# CA318
# Advanced Algorithms and AI Search

## Introduction to Dynamic Programming

# Dynamic Programming

Memoization; Subproblems; Fibonacci Sequence

# Dynamic Programming

- Dynamic Programming is a general and sometimes quite powerful method for the development of algorithms
- It can be used to improve the complexity of Brute Force algorithms
  - By removing redundant computations
  - By “remembering” computational states
    - In the form of memos
    - This introduces the term **memoization**

# Memoization and Subproblems

As mentioned, Dynamic Programming introduces us to two new concepts in algorithm design:

- Memoization
- Subproblems

# Aside

- The term Dynamic Programming is something of a misnomer:
  - The word programming => optimization
  - Dynamic => [sometimes] recursion

# Fibonacci numbers

A useful starting point for DP.

# Fibonacci Sequence

We are all familiar with the **Fibonacci Sequence**.

It is computed using the well known recurrence:

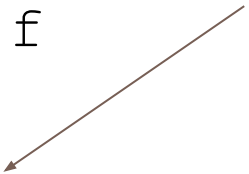
$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Goal: Compute $F_n$ for some arbitrary $n$

Note: This definition is **recursive**, as we shall see now

# Naive Recursive Implementation

```
fib(n):
    if (n == 0): f = 0
    else if (n == 1): f = 1
    else: f = fib(n-1) + fib(n-2)
    return f
```

