Dynamic Programming

Giulio Ermanno Pibiri

Department of Environmental Sciences, Informatics and Statistics
Ca' Foscari University of Venice
giulioermanno.pibiri@unive.it

What is “Dynamic Programming”?

- **Dynamic Programming (DP)** is a technique that combines the correctness of complete search and the efficiency of greedy algorithms.
- The term was coined by Richard Bellman in the 1950s. *“Programming”* in this context refers to writing results to an array, not to writing computer code. *“Dynamic”* instead refers to the fact that a DP algorithm solves different sub-problems, hence it has a “dynamic” nature.
- It is a **technique**, not a specific algorithm.
- **Our goal for today:** understand **when and how to apply DP** to solve problems, using several examples.

Overview

- General concepts
- Warm-up: Fibonacci numbers, recursion tree
- Problems: coins, rod cutting, shortest paths on DAGs
- Properties: optimal sub-structure and problem overlaps
- More problems: longest increasing subsequence, edit distance

Introduction

- DP typically applies to **optimization problems**, where we make a set of choices to arrive to an optimal solution.

- *An* optimal solution (note: not *the* optimal solution — a problem may have many optimal solutions) usually asks to minimize/maximize some value.

Introduction

- DP typically applies to **optimization problems**, where we make a set of choices to arrive to an optimal solution.

- *An* optimal solution (note: not *the* optimal solution — a problem may have many optimal solutions) usually asks to minimize/maximize some value.

- Key insight into the technique: as we make choices, **sub-problems of the same form often arise again**.

- **Idea.** Solve each of these problems only **once** and store the solution somewhere (e.g. in an array or hash table) for later use. Should the same sub-problem arise again, you will already have the solution computed! A good example of a **space-time tradeoff**.

Introduction

- DP typically applies to **optimization problems**, where we make a set of choices to arrive to an optimal solution.

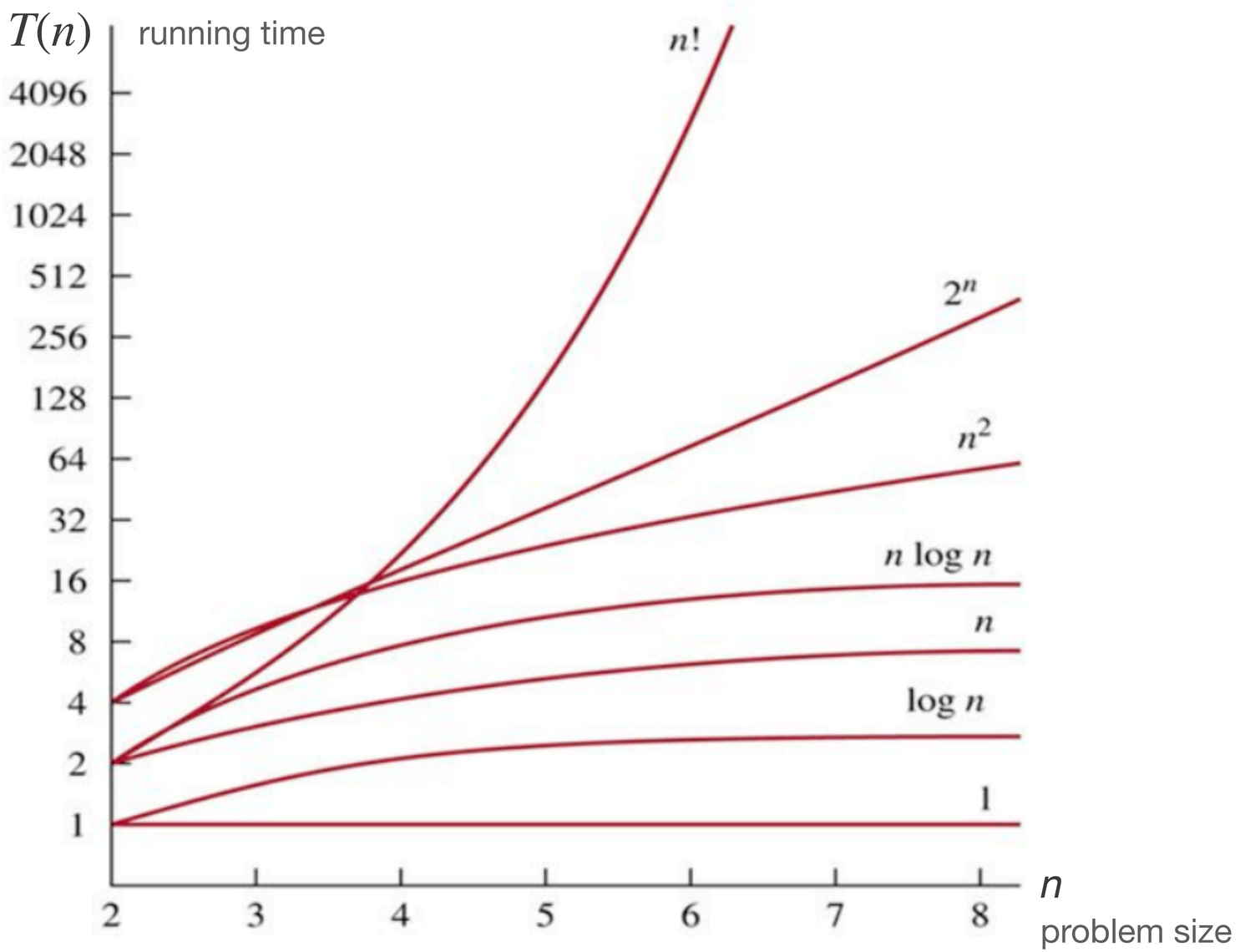
- *An* optimal solution (note: not *the* optimal solution — a problem may have many optimal solutions) usually asks to minimize/maximize some value.

- Key insight into the technique: as we make choices, **sub-problems of the same form often arise again**.

