CS 5112: Data Structures and Algorithms for Applications

Administrivia

- Course Staff:
    - Instructors: Prof. Ramin Zabih (rdz@cs.cornell.edu) and Greg Zecchini (gez3@cornell.edu)
    - TA: Gengmo Qi (gq35@cornell.edu)
    - Consultants/Graders: TBD, office hours schedule forthcoming
- Links to course website and Slack workspace will be emailed out in the next few days

Basic Course Information

- CS5112 work will be constant but not time intensive
  - Roughly 4 programming assignments
  - Weekly quizzes
- Prelim on 10/9 (tentative) and final on 12/9 (last day of class)
  - Exams will be in-class and closed book
- Ramin and Greg will lecture, with the possible occasional guest

Academic Integrity

- Each student is expected to abide by the Cornell University Code of Academic Integrity
  - http://theuniversityfaculty.cornell.edu/academic-integrity/
- Any work submitted by a student in this course for academic credit will be the student's own work.
  - Exception: some assignments may be designed for groups of two, in which case the group will obviously submit shared work together
- We take this very seriously. Students have been expelled from Cornell for violations. Copying code is easy to catch.

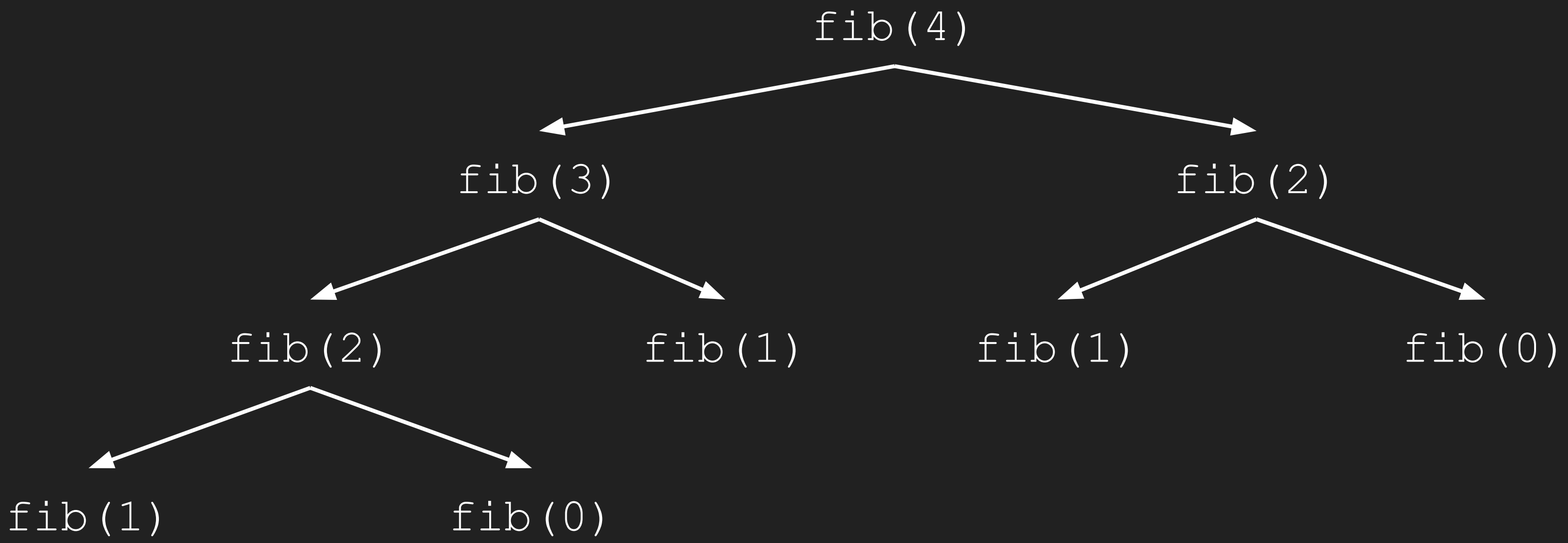
Dynamic Programming

Fibonacci Sequence

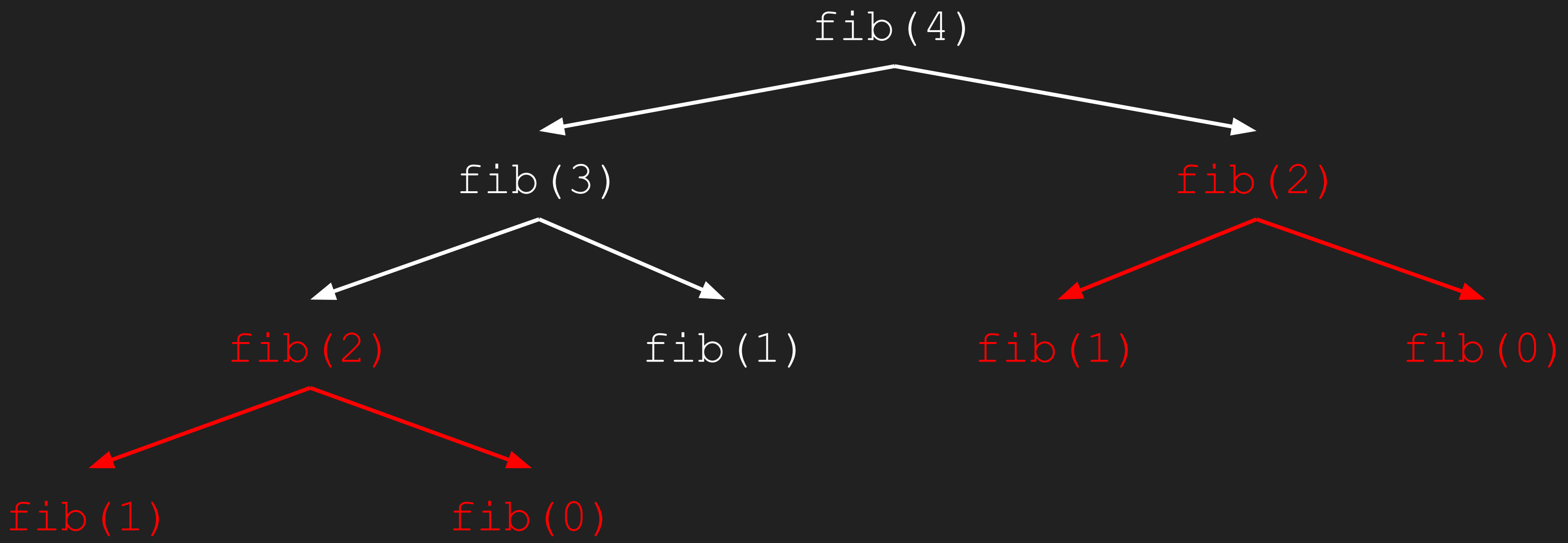
- 0 1 1 2 3 5 8 13 21 …
- First two numbers are 0 and 1, all subsequent numbers are the sum of the two prior numbers