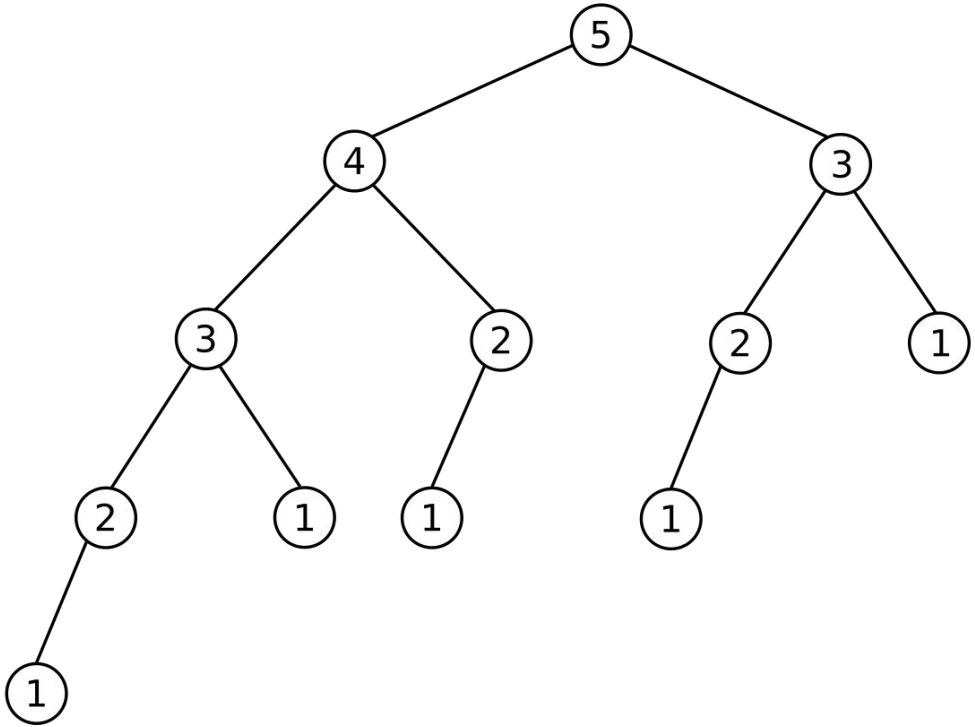


Recursive component

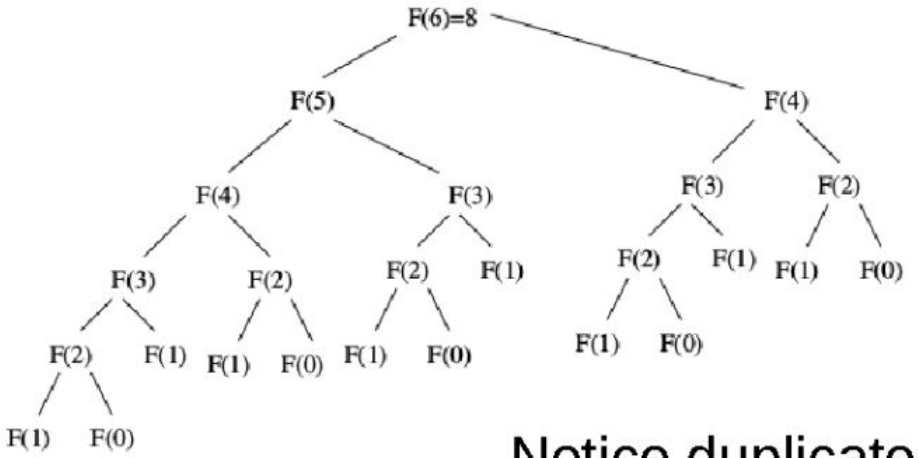
# Complexity Calculating $F_5$

Any suggestions for the complexity of this naive implementation?

Does it remind you of anything?



# Problems with Naive Fibonacci




Notice duplicate work:

- f(4) calculated twice
- f(3) calculated 3 times
- f(2) calculate 5 times

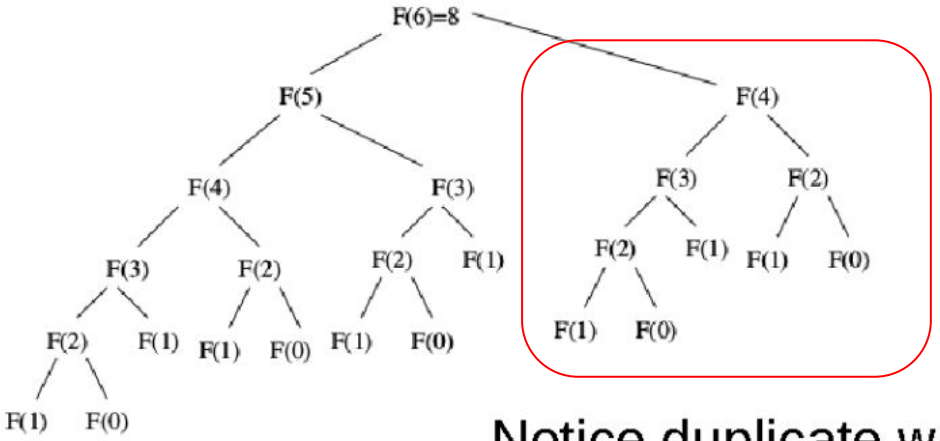
# How much duplicate work is done?

We can approximate the answer by using our knowledge of trees where the number of nodes is:

$2^{n-1}$

But we note that redundant calculations dont just happen in one place, they happen all over. Therefore the redundancy is at least

Which is not good




Notice duplicate work:

- f(4) calculated twice
- f(3) calculated 3 times
- f(2) calculate 5 times

# Memoization

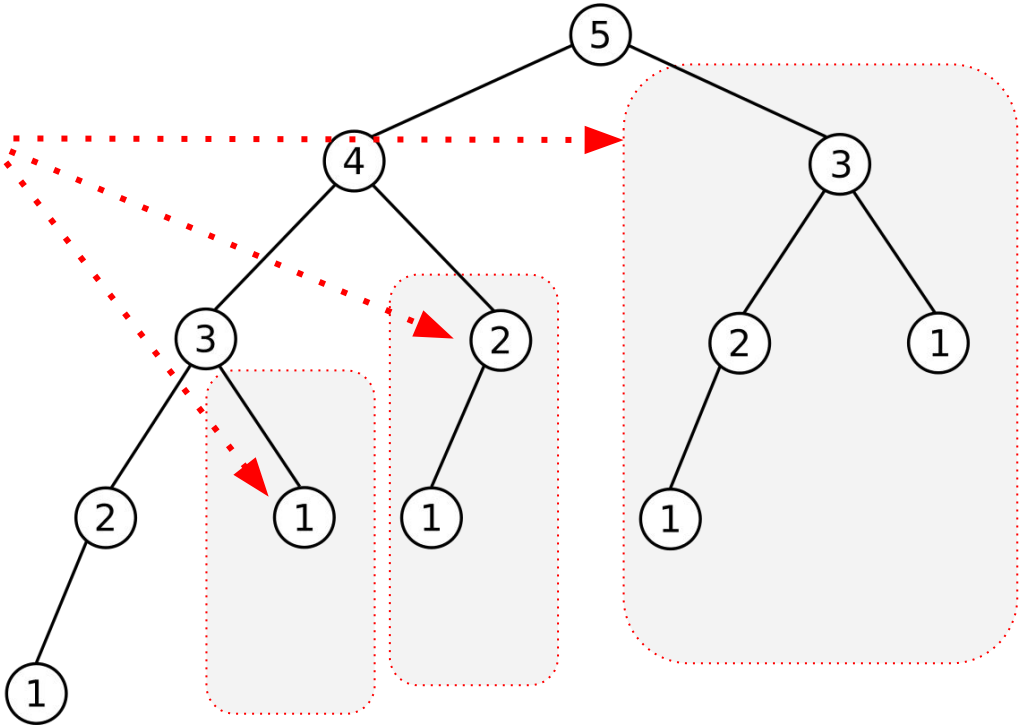
- The idea of memoization is to introduce a data structure (usally called a dictionary):
  - to store the results of already computed values.
- This dictionary has two important properties
  - Constant time lookup speed **O(1)**
  - Constant time insertion speed **O(1)**