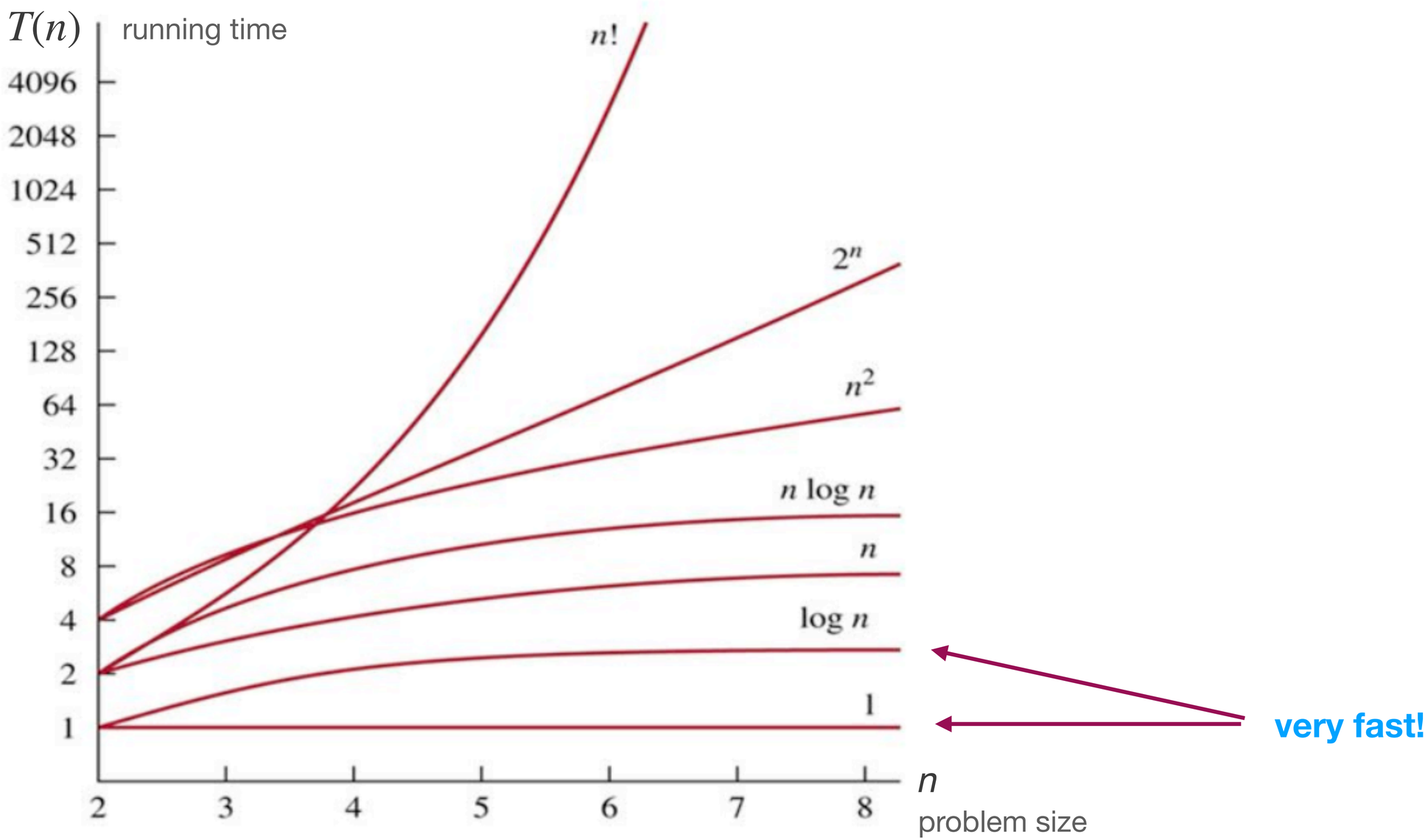
- **Idea.** Solve each of these problems only **once** and store the solution somewhere (e.g. in an array or hash table) for later use. Should the same sub-problem arise again, you will already have the solution computed! A good example of a **space-time tradeoff**.

- This idea often translates **exponential-time** algorithms into **polynomial-time** algorithms.

