CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Lecture 8: Hashing, Dynamic Programming

Hash Tables: Chaining

How to handle the collisions?

Chaining

- The hash table is an array with each cell being a linked-list
- Given the array A[1:n]:
  - For every i, we compute b(i) = h(A[i]) and add A[i] to the tail of the linked-list at T[b(i)]
- Given x to be searched:
  - We iterate over elements of the linked-list T[h(x)] to find x or output it does not exist

Chaining : Example

- $h(x) = (x \mod 4)$+1

n=8 **m=4**

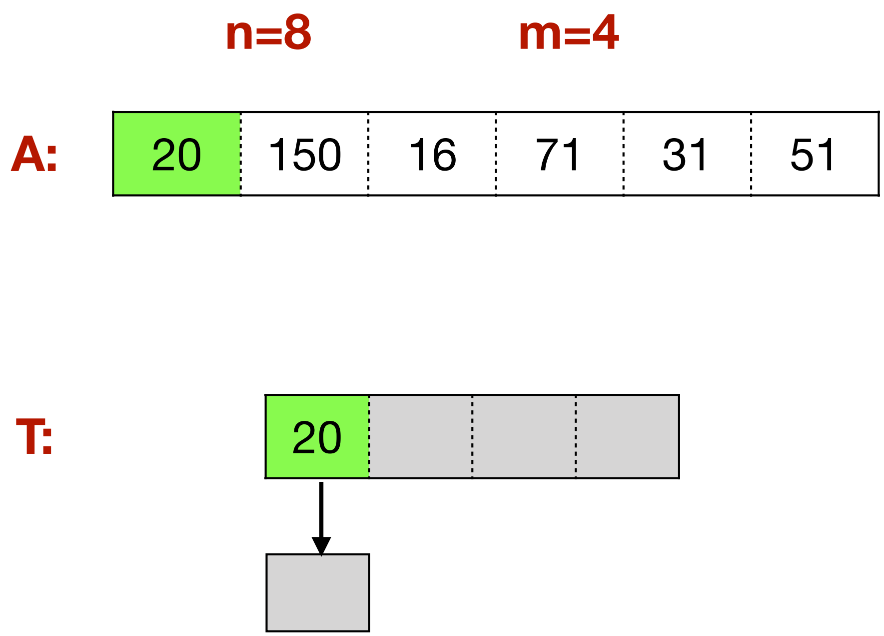
A:

20	150	16	71	31	51

T:

Chaining : Example

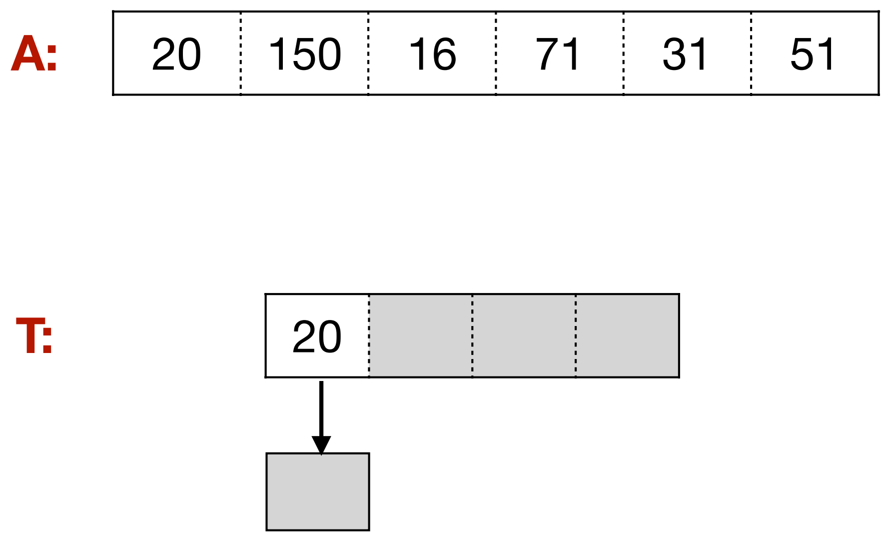
- $h(x) = (x \mod 4)+1$
- h(20) = 1



Chaining : Example

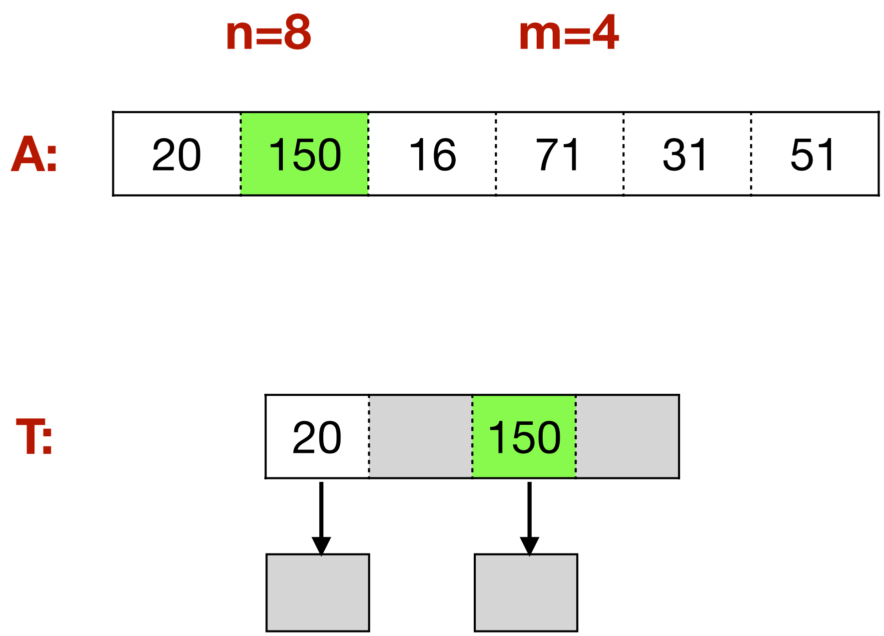
- $h(x) = (x \mod 4)+1$
- h(20) = 1

n=8 **m=4**



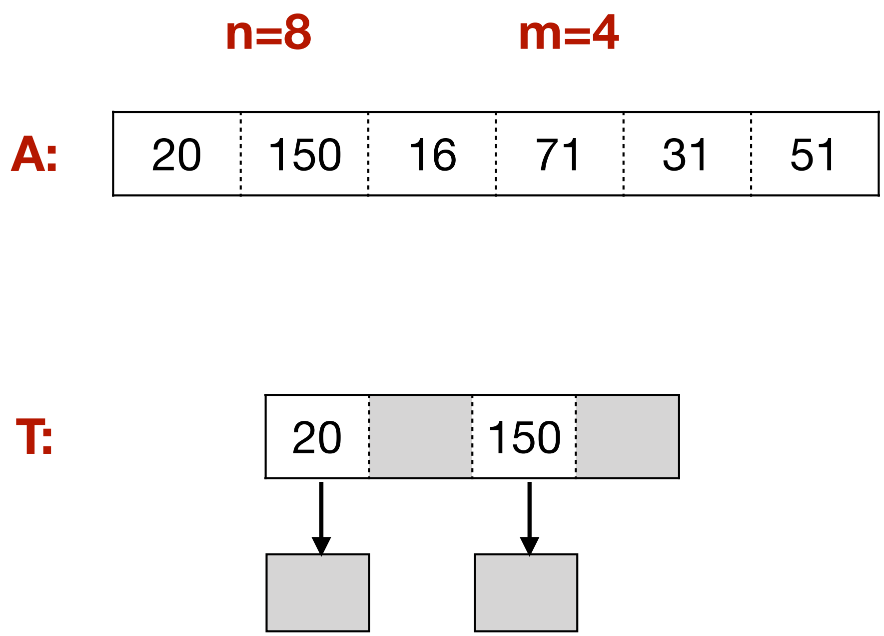
Chaining : Example

- $h(x) = (x \mod 4)$+1
- h(20) = 1
- h(150) = 3



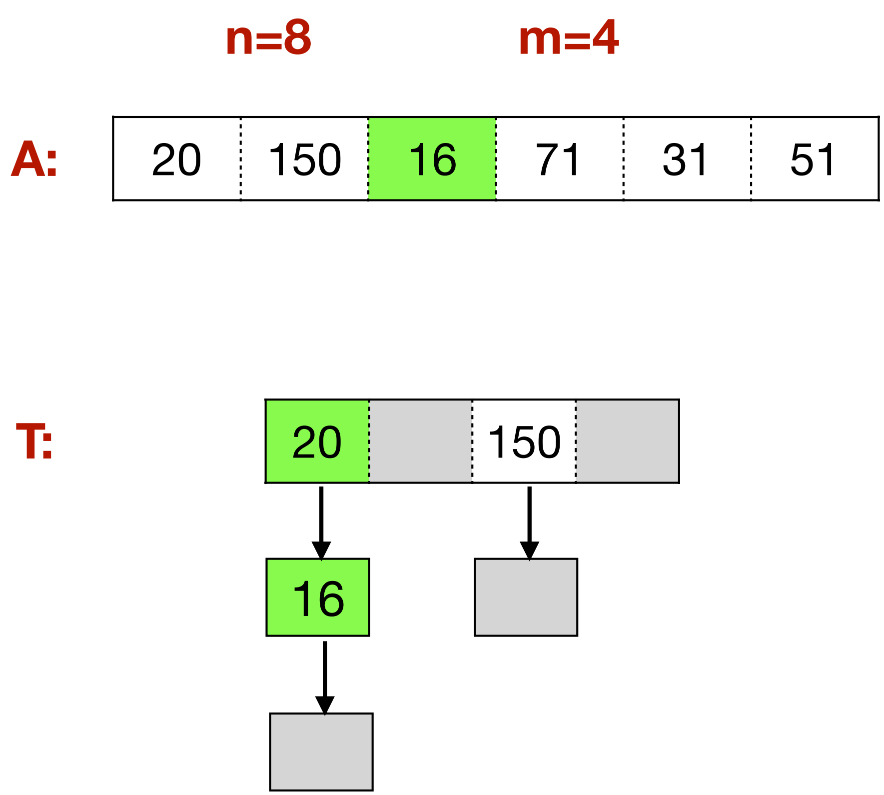
Chaining : Example

- $h(x) = (x \mod 4)+1$
- $h(20) = 1$
- $h(150) = 3$



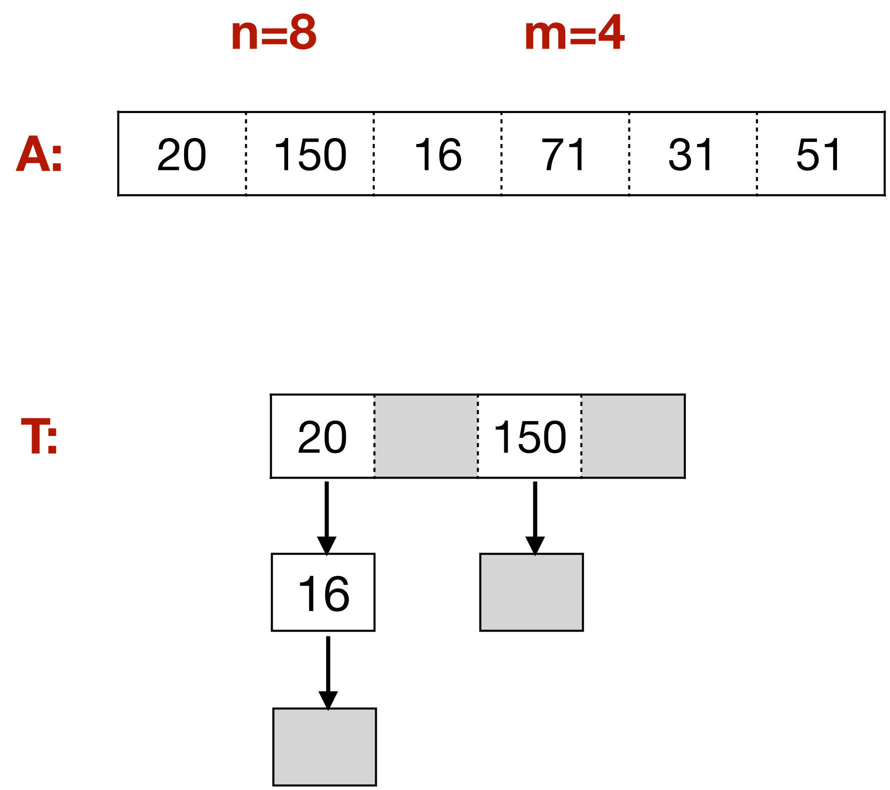
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1



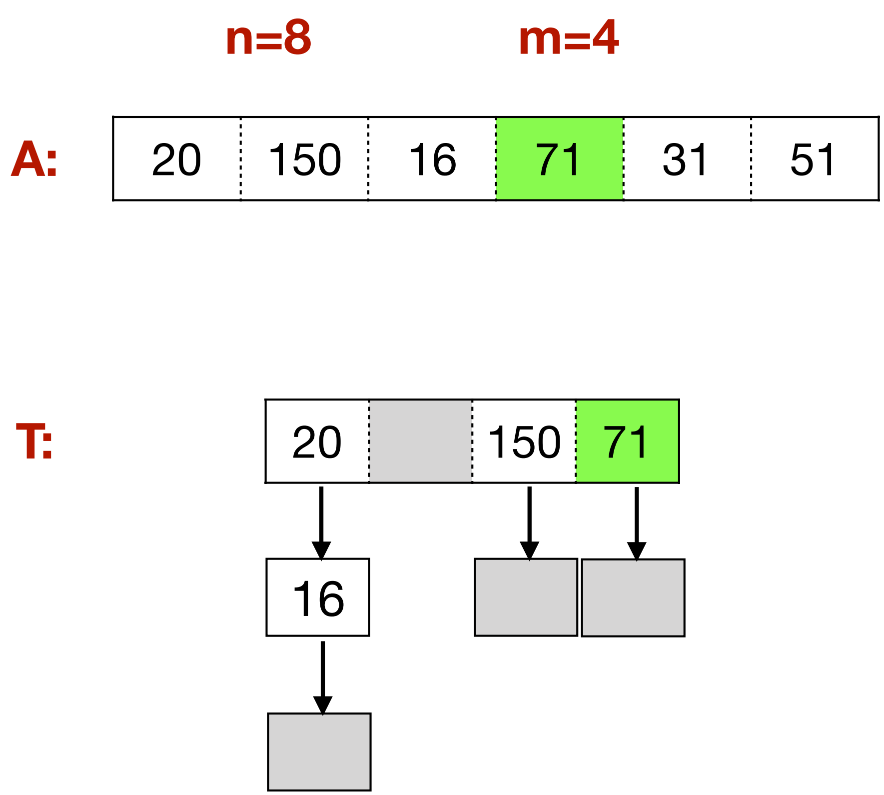
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1



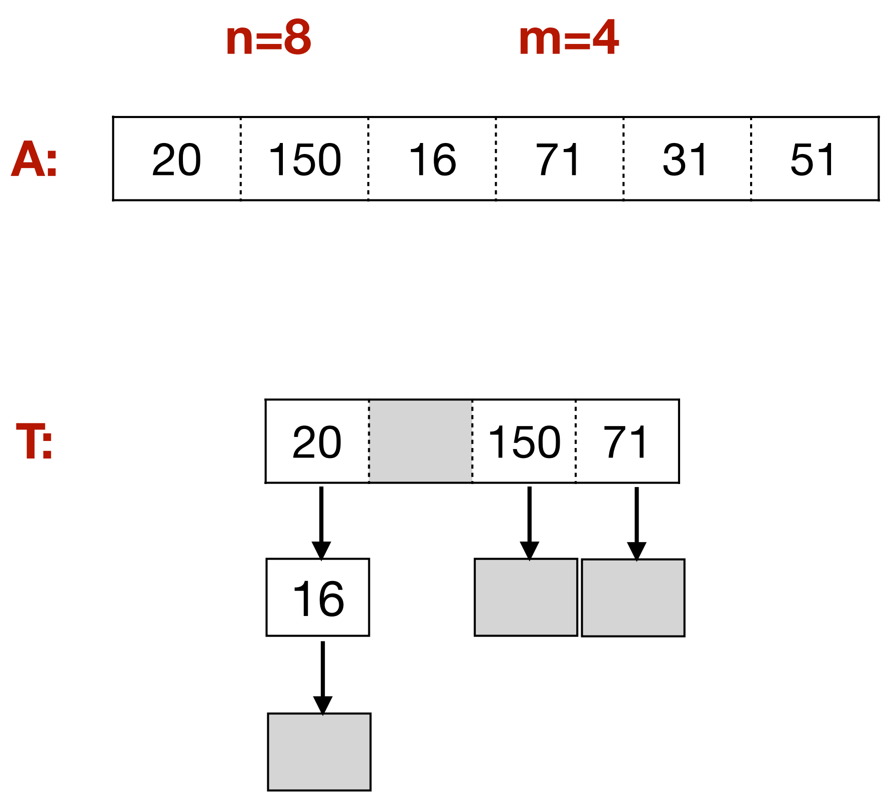
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) =1
- h(71) = 4



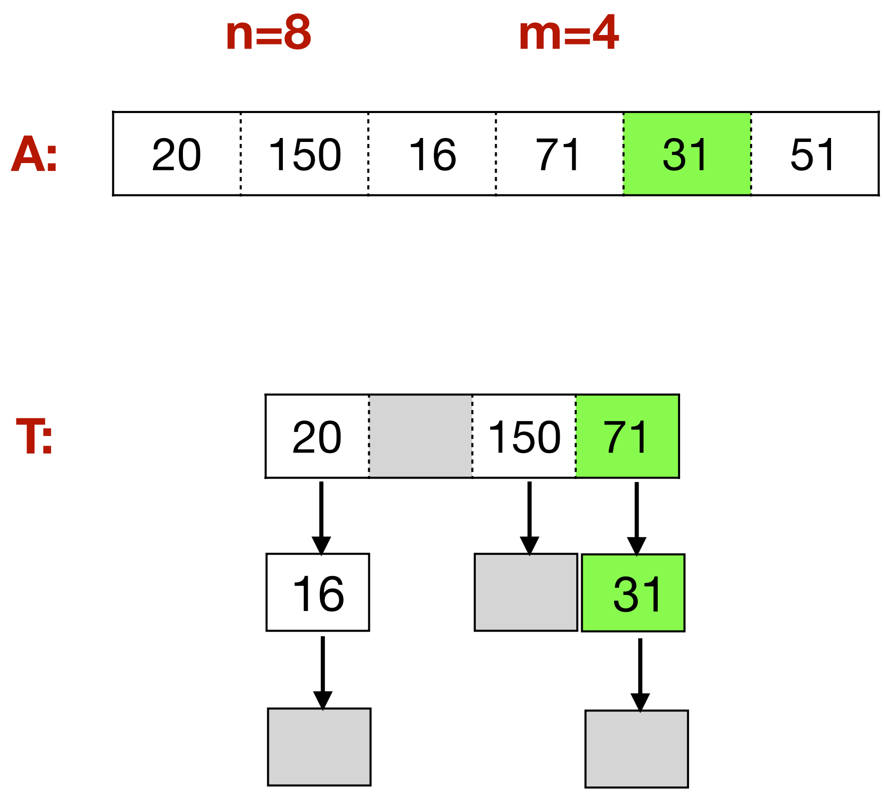
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1
- h(71) = 4