# Memoized Fibonacci

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else:
    if (n == 0): f = 0
    else if (n == 1): f = 1
        else: f = fib(n-1) + fib(n-2)
        memo[n] = f
    return f
```

# Improvements

Whole sections of the recurrence tree are eliminated with this small improvement



# Improvement

`fib(k)` only recurses the first time it is called, for all *k*. All other calculations of *k* are O(1) lookups

The total number of non-memoized calls for any number **n** is `fib(1),fib(2),...,fib(n)` As we can see from the previous slide

Therefore **the complexity of non-memoized work is … ?**

# How much work in `fib(k)` ?

Lets stop for a minute and consider how much work is done in the function `fib(k)`

**...any ideas?**

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else:
        if (n == 0): f = 0
        else if (n == 1): f = 1
        else: f = fib(n-1) + fib(n-2)
        memo[n] = f
    return f
```

# Complexity for Memoized Fibonacci

$$fib(n) = \Theta(n)$$

In fact, there is a better complexity for computing fibonacci, that uses on the order of

$$fib(n) = \Theta(log\ n)$$

But this is outside the scope of CA318

# Subproblems and Memoization

- So we have seen memoization.
- Sub-problems are just small problems along the way to solving the large problem.
- So in the case of `fib(5)` the subproblems are:
  - `fib(4)`
  - `fib(3)`
  - `fib(2)`
  - `fib(1)`
- So we memoize the subproblems
- Subproblems are **not** the goal!

# Subproblems

- The subproblems were solved recursively.
- Therefore, we can conclude by saying that dynamic programming is
  - Memoization, and
  - Recursion
- Dynamic Progrmamming - the running time of any solution is:
  - **# of subproblems x time spent per subproblem, eg,**

$$fib(n) = n\ \Theta(1)$$

# Subproblems

- Please remember, subproblems are not (mathematically speaking) different function calls in the code
  - **or** different parts of a program taken conditionally based on the code execution path
- Subproblems are **repeatedly executing identical blocks of code**

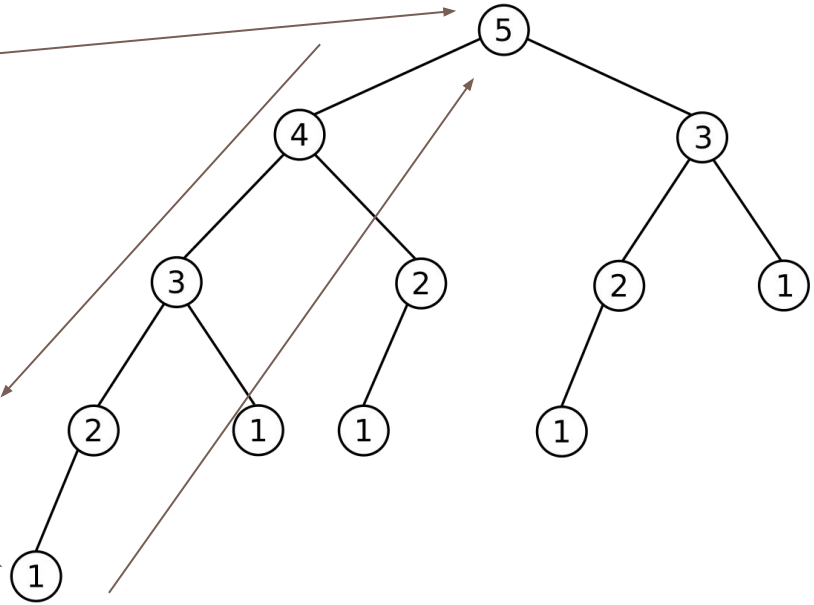
# Bottom-up DP Algorithms

An alternative to recursion/memoization

# Recursion Starts at the top

The fibonacci tree generation started at the top and recursed until it could go no further. This is **top-down**