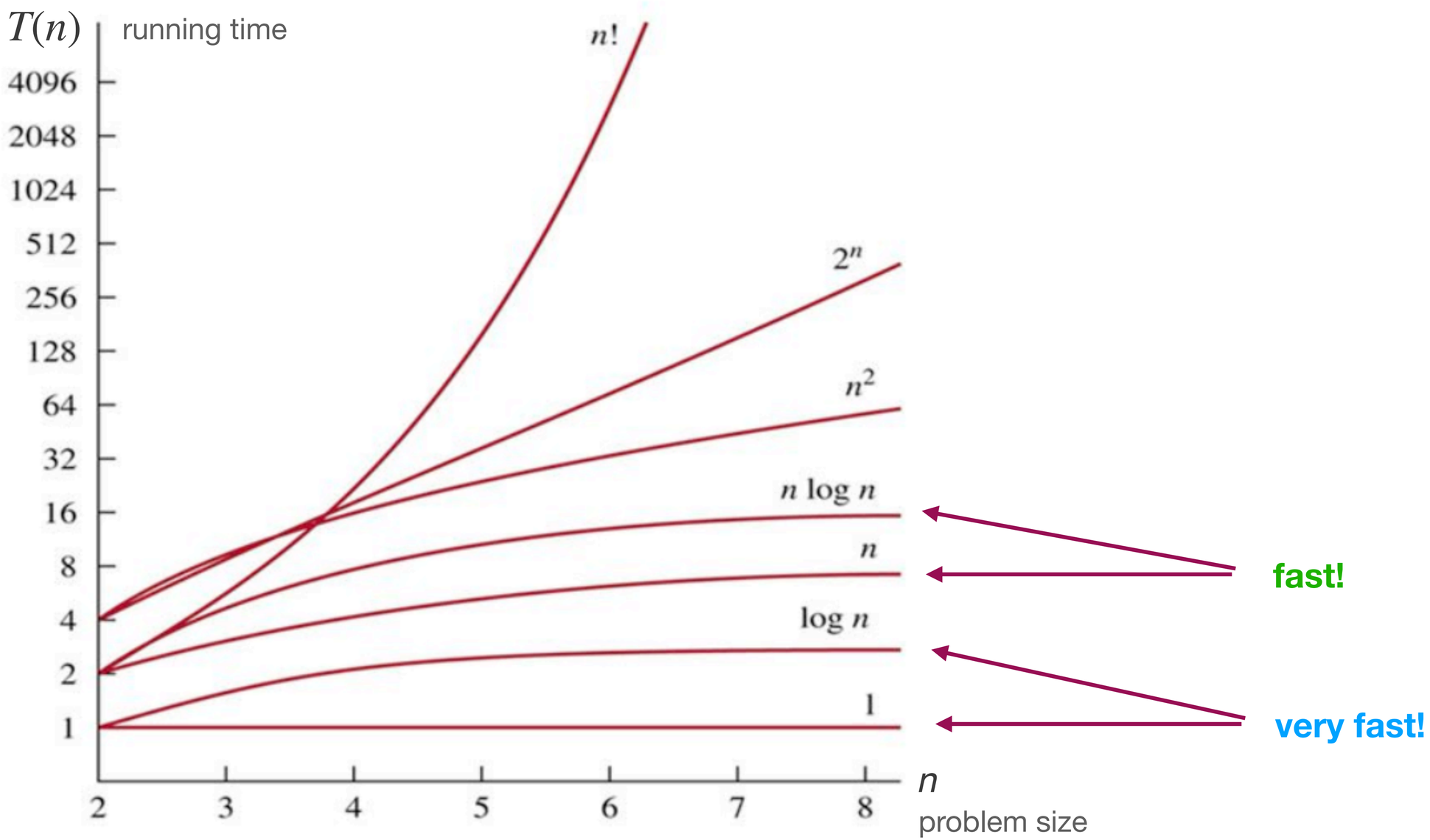
Growth of running time



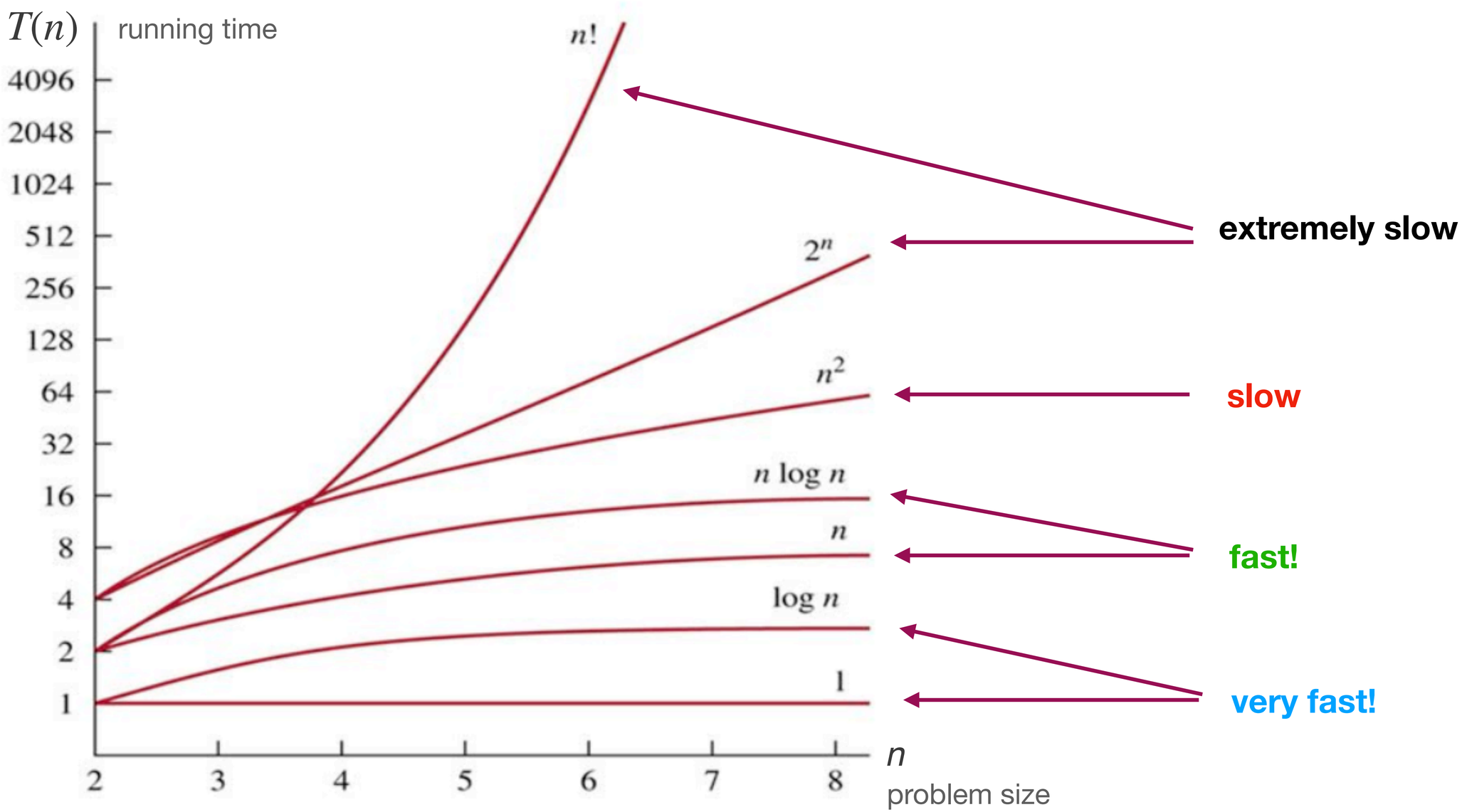
Growth of running time



Growth of running time



Growth of running time



Introduction

- We will see that many problems solvable via DP have a clean **recursive definition** of the optimal solution (optimal sub-structure).

- After we write the solution in this form, we can easily implement a recursive algorithm to solve the problem, but...

Introduction

- We will see that many problems solvable via DP have a clean **recursive definition** of the optimal solution (optimal sub-structure).
- After we write the solution in this form, we can easily implement a recursive algorithm to solve the problem, but...
- The algorithm will do a lot of **redundant work**, actually **re-computing** the solution to sub-problems it has already encountered.
- Let’s consider a simple example to warm up: **Fibonacci numbers**.

Fibonacci numbers

- The so-called *Fibonacci numbers* is the following infinite sequence:

$$0,1,1,2,3,5,8,13,21,34,55,\ldots$$

- Let us call $F(n)$ the n-th Fibonacci number, for $n \geq 0$.
- We have $F(0) = 0$ and $F(1) = 1$.
- What is $F(n)$ for $n \geq 2$?

Leonardo da Pisa (*Fibonacci*)
1170 — 1242, Pisa

Fibonacci numbers

- The so-called *Fibonacci numbers* is the following infinite sequence:

$$0,1,1,2,3,5,8,13,21,34,55,\ldots$$

- Let us call $F(n)$ the n-th Fibonacci number, for $n \geq 0$.
- We have $F(0) = 0$ and $F(1) = 1$.
- What is $F(n)$ for $n \geq 2$?
- In general

$$F(n) = \begin{cases} F(n-2) + F(n-1) & \text{if } n \geq 2 \\ n & \text{otherwise} \end{cases} .$$

Leonardo da Pisa (*Fibonacci*)
1170 — 1242, Pisa

Fibonacci numbers

- **Problem.** Given an integer $n \geq 0$, compute $F(n)$.
- Here is a simple **iterative** implementation.

```
3   uint64_t fibonacci_iterative(const uint64_t n) {
4       uint64_t F_n_2 = 0;
5       uint64_t F_n_1 = 1;
6       for (uint64_t i = 0; i != n; ++i) {
7           uint64_t F_n = F_n_2 + F_n_1;
8           F_n_2 = F_n_1;
9           F_n_1 = F_n;
10      }
11      return F_n_2;
12  }
```

Fibonacci numbers

- Let’s consider the following recursive implementation.

```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```

which is exactly the “translation” in code of the previous recurrence relation

$$F(n) = \begin{cases} F(n-2) + F(n-1) & \text{if } n \geq 2 \\ n & \text{otherwise} \end{cases}.$$

Fibonacci numbers

- Let’s consider the following recursive implementation.

```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```

which is exactly the “translation” in code of the previous recurrence relation

$$F(n) = \begin{cases} F(n-2) + F(n-1) & \text{if } n \geq 2 \\ n & \text{otherwise} \end{cases}.$$

- **Q.** What is the complexity of this solution?

Recursion tree

Example for `fibonacci_recursive(4)`.

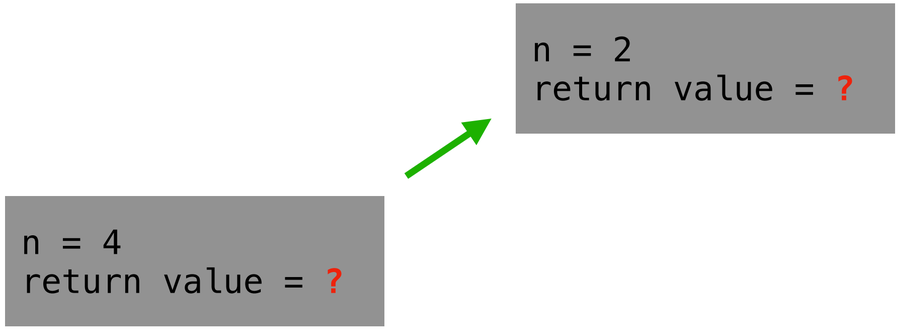
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```

```
n = 4
return value = ?
```

Recursion tree

Example for `fibonacci_recursive(4)`.