Chaining : Example

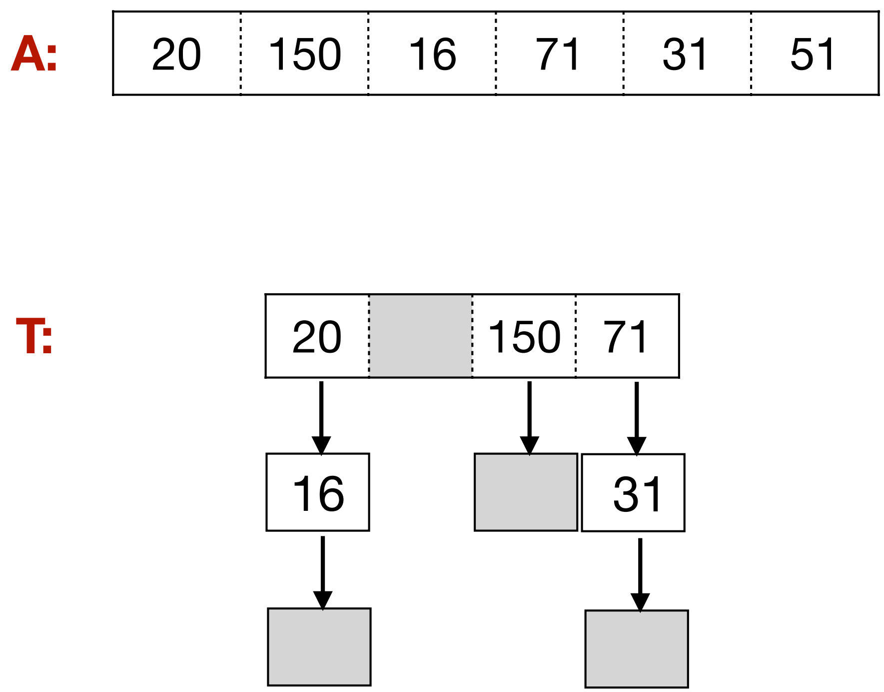
- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1
- h(71) = 4
- h(31) = 4



Chaining : Example

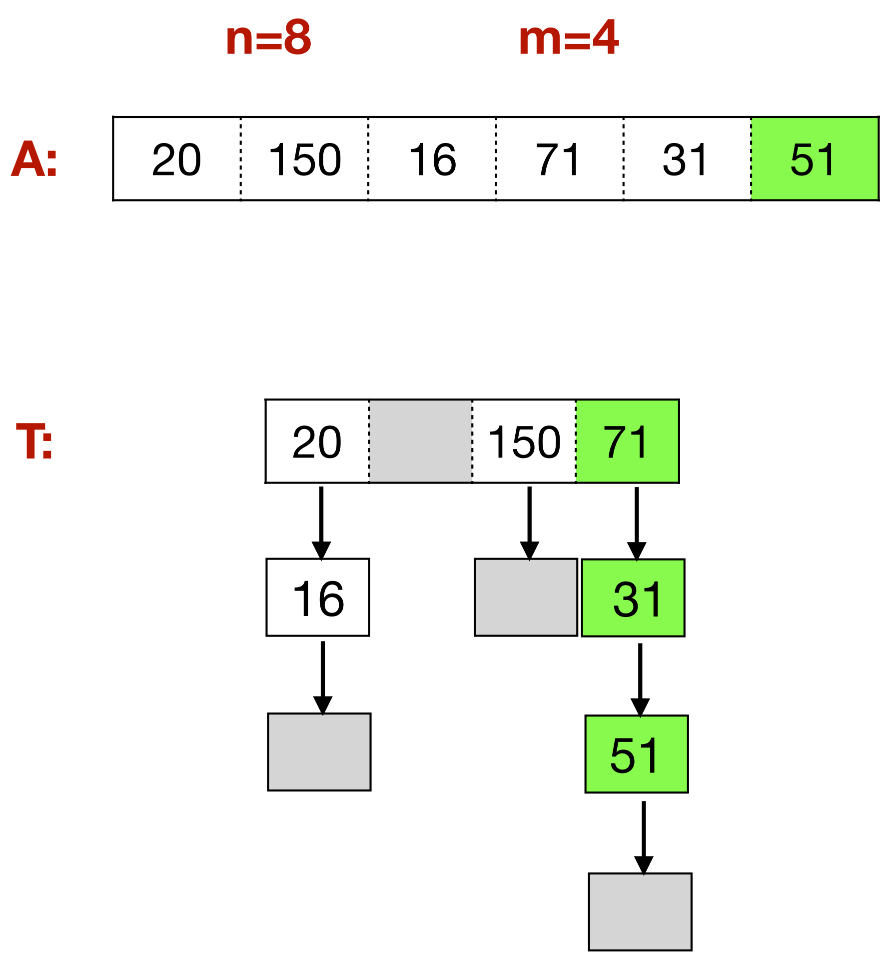
- $h(x) = (x \mod 4)$+1
- h(20) = 1
- h(150) = 3
- h(16) = 1
- h(71) = 4
- h(31) = 4

n=8 **m=4**



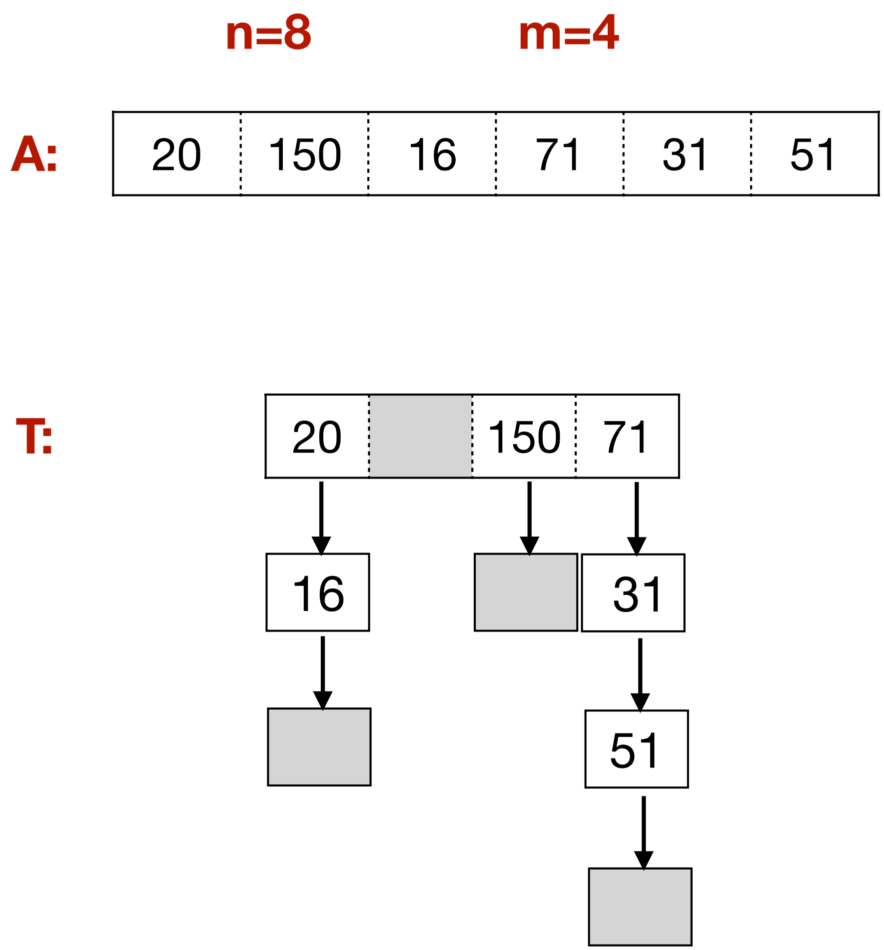
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1
- h(71) = 4
- h(31) = 4
- h(51) = 4



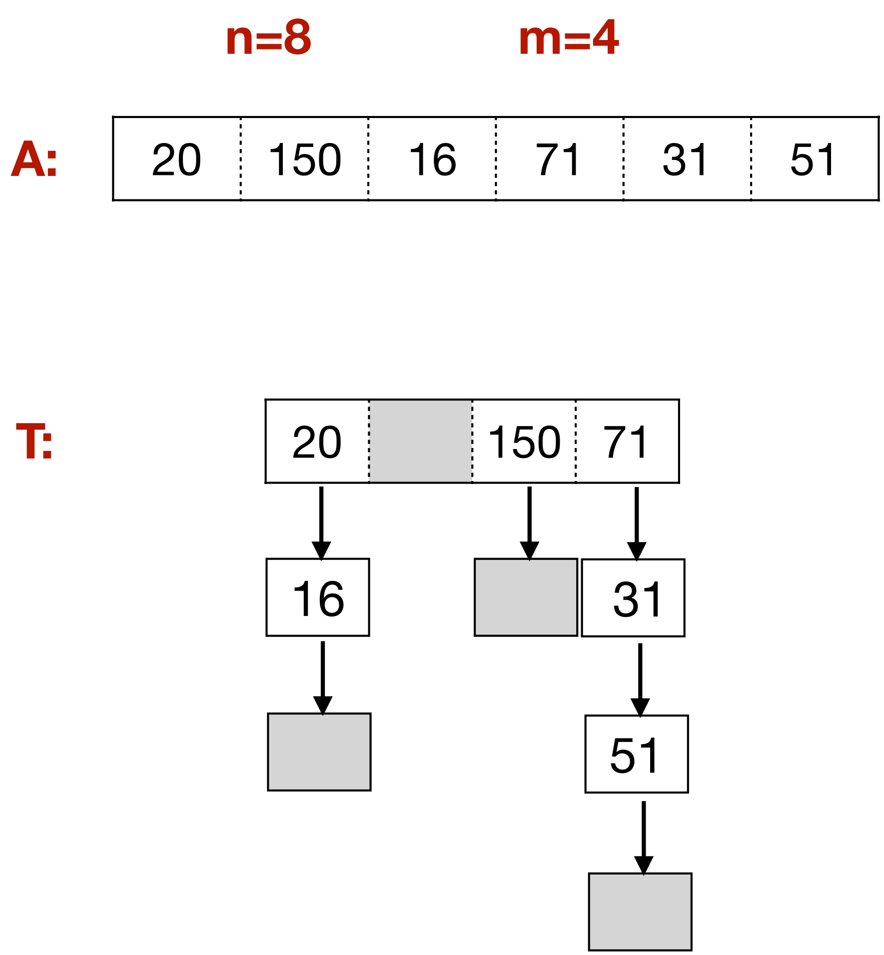
Chaining : Example

- $h(x) = (x \mod 4)+1$
- h(20) = 1
- h(150) = 3
- h(16) = 1
- h(71) = 4
- h(31) = 4
- h(51) = 4



Chaining : Example

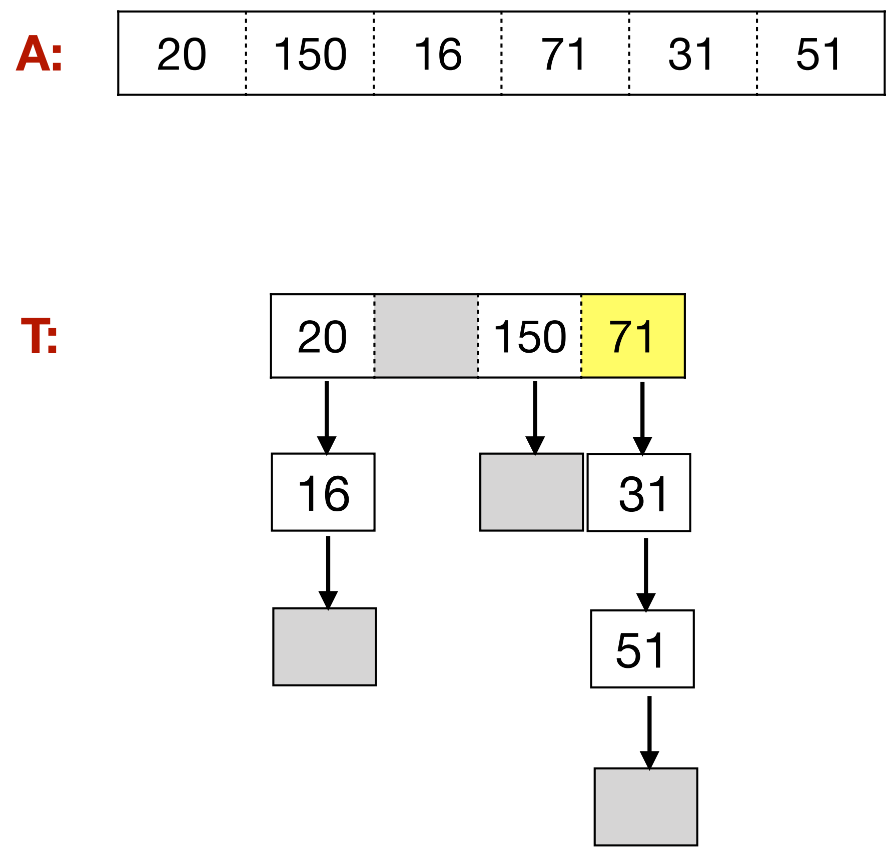
- $h(x) = (x \mod 4)+1$
- Search for x=31
  - h(31) = 4



Chaining : Example

- $h(x) = (x \mod 4)$+1
- Search for x=31
  - h(31) = 4

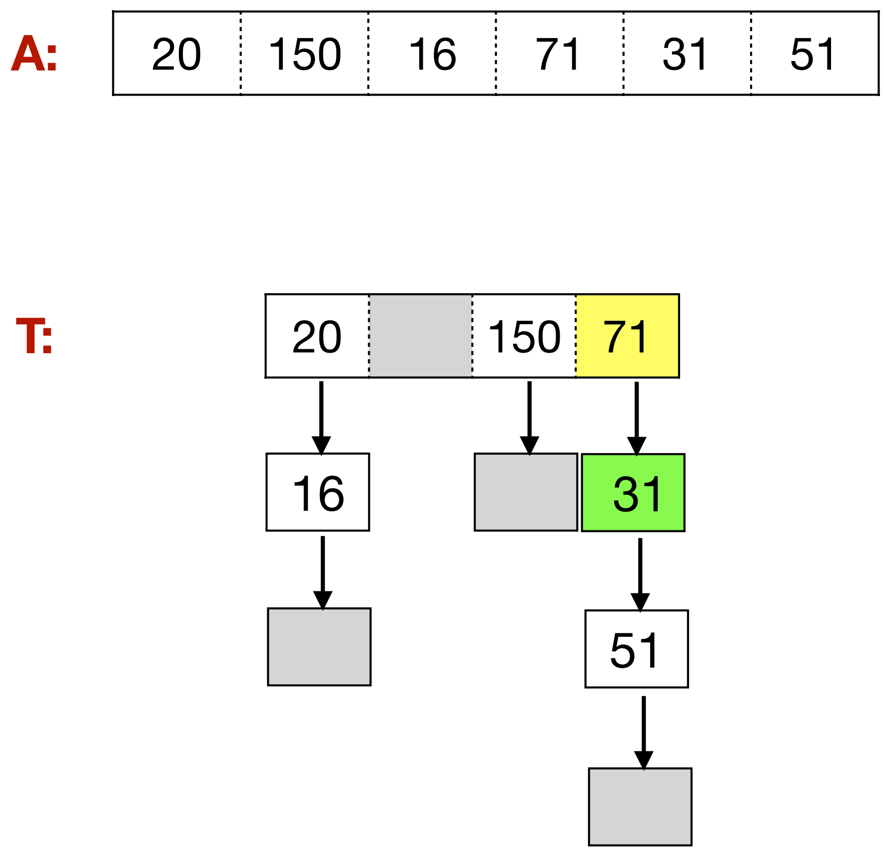
n=8 m=4



Chaining : Example

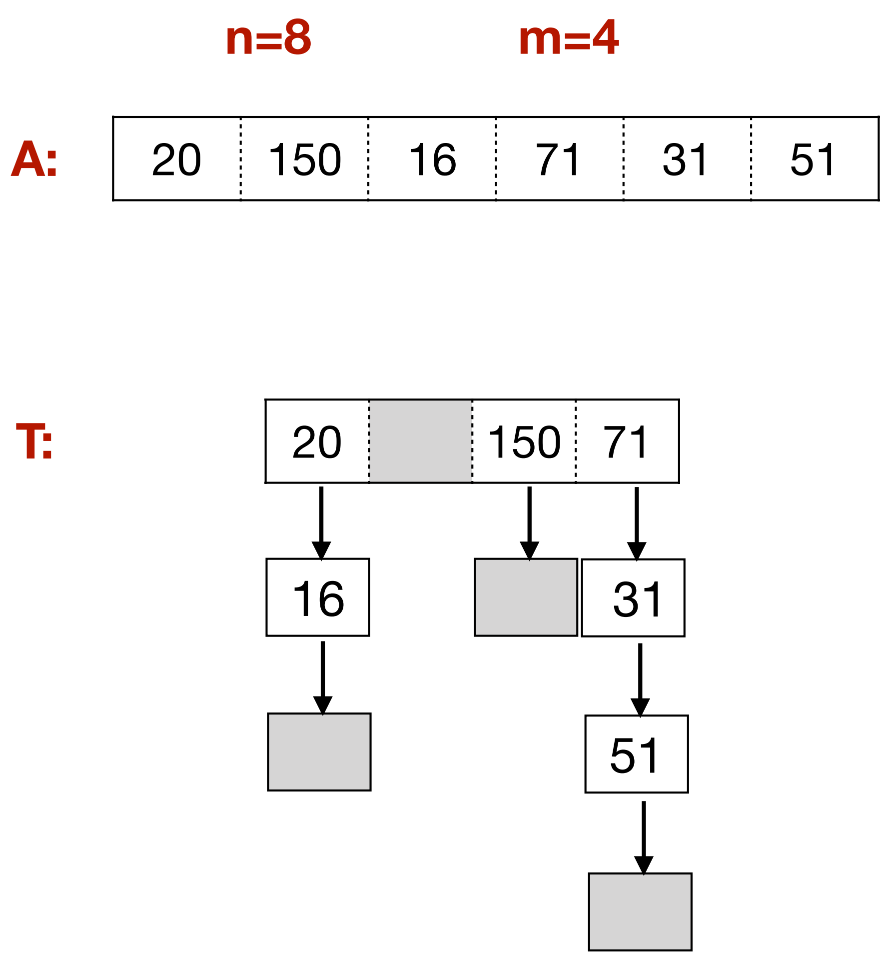
- $h(x) = (x \mod 4)+1$
- Search for x=31
  - h(31) = 4