- How do we find the n^{th} fibonacci number?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)
```

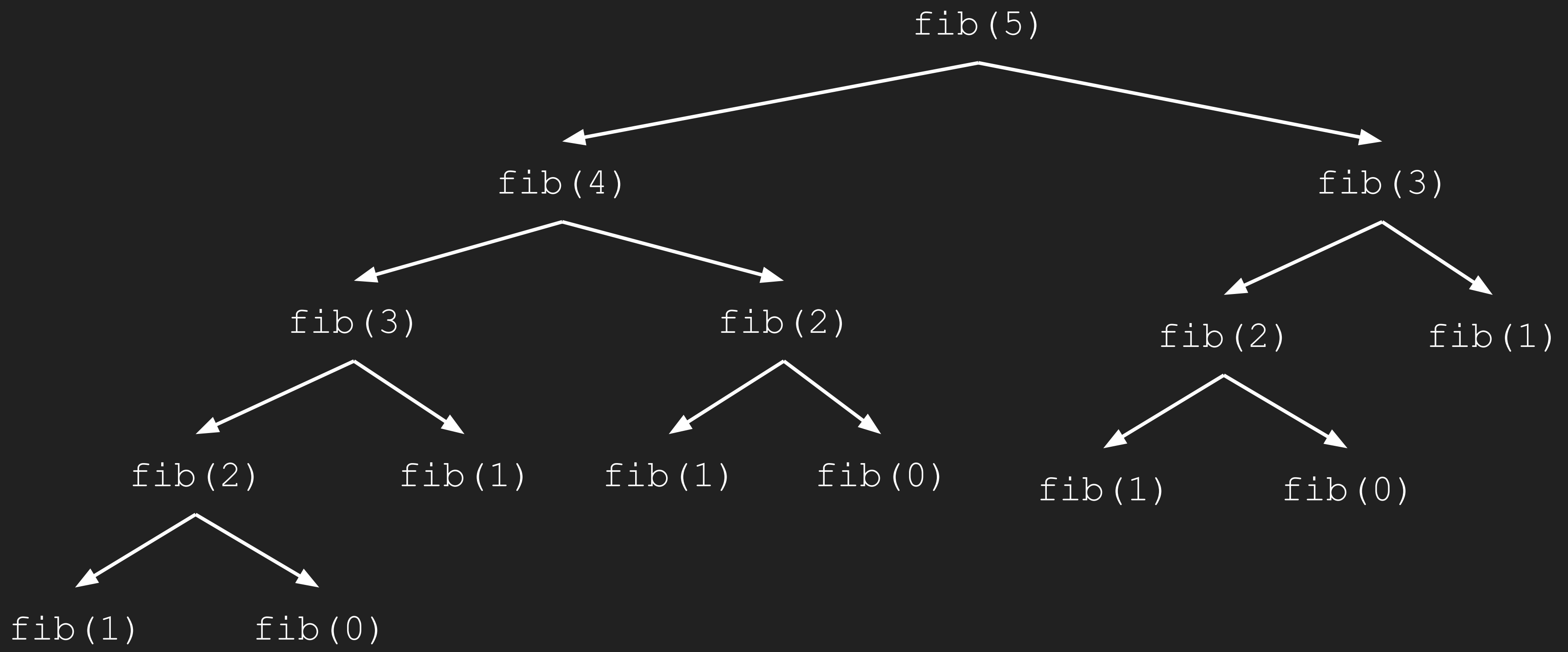
Fibonacci Sequence



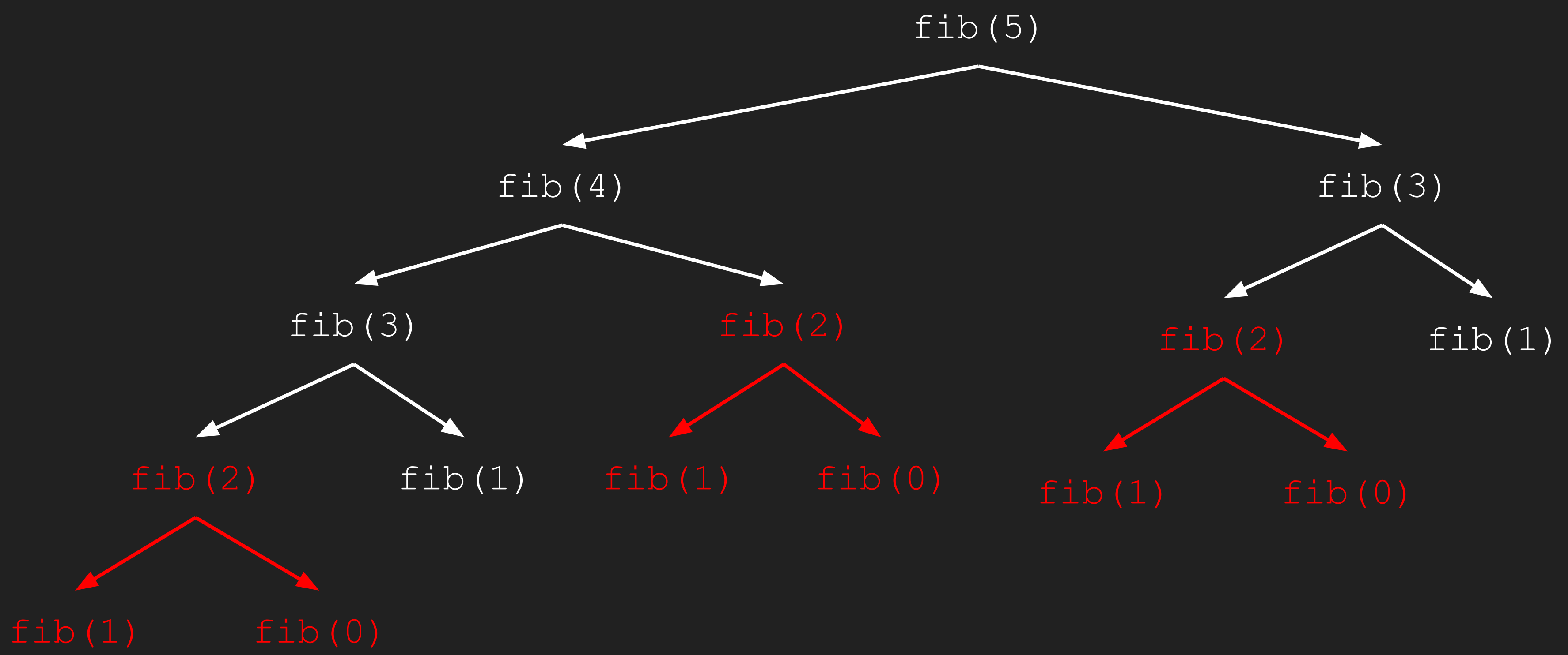
Fibonacci Sequence



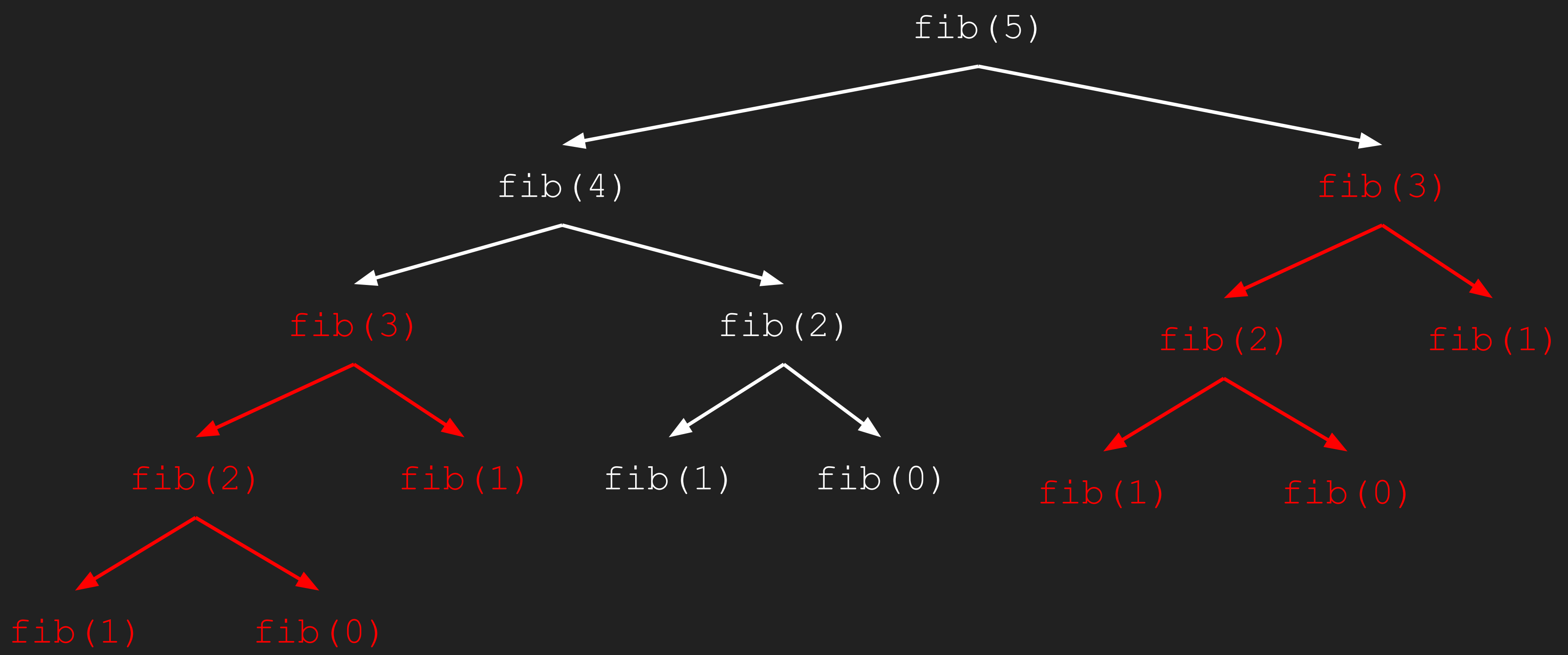
Fibonacci Sequence



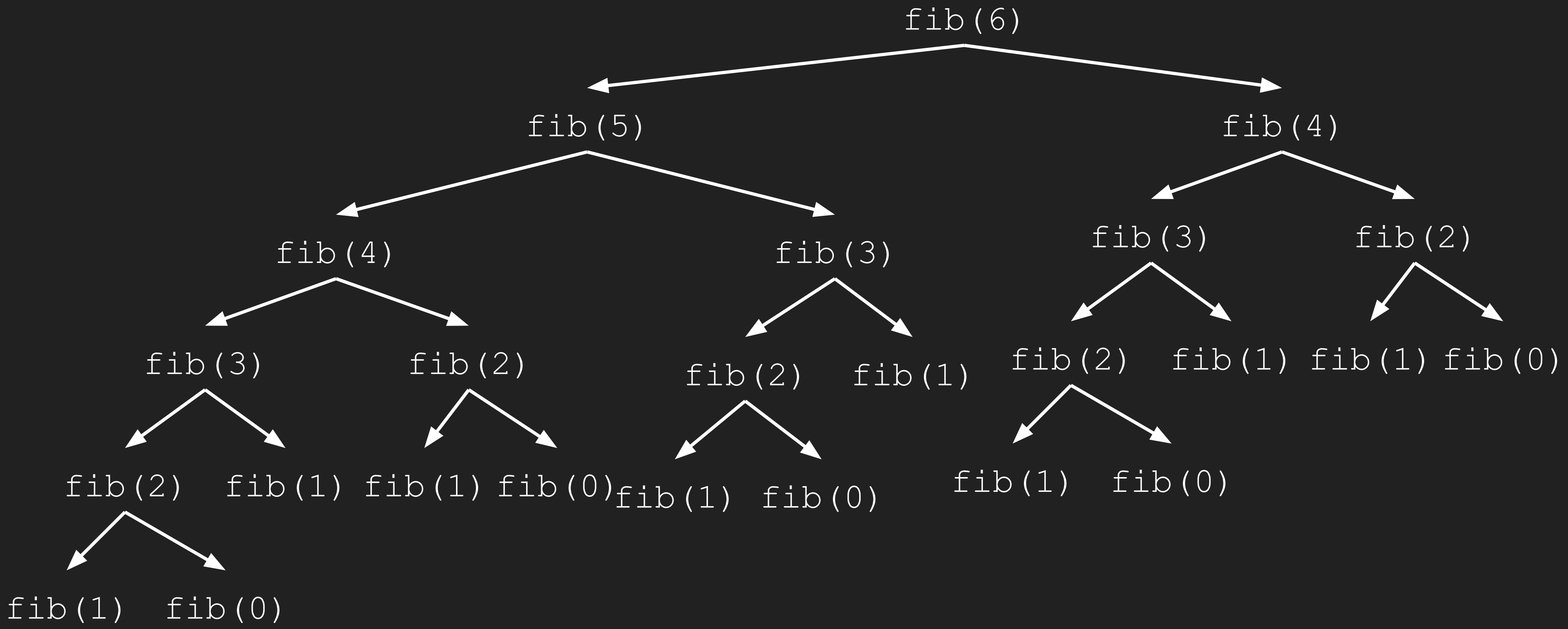
Fibonacci Sequence



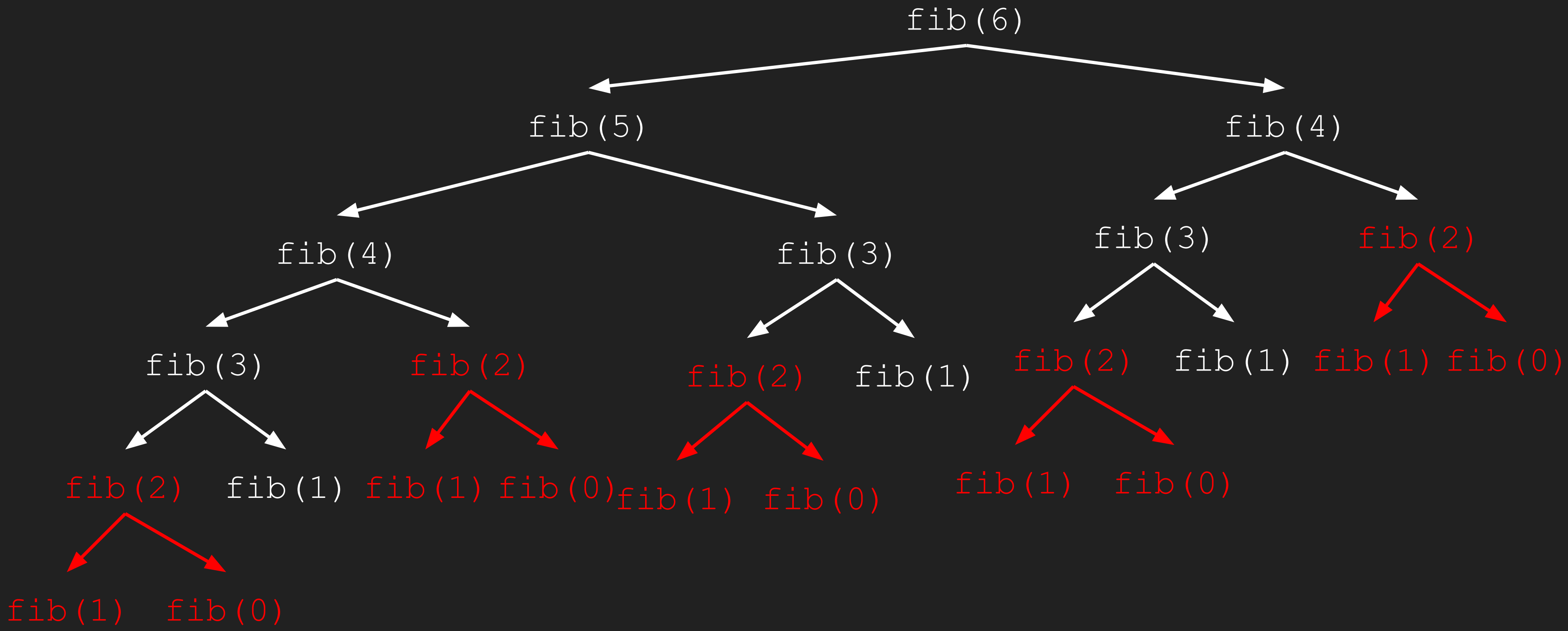
Fibonacci Sequence



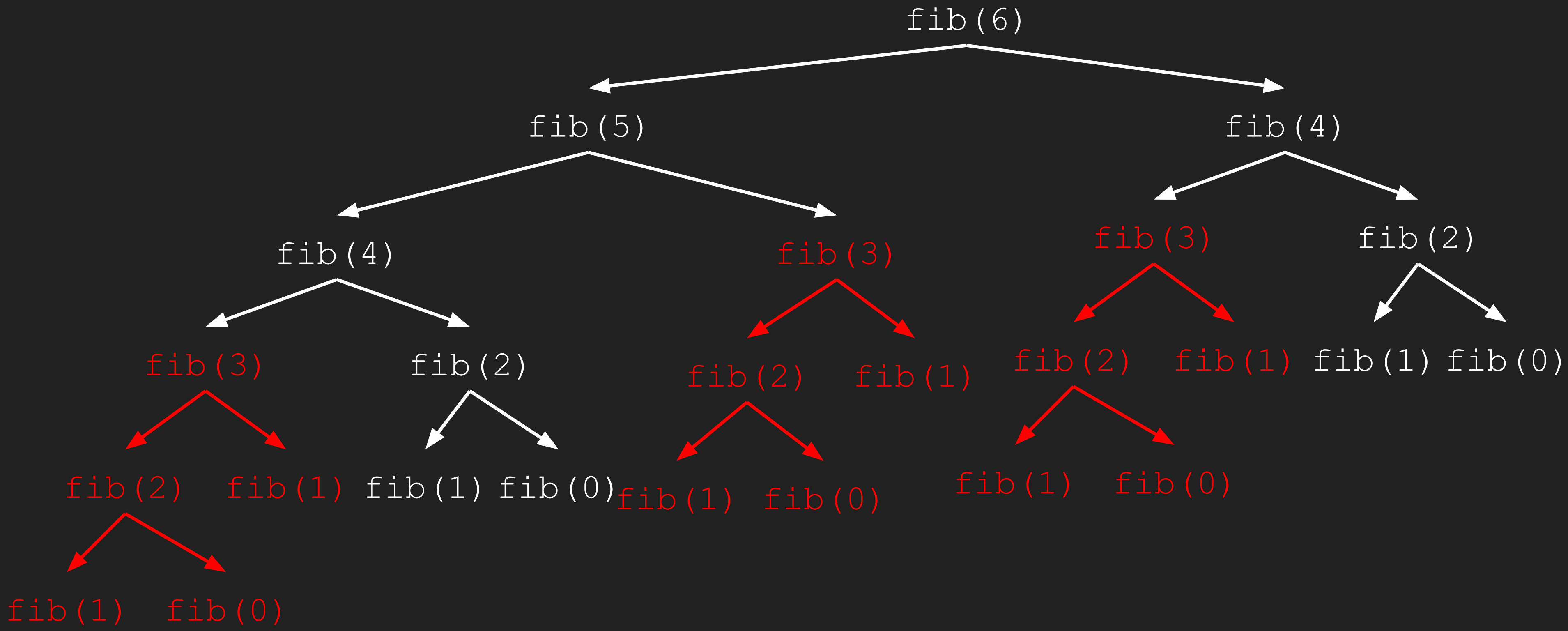
Fibonacci Sequence



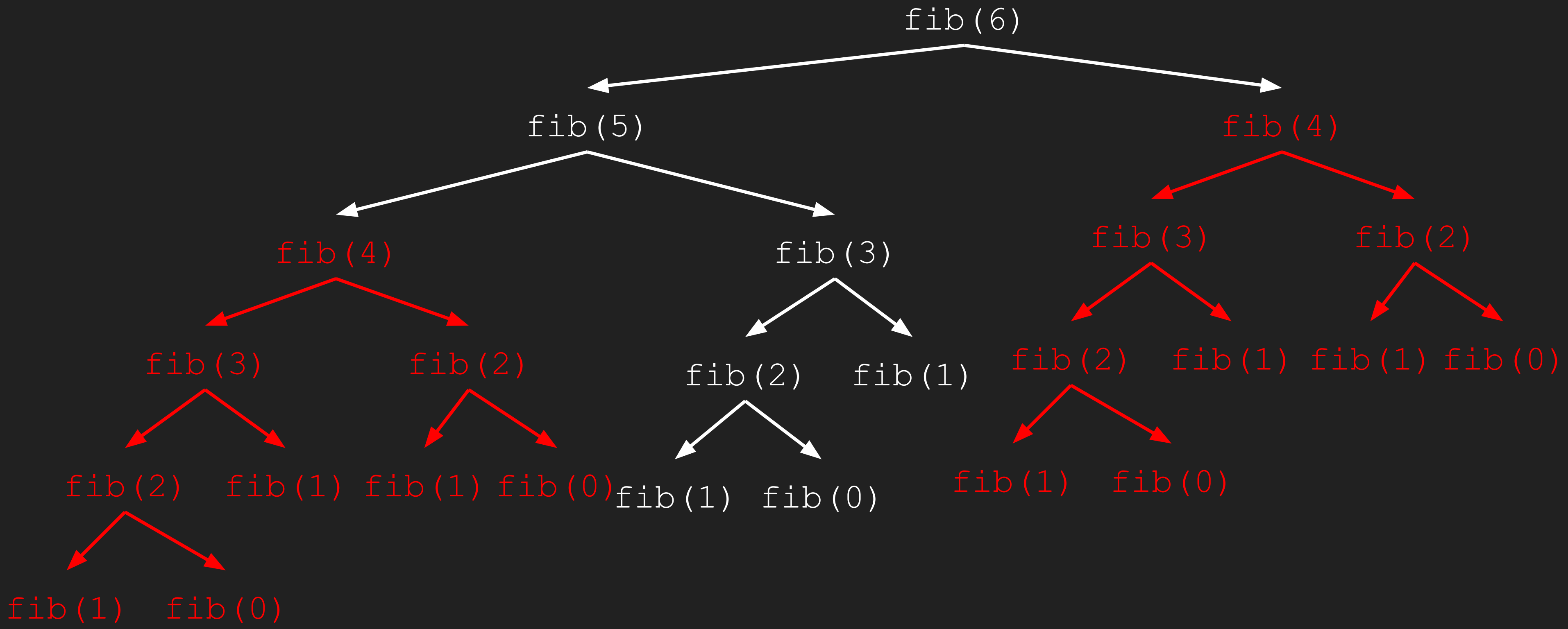
Fibonacci Sequence