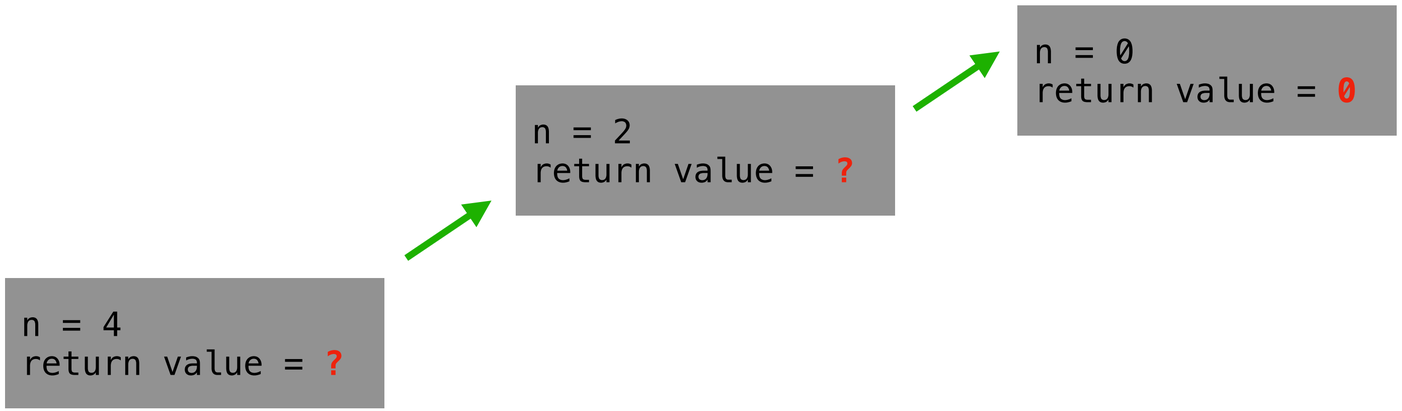
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

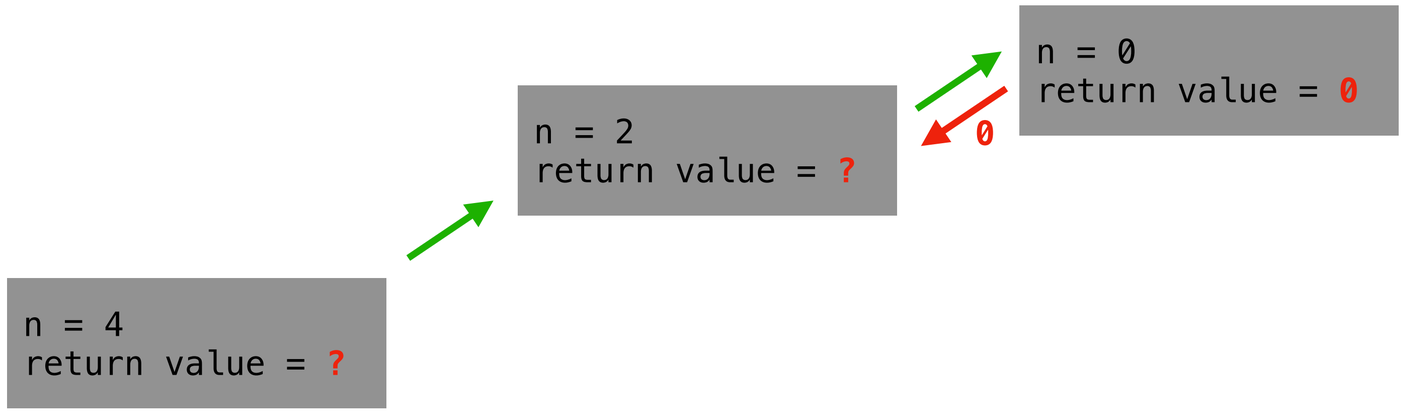
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

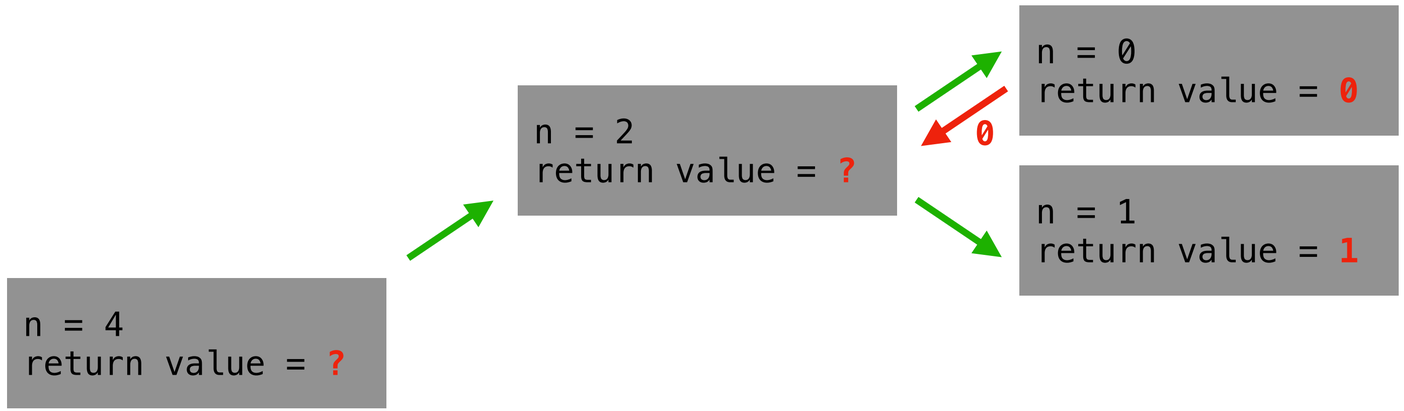
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

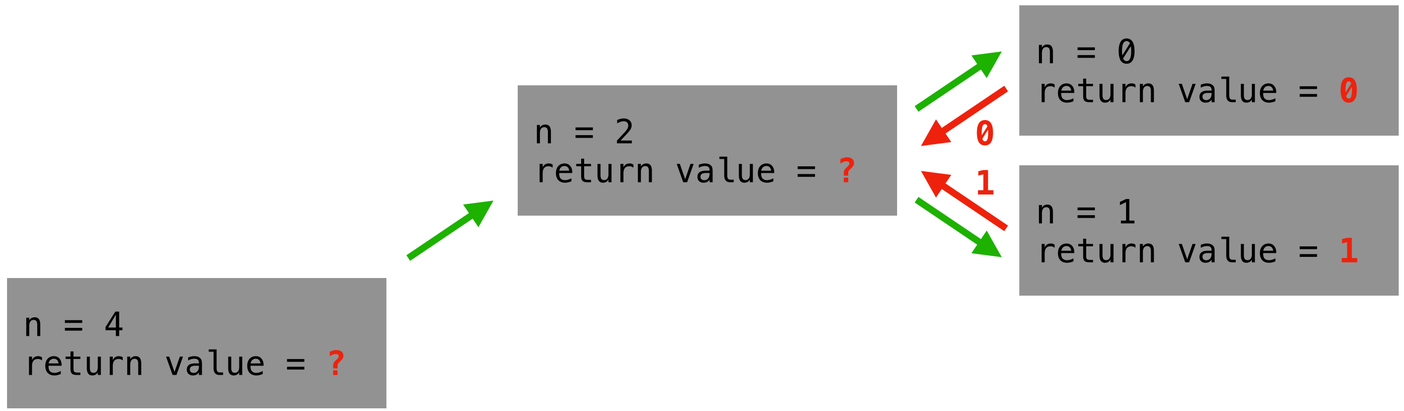
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

