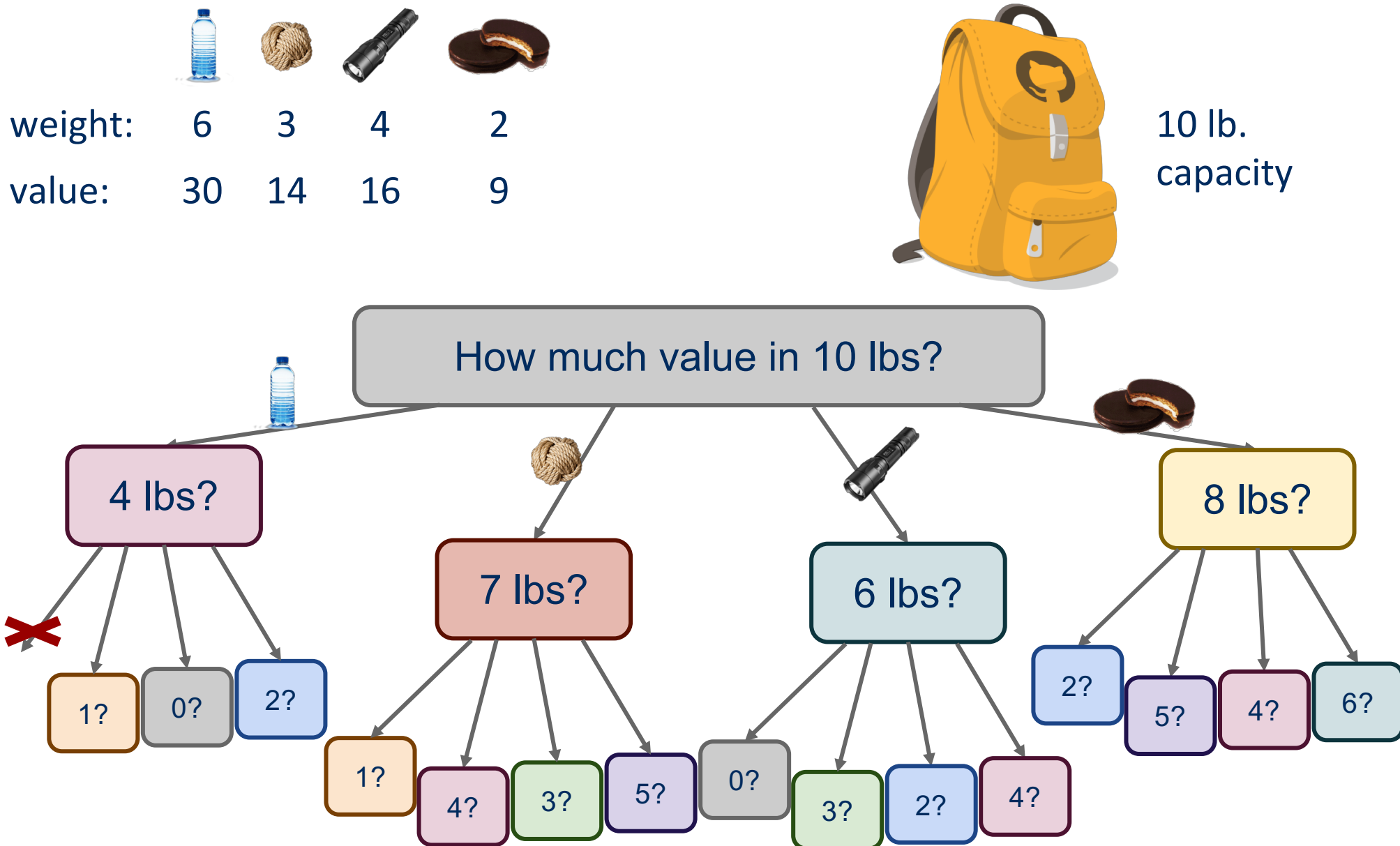
The unbounded knapsack problem

- 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 we can fit in the knapsack if we assume we have unbounded copies of each item?

Recursive example



Recursive example



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

Bottom-up solution

```
K[0] = 0

for (l = 1; l <= L; l++) {
    int max = 0;

    for (i = 0; i < n; i++) {
        if (wi <= l && vi + K[l - wi]) > max) {
            max = vi + K[l - wi];
        }
    }

    K[l] = max;
}
```

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!

The 0/1 knapsack problem

- What if we have a finite set of items that each has a weight and value?
 - Two choices for each item:
 - Goes in the knapsack
 - Is left out

0/1 Recursive example

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



How much value in 10 lbs?