Fibonacci Sequence



Fibonacci Sequence



Fibonacci Sequence

- Obvious algorithm ends up duplicating a lot of work!
- Runtime complexity is $O(2^n)$ - not good!
- How can we avoid doing duplicate work?

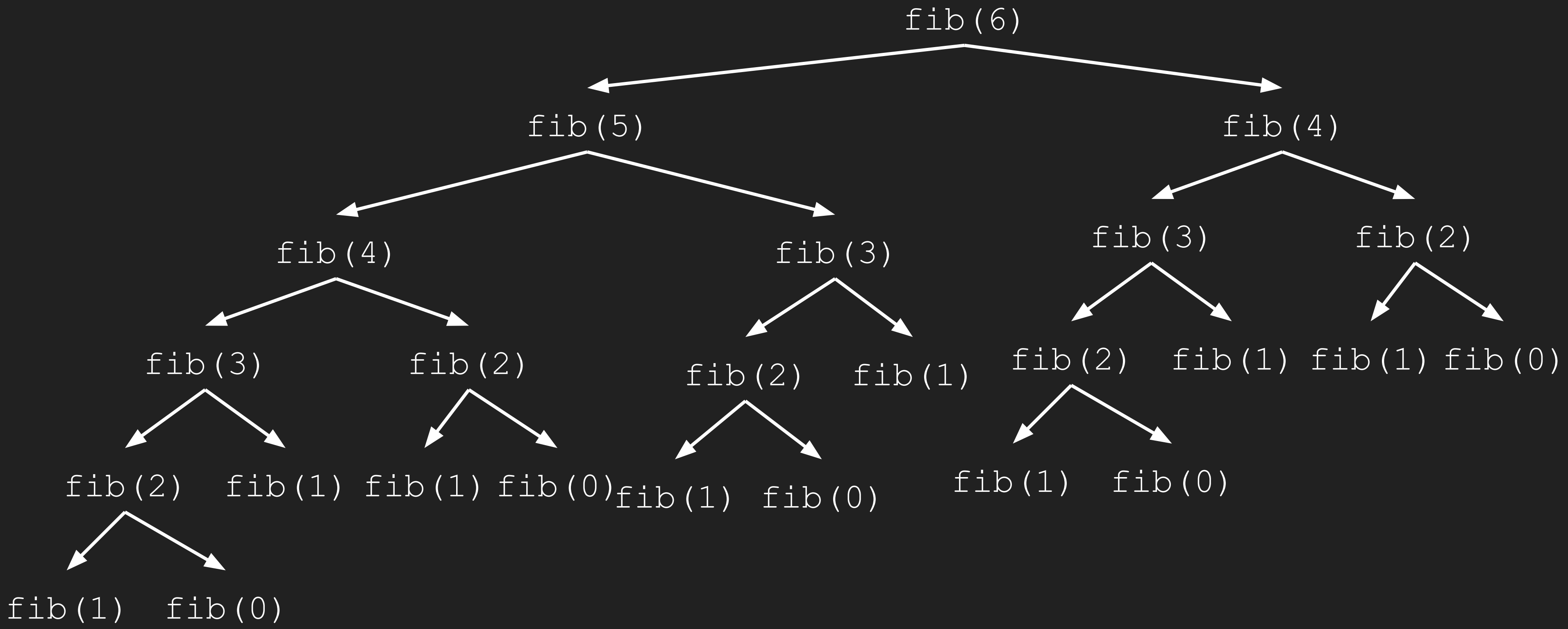
Runtime Complexity

- Essentially how the amount of computation the algorithm does scales with the size of the input.
  - i.e., as the input gets bigger, how much worse does the algorithm perform?

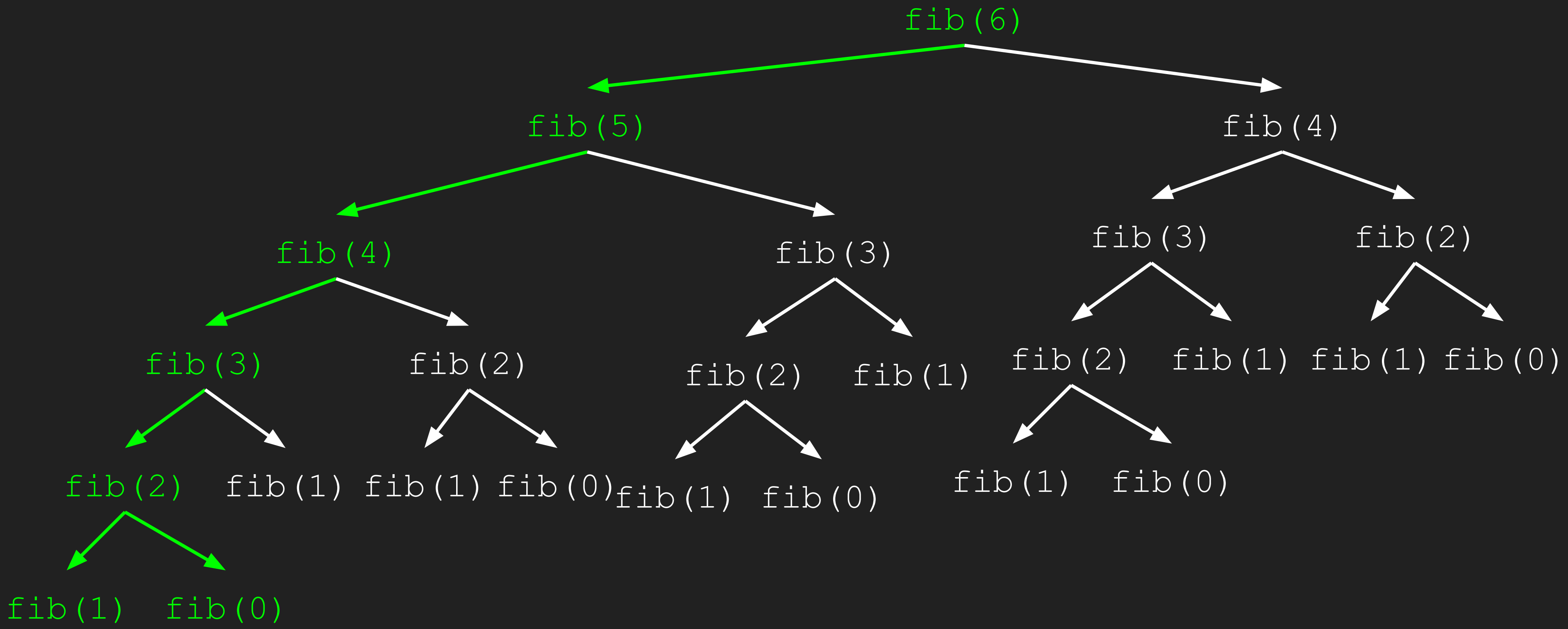
