Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

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

Announcements

- Upcoming Deadlines
  - Lab 9 and Homework 9: next Monday 11/21 @ 11:59 pm
  - Assignment 3: ~~Monday 11/28~~ Friday 12/9 @ 11:59 pm
  - Assignment 4: Friday 12/9 @ 11:59 pm

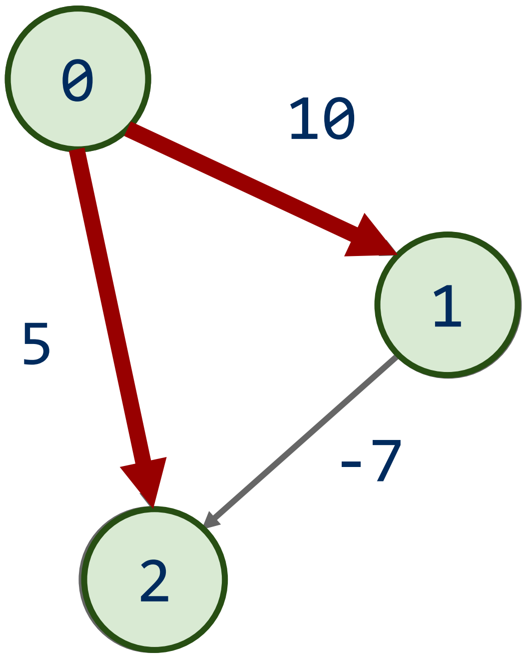
Previous lecture

- Weighted Shortest Paths problem
  - Dijkstra's shortest paths algorithm
  - Bellman-Ford's shortest paths algorithm

This Lecture

- Dynamic Programming

Dijkstra's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	5	0

Incorrect!

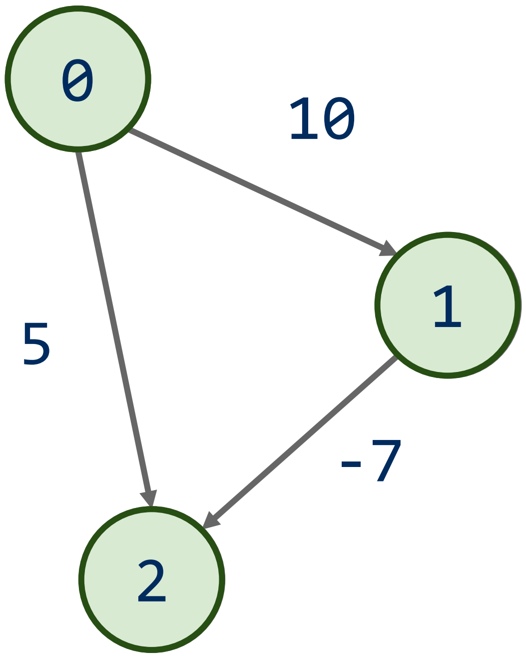
Analysis of Dijkstra's algorithm

Dijkstra's is correct only when all edge weights >= 0

Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty:
  - cur = pop a vertex from Q
  - For each non-parent neighbor x of cur:
    - Compute distance from start to x through cur
      - distance[cur] + weight of edge between cur and x
    - if computed distance < distance[x]
      - Update distance[x]
      - add x to Q if not already there

Bellman-Ford's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	3	1

FIFO Q:

0
1
2

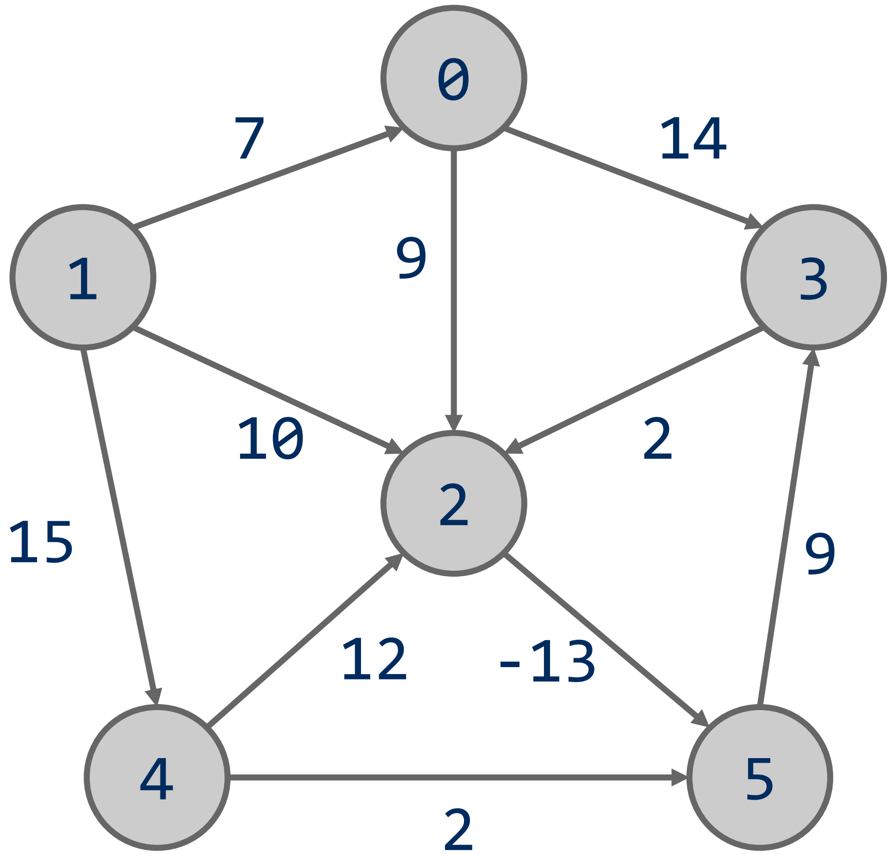
Correct!

Analysis of Bellman-Ford's algorithm

Bellman-Ford's is correct even when there are negative edge weights in the graph but what about negative cycles?

- a negative cycle is a cycle with a negative total weight

Bellman-Ford’s example with a negative cycle



Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty **and no negative cycle has been detected**:
  - cur = pop a vertex from Q
  - For each non-parent neighbor x of cur:
    - Compute distance from start to x through cur
      - distance[cur] + weight of edge between cur and x
    - if computed distance < distance[x]
      - Update distance[x]
      - add x to Q if not already there
  - check for a negative cycle in the current Spanning Tree every v edges

Let’s change focus into a different type of problems

- We will get back to graphs after the break!

