Dynamic Programming

Joonho Kim

Instructions

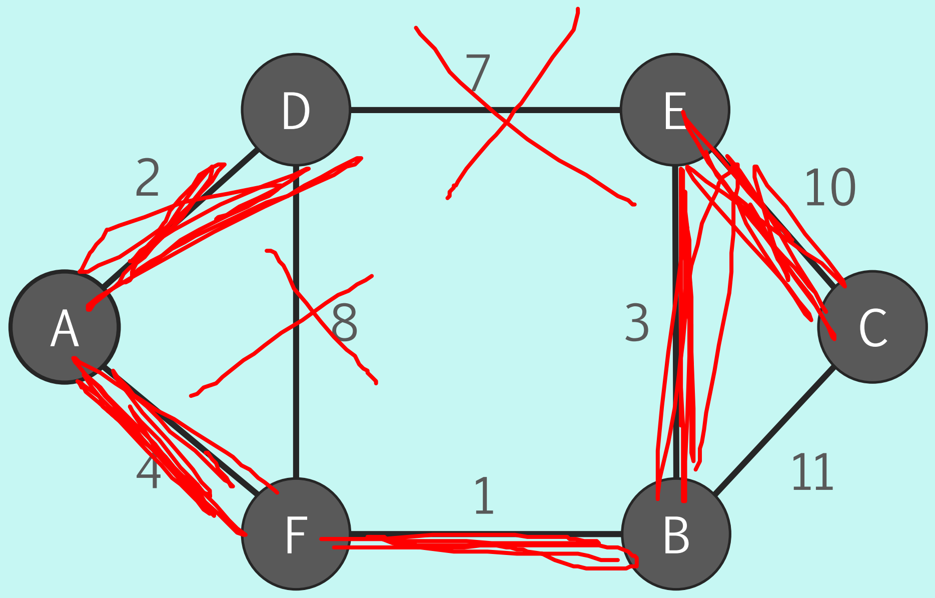
- There will be questions on these slides.   Please have a clean piece of paper to write your answers.   Write your name on the top right corner for our record.   At the end of lecture, we will collect these pieces of paper for your participation grade. Scribes should get ready to scribe.

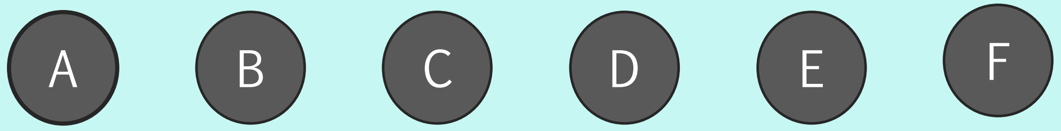
Announcements

- Scribes please
  - Write names on white board to remember
- Homework 8 Graphs is Due Thur

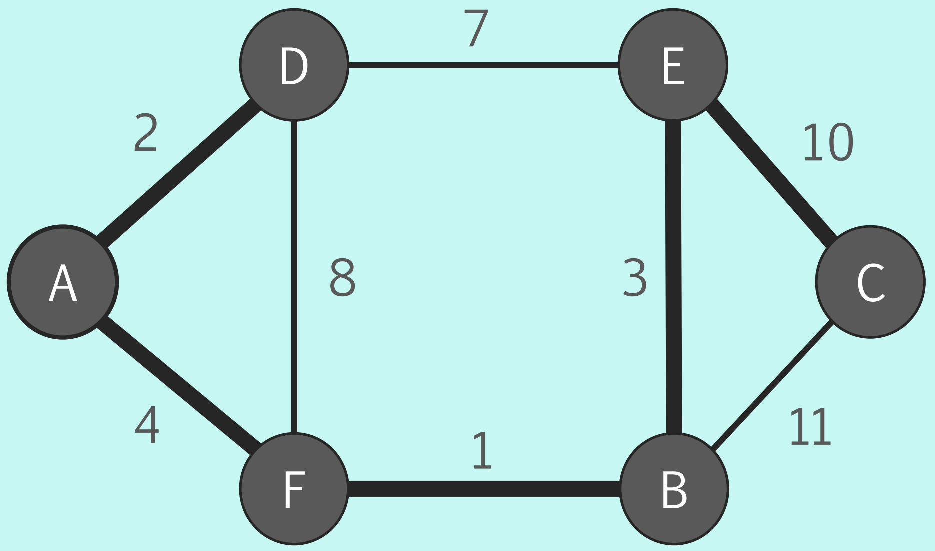
Last Time…

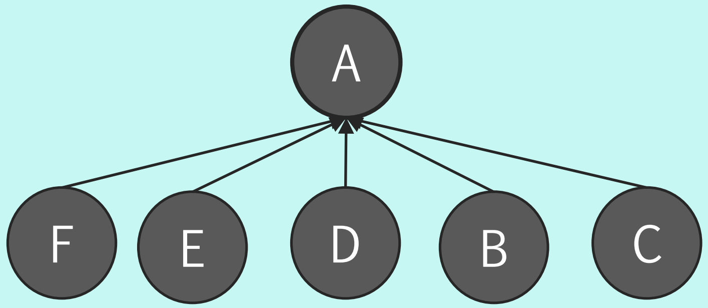
- Take 5 min to write down a the following:
  - Perform Kruskal's w/ disjoint sets
    - If two sets have the same rank, the node with the former alphabetical character comes first.   A <- B





- Take 5 min to write down a the following:
  - Perform Kruskal's w/ disjoint sets
    - If two sets have the same rank, the node with the former alphabetical character comes first. A <- B





Fibonacci

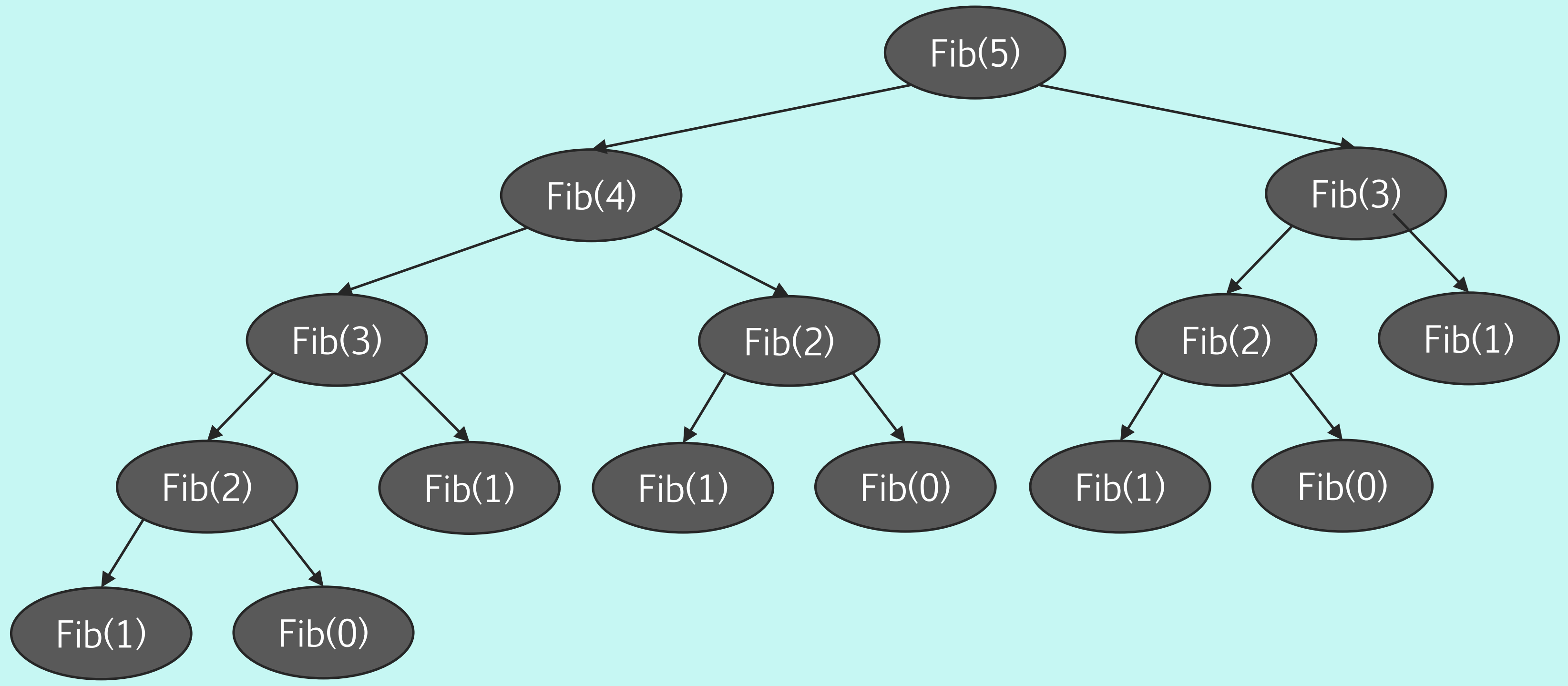
- Fibonacci is a recursive equation in the form of:

  -
    ```
    Fib(n) = Fib(n-1) + Fib(n-2)
    Fib(0) = 0
    Fib(1) = 1
    ```
- The code would look something like this:

-
  ```
  int Fib(n):
    if (n == 0) return 0;
    if (n == 1) return 1;
    ans = Fib(n-1) + Fib(n-2)
    return ans
  ```

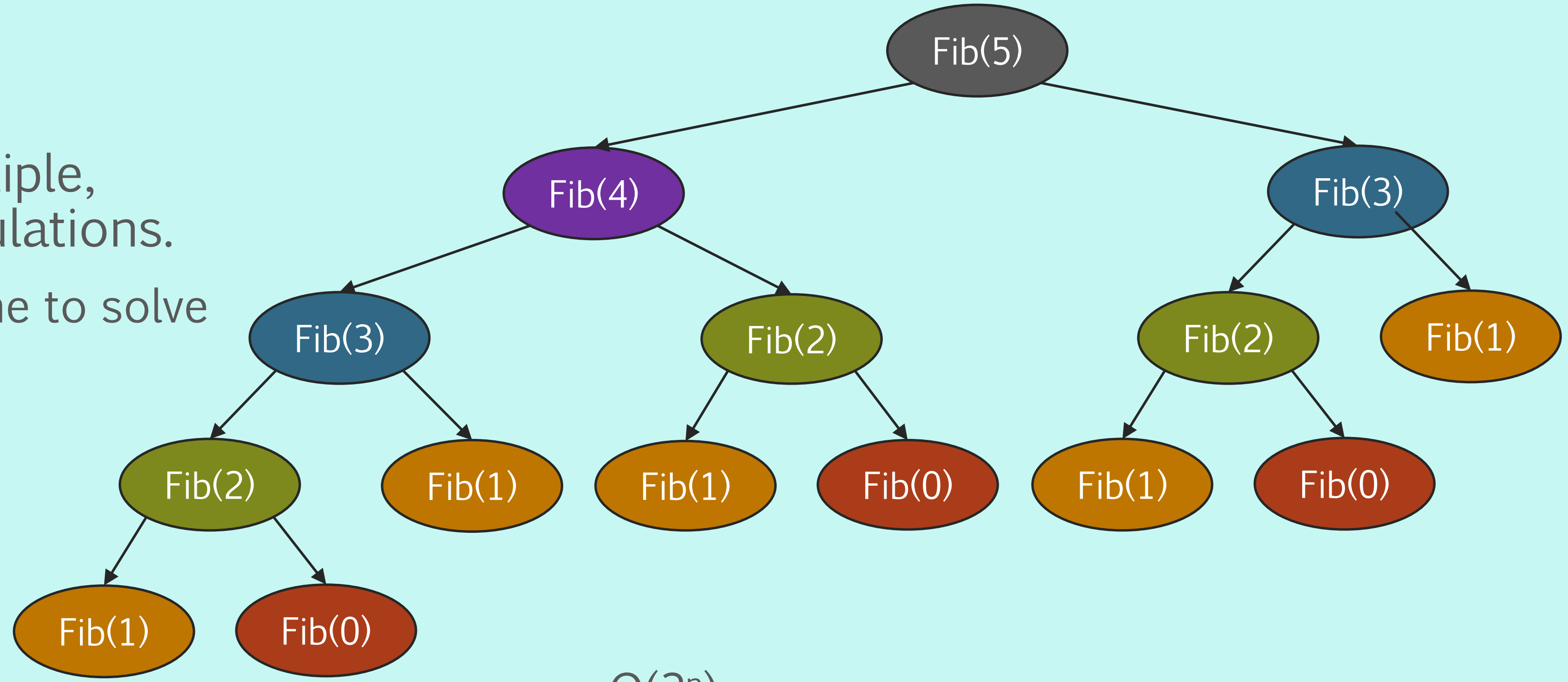
Fibonacci Recursive

- `Fib(5);`



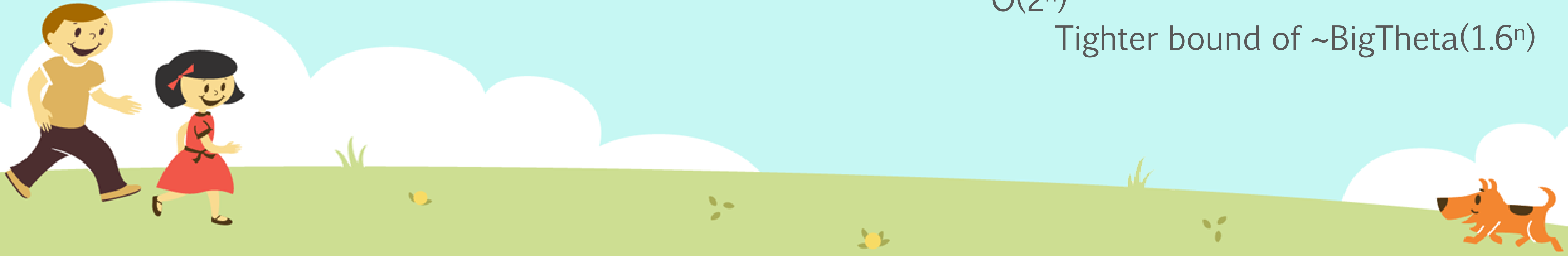
Fibonacci Recursive

- `Fib(5);`
- There are multiple, repeated calculations.
  - Excessive time to solve this.



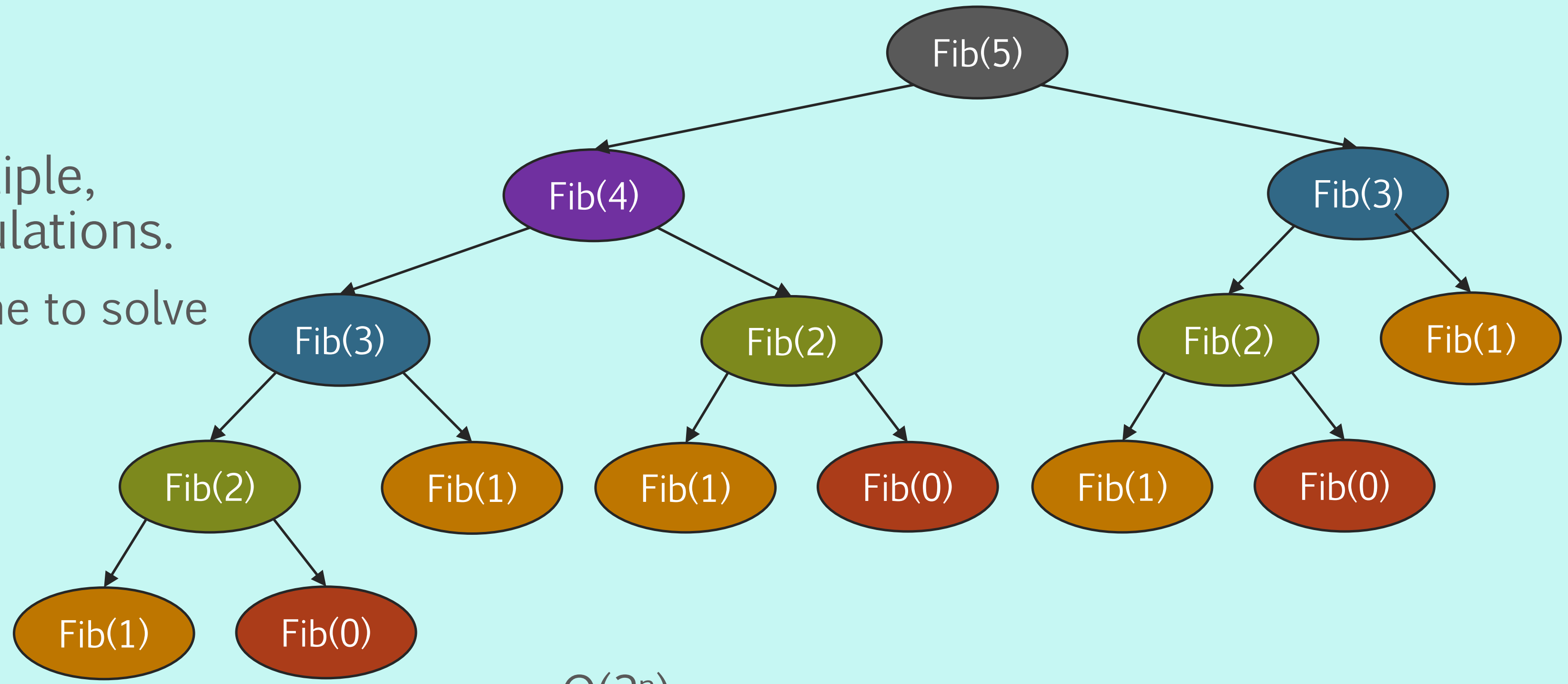
$O(2^n)$

Tighter bound of ~BigTheta(1.6^n)



Fibonacci Recursive

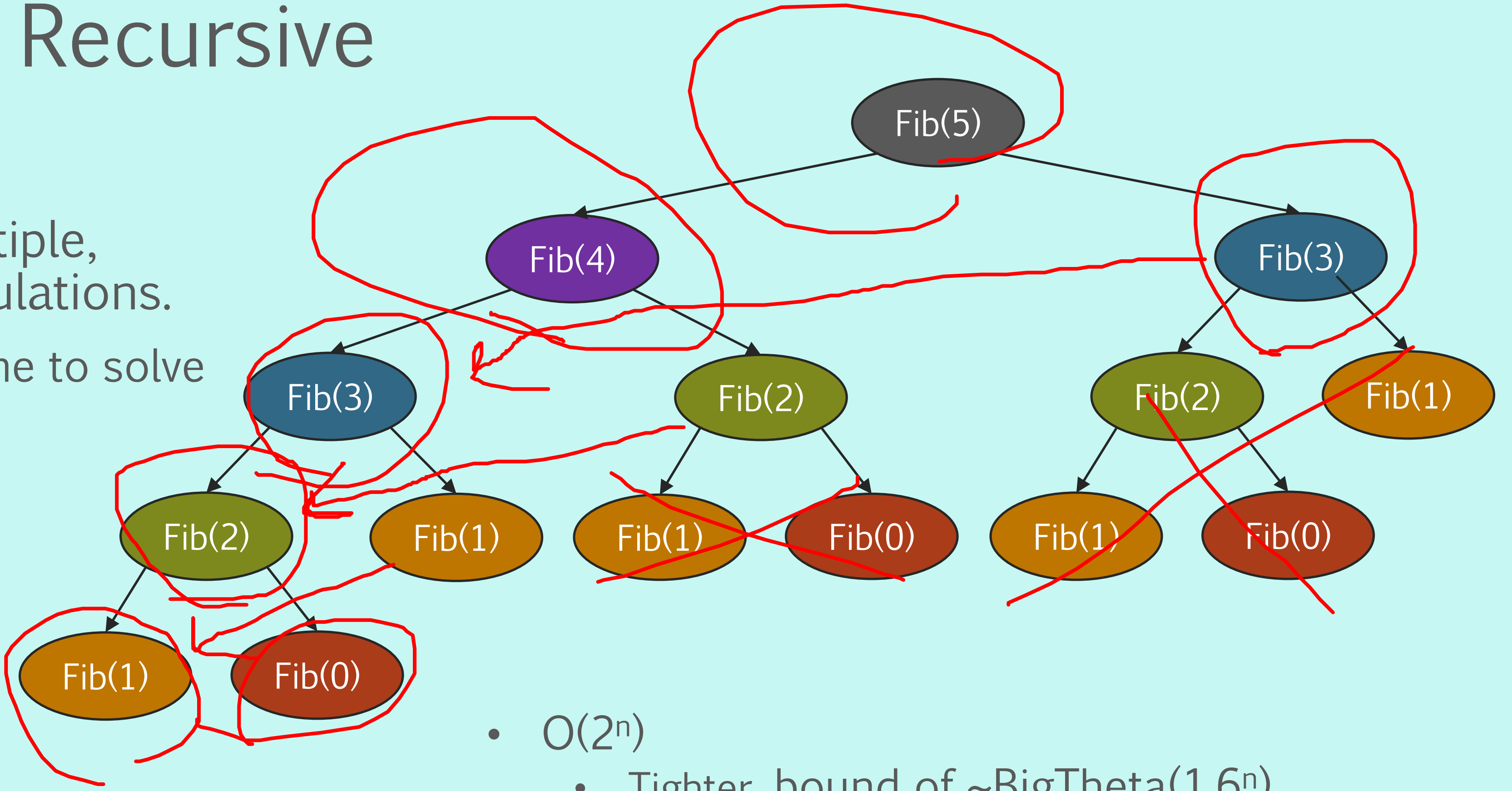
- `Fib(5);`
- There are multiple, repeated calculations.
  - Excessive time to solve this.



- $O(2^n)$
  - Tighter bound of ~BigTheta(1.6^n)
- What if we saved

Fibonacci Recursive

- `Fib(5);`
- There are multiple, repeated calculations.
  - Excessive time to solve this.



- $O(2^n)$
  - Tighter bound of ~BigTheta(1.6^n)
- What if we saved

Dynamic Programming

- A method of problem solving by breaking down complex problems into smaller sub-problems.
- Dynamic Programming has two parts:
  - Optimal Substructure – DP is usually applied to **optimization problems**: problems requiring some minimum or maximum value. The optimal solution to a problem contains the optimal solution to subproblems.
  - Overlapping Sub-problems – Sub-problems are revisited repeatedly when solving the problem.

Dynamic Programming

- There are two types of Dynamic Programming:
  - Top-down – Solve from the largest problem to smaller problems. Along the way, if a sub-problem has not been solved, then continue solving the sub-problem with deeper sub-sub-problems. Once a sub-problem has been solved, **memoize** or save the value in some table. When attempting to solve a sub-problem, check to see if the solution has been memoized.
    - Saving the solution to a sub-problem is called **memoization**.*
  - Bottom-up – Start with the smaller problems and solve increasingly larger problems. The results of smaller problems are saved in a table to compute a larger sub-problem.

**Memoization = taking a memo*

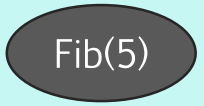
Top-Down Fibonacci