10 lbs?



4lbs?



10 lbs?



7 lbs?



4 lbs?



1 lbs?



10 lbs?



7 lbs?



4 lbs?



1 lbs?



6 lbs?



3 lbs?



0 lbs?



Recursive solution

```
int knapSack(int[] wt, int[] val, int L, int n) {  
    if (n == 0 || 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

```
int knapSack(int wt[], int val[], int L, int n) {  
    int[][] K = new int[n+1][L+1];  
    for (int i = 0; i <= n; i++) {  
        for (int l = 0; l <= L; l++) {  
            if (i==0 || l==0){ K[i][l] = 0 };  
            else if (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];  
}
```

To review...

- Questions to ask in finding dynamic programming solutions:
 - Does the problem have optimal substructure?
 - Can solve the problem by splitting it into smaller problems?
 - Can you identify subproblems that build up to a solution?
 - Does the problem have overlapping subproblems?
 - Where would you find yourself recomputing values?
 - How can you save and reuse these values?

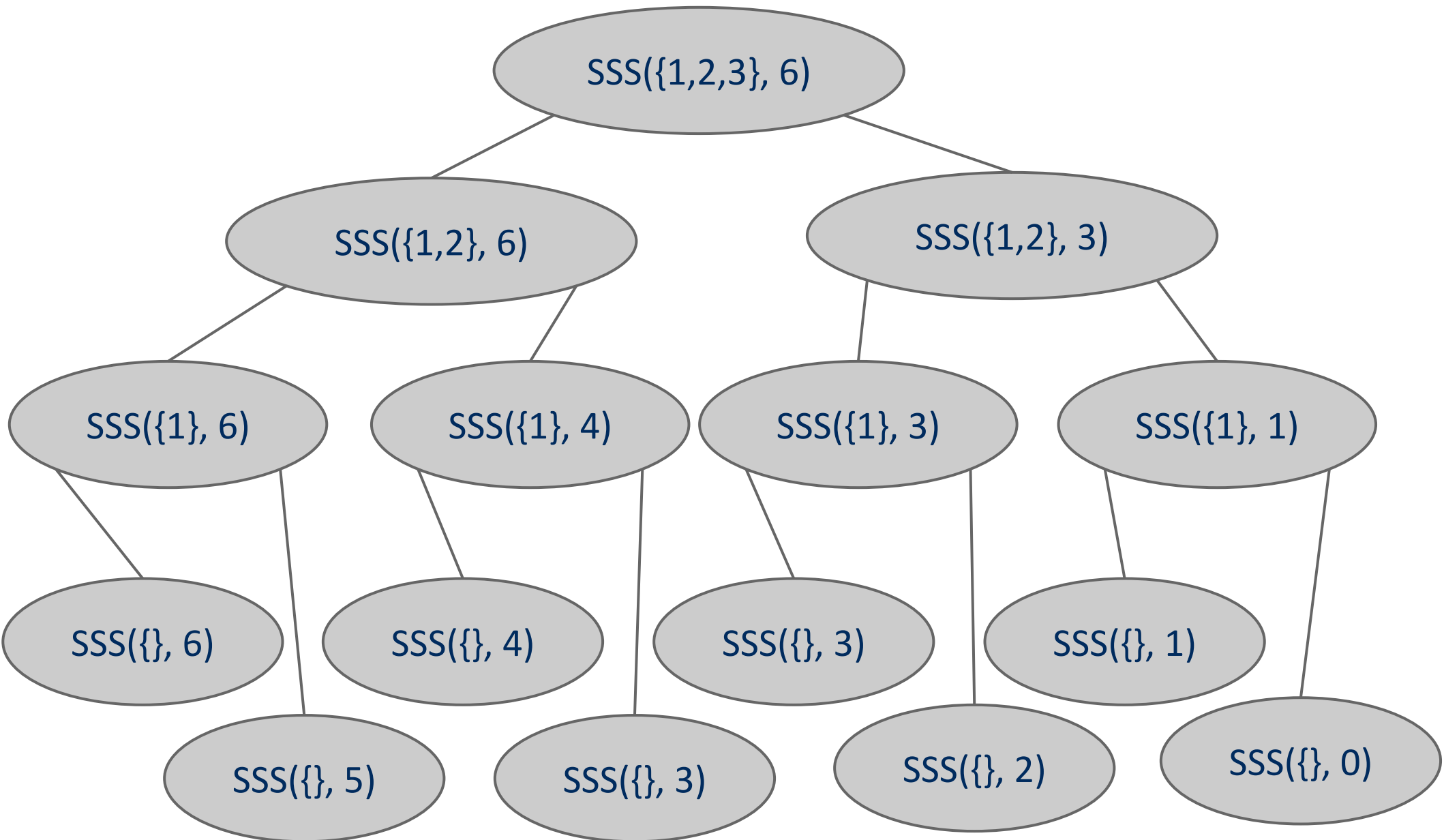
The change-making problem

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

Subset sum

- Given a set of non-negative integers S and a value k , is there a subset of S that sums to exactly k ?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {  
    if (sum == 0)  
        return true;  
    if (sum != 0 && n == 0)  
        return false;  
    if (set[n-1] > sum)  
        return SSS(set, sum, n-1);  
    return SSS(set, sum, n-1)  
        || SSS(set, sum-set[n-1], n-1);  
}
```