But another approach is **bottom-up**. Start here and work your way up. This is called **bottom-up**, but does not use recursion



# Bottom-up

If you think about it, bottom-up does not need to use recursion. It can use another approach. So we **unroll** the recursion and write it into a `for`-loop instead

```
fib = {}
fibBU(n):
    for k in range(1:n):
        if k <= 2: f = 1
        else:
            f = fib[k-1] + fib[k-2]
            fib[k] = f
    return fib[k]
```

The same code as the recursive definition

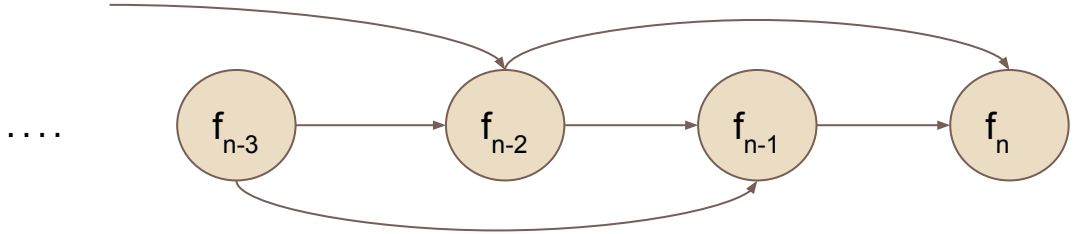
# Complexity Check

- Whats the complexity of this approach?
- Is it faster or slower than the recursive definintion?
- Which do you prefer?

# Equivalent Computation

In the general cases:

- Bottom-up does exactly the same computation as recursion
- It is merely a topological sort of the subproblem dependency DAG



# Space Efficiency

- Bottom-up also has another advantage:
- It is more efficient with space (storage)
- Of course, space is not always a concern nowadays
  - But if **n** is extremely large, then space efficiency is always a good idea

**What is the space complexity of the bottom up approach?**

# Calculating Complexity

- As mentioned the complexity for this algorithm is the same as the recursive solution
- But something rather interesting to note here,
  - the complexity is obvious
  - there is no recurrence, no memoization,
- so the complexity calculation is trivial:

$$fibBU(n) = n\ \Theta(1)$$

# Greedy Algorithms

Making change;  Local Optimas; Knapsack Problem

# Greedy Algorithms

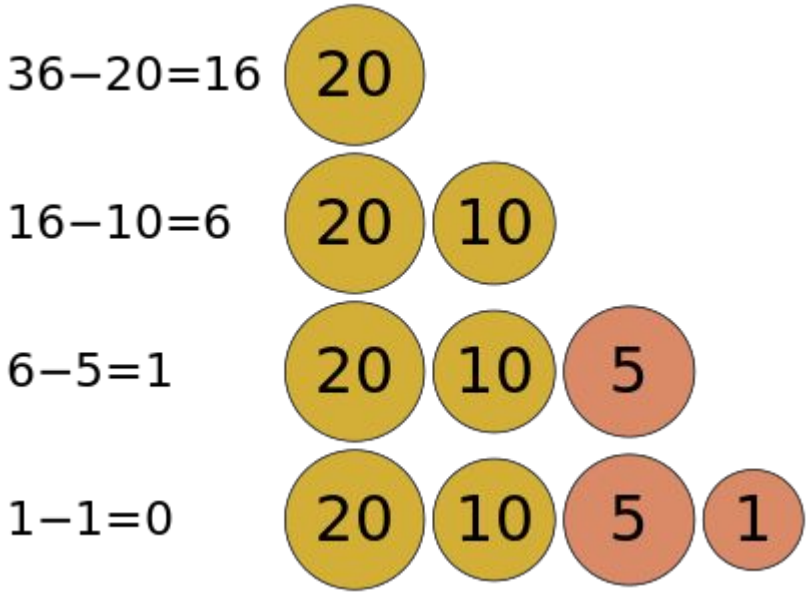
## Greedy algorithm

From Wikipedia, the free encyclopedia

A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.[1] In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