```
TDFib(n):
   memo[0 … n] = -1
   memo[0] = 0
   memo[1] = 1
   return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
   if (memo[n] != -1) return memo[n]
   nm1 = TDFibRecurse(n-1, memo[])
   nm2 = TDFibRecurse(n-2, memo[])
   memo[n] = nm1 + nm2
   return memo[n]
```

Trace through the code on paper for TopDownFib(5)

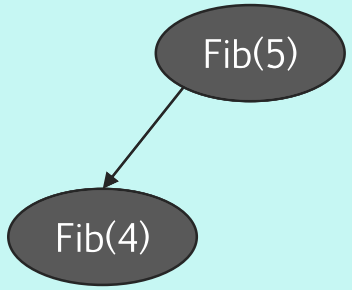
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



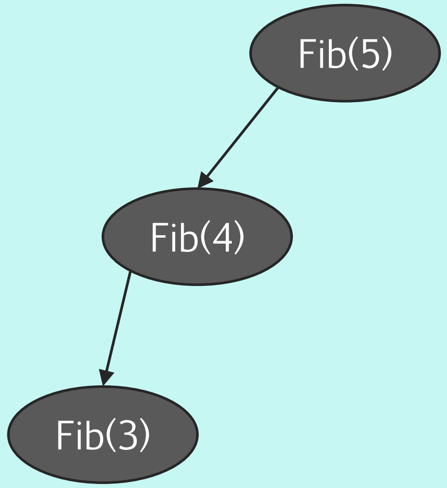
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



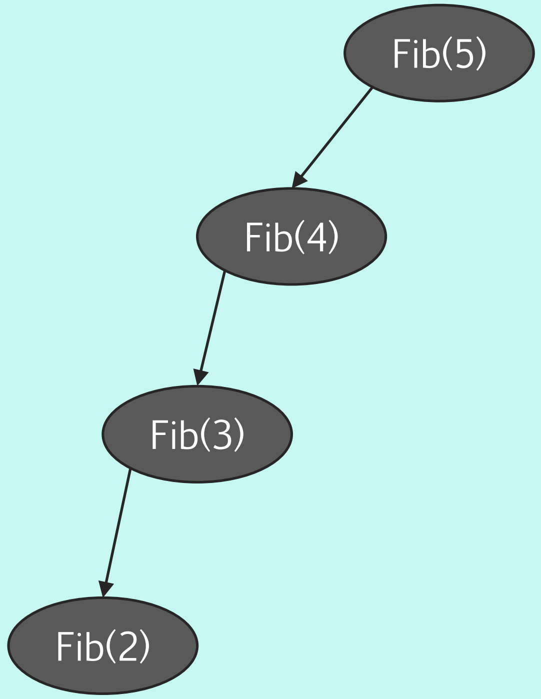
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



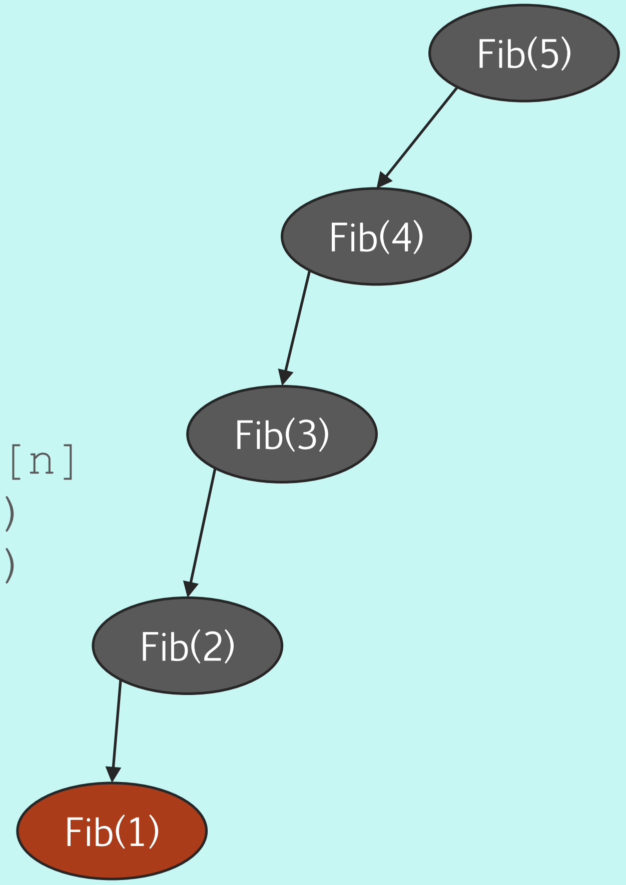
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  **return** TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



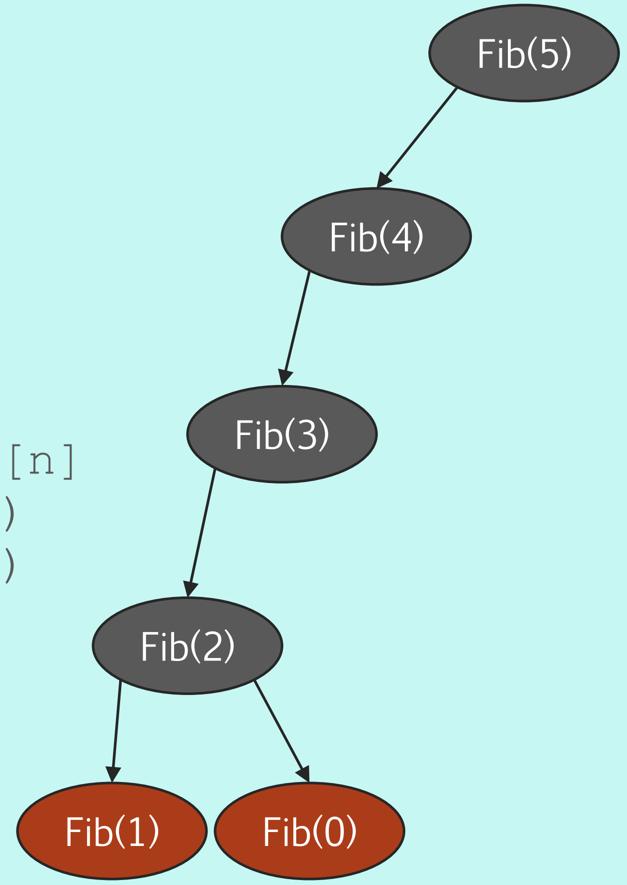
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



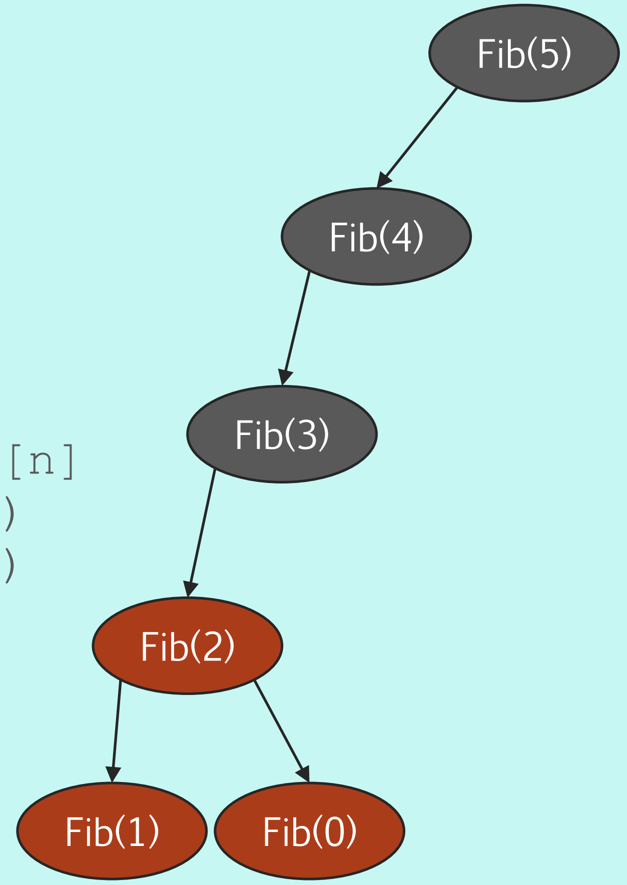
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Top-Down Fibonacci

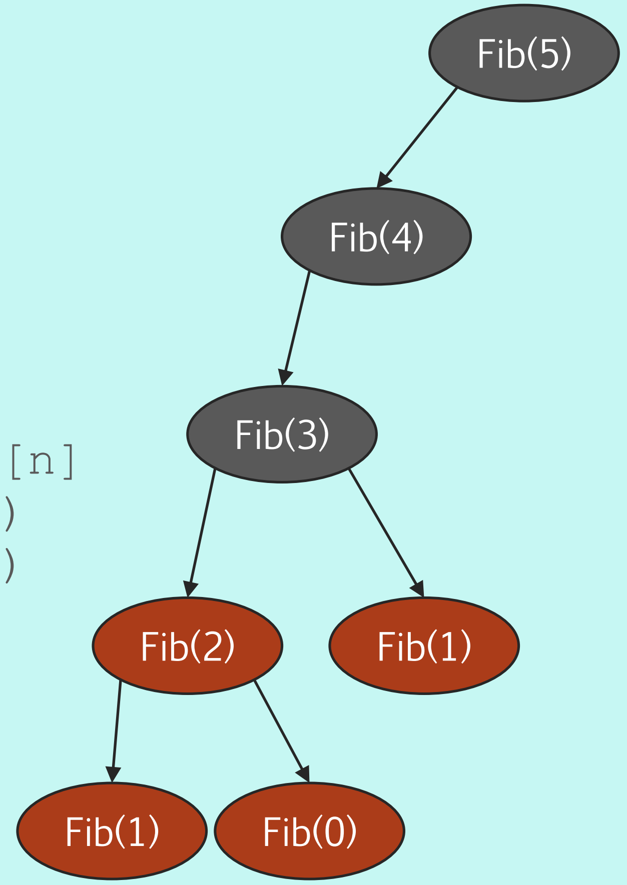
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Fib(2) is calculated and memoized

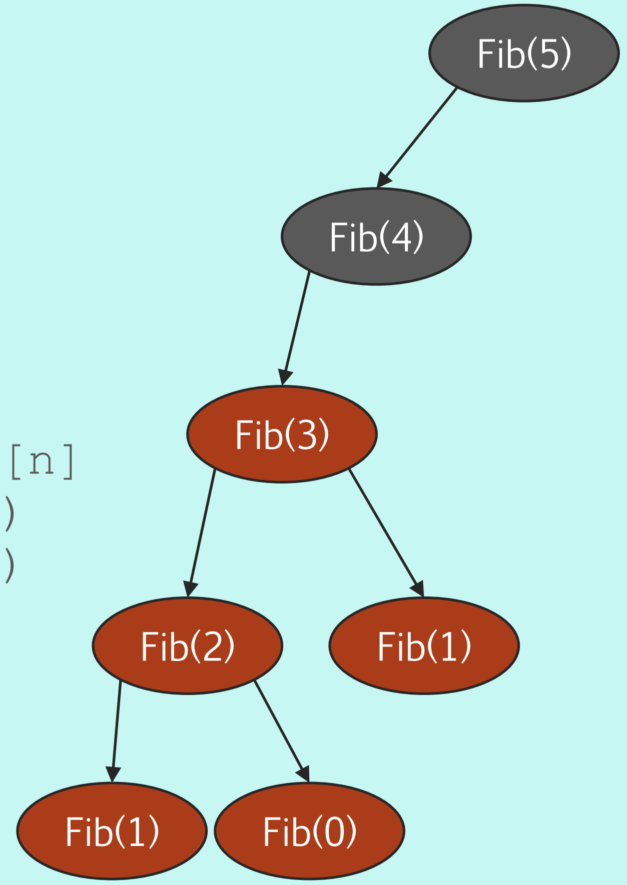
Top-Down Fibonacci