n=8 **m=4**



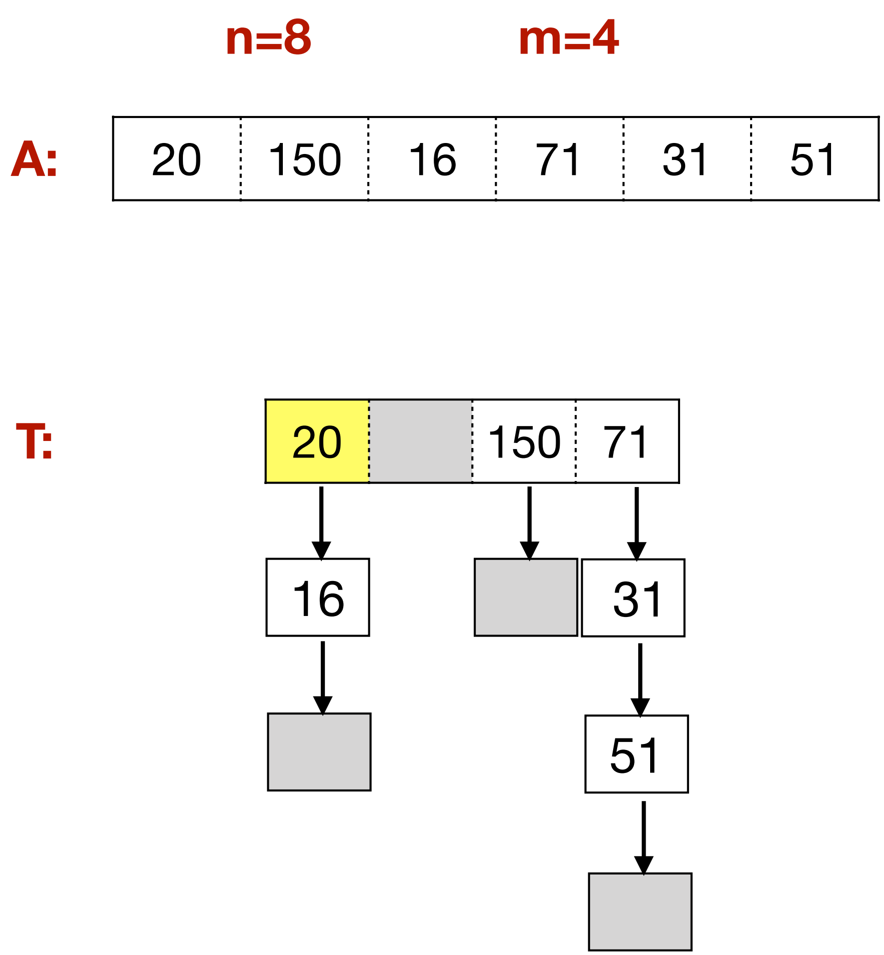
Chaining : Example

- $h(x) = (x \mod 4)+1$
- Search for x=31
  - h(31) = 4
- Search for x=64
  - h(64)=1



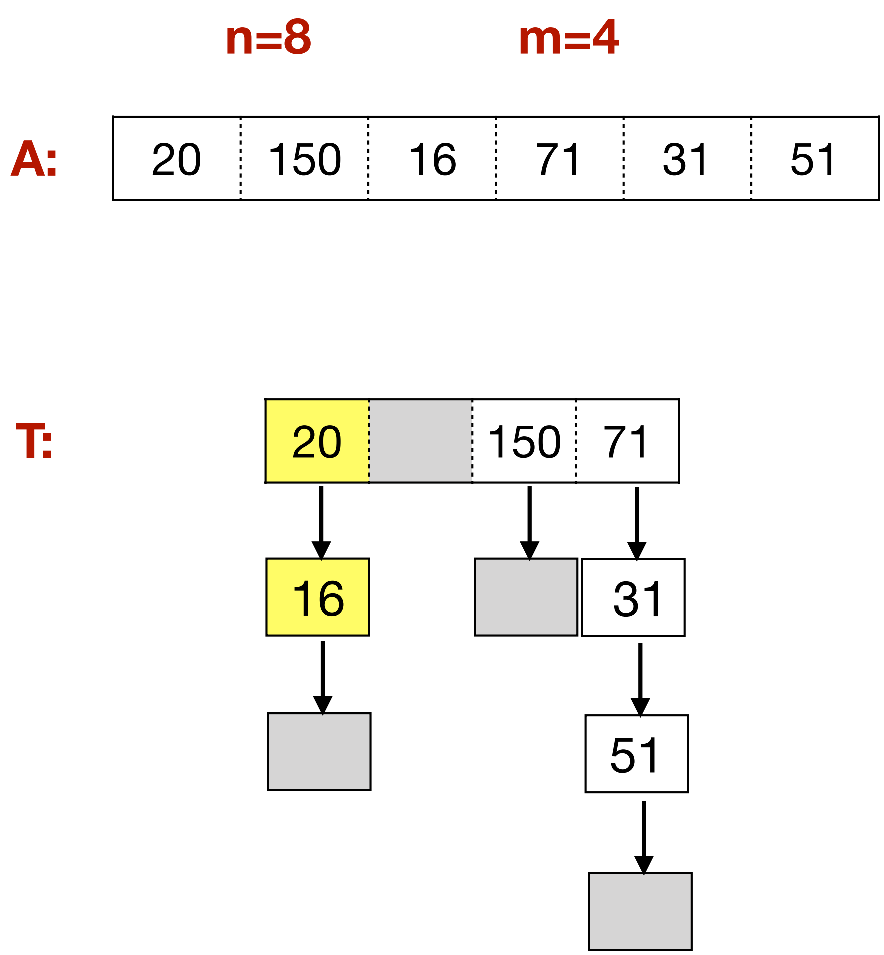
Chaining : Example

- $h(x) = (x \mod 4)+1$
- Search for x=31
  - h(31) = 4
- Search for x=64
  - h(64)=1



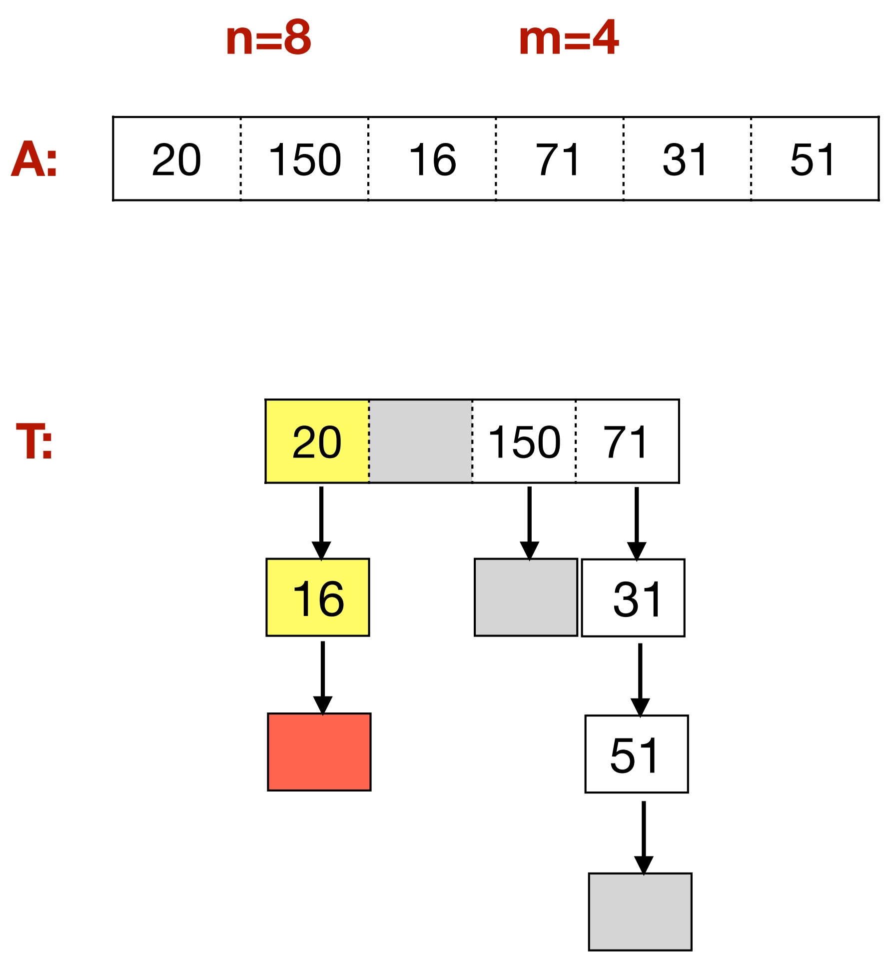
Chaining : Example

- $h(x) = (x \mod 4)$+1
- Search for x=31
  - h(31) = 4
- Search for x=64
  - h(64)=1



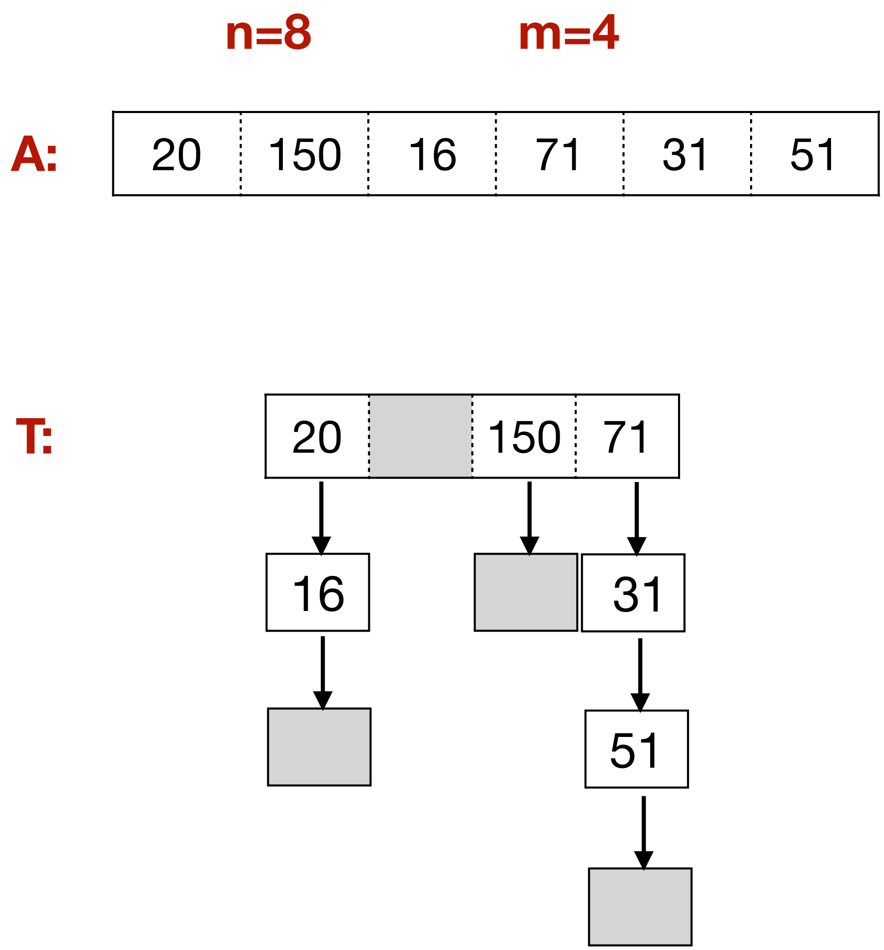
Chaining : Example

- $h(x) = (x \mod 4)+1$
- Search for x=31
  - h(31) = 4
- Search for x=64
  - h(64)=1



Chaining : Example

- $h(x) = (x \mod 4)$+1
- Search for x=31
  - h(31) = 4
- Search for x=64
  - h(64)=1



Chaining: Proof of Correctness

- Every element A[i] of is added to the linked-list of T[h(A[i])] and no other number appears in the linked-list
- For any number x, it can only be in the linked-list T[h(x)] and we search the entire list for it

Chaining: Runtime Analysis

- What is worst-case runtime of **search**(x)?

Chaining: Runtime Analysis

- What is worst-case expected runtime of **search**(x)?

Chaining: Runtime Analysis

- Worst-case expected runtime of **search**(x) using near-universal hash functions:
  - Define $\ell(x)$ as the number of elements in A[1:n] mapped to x by the hash function h
  - Runtime of **search**(x) is $O(1 + \ell(x))$
- Worst-case expected runtime of **search**(x) is $O(1 + \mathbf{E}_{h \in \mathscr{H}}[\ell(x)])$

Chaining: Runtime Analysis

- So worst-case expected runtime is:

$$O(1 + \mathbf{E}_{h \in \mathscr{H}}[\ell(x)]) = O(1 + \frac{n}{m})$$

Called **Load**
ratio of number of elements to hash to the size of hash table

Dynamic Programming:
It is just Smart Recursion

Dynamic Programming?

- A technique for speeding up recursive algorithms
- Proof of correctness: exactly the same as recursive algorithms
- Runtime? Will become much faster typically

Fibonacci Numbers

Recursive Algorithm

- **RecFibo**(n):
  - If n=1 or n=2, return 1
  - Otherwise, return **RecFibo**(n-1) + **RecFibo**(n-2)

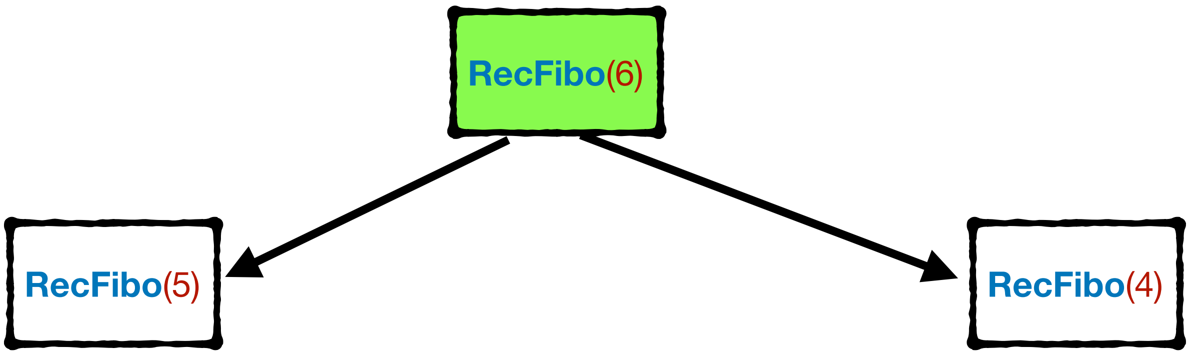
Recursive Algorithm

- **RecFibo**(n):
  - If n=1 or n=2, return 1
  - Otherwise, return **RecFibo**(n-1) + **RecFibo**(n-2)
- Proof of correctness: This is literally the same formula as F(n)!
- Runtime analysis?

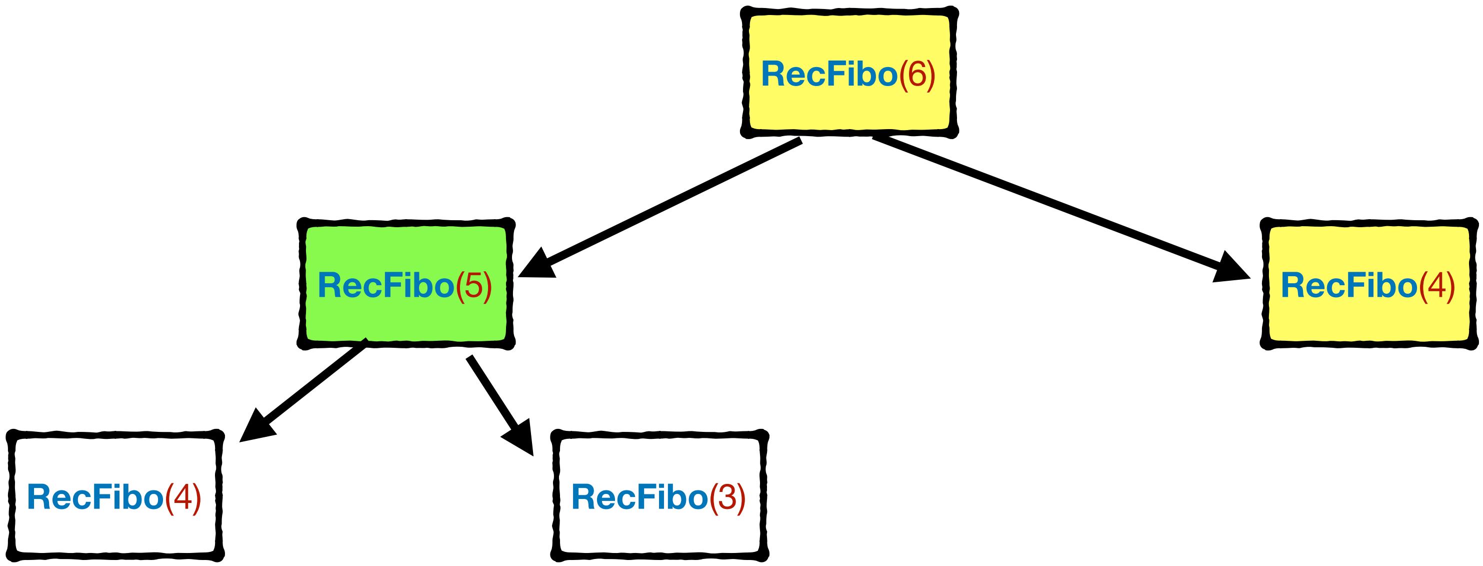
RecFibo(6):

RecFibo(6)