# Example

We want to make 36c from the coins 20,10,5,1 as shown here, using as few coins as possible



# Naive Implementation

An naive implementation that does not use any dynamic programming efficiencies seen earlier

```
def make_change(amount, coins):
    coins.sort(reverse=True)
    used = [0] * len(coins)

    for index, coin in enumerate(coins):
        while amount >= coin:
            used[index] += 1
            amount -= coin

    return used
```

# What change is computed

For the amount A = 767

- Coin set C = [ 200,100, 50, 20,10, 5, 2,1 ]
- Output: O = [ 3, 1, 1, 0, 1, 1, 1, 0 ]
- Which is correct

For the amount A = 101

- Coin set C = [ 200,100, 50, 20,10, 5, 2,1 ]
- Output: A = [ 0, 1, 0, 0, 0, 0, 0, 1 ]
- Which is correct

# Greedy makes a mistake

For the amount A = 6

- Coin set C = [ 4,3,1 ]
- Output: O = [ 1, 0, 2 ]
- Which is incorrect - it results in too many coins being used

What has happened here is that greedy has made locally optimal decisions until it no longer can. But it has lost sight of the objective. We need a new method.

# Problems

- Obviously, the greedy algorithm as shown here fails in certain cases.
- This is qualitatively different to naive fibonacci - which worked correctly in every case, but with a bad complexity
- But we can still use Dynamic Programming here:
  - Recursion
  - Memoize

# Greedy fails

For the amount 36

- Coin set [ 25, 10, 4, 2 ]
- Output: [ 1, 1, `---  ERROR---` ]
  - So greedy simply fails
  - Telling us no solution exits
  - But a solution **does** exist
- So now we can see where greedy returns a solution that is not optimal, and greedy simply fails (above)
- Greedy is not guaranteed to work - so we need another approach

# Dynamic Programming

If a solution does exist, DP is guaranteed to **always find the optimal solution**

This is in contrast to greedy algorithms that offer no such assurance

# Computation Tree

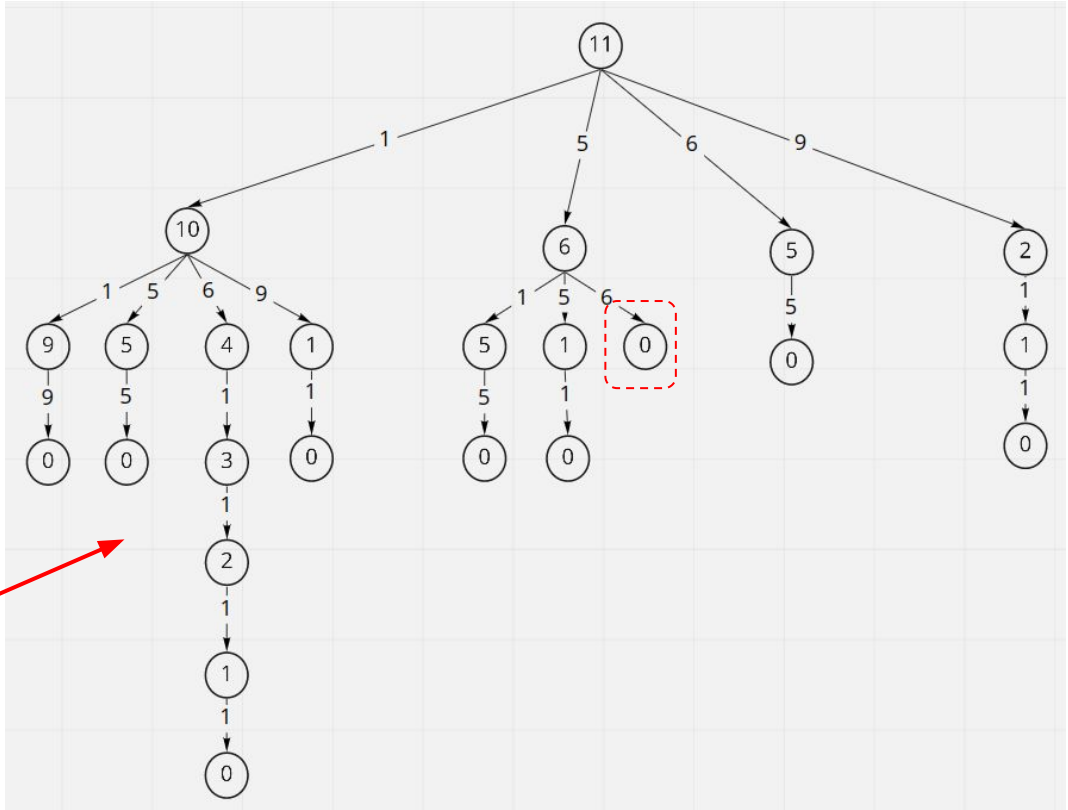
Lets look at this again, using the following amount and coin configuration