```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Top-Down Fibonacci

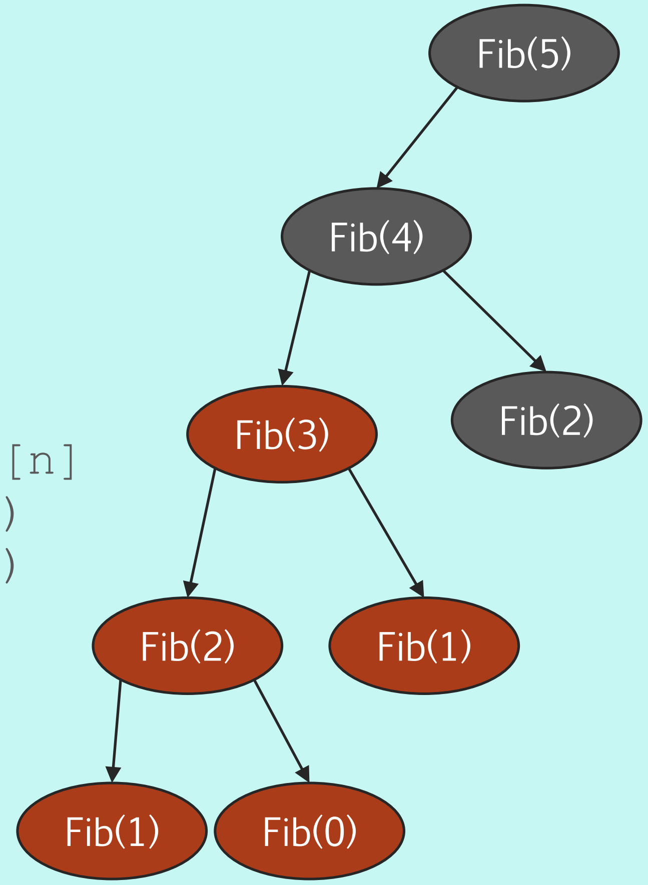
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Fib(3) is calculated and memoized.

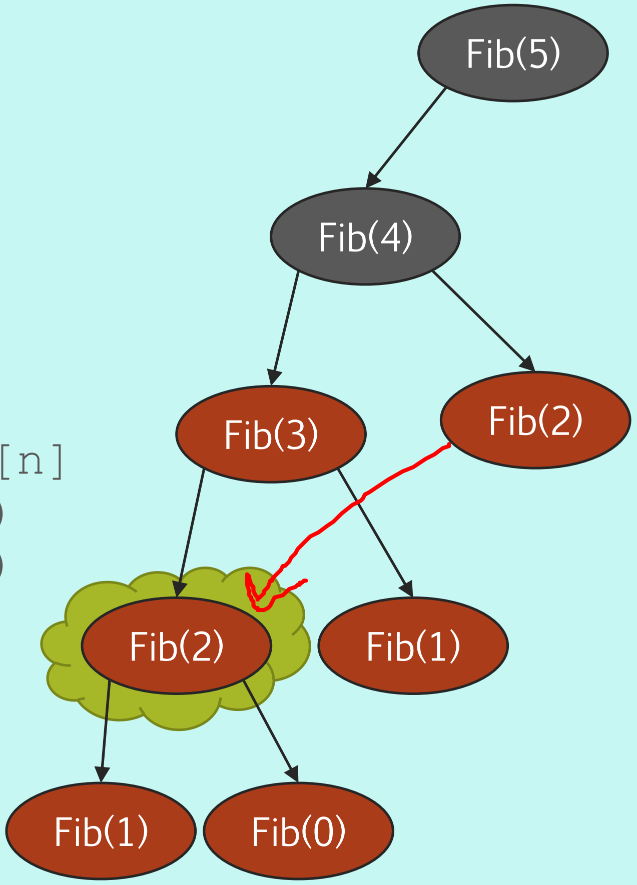
Top-Down Fibonacci

```
TDFib(n):
    memo[0 … n] = -1
    memo[0] = 0
    memo[1] = 1
    **return** TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
    **if** (memo[n] != -1) return memo[n]
    nm1 = TDFibRecurse(n-1, memo[])
    nm2 = TDFibRecurse(n-2, memo[])
    memo[n] = nm1 + nm2
    **return** memo[n]
```