Runtime Complexity	Rough implication*
$O(1)$	awesome!
$O(\log n)$	fantastic!
$O(n)$	great!
$O(n \log n)$	pretty good!
$O(n^2)$	ok!
$O(2^n)$	very bad!
$O(n!)$	extremely bad!
$O(n^n)$	complete disaster!

*Caveat: sometimes it's just the best you can do

Fibonacci Sequence



Fibonacci Sequence

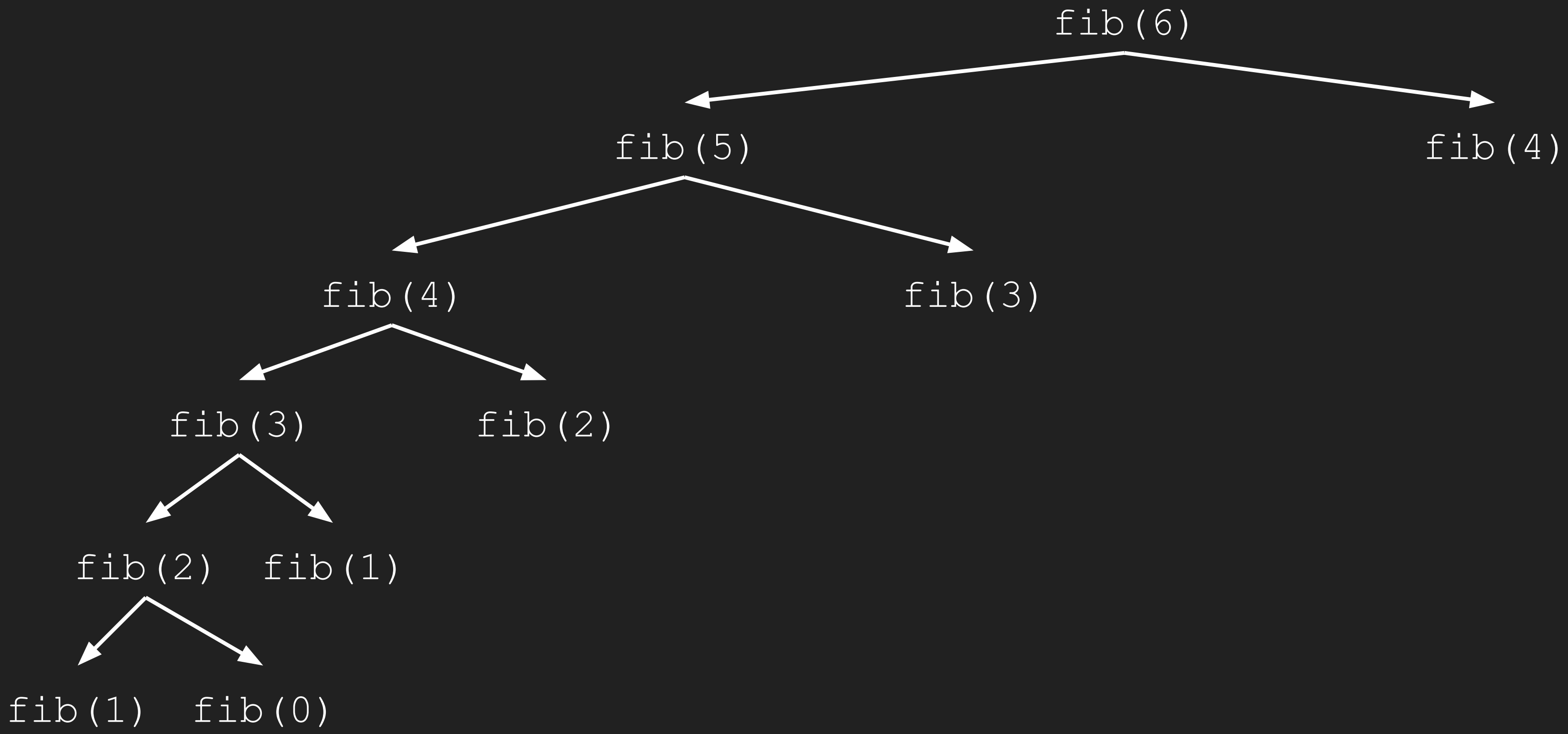


Fibonacci Sequence

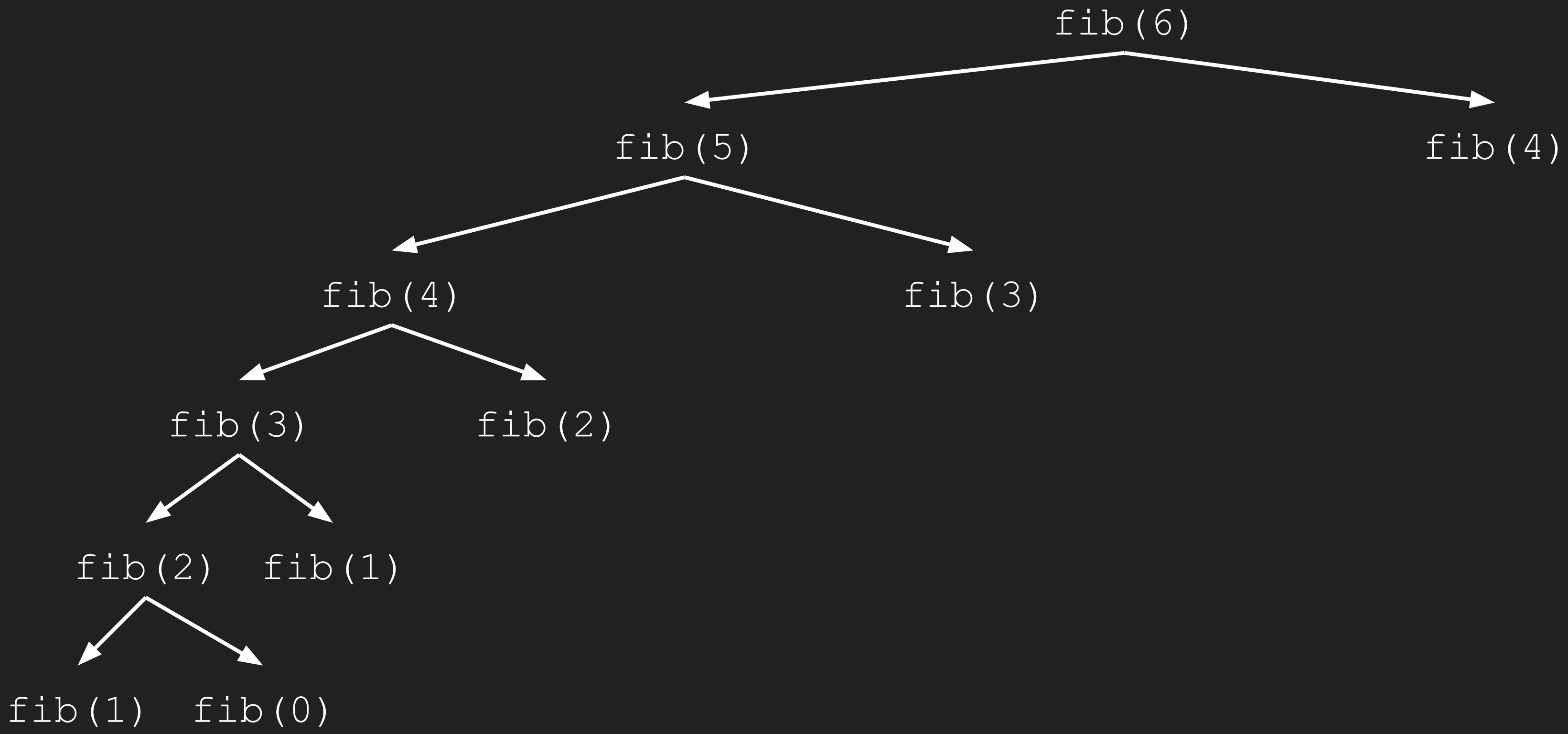
- Big idea: memoization
  - Essentially: save the work you’ve done in the past so you can reuse it later