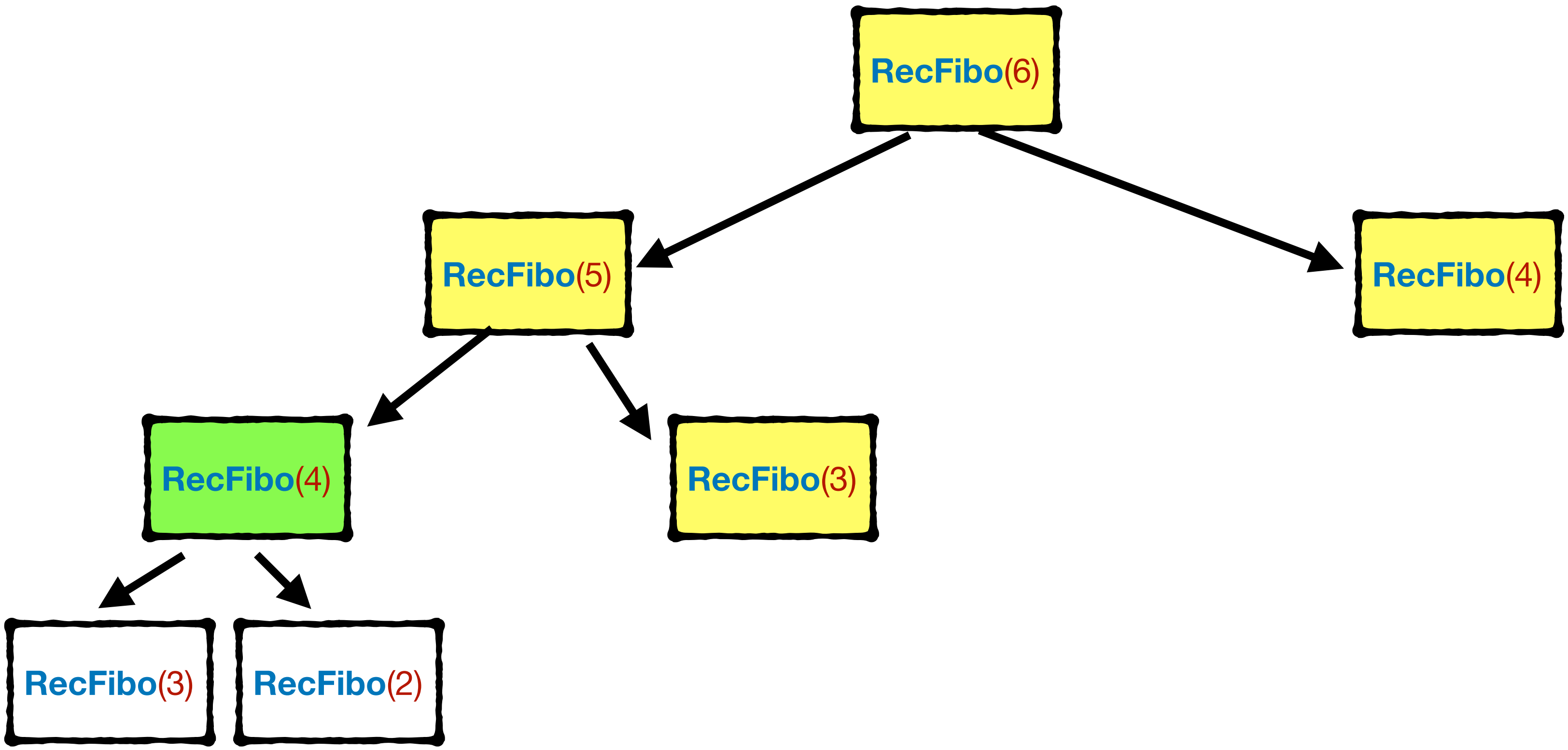
RecFibo(6):



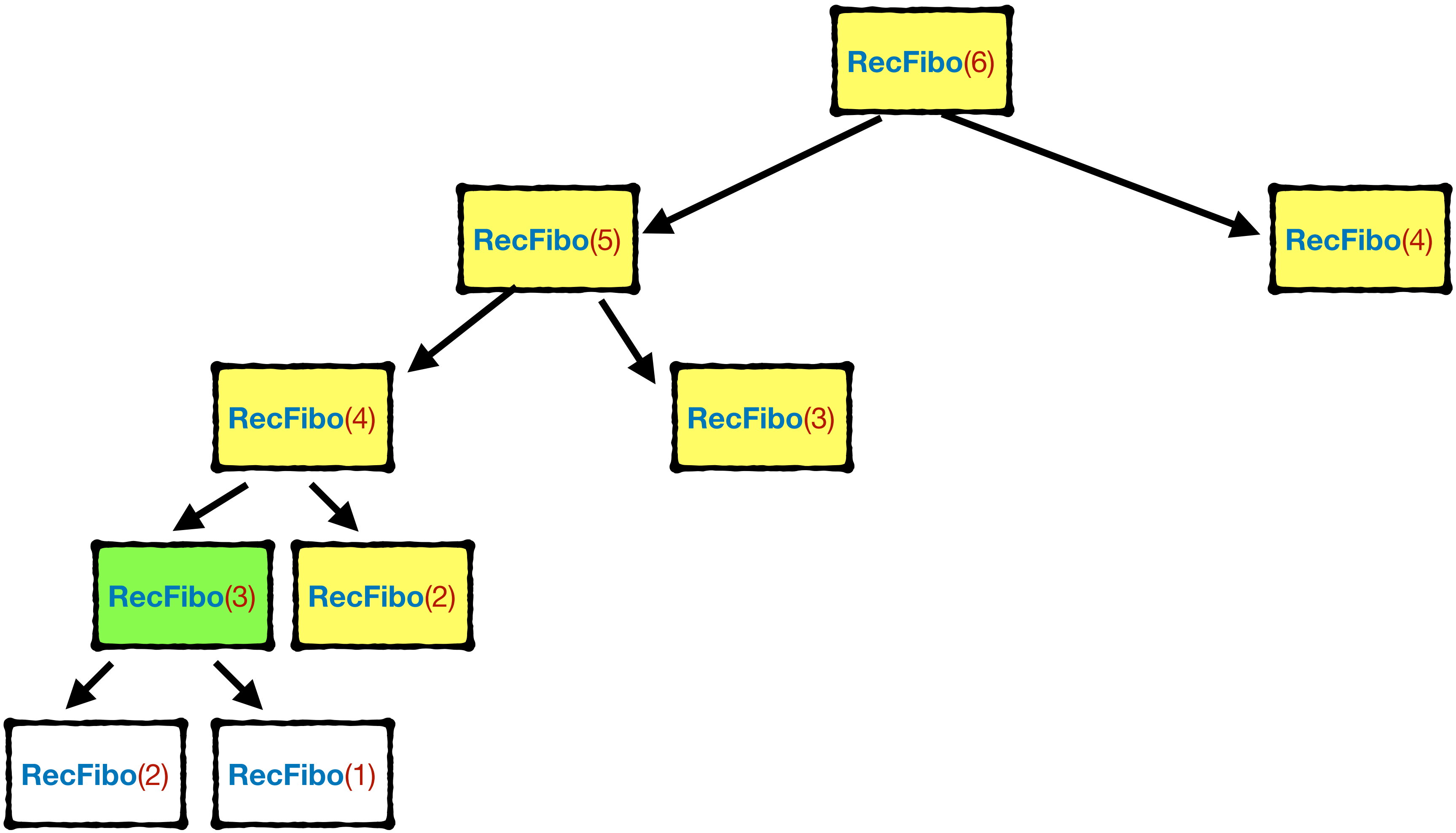
RecFibo(6):



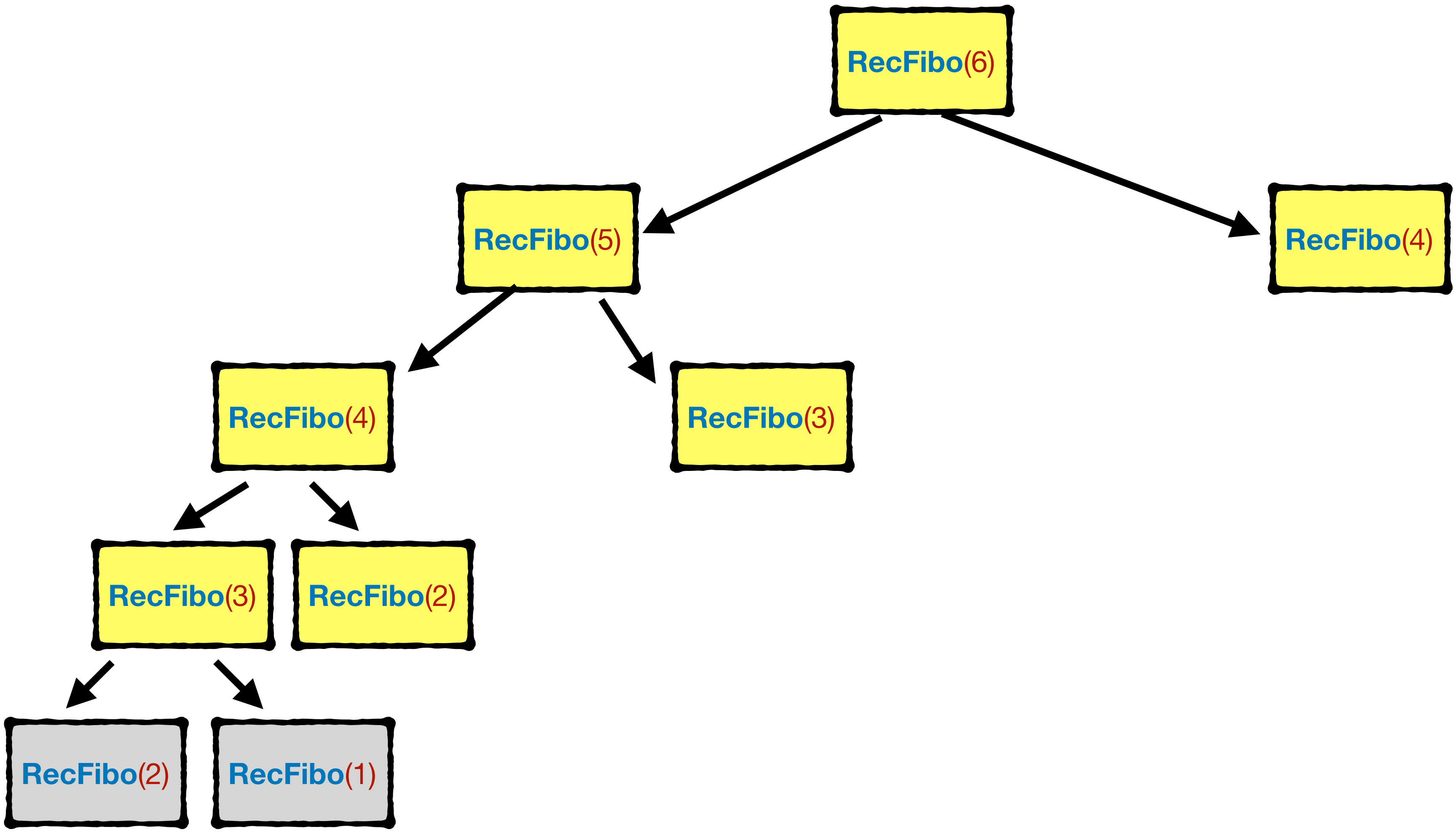
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)

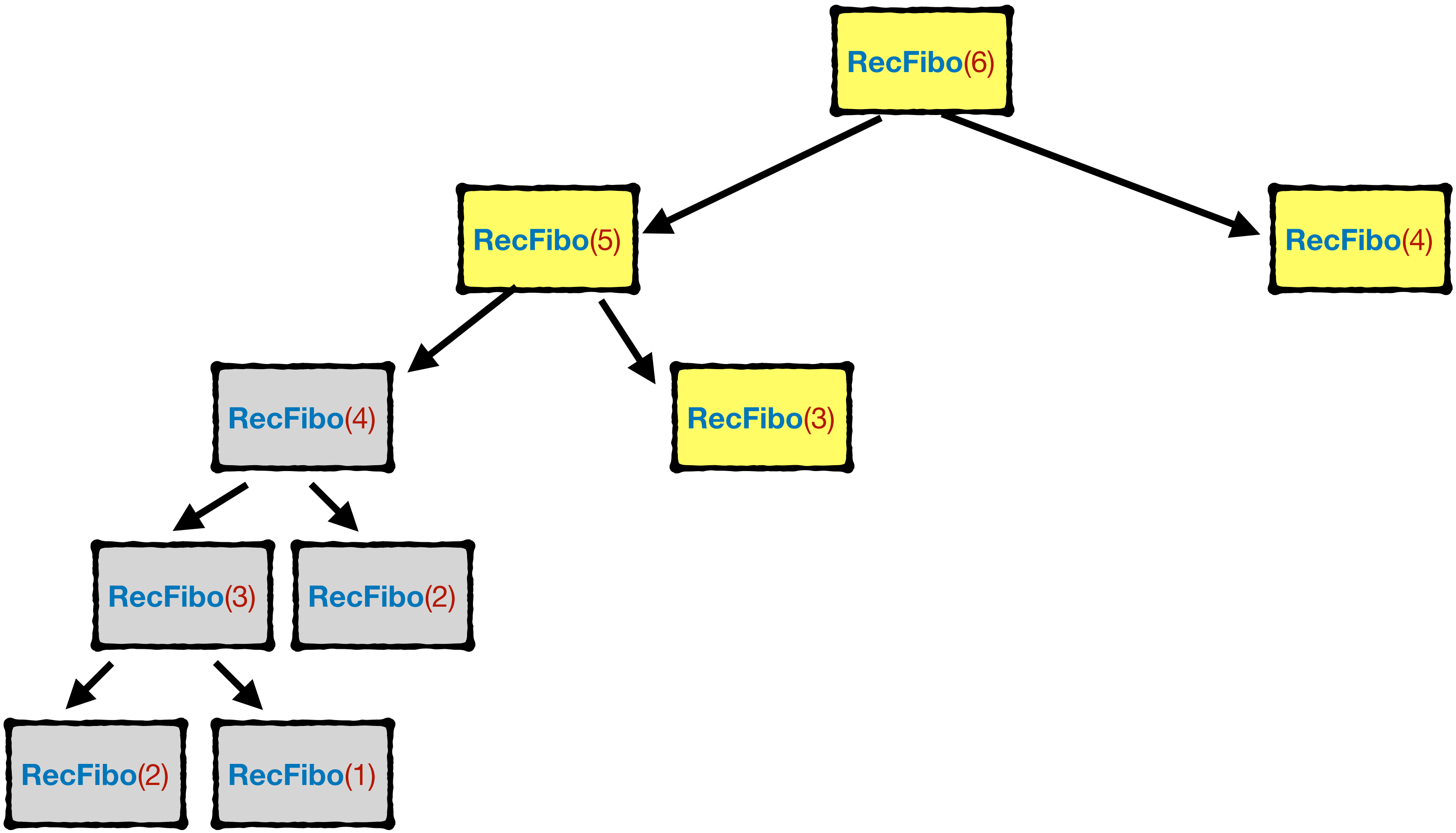
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)

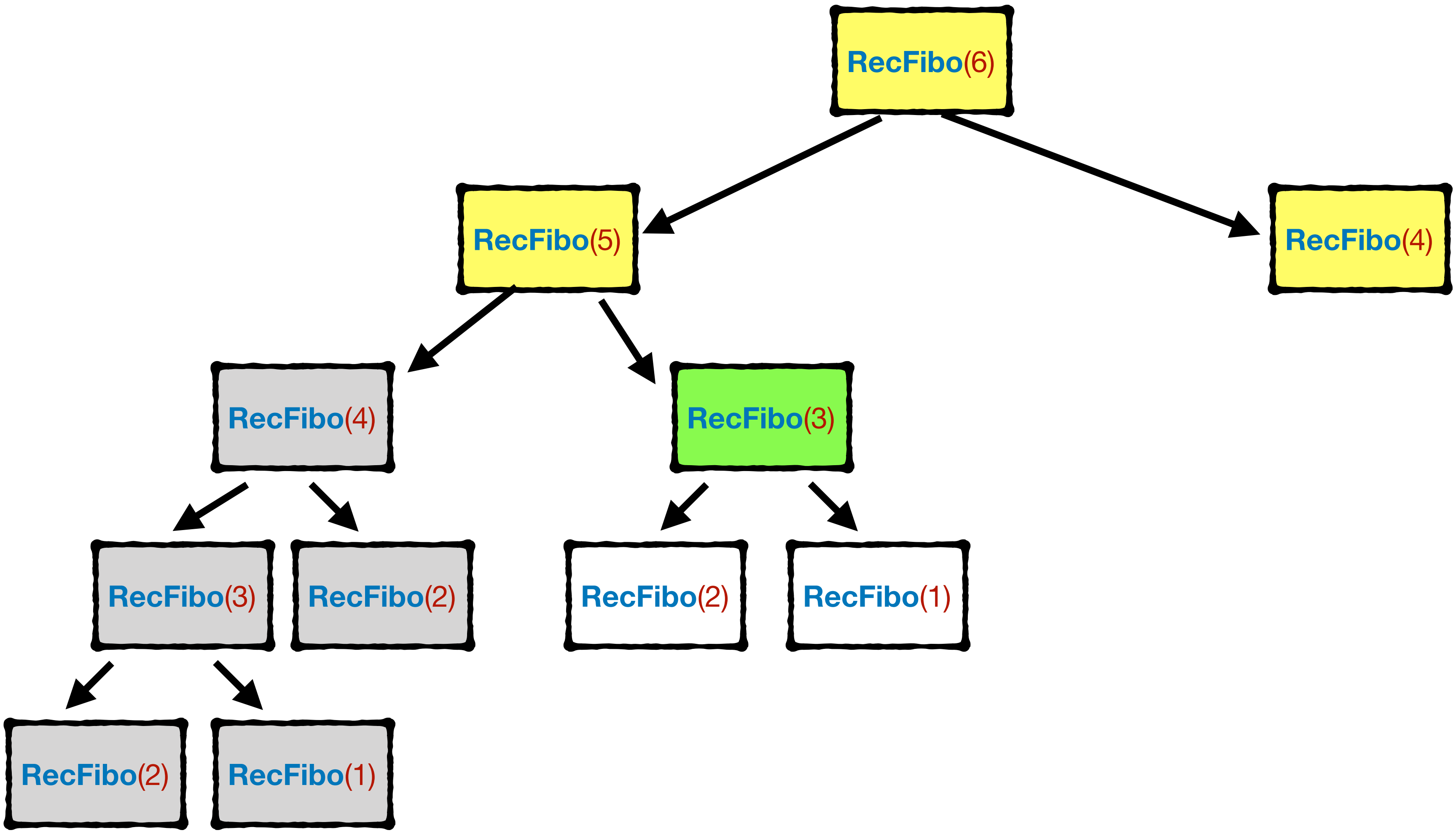
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)