Top-Down Fibonacci

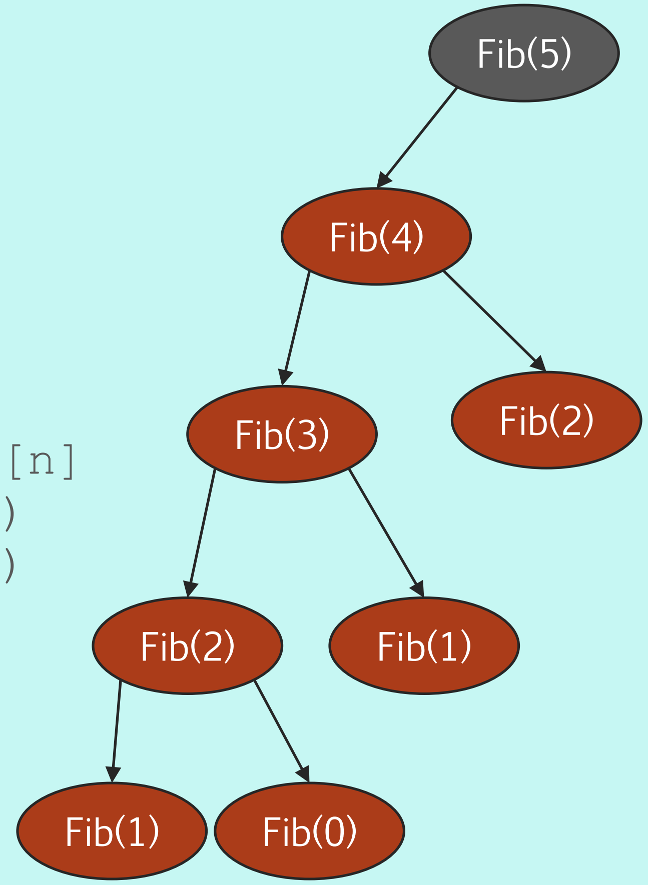
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Fib(2) was already calculated and memoized earlier, so we don't need to recompute Fib(2)

Top-Down Fibonacci

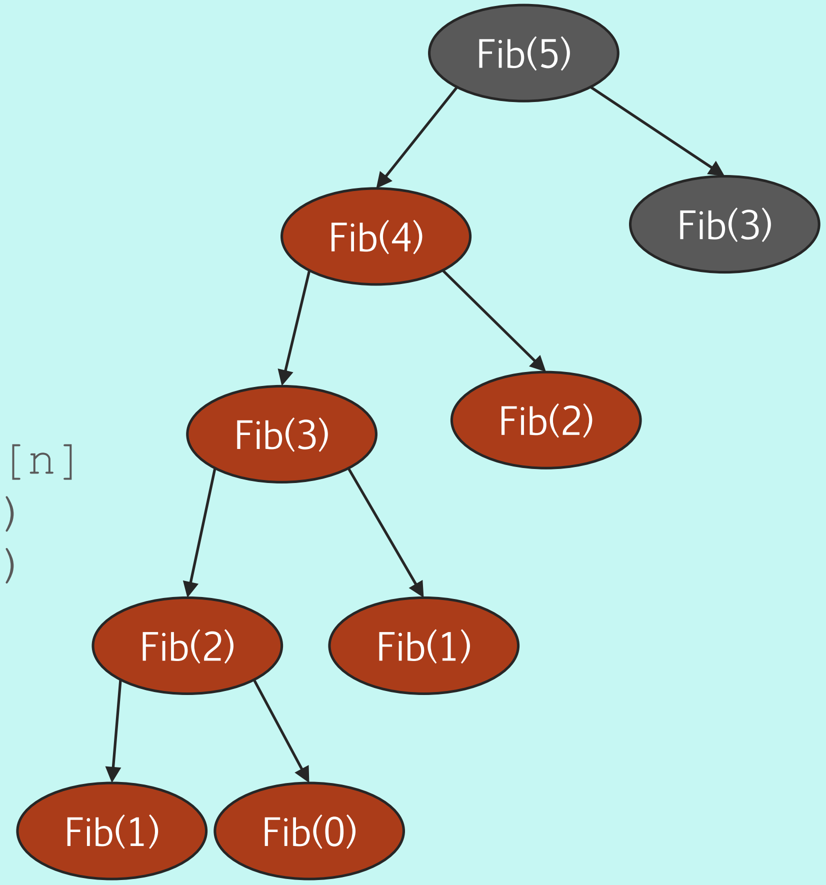
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Fib(4) is calculated and memoized.

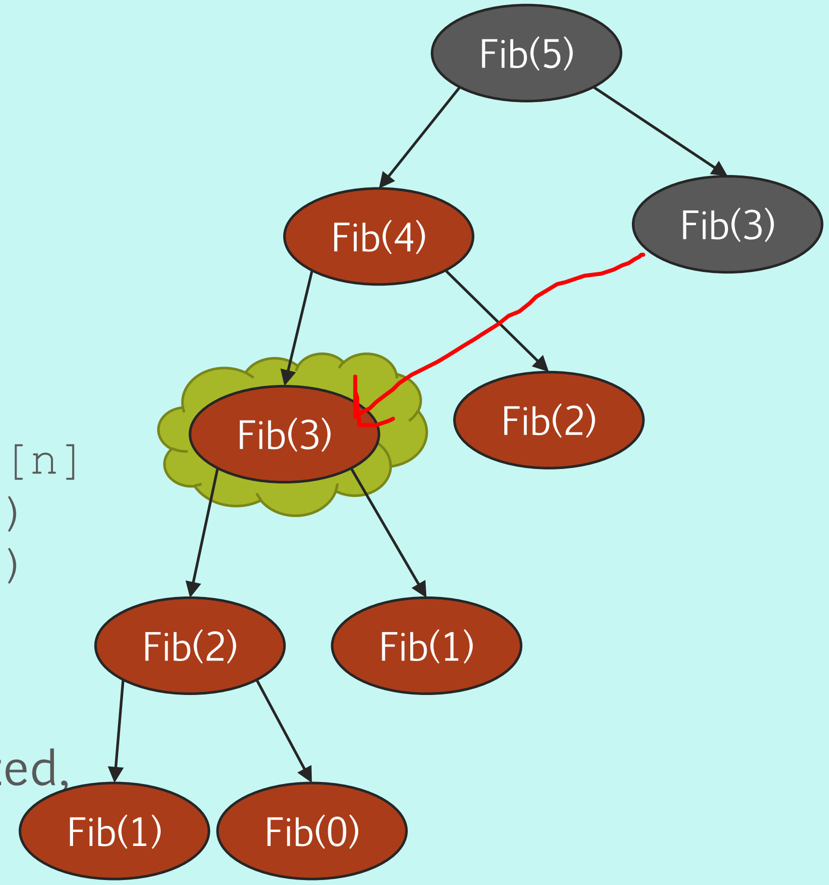
Top-Down Fibonacci

```
TDFib(n):
    memo[0 … n] = -1
    memo[0] = 0
    memo[1] = 1
    return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
    if (memo[n] != -1) return memo[n]
    nm1 = TDFibRecurse(n-1, memo[])
    nm2 = TDFibRecurse(n-2, memo[])
    memo[n] = nm1 + nm2
    return memo[n]
```



Top-Down Fibonacci

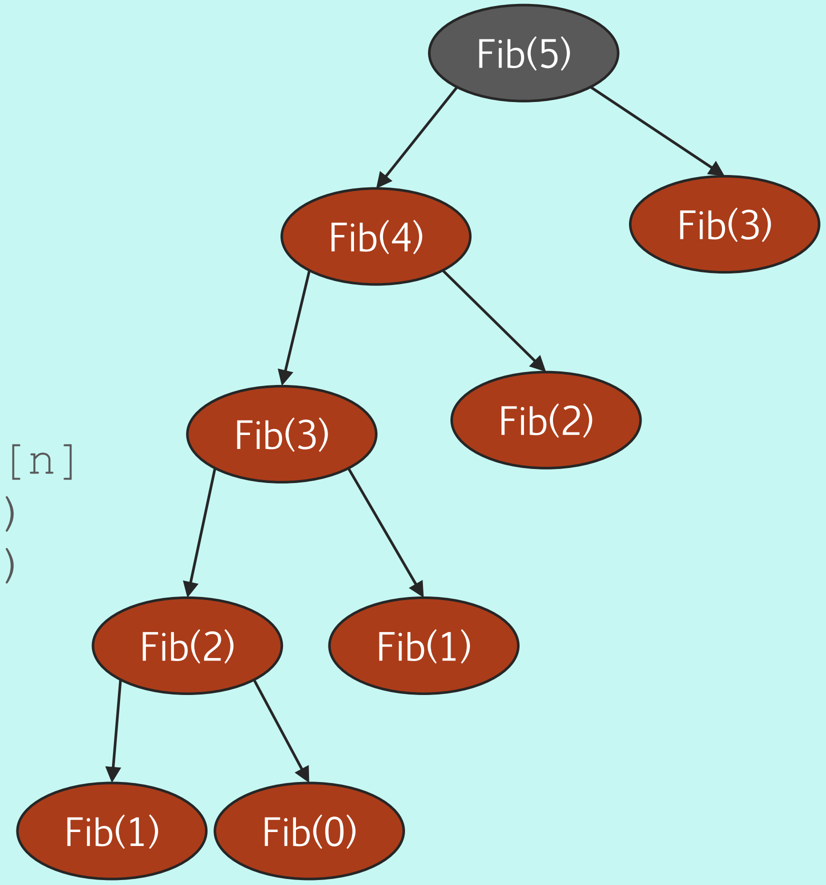
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



Fib(3) was already calculated and memoized, so we don't need to re-compute Fib(3).