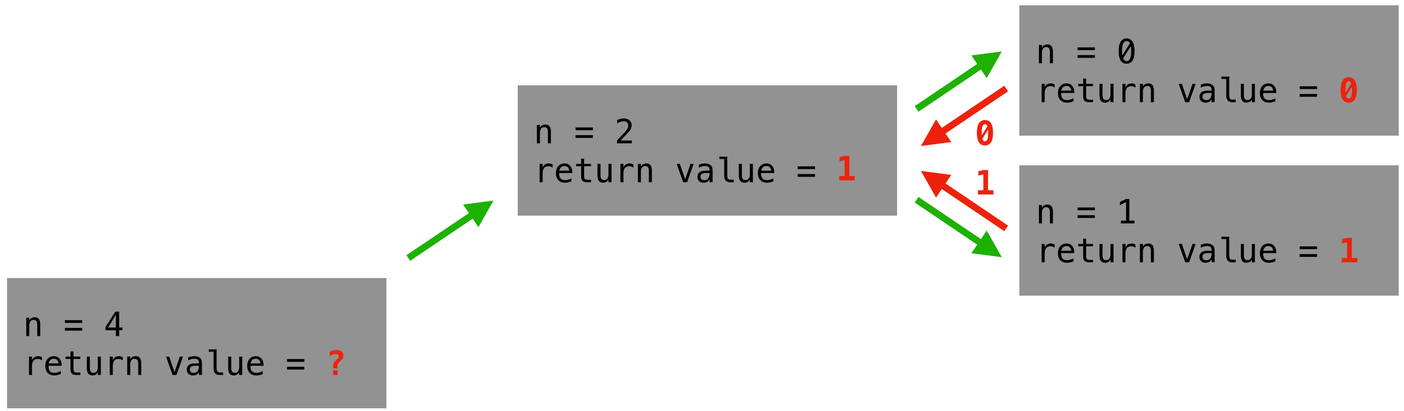
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

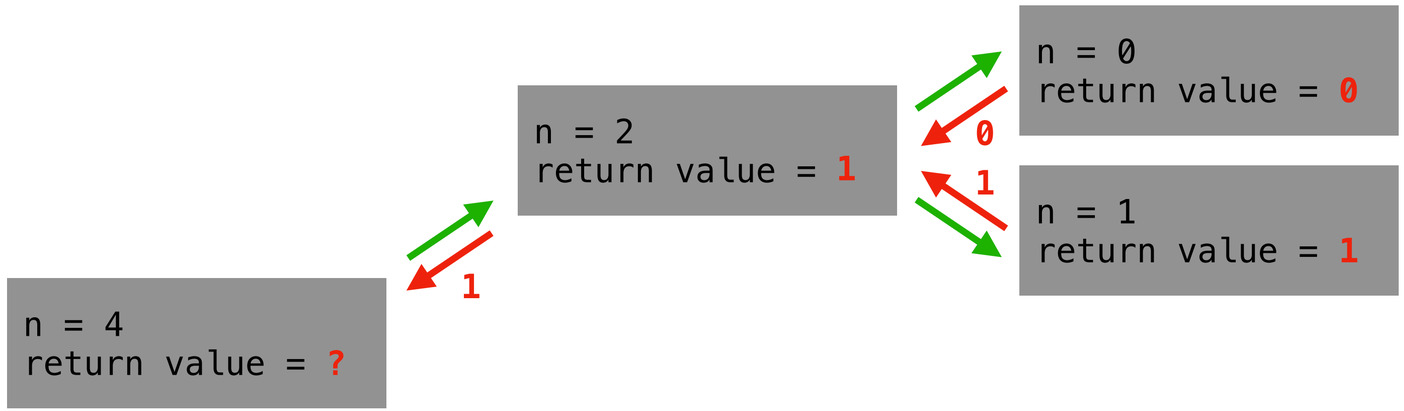
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

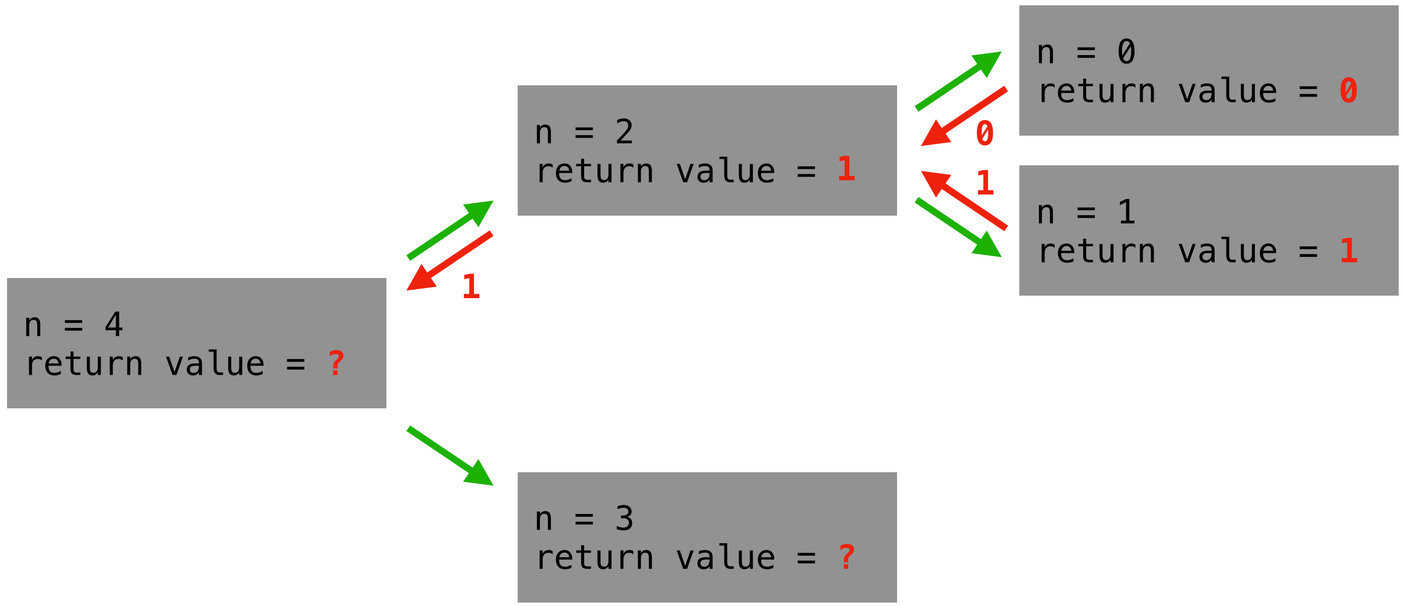
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