RecFibo(6):

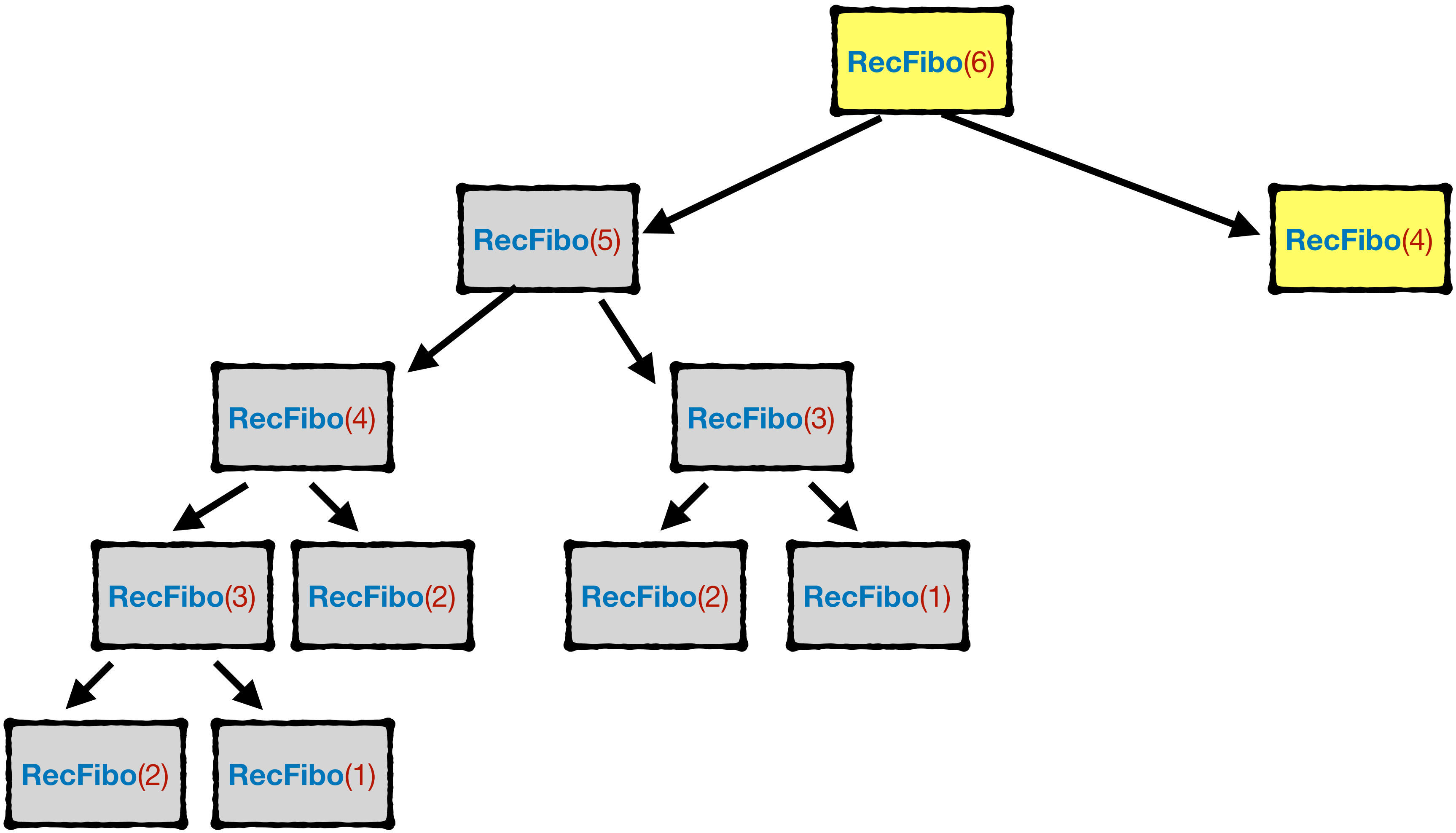

RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)

RecFibo(6):

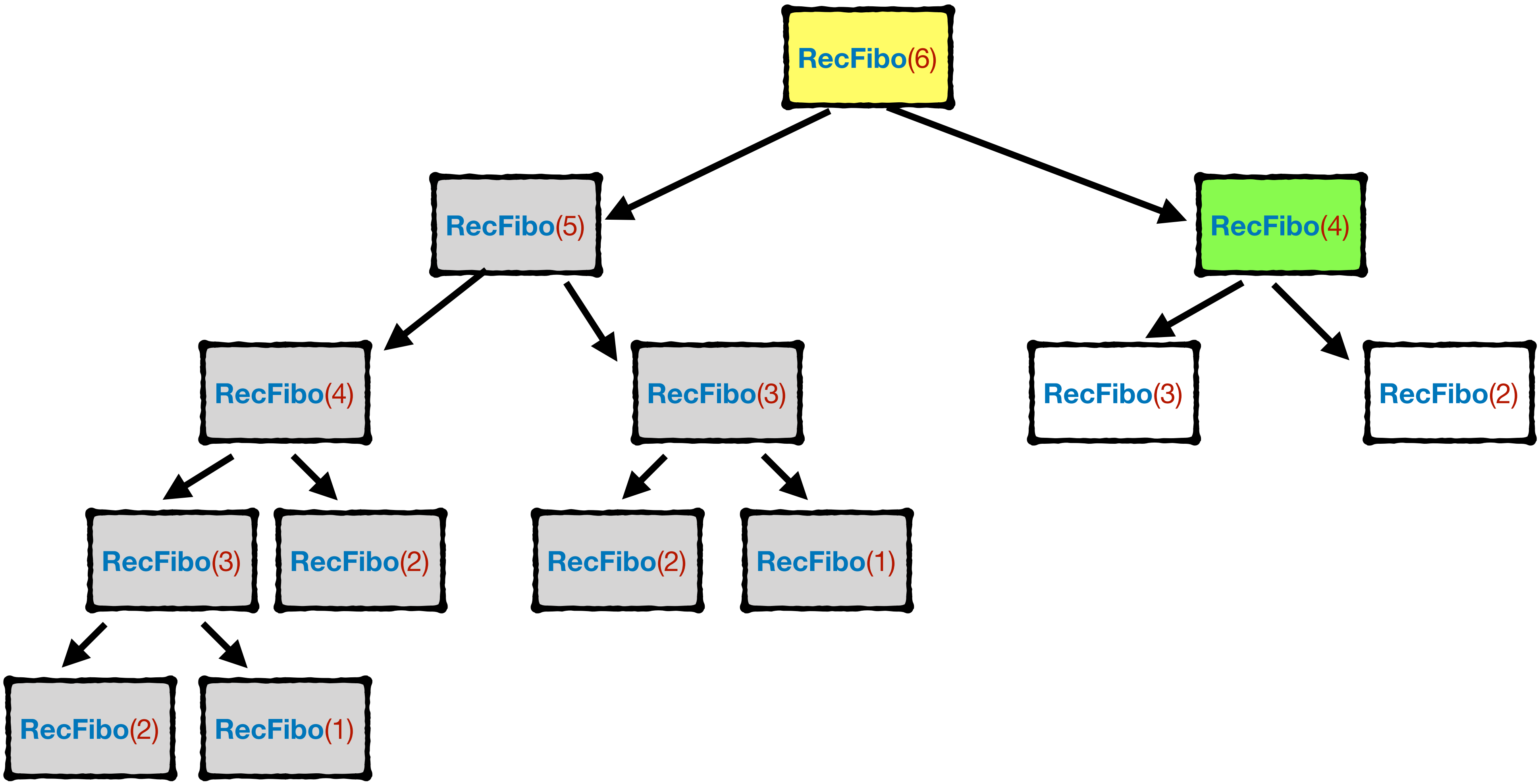


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)

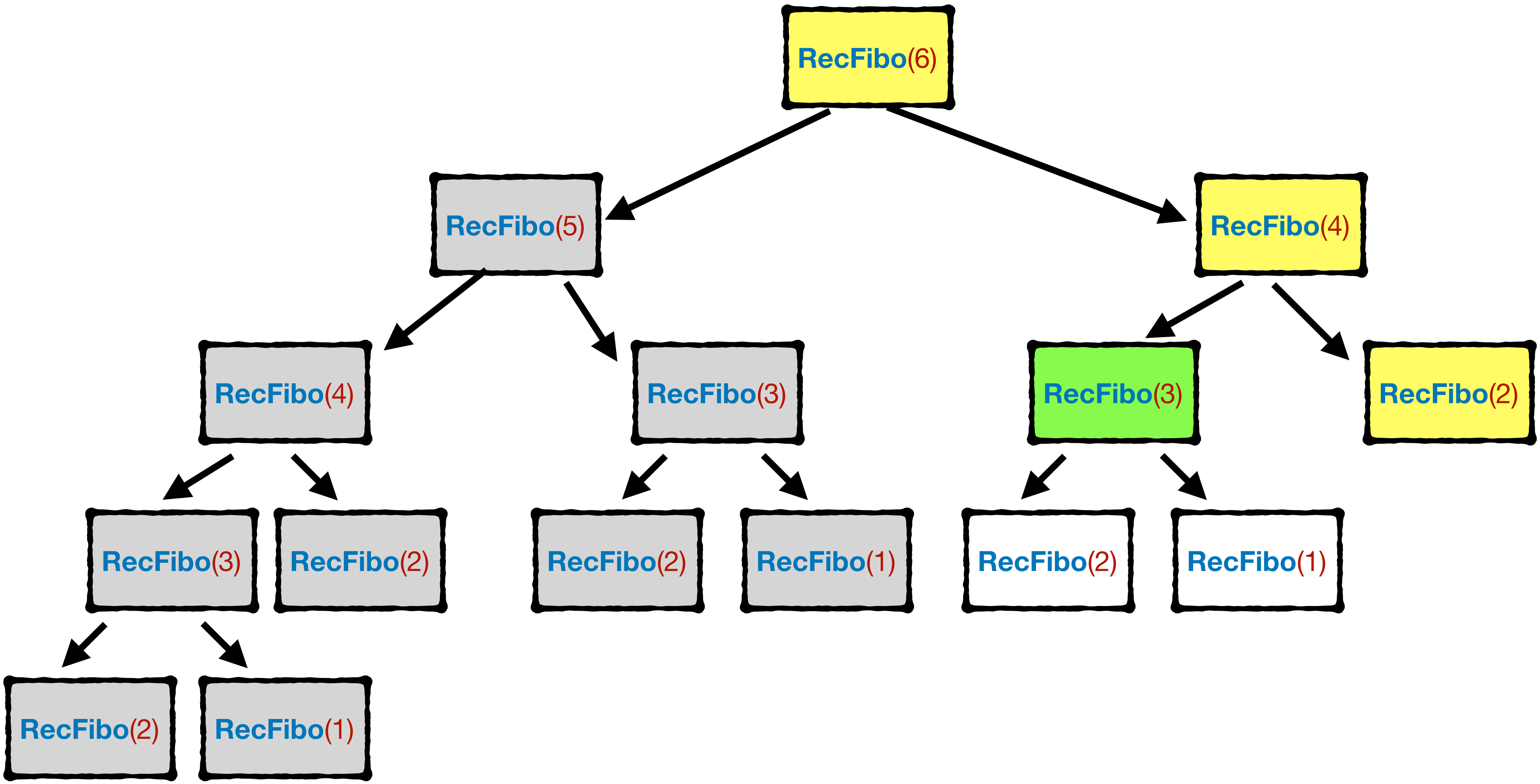
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)

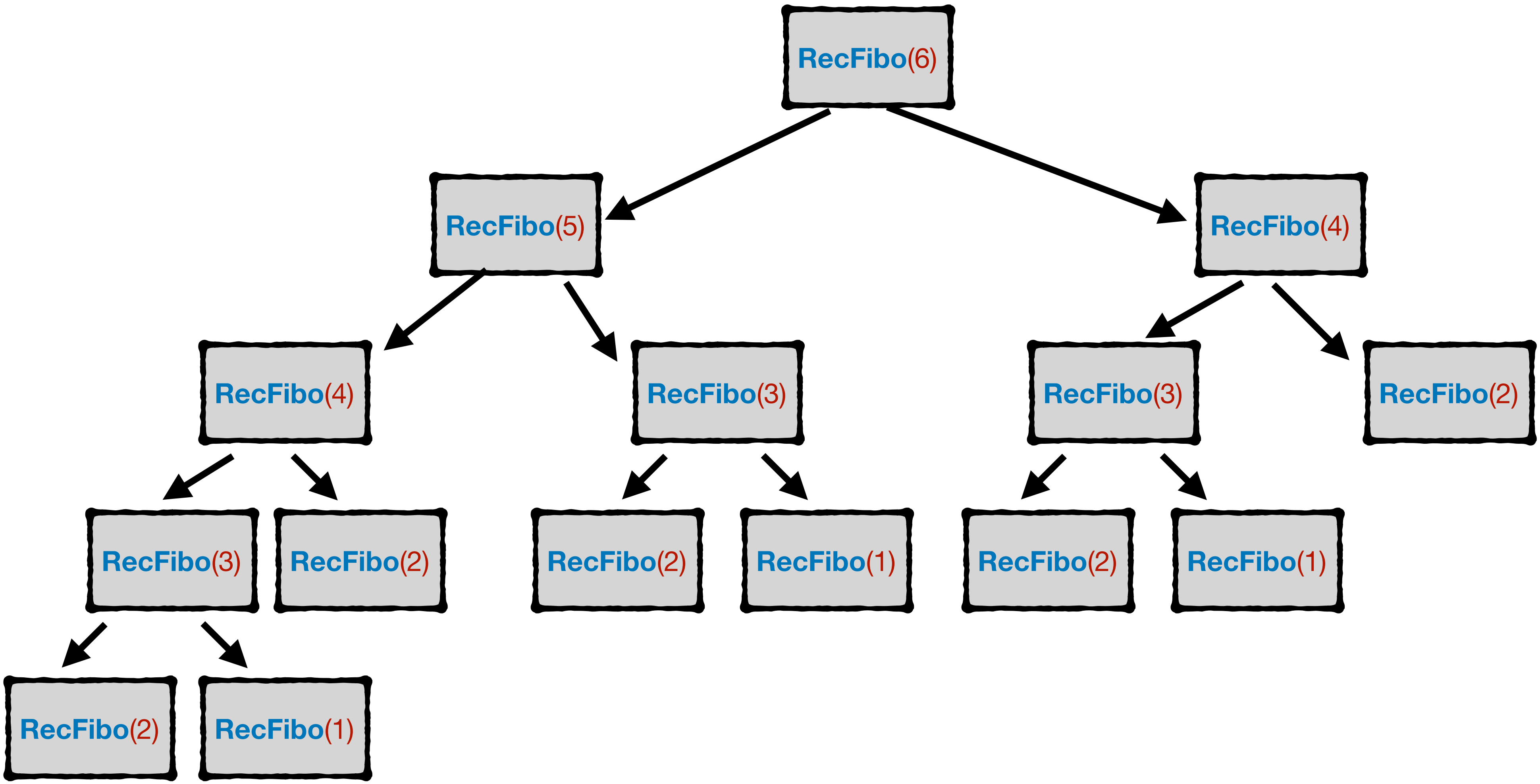
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)

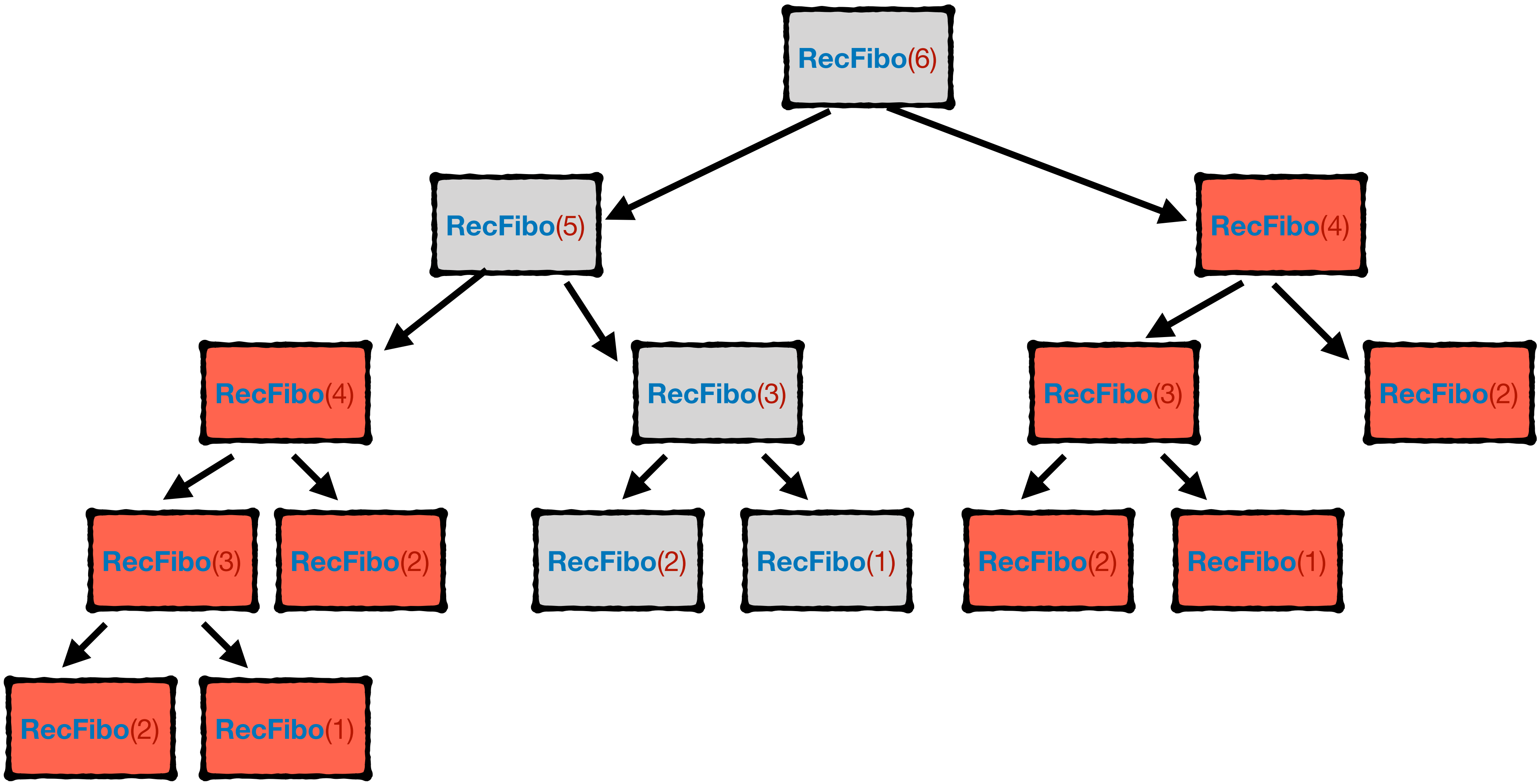
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)