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

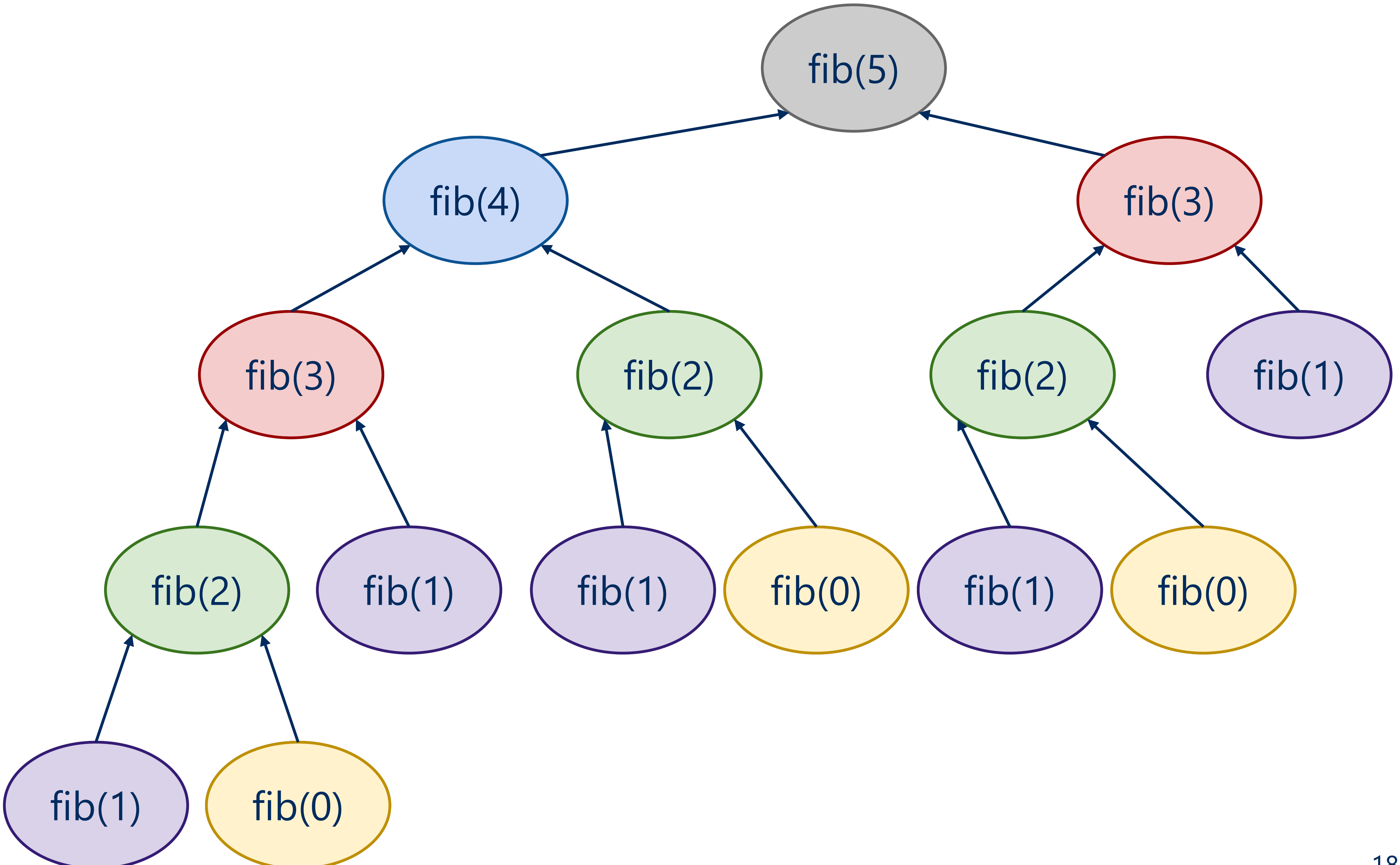
- Consider computing the Fibonacci sequence:

```
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?

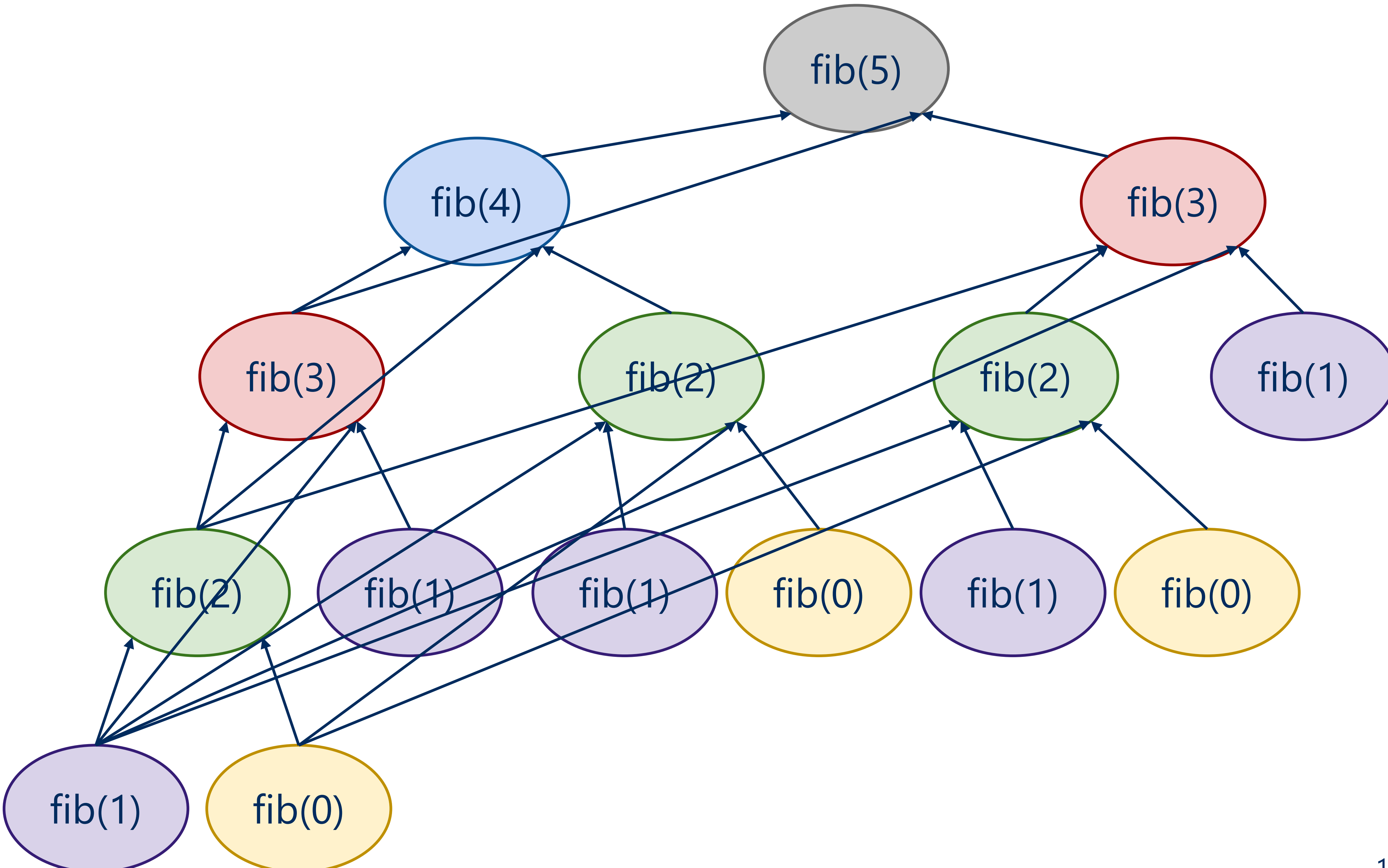