- What would a memoization data structure look like?

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[sum+1][n+1];
    for (int i = 0; i <= n; i++) subset[0][i] = true;
    for (int i = 1; i <= sum; i++) subset[i][0] = false;
    for (int i = 1; i <= sum; i++) {
        for (int j = 1; j <= n; j++) {
            subset[i][j] = subset[i][j-1];
            if (i >= set[j-1])
                subset[i][j] = subset[i][j]
                    || subset[i -
set[j-1]][j-1];
        }
    }
    return subset[sum][n];
}
```

Longest Common Subsequence

- Given two sequences, return the longest common subsequence
 - 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 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

```
int LCSLength(String x, String y) {  
    int[][] m = new int[x.length + 1][y.length + 1];  
    for (int i=0; i <= x.length; i++) {  
        for (int j=0; j <= y.length; j++) {  
            if (i == 0 || j == 0) m[i][j] = 0;  
            if (x.charAt(i) == y.charAt(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[x.length][y.length];  
}
```

Case study: database query optimization

- SQL is a very popular database query language
 - It is *declarative*
 - User's don't specify what the database should do to answer their query, just what they want
 - The *query optimizer* then finds the best plan to find the result of the user's query
 - Dynamic programming is a popular approach to query optimization

Database query optimization example

- Example database:

Customers

CID	Name
10	Nick
20	Bill
30	Erin

Addresses