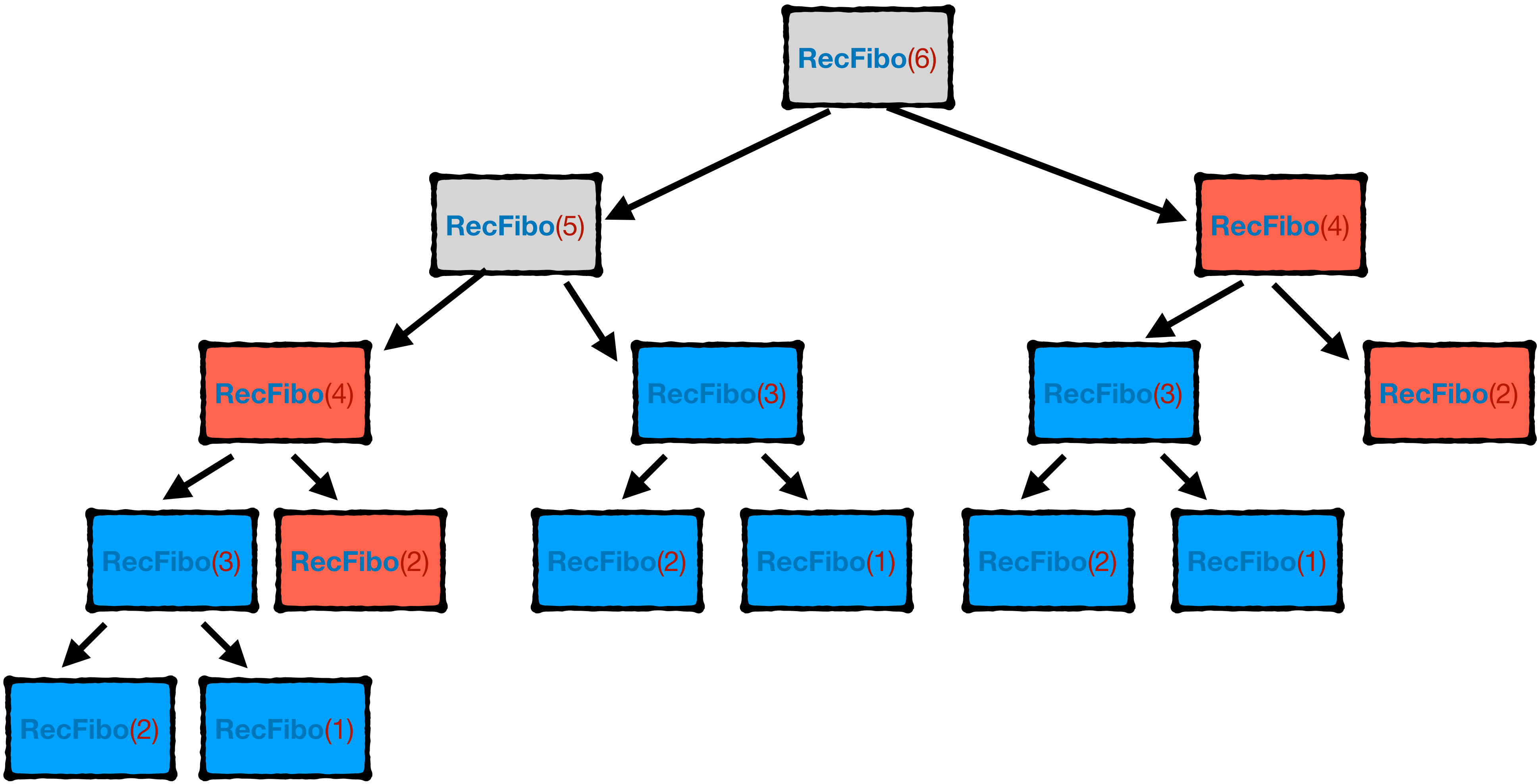
RecFibo(6):


RecFibo(6)
RecFibo(5)
RecFibo(4)
RecFibo(4)
RecFibo(3)
RecFibo(3)
RecFibo(2)
RecFibo(3)
RecFibo(2)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)
RecFibo(2)
RecFibo(1)

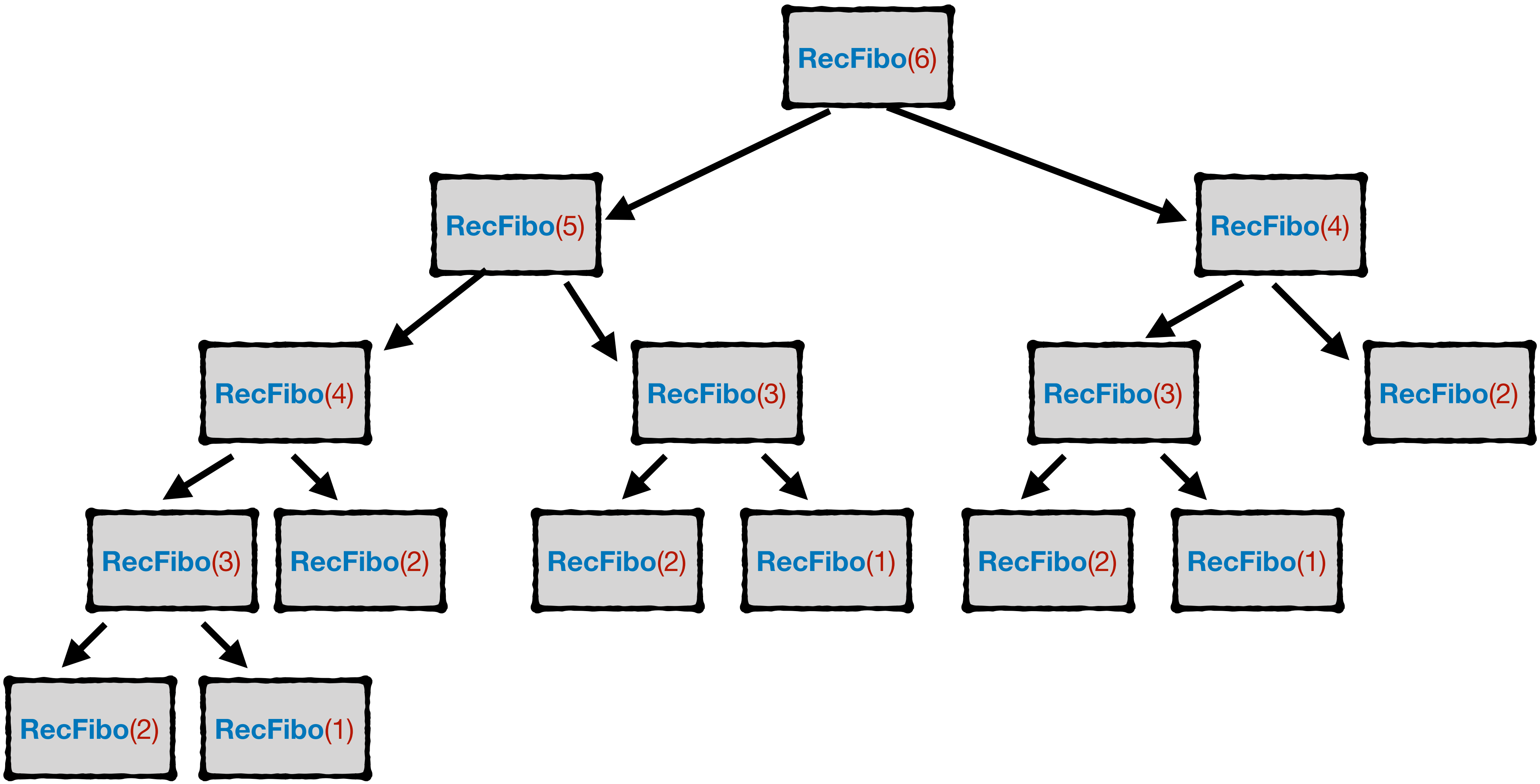
What is wrong with this figure?



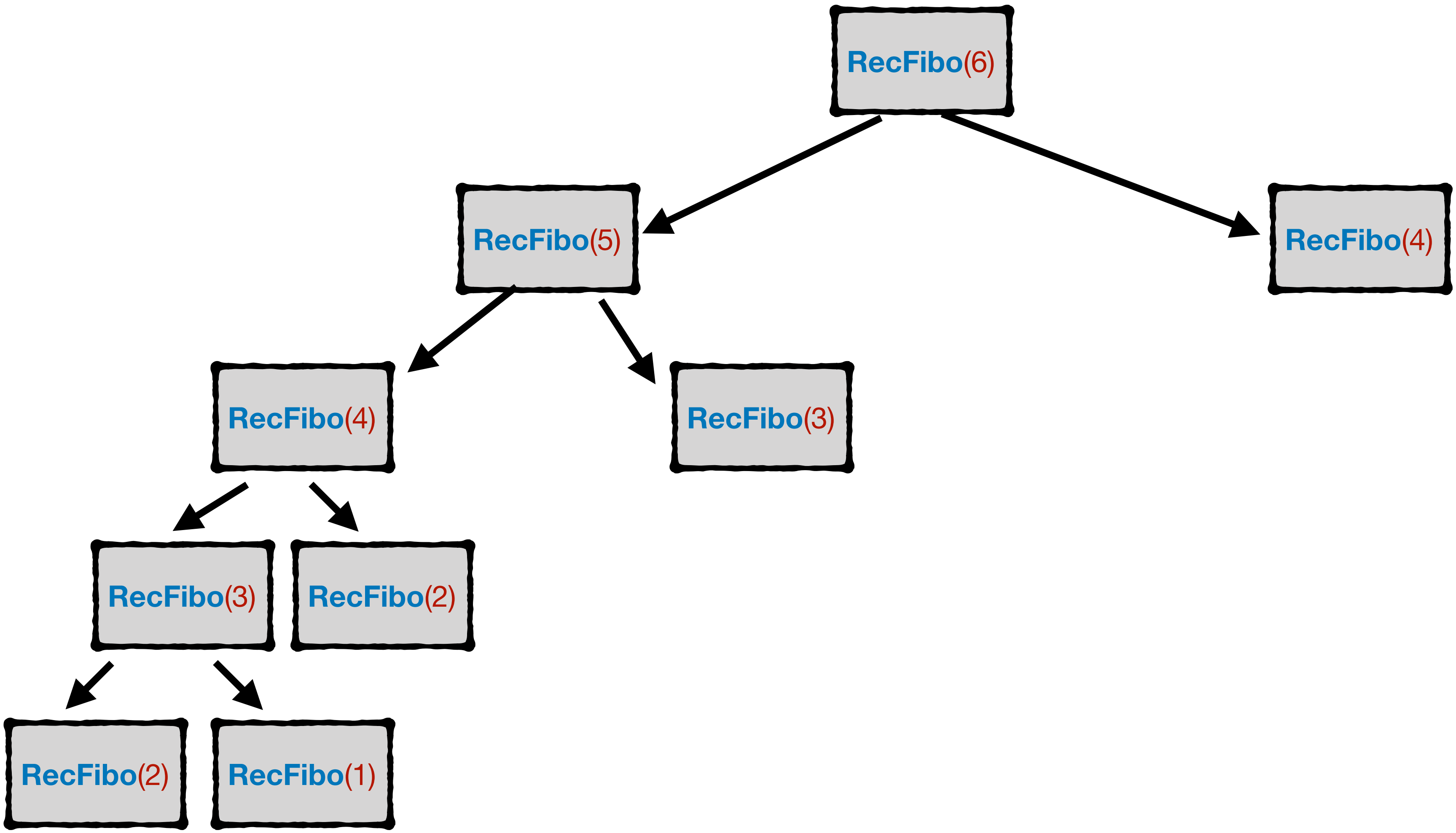
What is wrong with this figure?



An "ideal" figure?



An "ideal" figure?



Memoization

- Pick an array S[1:n] initialized with ‘undefined’ in every entry
- Whenever we compute F(i), store it in S[i]
- Next time, instead of recomputing F(i), just return S[i]

Fibonacci Numbers with Memoization

- Initialize an array S[1:n] with 'undefined' in every entry
- **MemFibo**(n):
  - If S[n] $\neq$ 'undefined', return S[n]
  - If n=1 or n=2, let S[n] = 1
  - Otherwise, let S[n] = MemFibo(n-1) + MemFibo(n-2)
  - Return S[n]

Fibonacci Numbers with Memoization

- Proof of Correctness:
  - Nothing has changed in the logic of algorithm between **MemFibo** and **RecFibo**
  - So **MemFibo** is also correct

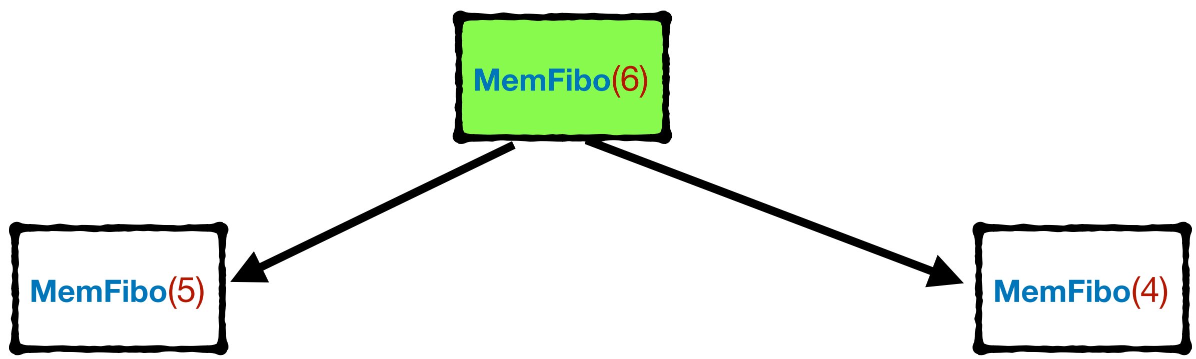
Fibonacci Numbers with Memoization

- Runtime Analysis:
  - We spend $O(1)$ time, ignoring recursive calls, for each subproblem **MemFibo**(m) for any $1 \leq m \leq n$
  - We will **not** compute a subproblem more than once
  - So the runtime is O(n)

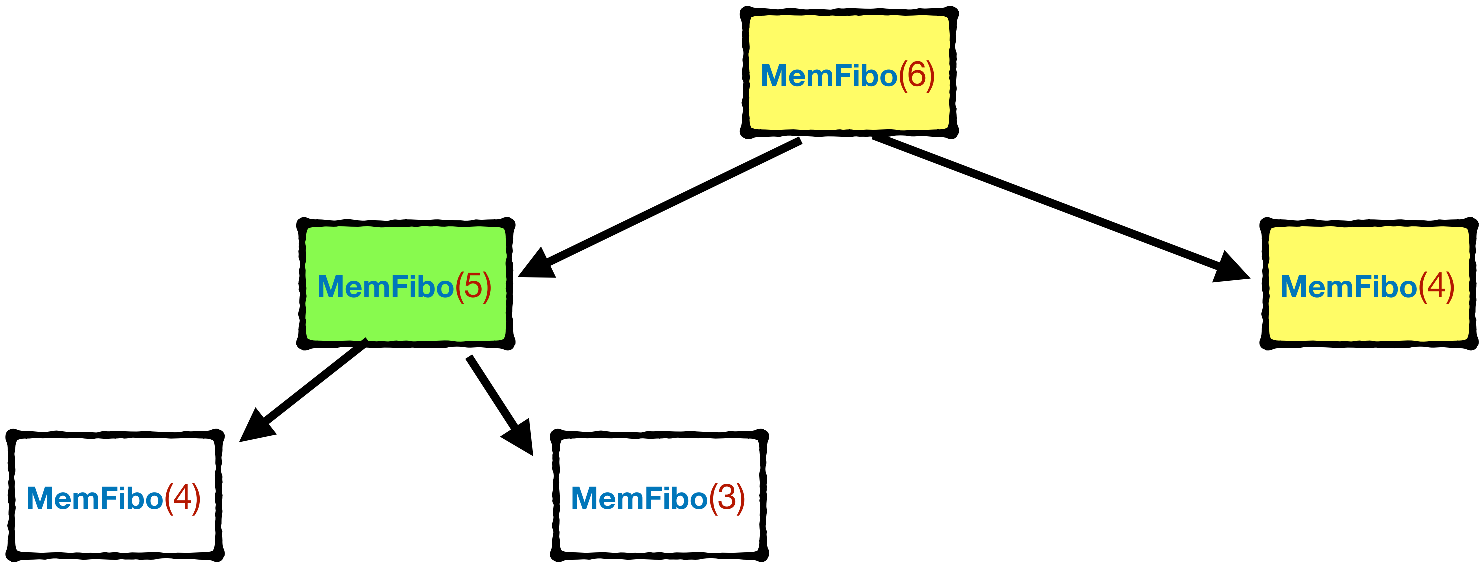
MemFibo(6)

MemFibo(6)