```
mem = {0:0, 1:1}
def fib(n):
    if n not in mem:
        mem[n] = fib(n-1) + fib(n-2)
    return mem[n]
```

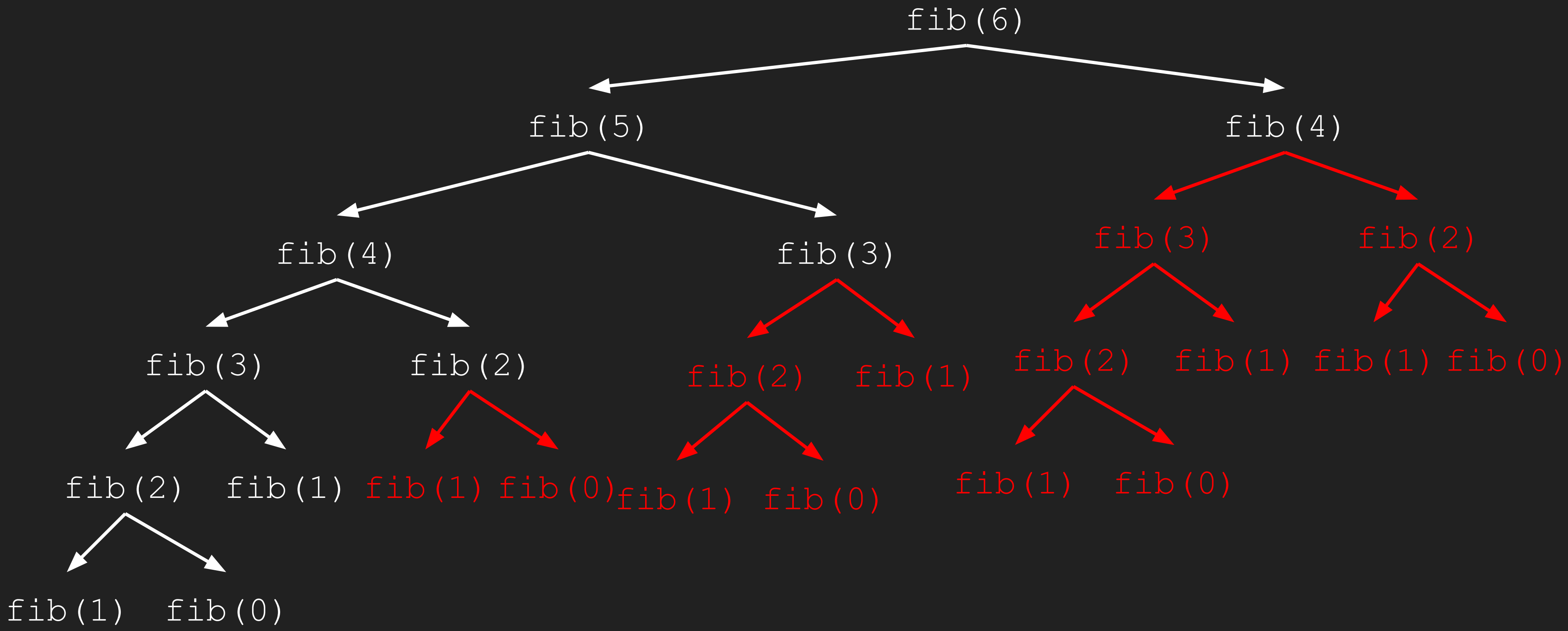
Fibonacci Sequence with Memoization



Fibonacci Sequence with Memoization



Fibonacci Sequence



Fibonacci Sequence

- New runtime with memoization: $O(n)$
- Going from $O(2^n)$ to $O(n)$ is a **massive** improvement!

Dynamic Programming

- Useful technique to solve problems that have an "optimal substructure."
  - i.e. an optimal solution to a problem can be built from optimal solutions to subproblems
  - Ex. fib(n-1) and fib(n-2) can be used to calculate fib(n)
- Dynamic Programming also requires "overlapping subproblems."
  - i.e. there is shared work in the recursive calls
  - Ex. fib(n) = fib(n-1) + fib(n-2)   <- notice that fib(n-1) can be expanded to also need fib(n-2)
  - Note: if subproblems don't overlap, you may still be able to develop a "Divide and Conquer" algorithm

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)
```

```
mem = {0:0, 1:1}
def fib(n):
    if n not in mem:
        mem[n] = fib(n-1) + fib(n-2)
    return mem[n]
```