Top-Down Fibonacci

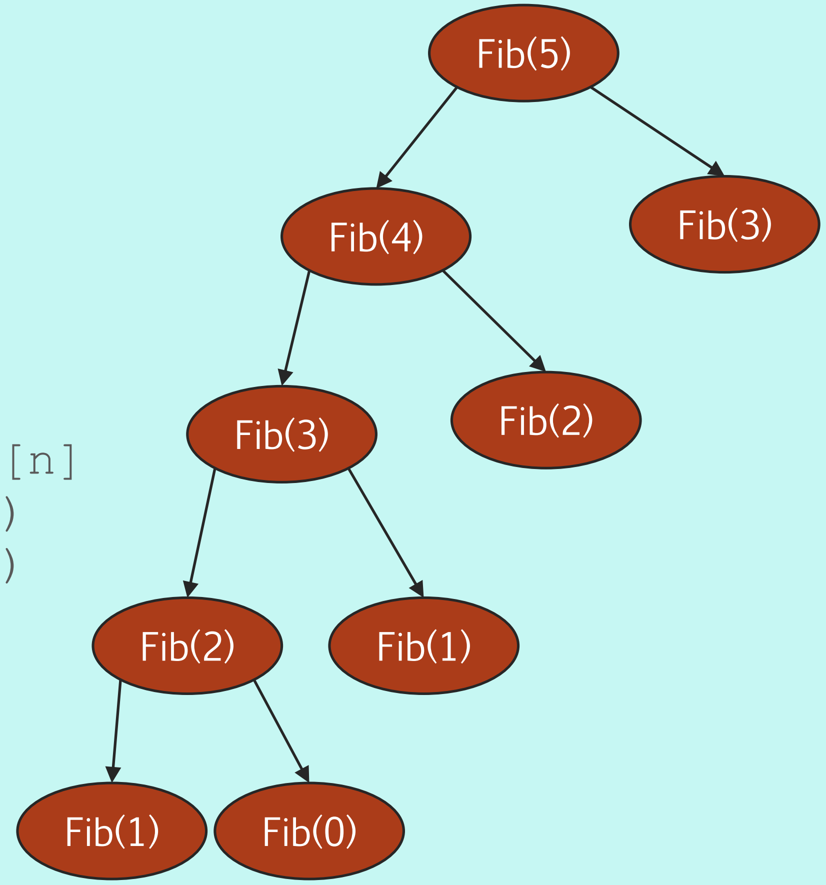
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```



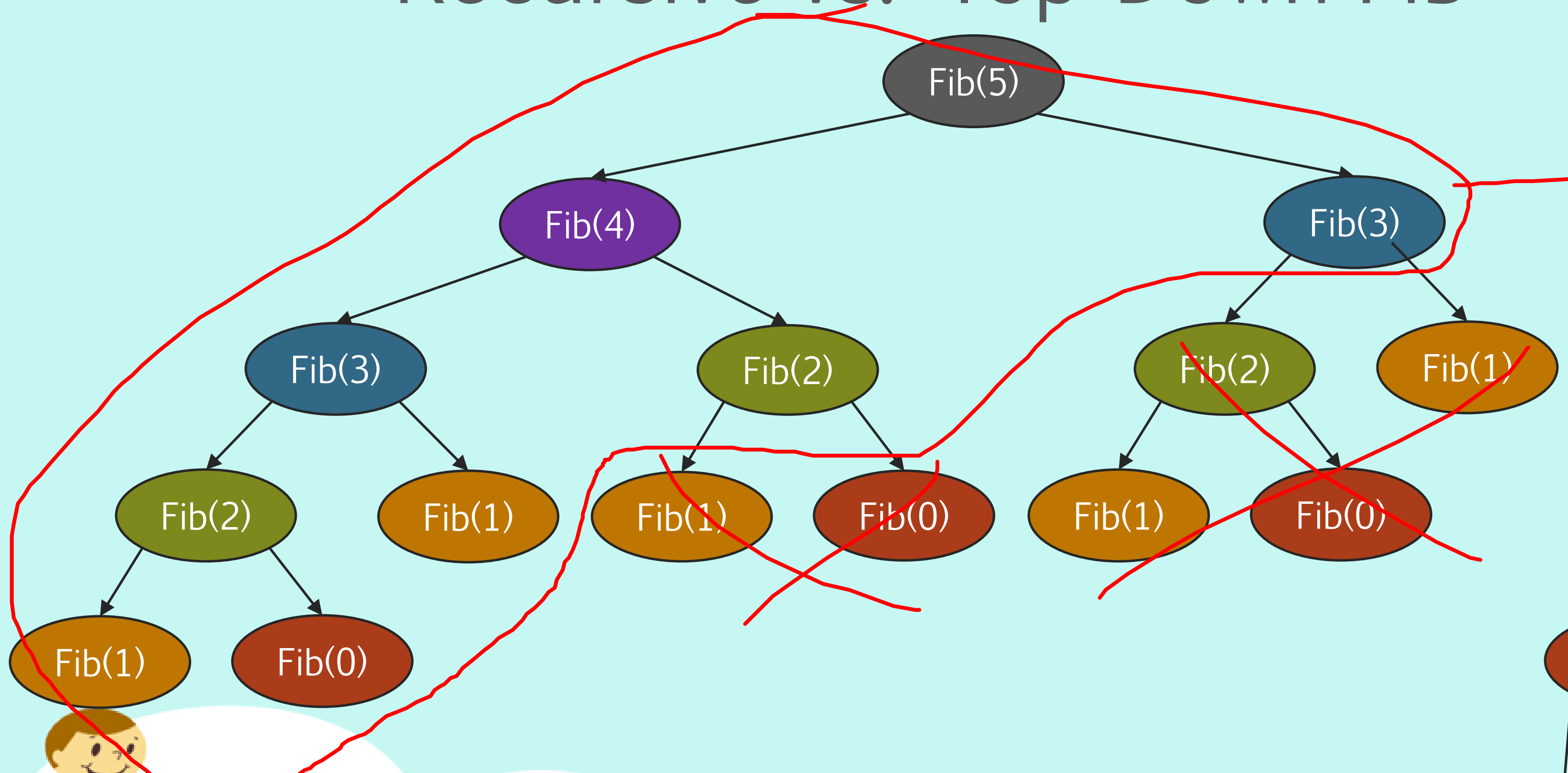
Top-Down Fibonacci

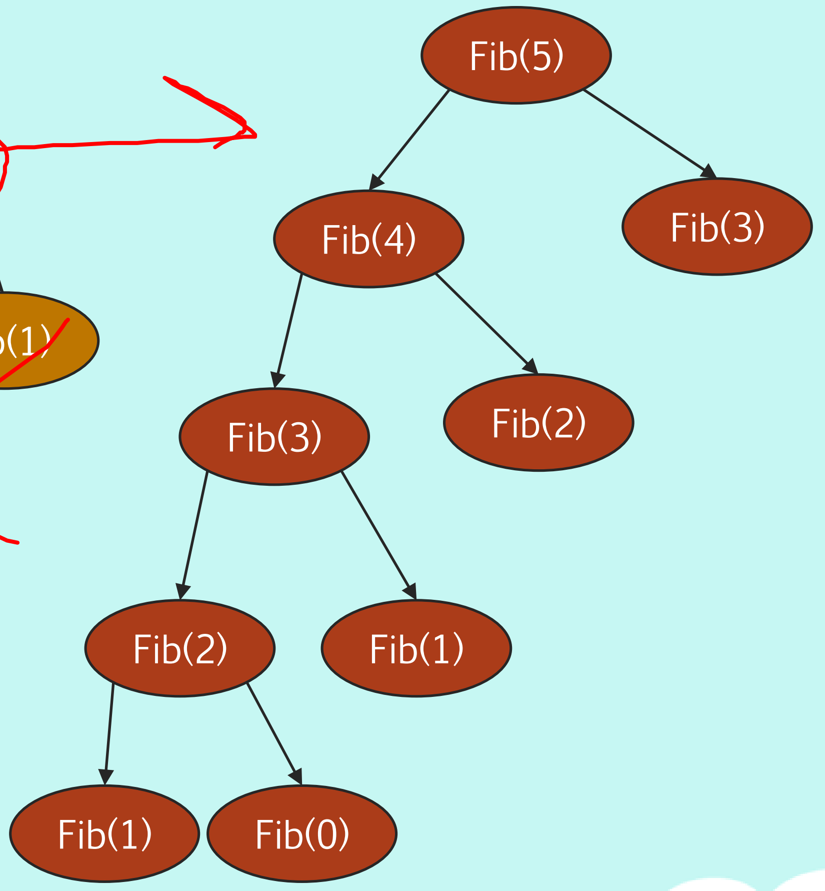
```
TDFib(n):
  memo[0 … n] = -1
  memo[0] = 0
  memo[1] = 1
  return TDFibRecurse(n, memo)

TDFibRecurse(n, memo[]):
  if (memo[n] != -1) return memo[n]
  nm1 = TDFibRecurse(n-1, memo[])
  nm2 = TDFibRecurse(n-2, memo[])
  memo[n] = nm1 + nm2
  return memo[n]
```

We now compute Fib(5).