$$\overset{+}{0,\ 1},\ \underset{+}{1,\ 2},\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$$

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

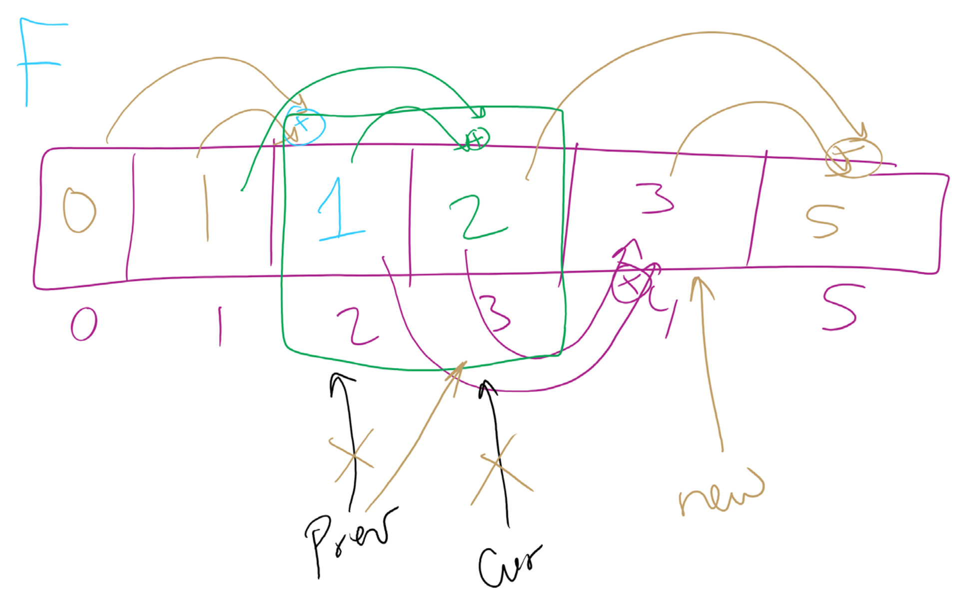
```
int[] F = new int[n+1];
   F[0] = 0;
   F[1] = 1;
   for(int i = 2; i <= n; i++) { F[i] = -1 };