AID	Loc	Cust
100	PGH	30
200	NYC	20
300	PGH	10

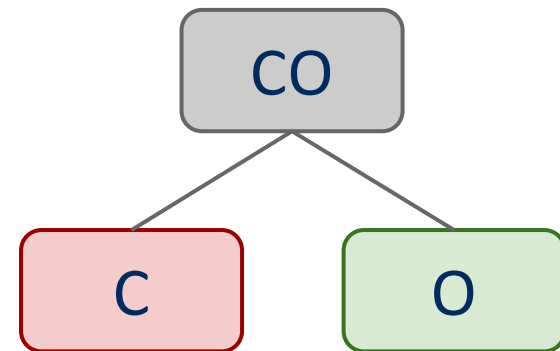
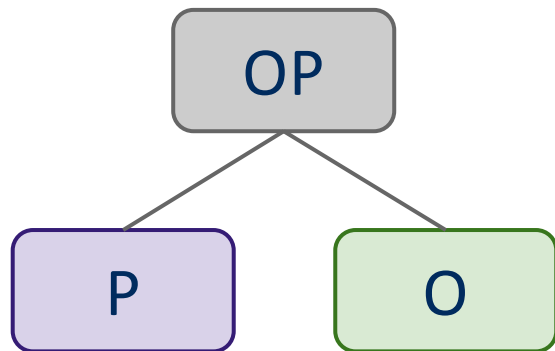
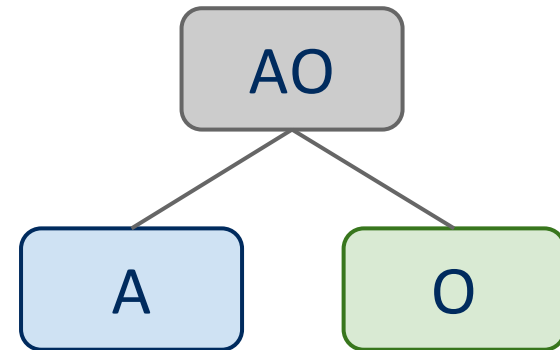
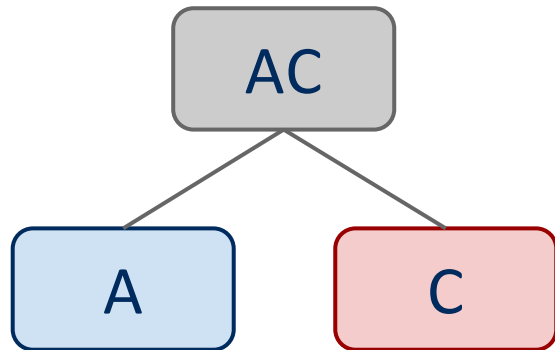
Products

PID	Name	Price
1	Book	3
2	DVD	10
3	LP	30

Orders

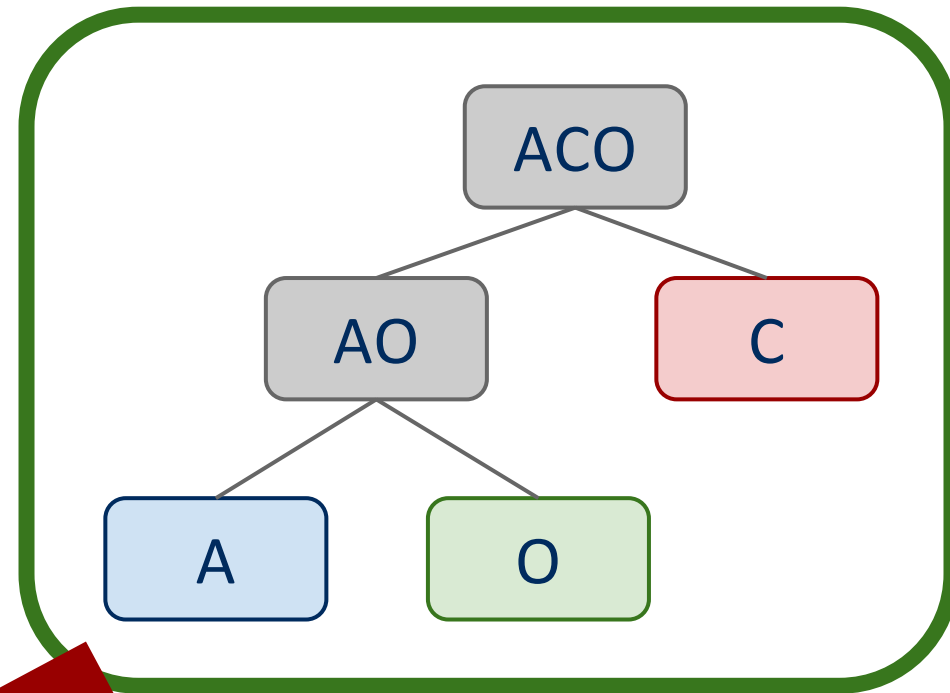
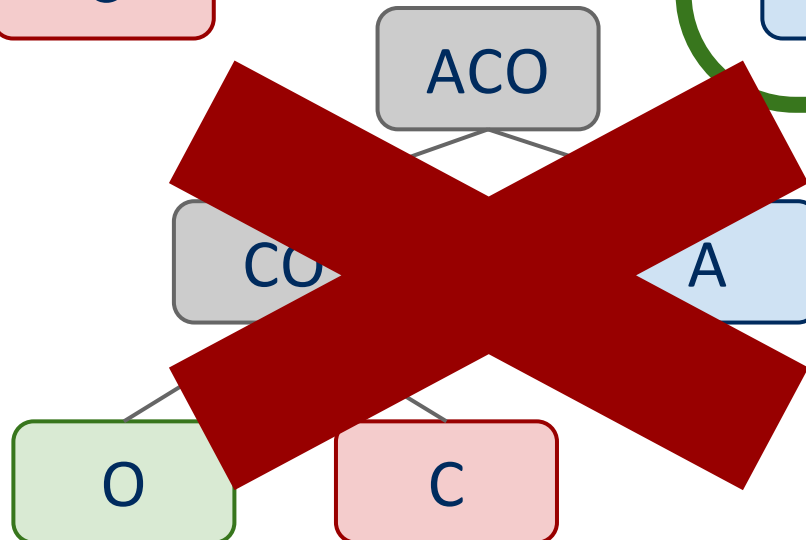
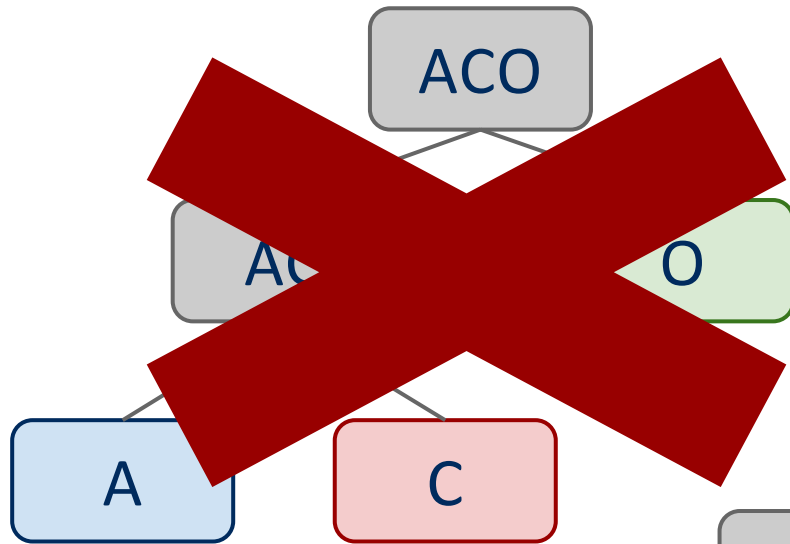
OID	Prod	Num	Cust	Addr
1000	3	3	20	200
2000	3	10	10	300
3000	3	30	30	100

First, find plans to join pairs of tables

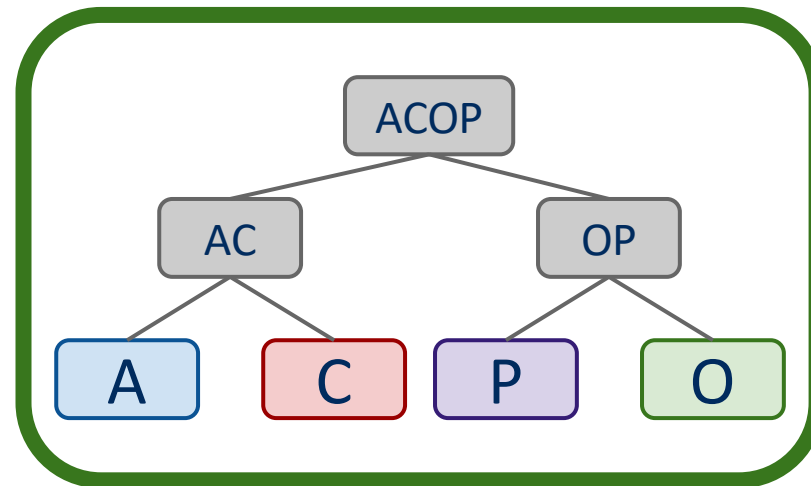
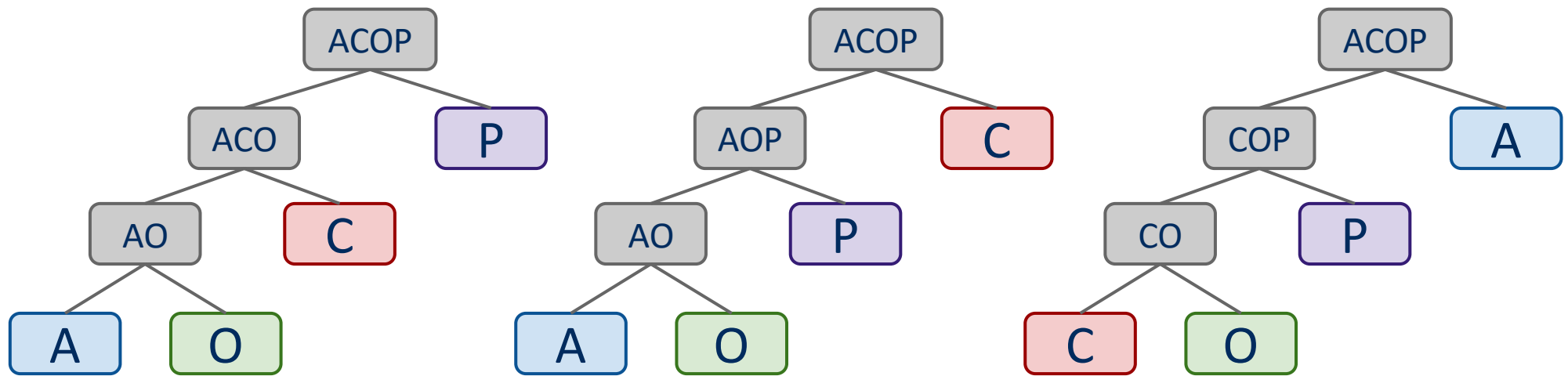


Next, find best plans to join triplets of tables

- ACO, for example:



Continue until you find all n-way plans



Already know this is
the most efficient ACO
plan, don't need to try
any others!