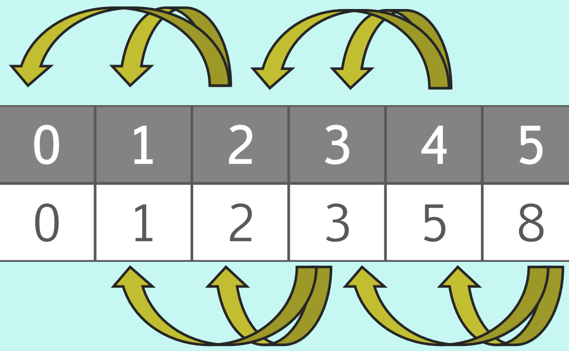
Recursive vs. Top-Down Fib





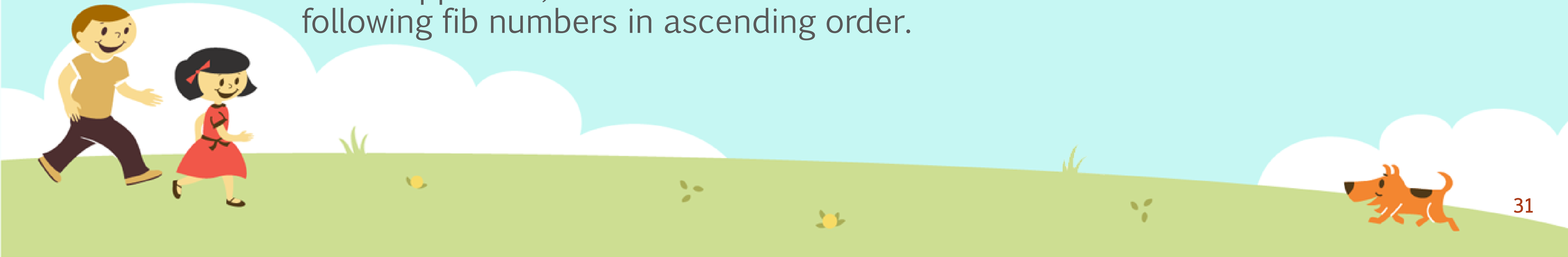
Bottom-Up Fibonacci

```
BUFib(n):
  if (n == 0) return 0
  if (n == 1) return 1
  save[0 … n]
  save[0] = 0
  save[1] = 1
  for i = [2 … n]:
    save[i] = save[i-1] + save[i-2]
  return save[n]
```



i	0	1	2	3	4	5	6	7	8
Fib(i)	0	1	2	3	5	8	13	21	34

In this approach, we start with the smallest fib numbers first and solve for following fib numbers in ascending order.



Wand Cutting (NOT ON EXAM)

- Back in the day, Georgia Tech offered a degree called Wizardy and Magic. However, due to budget cuts, the track has been closed and you decide to switch to CS. Along the way, you decide a way to earn some extra cash is to sell your wand.

- While sitting in CS 1332, you find out your instructor is buying old wands either cut up or whole. He shows you the prices of sections of wand you can sell.

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

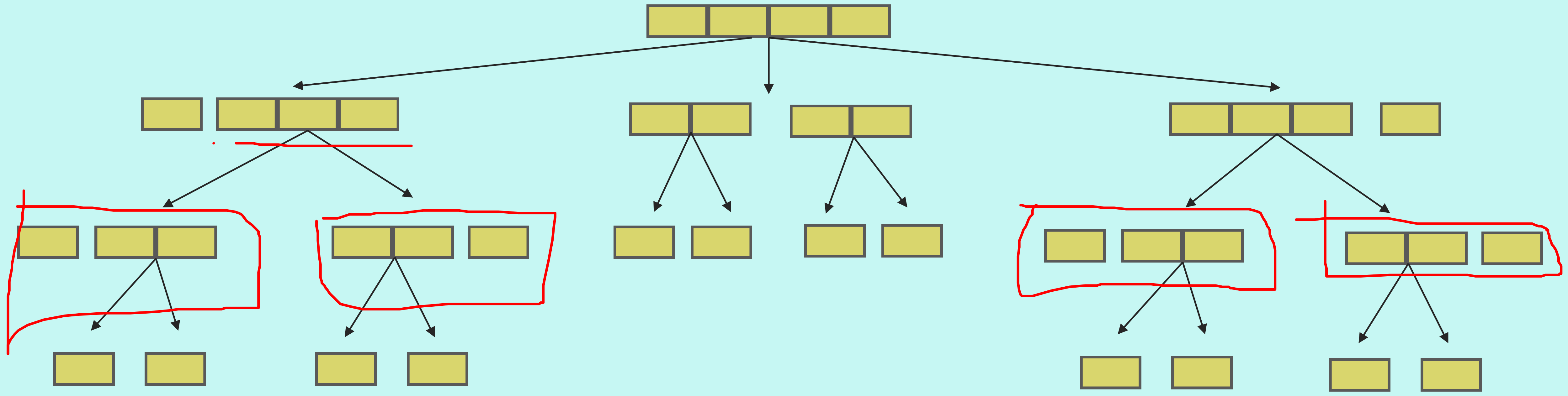
- Given your wand of length n, what's the most money you can make from cutting up your wand?

Wand Cutting

- Say n = 4, there are multiple ways to cut the wand.
  - Sell as (4) inch $9
  - Sell as (2, 2) inch: $5 + $5 = $10
  - Sell as (1, 1, 2) inch: $1 + $1 + $5 = $7
- Since you're a CS major, you decide to write a program to figure out how much you can make.

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Wand Cutting Options



Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

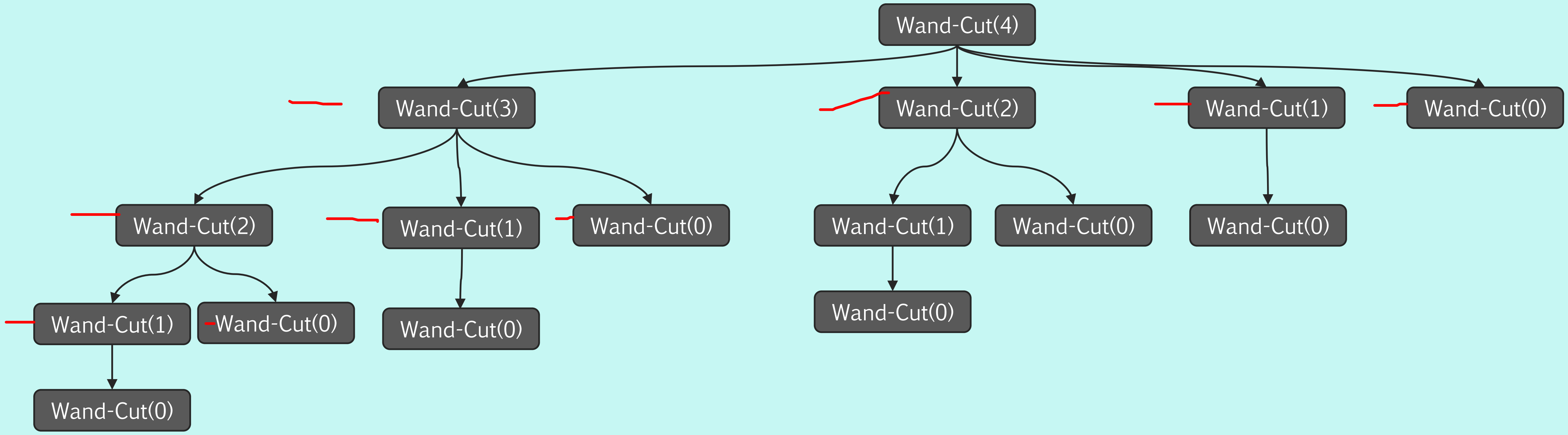
Wand Cutting Code

- We need to explore every combination of cuts to see what's the best.

```
WandCut(prices, n):
  if (n == 0)
    return 0
  maxProfit = -1
  for i = [1 … n]:
    maxProfit = max(maxProfit, prices[i] + WandCut(prices, n-i))
  return maxProfit
```

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Wand Cutting Tree



Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Wand Cutting Code

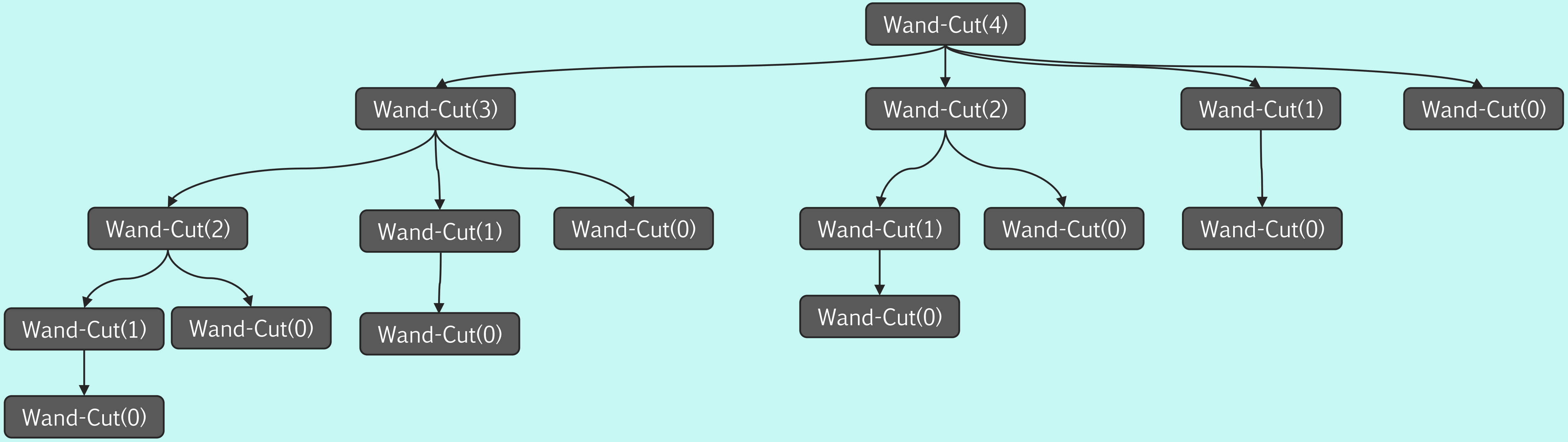
- We need to explore every combination of cuts to see what's the best.

```
WandCut(prices, n):
  if (n == 0)
    return 0
  maxProfit = -1
  for i = [1 … n]:
    maxProfit = max(maxProfit, prices[i] + WandCut(prices, n-i))
  return maxProfit
```

- This algorithm runs in O(2^n). For a wand of length n, you have n-1 possible cuts: don't cut (0) or cut (1).
  - This is the same as the number of binary combinations of n-1 bits.