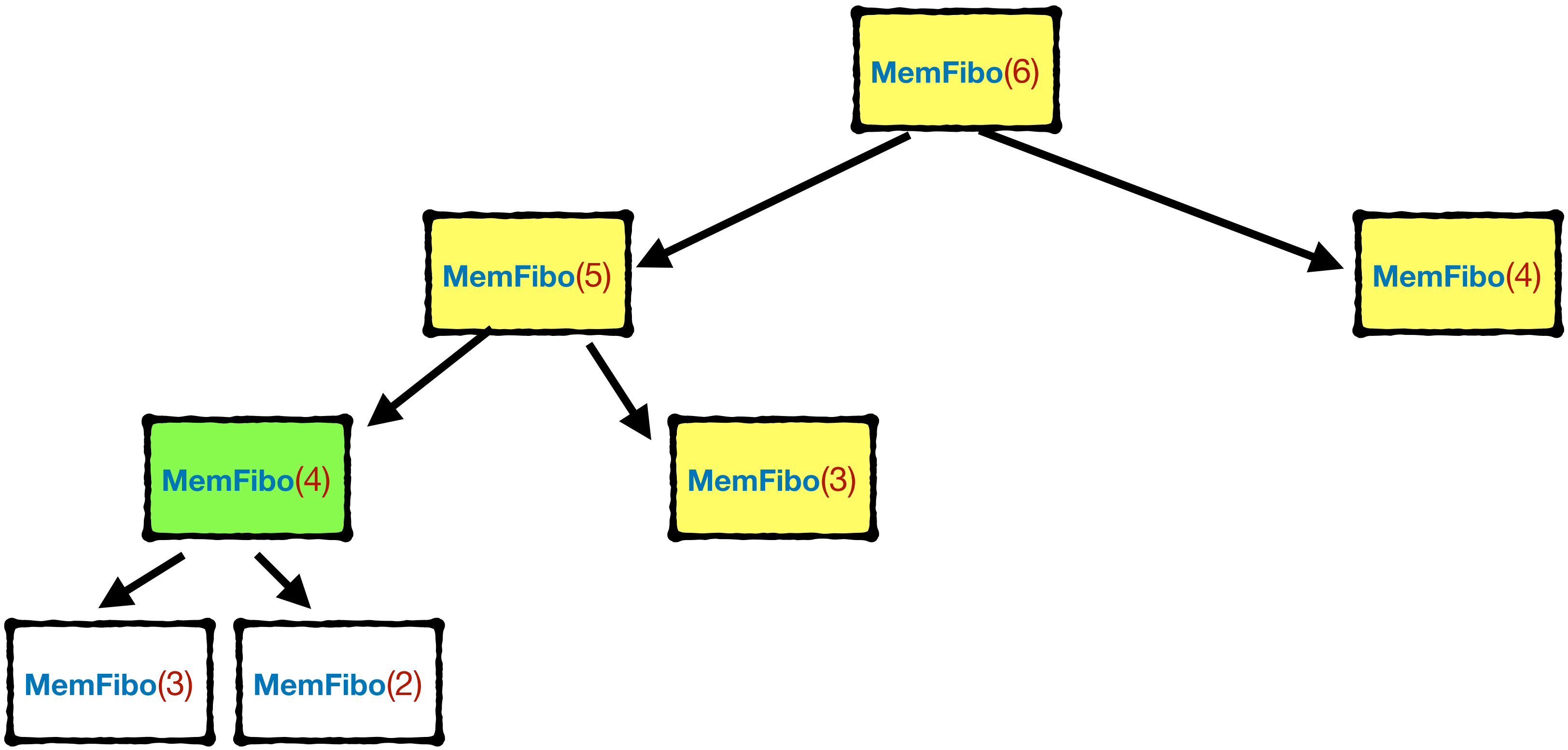
MemFibo(6)



MemFibo(6)

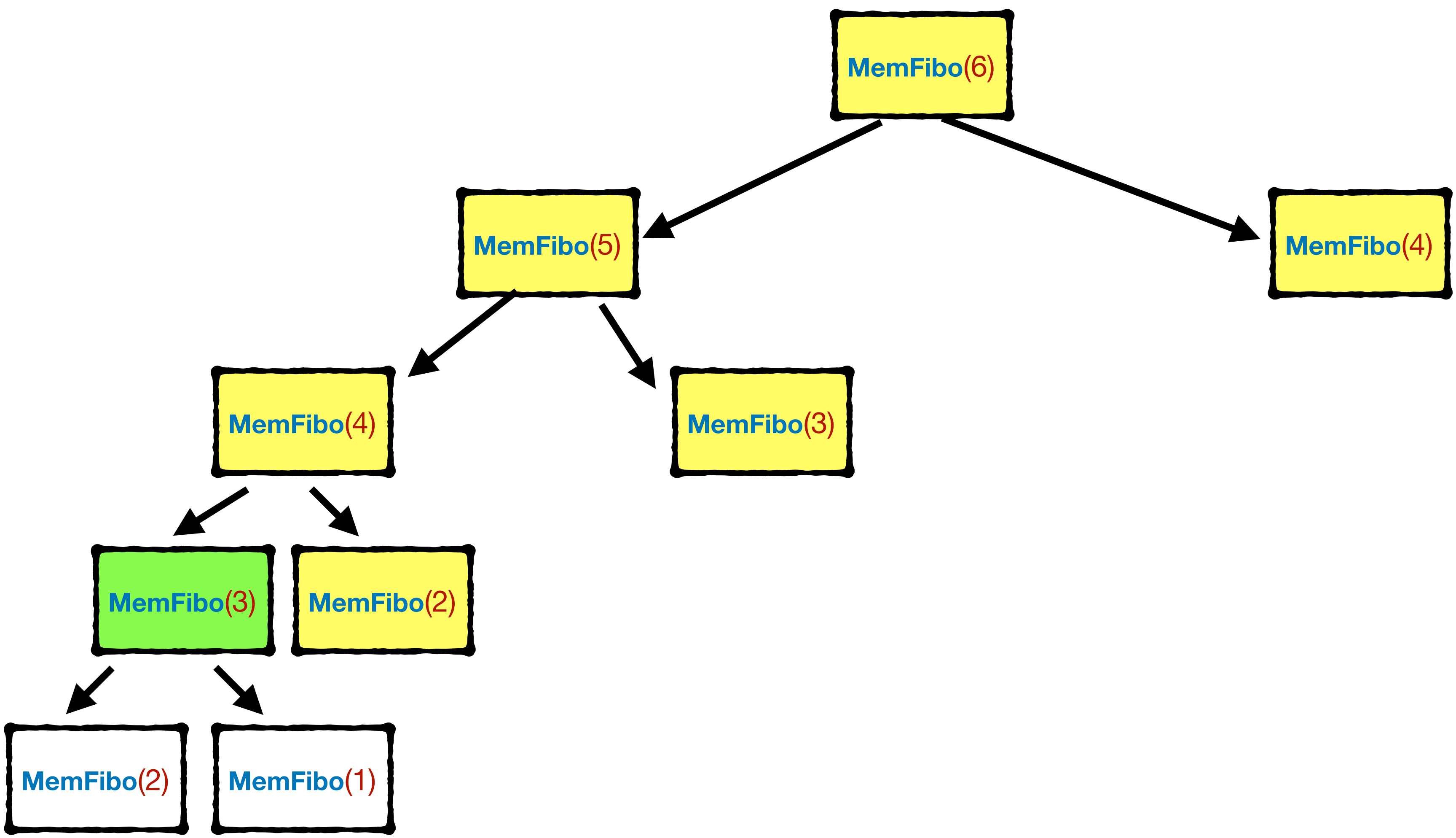


MemFibo(6)

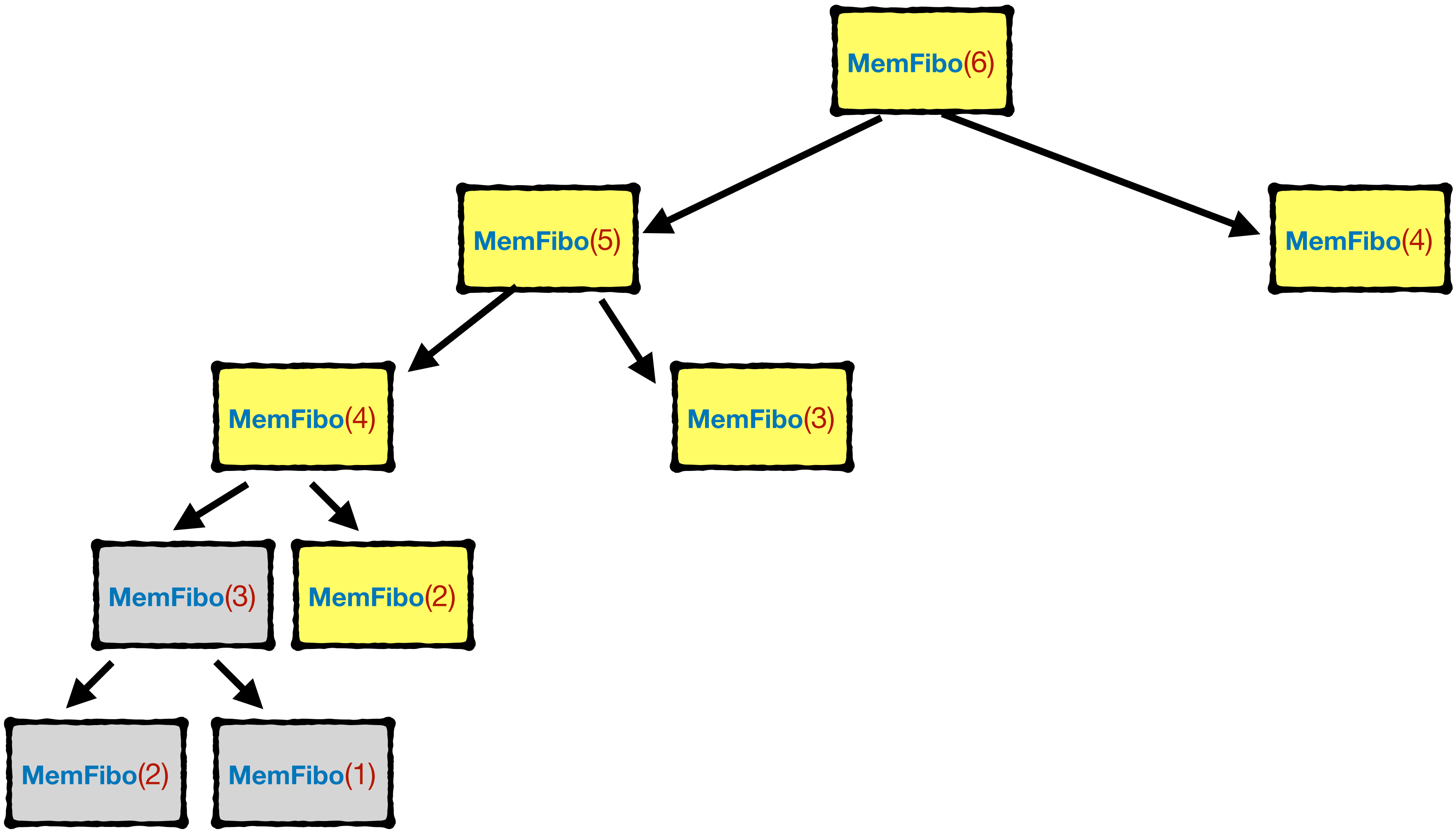


MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)

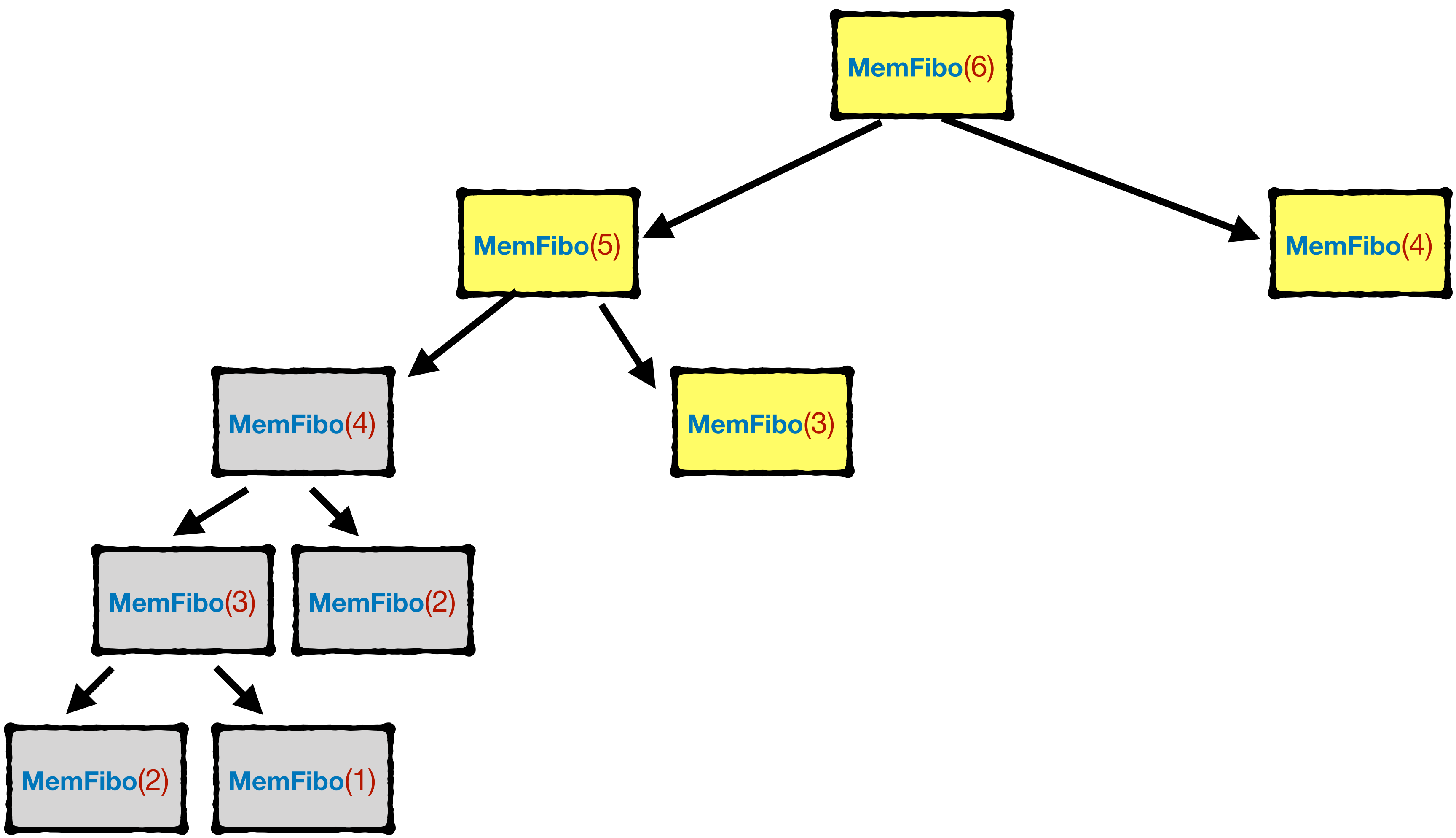
MemFibo(6)


MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

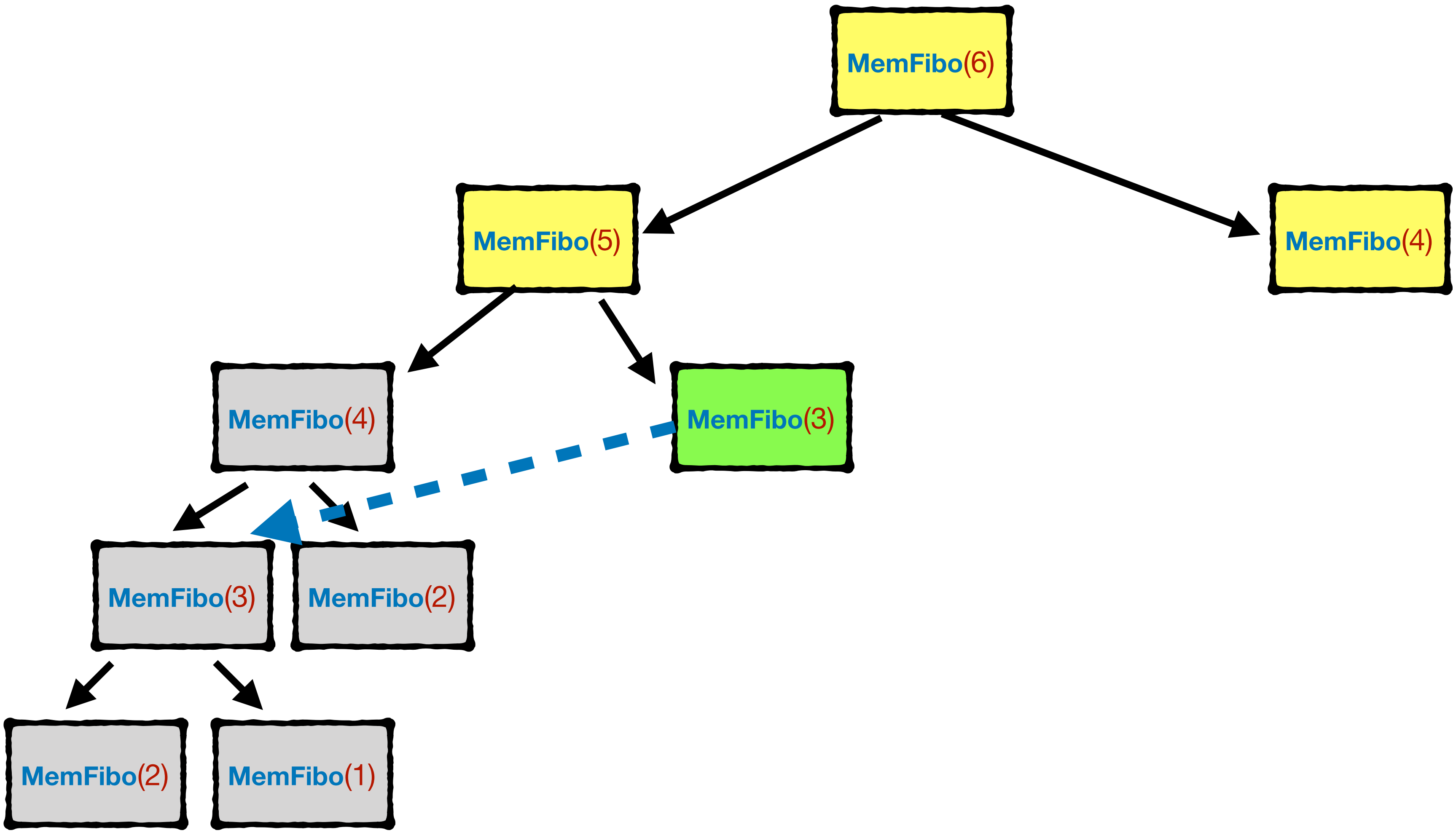
MemFibo(6)


MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

MemFibo(6)