DP Example: Longest Common Subsequence

- Define a subsequence of a string *s* to be a string *s'* where all characters of *s'* appear in *s* and are in the same order in both *s* and *s'*.
  - Example: MTA, H, ATTN, HAT are all subsequences of MANHATTAN, but TAM is not
- Problem statement: given two strings *s* and *t*, find the longest subsequence common to both strings.
  - Example: if our strings are ITHACA and MANHATTAN, the LCS would be HAA
- Brute force: enumerate all subsequences of *s* and check if each is a subsequence of *t*.
  - Runtime complexity: $O(2^n)$

DP Example: Longest Common Subsequence

- Does this problem have an optimal substructure?
- Observation #1:
  - If at least one of *s* or *t* is the empty string, then *LCS(s, t)* is also the empty string

DP Example: Longest Common Subsequence

- Observation #2:
    - Consider the case where *s* and *t* end in the same letter. Example: MANHATTAN and MADMEN
    - Since we know they both end in N, let's guess that *LCS(*MANHATTAN, MADMEN*)* ends in N
    - Consider *LCS(*MANHATTA, MADME*)*
        - By inspection, this equals MA
    - Therefore *LCS(*MANHATTA, MADME*)* + N = MAN = *LCS(*MANHATTAN, MADMEN*)*
    - More generally,

If $s_n = t_m$,

$$LCS(s_1 \ldots s_n, t_1 \ldots t_m) = LCS(s_1 \ldots s_{n-1}, t_1 \ldots t_{m-1}) + t_m$$

DP Example: Longest Common Subsequence

- Observation #3:
  - Consider the case where *s* and *t* do NOT end in the same letter. Example: MANHATTAN and ITHACA
  - Case 1: *LCS(*MANHATTAN, ITHACA*)* does NOT end in N
    - If so, we don't need it, so *LCS(*MANHATTAN, ITHACA*)* = *LCS(*MANHATTA, ITHACA*)*
  - Case 2: *LCS(*MANHATTAN, ITHACA*)* ends in N
    - If so, we don't need the A at the end of ITHACA, so *LCS(*MANHATTAN, ITHACA*)* = *LCS(*MANHATTAN, ITHAC*)*
  - But… we don't know which case is true *a priori*
  - So, generally:

If $s_n \neq t_m$,

$$LCS(s_1 \ldots s_n, t_1 \ldots t_m) = max(LCS(s_1 \ldots s_{n-1}, t_1 \ldots t_m) + LCS(s_1 \ldots s_n, t_1 \ldots t_{m-1}))$$

DP Example: Longest Common Subsequence

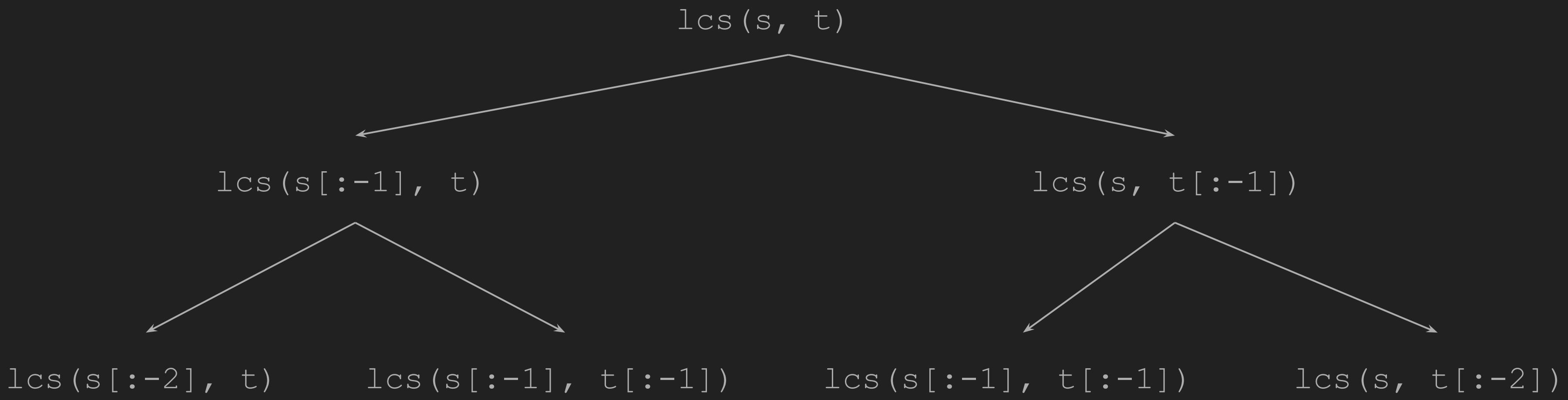
$$LCS(s_1...s_n, t_1...t_m) = \begin{cases} \text{“ ”} & \text{if } n = 0 \text{ or } m = 0 \\ LCS(s_1...s_{n-1}, t_1...t_{m-1}) + t_\mathrm{m} & \text{if } s_n = t_m \\ max(LCS(s_1...s_{n-1}, t_1...t_m), LCS(s_1...s_n, t_1...t_{m-1})) & \text{otherwise} \end{cases}$$

Does this problem have an optimal substructure? Yes!

LCS: Naive Implementation

```
def lcs(s, t):
    if len(s) == 0 or len(t) == 0:
        return “”
    if s[-1] == t[-1]:
        return lcs(s[:-1], t[:-1]) + t[-1]
    tmp1 = lcs(s[:-1], t)
    tmp2 = lcs(s, t[:-1])
    return tmp1 if len(tmp1) > len(tmp2) else tmp2
```

LCS: Naive Implementation



LCS: Naive Implementation

```
lcs(s, t)

lcs(s[:-1], t)                              lcs(s, t[:-1])

lcs(s[:-2], t)   lcs(s[:-1], t[:-1])   lcs(s[:-1], t[:-1])   lcs(s, t[:-2])
```

Runtime complexity: $O(2^n)$

LCS: Recursive Implementation with Memoization

```
mem = {}
def lcs(s, t):
    if (s, t) in mem:
        return mem[(s, t)]
    if len(s) == 0 or len(t) == 0:
        return “”
    if s[-1] == t[-1]:
        mem[(s, t)] = lcs(s[:-1], t[:-1]) + t[-1]
    else:
        tmp1 = lcs(s[:-1], t)
        tmp2 = lcs(s, t[:-1])
        mem[(s, t)] = tmp1 if len(tmp1) > len(tmp2) else tmp2
    return mem[(s, t)]
```

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”						
A						
G						
T						

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”					
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""					
G	""					
T	""					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”				
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”				
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A			
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A			
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A		
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""					
T	""					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”				
T	“”					

LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""				
T	""					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A			
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A			
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG		
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

LCS: Iterative Implementation with “table filling”

```
def lcs(s, t):
    matrix = [[“” for x in range(len(t)+1)] for y in range(len(s)+1)]
    for i in range(1, len(s)+1):
        for j in range(1, len(t)+1):
            if s[i-1] == t[j-1]:
                matrix[i][j] = matrix[i-1][j-1] + t[j-1]
            else:
                tmp1 = matrix[i-1][j]
                tmp2 = matrix[i][j-1]
                matrix[i][j] = tmp1 if len(tmp1) > len(tmp2) else tmp2
    return matrix[len(s)][len(t)]
```