A = 11, C = { 1, 5, 6, 9 }

We can see that the greedy algorithm fails to get the optimal solution (9,1,1) wheras we wanted (5 ,6).

Lets recast this in the form of a tree and use our recursion.

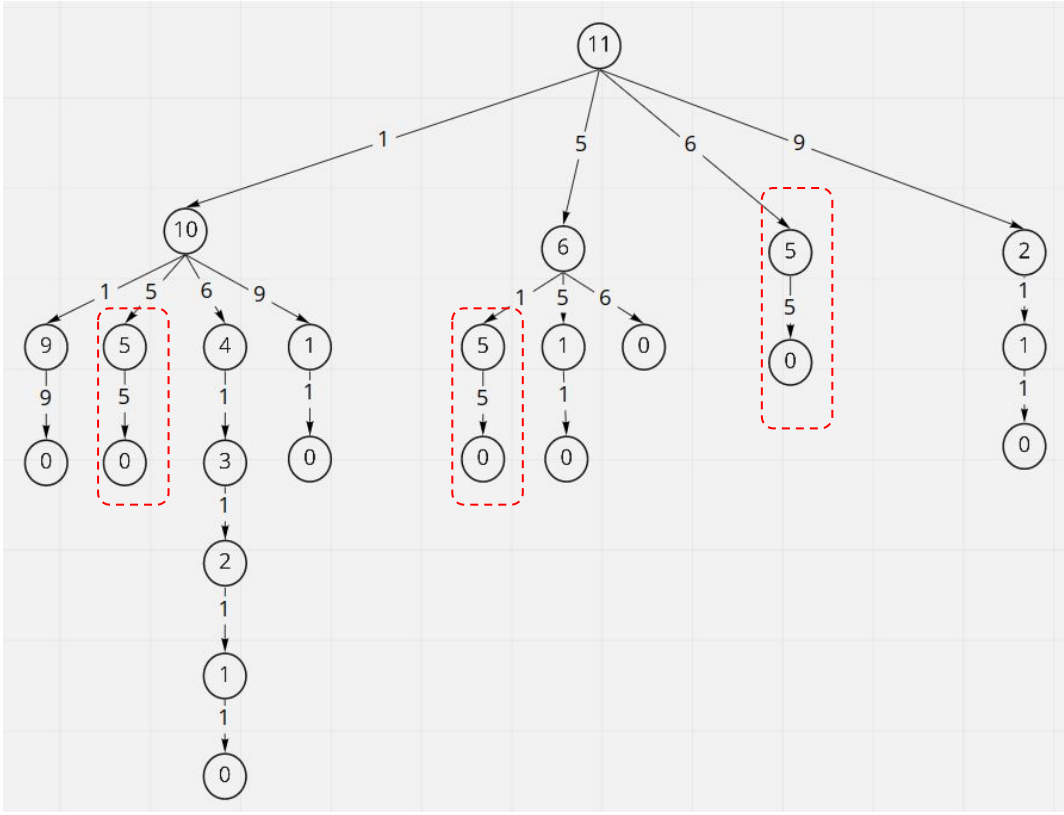
Q: What is the complexity of this tree as shown?



# Computation Tree

Once again, repeated computations are obvious.