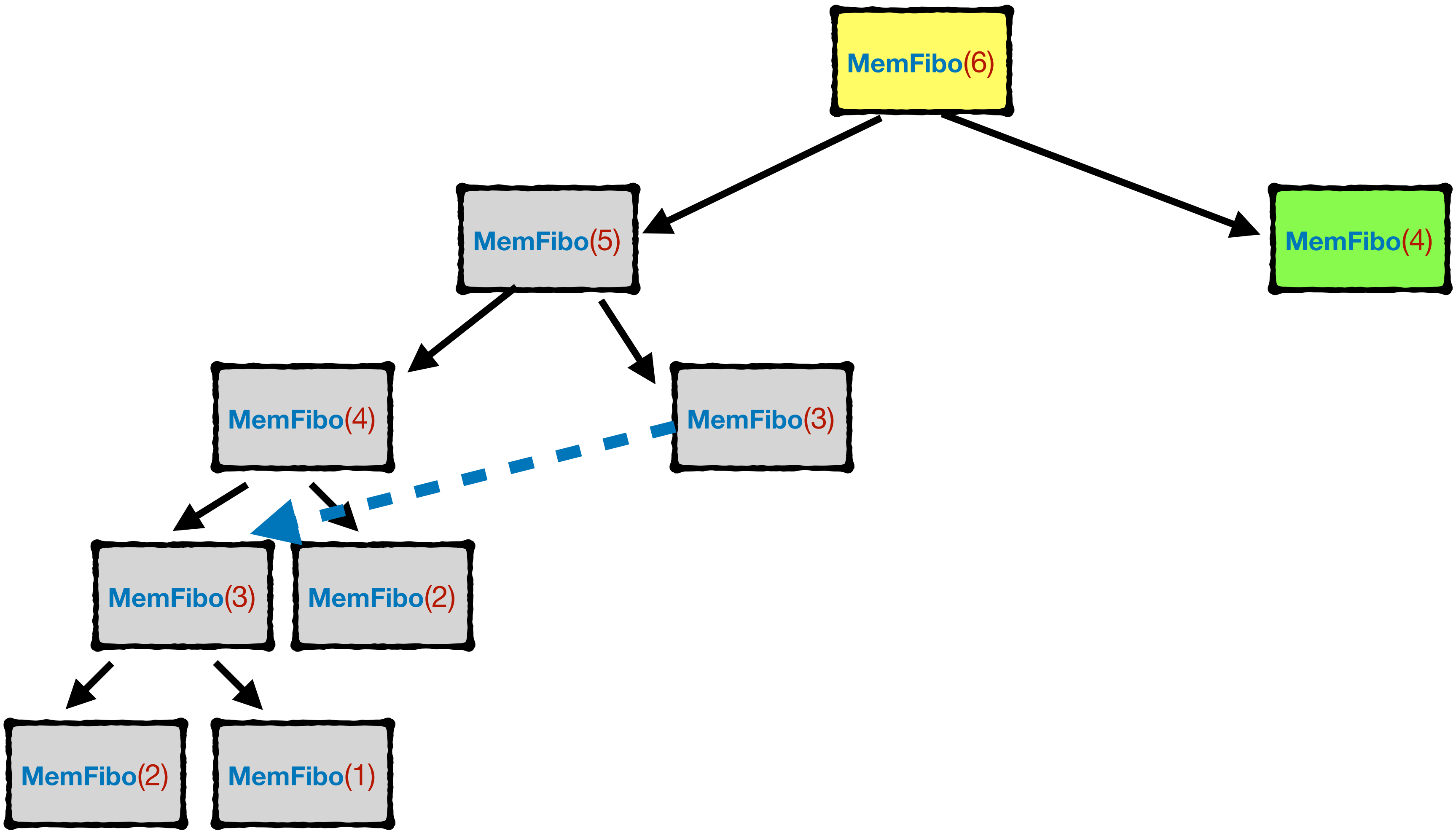

MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

MemFibo(6)



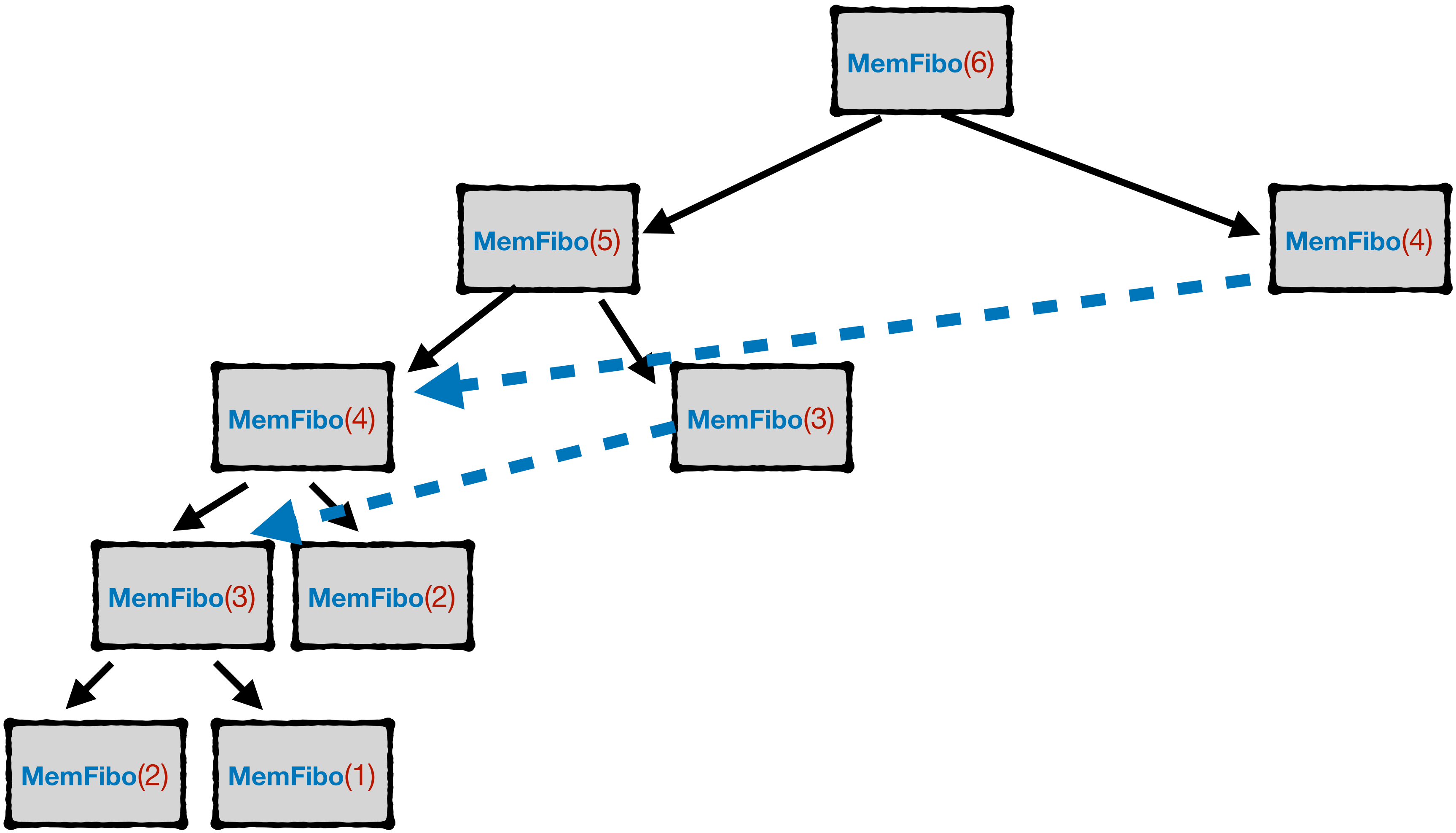
MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

MemFibo(6)



MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

MemFibo(6)


MemFibo(6)
MemFibo(5)
MemFibo(4)
MemFibo(4)
MemFibo(3)
MemFibo(3)
MemFibo(2)
MemFibo(2)
MemFibo(1)

Beyond Memoization?

- Memoization is a form of dynamic programming
- We literally only need to store the answers to our recursive calls so we do not recompute them
- It is usually called **top-down** dynamic programming
- There is also a notion of **bottom-up** dynamic programming
  - A way of stating dynamic programming without recursive calls

Bottom-Up Dynamic Programming

- We can literally fill up the array of stored answers ourselves
- **DynFibo**(n):
  - Create an empty array S[1:n]
  - Let S[1] = S[2] = 1
  - For i = 3 to n: let S[i] = S[i-1] + S[i-2]
  - Return S[n]

Bottom-Up Dynamic Programming

- Runtime analysis?
  - A for-loop of length $O(n)$
  - $O(1)$ runtime per each iteration of for-loop
  - $O(n)$ time in total then

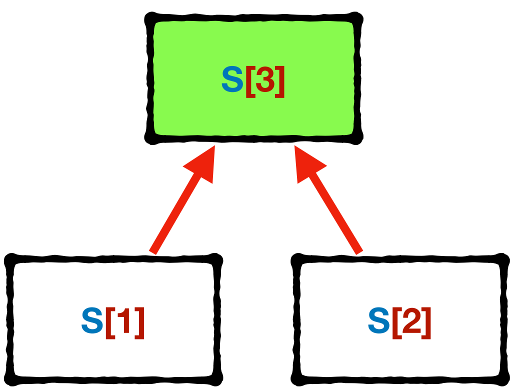
DynFibo(6)

S[1]

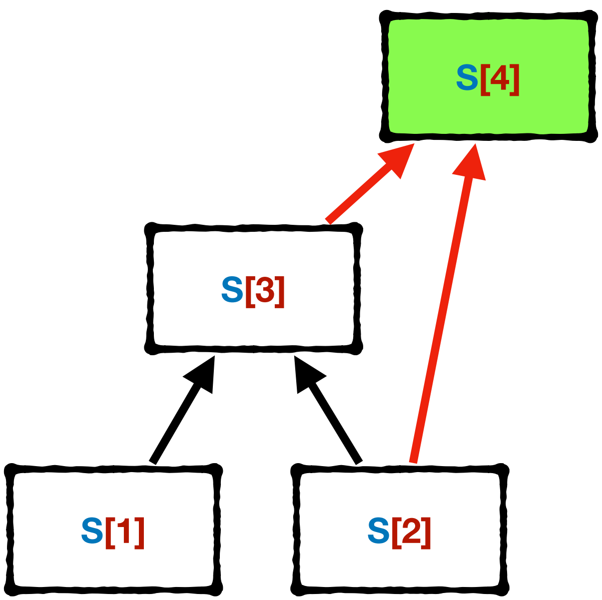
DynFibo(6)

S[1] S[2]

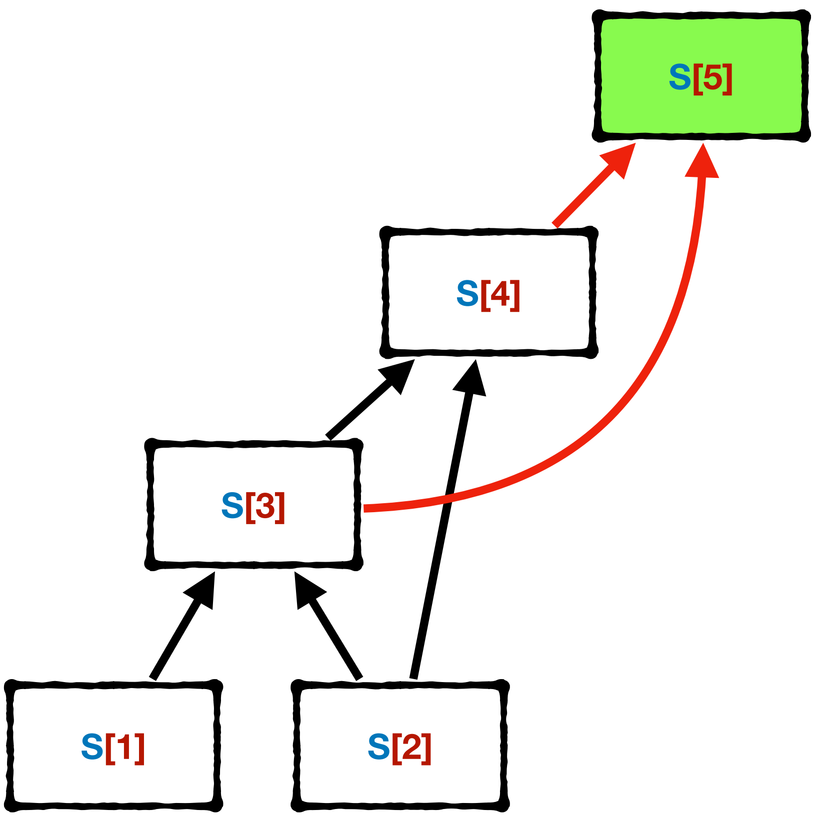
DynFibo(6)



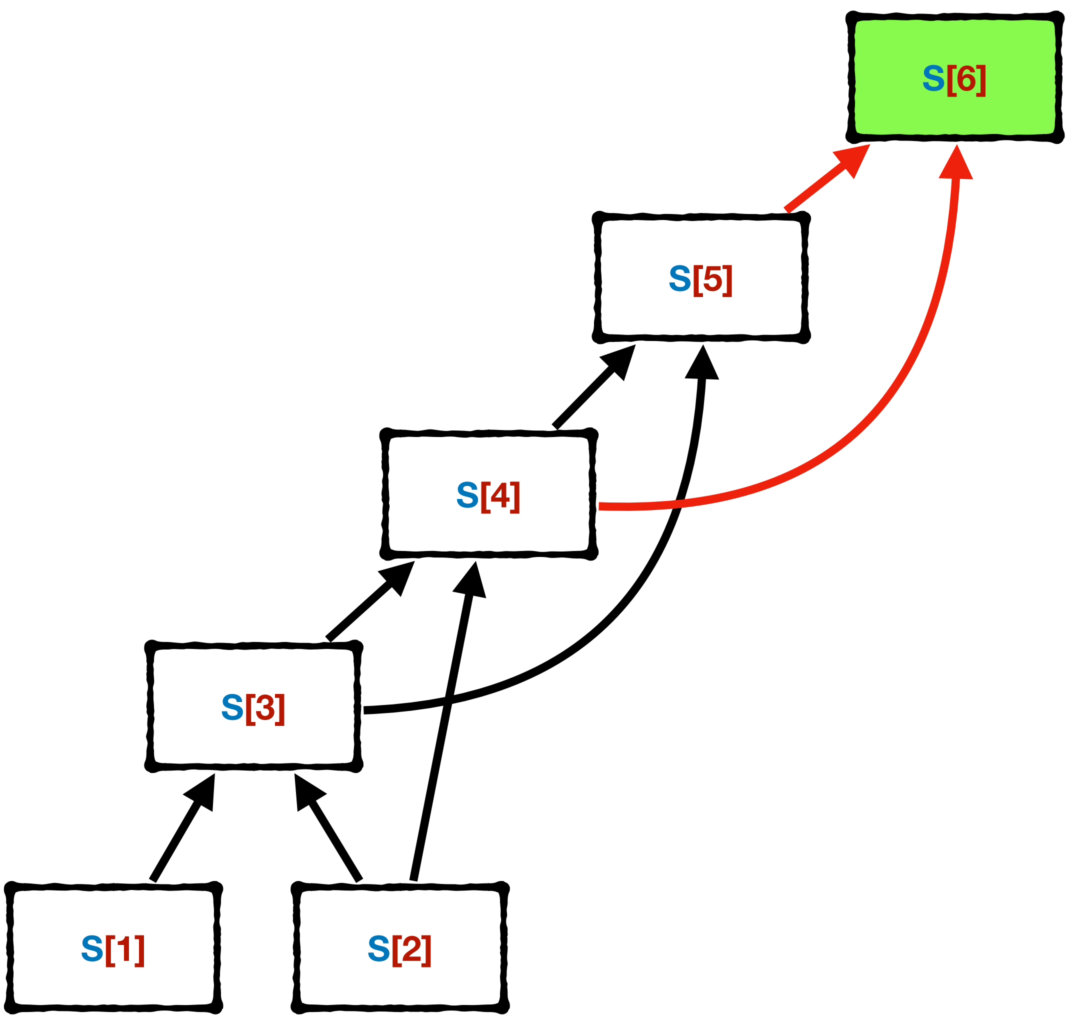
DynFibo(6)



DynFibo(6)



DynFibo(6)



Elements of Dynamic Programming

Dynamic Programming?

- Write a recursive formula for the problem we want to solve
- Write a recursive function that computes the recursive formula
- Use a table to store the answer of recursive function for each recursive call and do not recompute them

Dynamic Programming?

- Write a recursive formula for the problem we want to solve
- Write a recursive function that computes the recursive formula
- Use a table to store the answer of recursive function for each recursive call and do not recompute them
- Two ways:
  - Memoization: Top-Down Dynamic Programming
  - Iterative: Bottom-Up Dynamic Programming

Dynamic Programming?

- Write a recursive formula for the problem we want to solve
- Write a recursi… …sive formula
- Us…
  n…
- Tw…
  - Memoiza…
  - Iterative: Bottom-Up Dynamic Programming

Reminder: Goal of dynamic programming is to speed up computation nothing more

Writing Recursive Formula

- Step One: **Specification**
  - Answer to the question of **“What?”**
  - Describe the problem you want to solve using your formula in plain English
    - Example: “For every $1 \leq i \leq n$, the formula F(i) is supposed to be the i-th Fibonacci number”
  - Describe how the answer to the original problem can be obtained IF we have a solution to the recursive formula
    - Example: Return F(n)”

Writing Recursive Formula

- Step Two: **Solution**
  - Answer to the question of **"How?"**
  - Give a recursive formula for solving for the problem you described in specification by solving the instances of the same problem
    - Example: "$F(1) = F(2) = 1$; for any $i > 2$, $F(i) = F(i-1)+F(i-2)$"
  - Prove that this recursive formula indeed matches the specification provided in the previous step
    - Answer to the question of **"Why?"**

Writing Recursive Formula

- Prove that this recursive formula indeed matches the specification provided in the previous step:
  - Prove that base case of recursive formula is correct
  - Prove that larger values of formula are computed correctly from the smaller values

- Note: this is just induction in disguise

Next Steps?

- Step Three:
  - Use either memoization or bottom-up dynamic programming
  - The choice is entirely up to you
    - Just remember in bottom-up dynamic programming, you have to specify the order of evaluation also

Next Steps?

- Step Four:
  - Analyze the runtime of the algorithm
    - How many subproblems are there?
    - How much time solving each one takes?

- We are done! This is all there is to dynamic programming
- The only thing remained for us is to practice

Dynamic Programming

- Every time you do dynamic programming:
  - Clearly specify subproblems in plain English
  - Design a recursive formula for solving subproblems from smaller ones
  - Prove the correctness of your recursive formula
  - Turn the formula into a dynamic programming algorithm using either memoization or bottom-up approach
  - Analyze the runtime of your algorithm