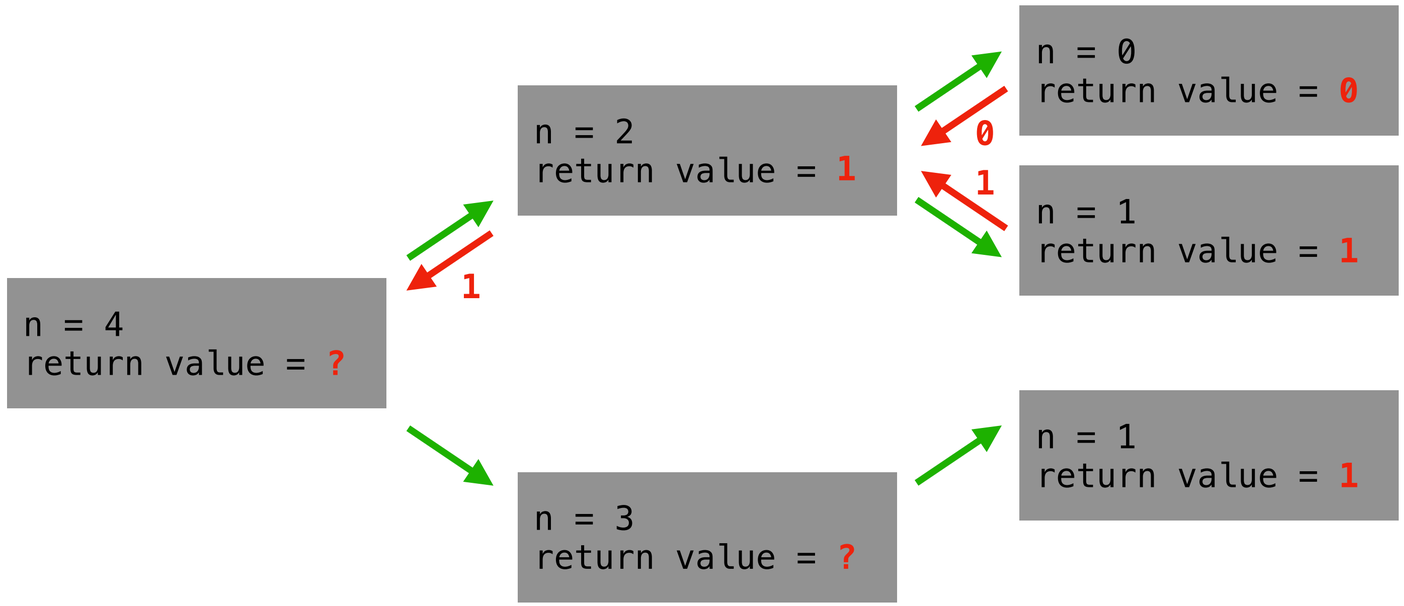
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

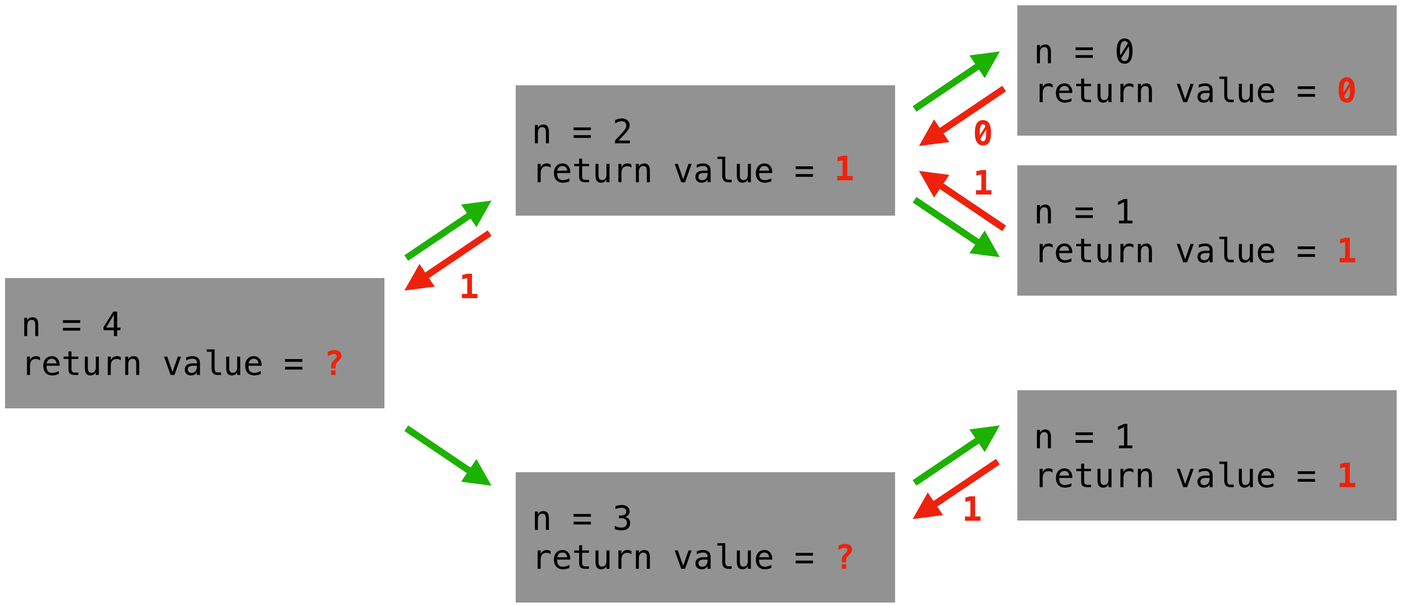
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