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

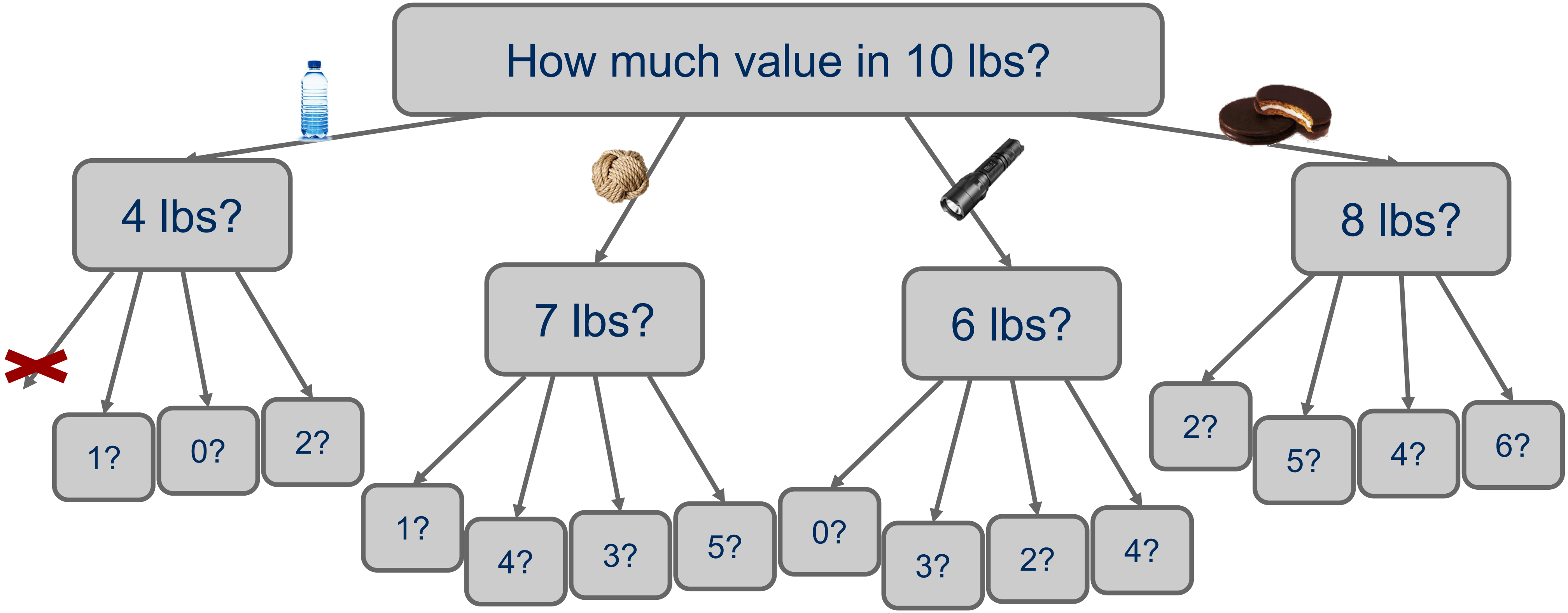
Problem of the Day Part 3: The unbounded knapsack problem

- 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 Solution

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

10 lb.
capacity



Recursive Solution

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

10 lb.
capacity