So recursion on its own does not solve this efficiently



# The importance of memo

- This solution has en extremely poor performance without the use of memo
- If you implement it without memo, it could take many seconds to solve the question
- With memo the performance will be instantaneous
- In this example it is slightly clearer if we use loop unrolling rather than recursion

# Example

Let us now apply the bottom up method to solve a new problem

A = 10, C = { 1, 5, 6, 9 }, “bottom-up” (no recursion):

$j^{th}$ Amount →

$i^{th}$ coin ↓

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 |
| 6 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 |
| 9 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 1 | 2 |

# Step by Step

A = 10, C = { 1, 5, 6, 9 }, “bottom-up” (no recursion):

$j^{th}$ Amount →

$i^{th}$ coin ↓

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf | Inf |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 |
| 6 | 0 | | | | | | | | | | |
| 9 | 0 | | | | | | | | | | |

# Bottom-upImplementation

The following is a dynamic programming implementation (with Python 3) which uses a matrix to keep track of the optimal solutions to sub-problems, and returns the minimum number of coins, or "Infinity" if there is no way to make change with the coins given.

A second matrix may be used to obtain the set of coins for the optimal solution.

Ref
https://en.wikipedia.org/wiki/Change-making_problem

```python
def _get_change_making_matrix(set_of_coins, r: int):
    m = [[0 for _ in range(r + 1)] for _ in range(len(set_of_coins) + 1)]
    for i in range(1, r + 1):
        m[0][i] = float('inf')  # By default there is no way of making change
    return m

def change_making(coins, n: int):
    """This function assumes that all coins are available infinitely.
    n is the number to obtain with the fewest coins.
    coins is a list or tuple with the available denominations.
    """
    m = _get_change_making_matrix(coins, n)
    for c in range(1, len(coins) + 1):
        for r in range(1, n + 1):
            # Just use the coin coins[c - 1].
            if coins[c - 1] == r:
                m[c][r] = 1
            # coins[c - 1] cannot be included.
            # Use the previous solution for making r,
            # excluding coins[c - 1].
            elif coins[c - 1] > r:
                m[c][r] = m[c - 1][r]
            # coins[c - 1] can be used.
            # Decide which one of the following solutions is the best:
            # 1. Using the previous solution for making r (without using coins[c - 1]).
            # 2. Using the previous solution for making r - coins[c - 1] (without
            #      using coins[c - 1]) plus this 1 extra coin.
            else:
                m[c][r] = min(m[c - 1][r], 1 + m[c][r - coins[c - 1]])
    return m[-1][-1]
```

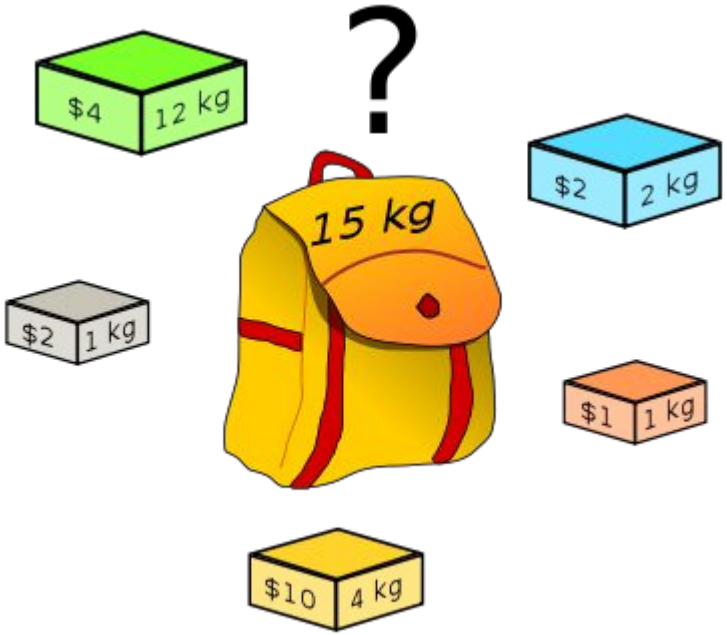
# Bottom-up Implementation

Using the previous example where we derived an estimate for the complexity of fibonacci, can you work out a complexity estimate for the recursive memoized making change implementation on the previous slide?

# Knapsack Problem

The knapsack problem is a problem in combinatorial optimization:

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



# Finally

The knapsack problem is similar to the coin changing problem, except that we have to model a weight **and** a cost property rather than a size.

Both of these problems are problems of **Optimization** - where you are looking for the maximum/minimum function. We will revisit this in a future lecture

# Thank you

Any Questions