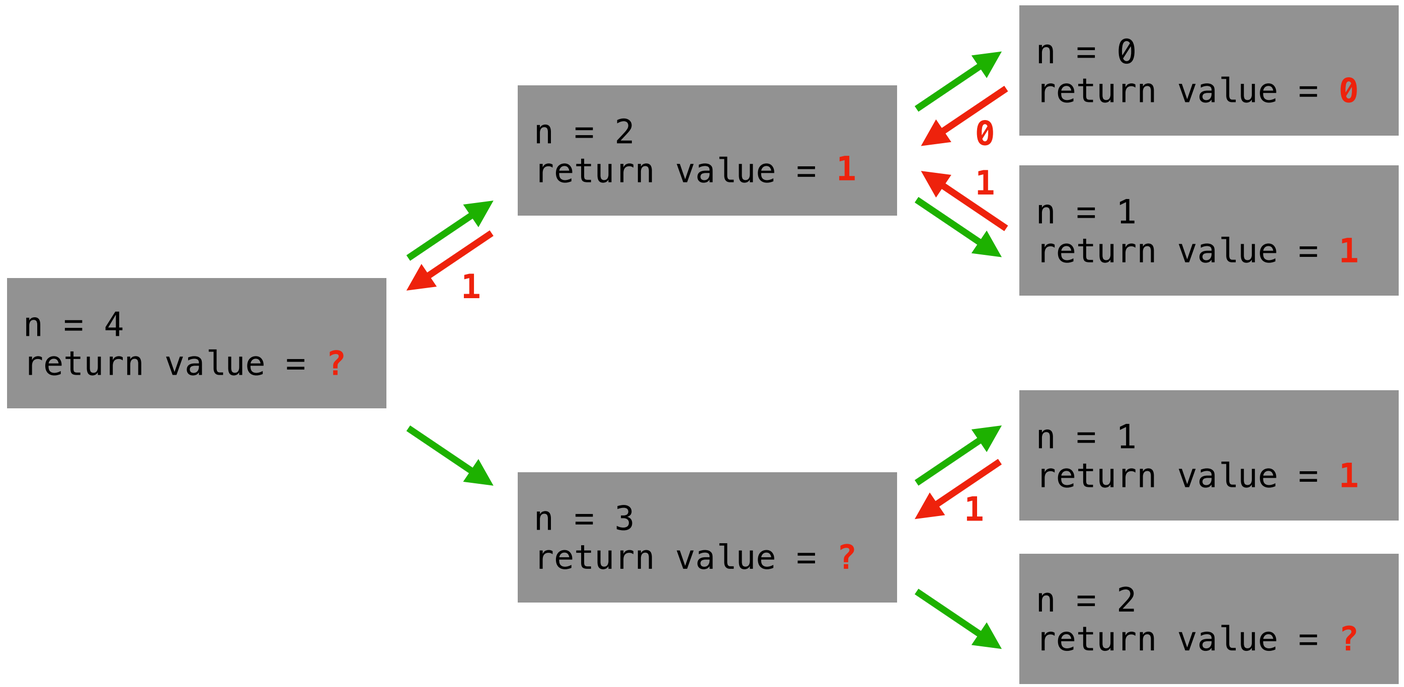
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

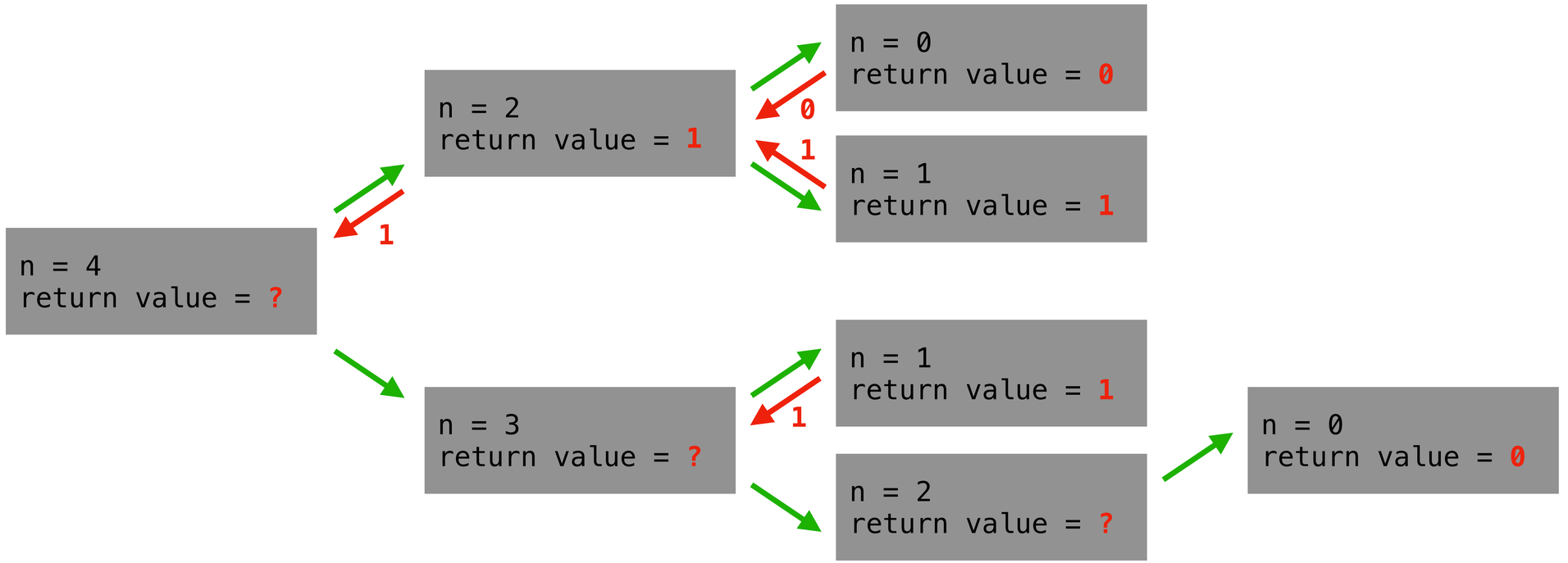
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

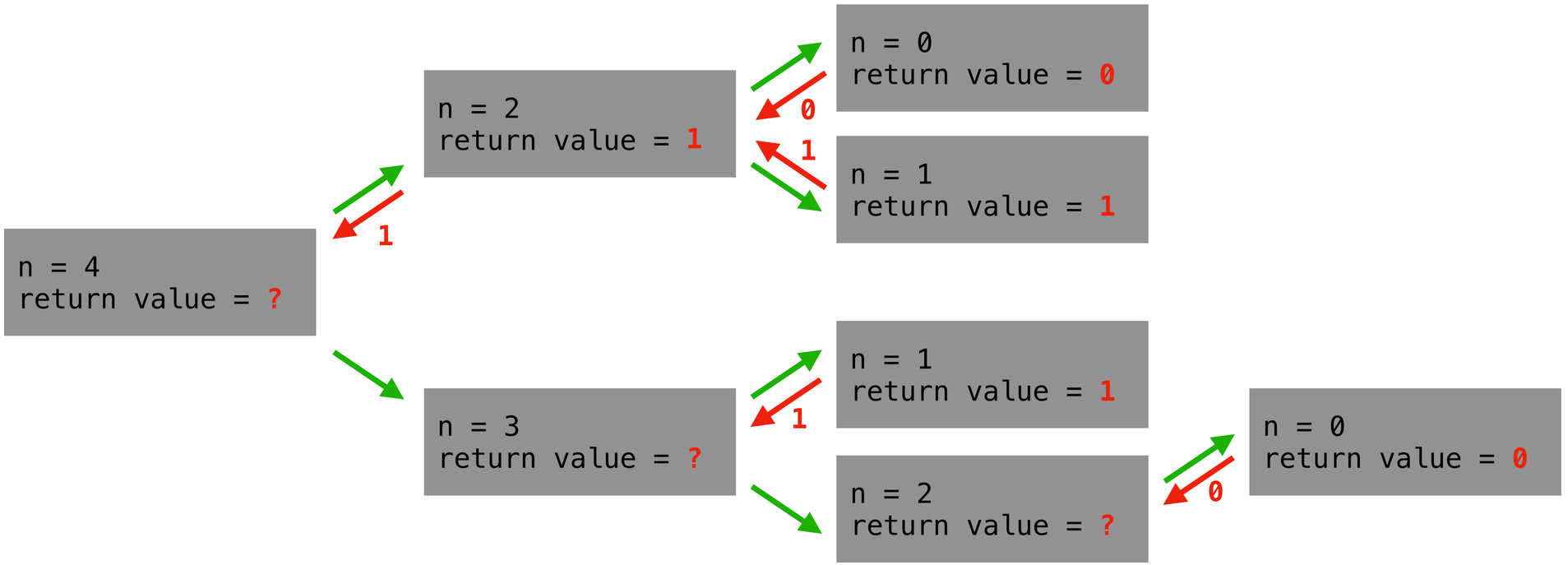
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