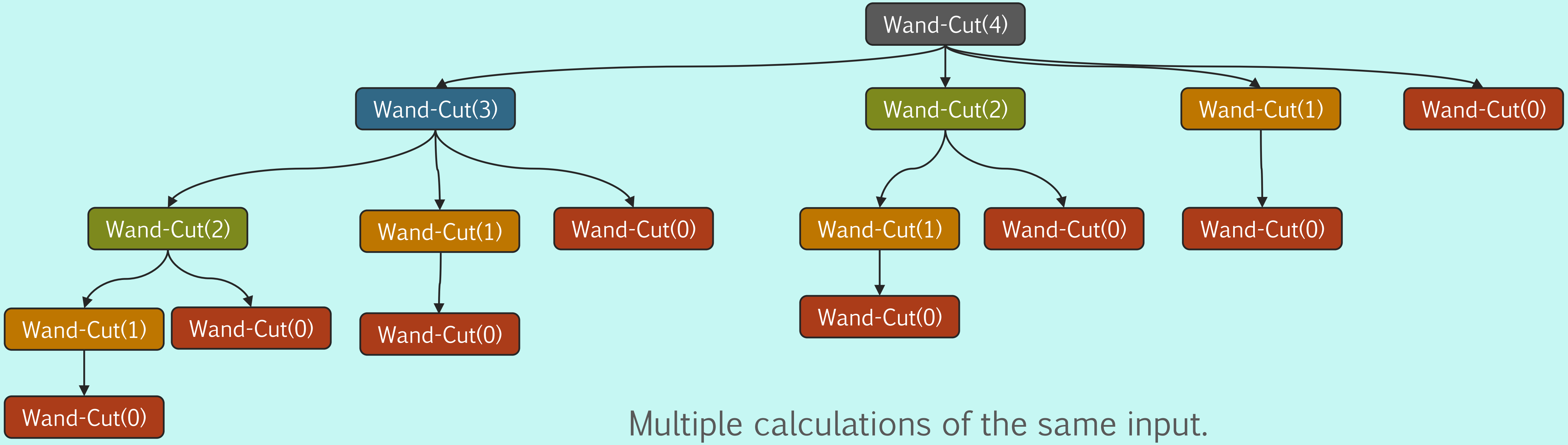
Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Wand Cutting Tree



Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Wand Cutting Tree



Multiple calculations of the same input.
This is redundant calculations.
Why not calculate one input **only once?**

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Dynamic Programming

- A method of problem solving by breaking down complex problems into smaller sub-problems.
- Dynamic Programming has two parts:
  - Optimal Substructure – DP is usually applied to **optimization problems**: problems requiring some minimum or maximum value. The optimal solution to a problem contains the optimal solution to subproblems.
  - Overlapping Sub-problems – Sub-problems are revisited repeatedly when solving the problem.

Dynamic Programming

- There are two types of Dynamic Programming:
  - Top-down – Solve from the largest problem to smaller problems. Along the way, if a sub-problem has not been solved, then continue solving the sub-problem with deeper sub-sub-problems. Once a sub-problem has been solved, **memoize** or save the value in some table. When attempting to solve a sub-problem, check to see if the solution has been memoized.
    - Saving the solution to a sub-problem is called **memoization**.*
  - Bottom-up – Start with the smaller problems and solve increasingly larger problems. The results of smaller problems are saved in a table to compute a larger sub-problem.

**Memoization = taking a memo*

Top-Down Wand Cutting

- Top-Down wand cutting is very similar to our recursive WandCut(n), except whenever we find the value of WandCut(i), where i : [0 … n], we'll save WandCut(i) in a table.
    - Whenever we encounter WandCut(i) again in our recursion, we can pull the value from the table immediately instead of calculating WandCut(i) again.
- With this idea, we will calculate WandCut(i) only once.

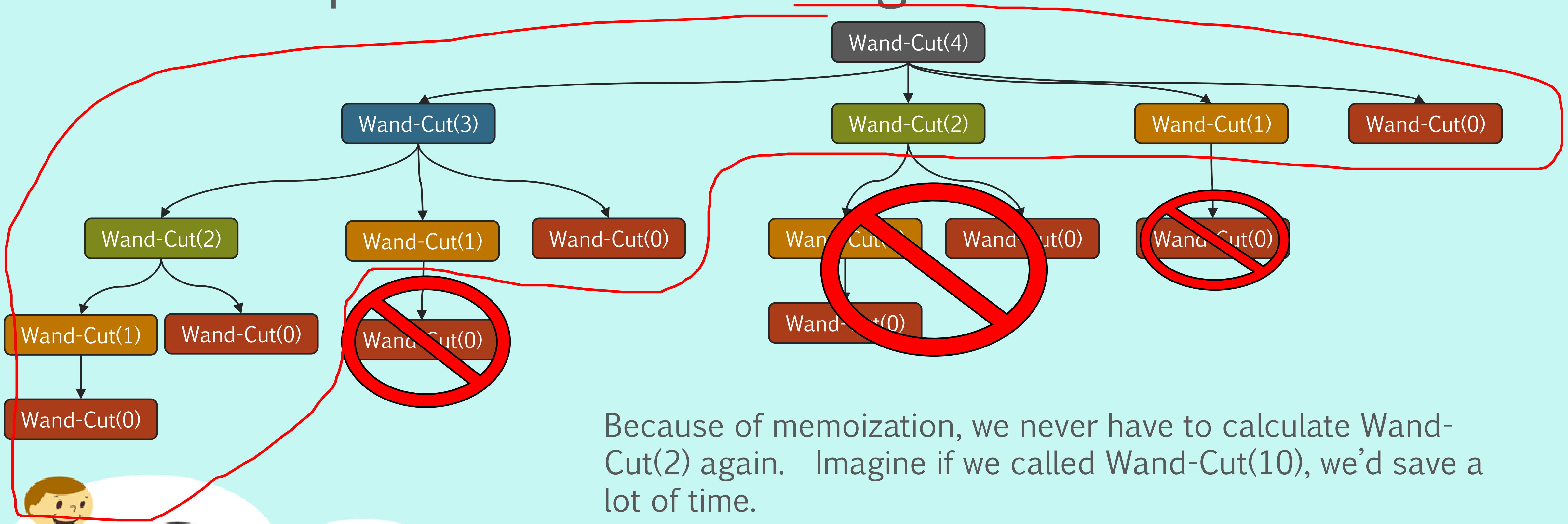
Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Top-Down Wand Cutting

```
WandCut(prices, n):
  mem[0 … n] = array          // Array for memoization
  for i = [0 … n]:            // Initialize our mem[]
    mem[i] = -1
  return TDWandCut(prices, n, mem)   // Recursive Call w/ mem

TDWandCut(prices, n, mem[])
  if (mem[n] != -1): return mem[n]   // Check for calculated value first
  if n == 0:
    p = 0
  else:
    p = -1
      for i = [1 … n]:
        p = max(p, prices[i] + TDWandCut(prices, n-i, mem))
  mem[n] = p     // Save calculated value in array.
  return mem[n]
```

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Top Down Wand Cutting Tree



Because of memoization, we never have to calculate Wand-Cut(2) again. Imagine if we called Wand-Cut(10), we'd save a lot of time.

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Bottom-Up Wand Cutting

- Bottom-Up wand cutting will start with calculating WandCut(0), WandCut(1), WandCut(2), and up till WandCut(n).
- By starting with 0, 1, 2, 3 … n, WandCut(i), where i : {0 … n}, we have already calculated WandCut(i-0), WandCut(i-1), … , WandCut(i-i).

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30

Bottom-Up Wand Cutting

```
BUWandCut(prices, n):
  save[0 … n] = array
  save[0] = 0
  for j = [1 … n]:
    p = -1
    for i = [1 … j]:
      p = max(p, prices[i] + save[j − i])
    save[j] = p
  return r[n]
```

$$\begin{tikzcd}
\text{save}[4] \arrow[d] \arrow[dd, bend right] \arrow[ddd, bend right] \arrow[dddd, bend right] \\
\text{save}[3] \arrow[d] \arrow[dd, bend left] \arrow[ddd, bend left] \\
\text{save}[2] \arrow[d] \arrow[dd, bend right] \\
\text{save}[1] \arrow[d] \\
\text{save}[0]
\end{tikzcd}$$

Length (in)	1	2	3	4	5	6	7	8	9	10
Price $	1	5	8	9	10	17	17	20	24	30