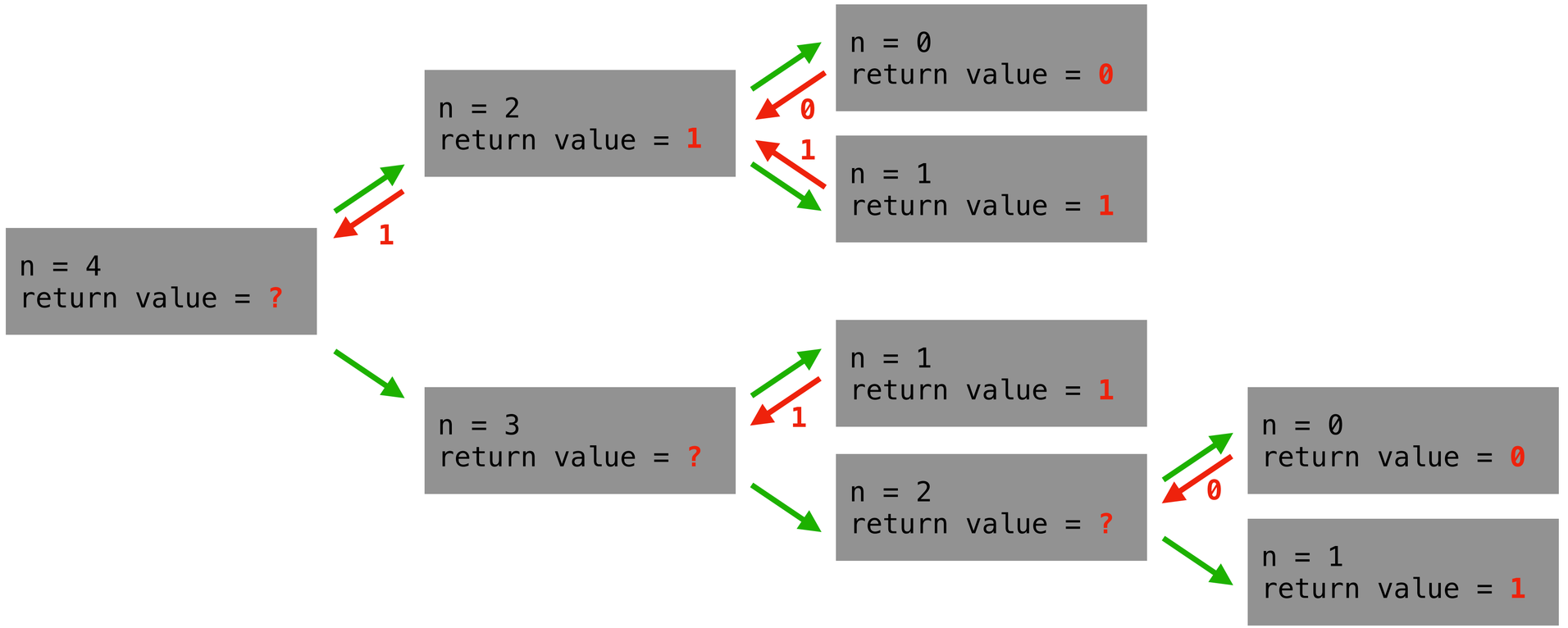
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

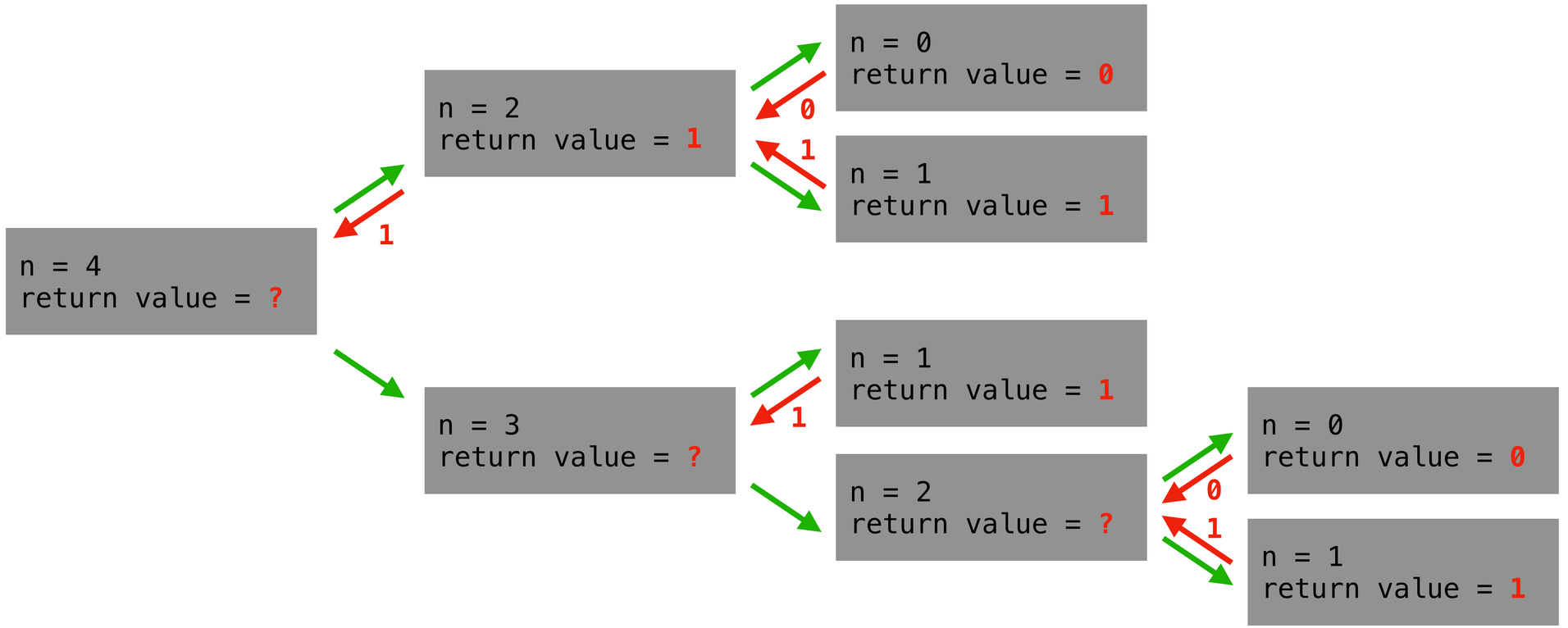
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

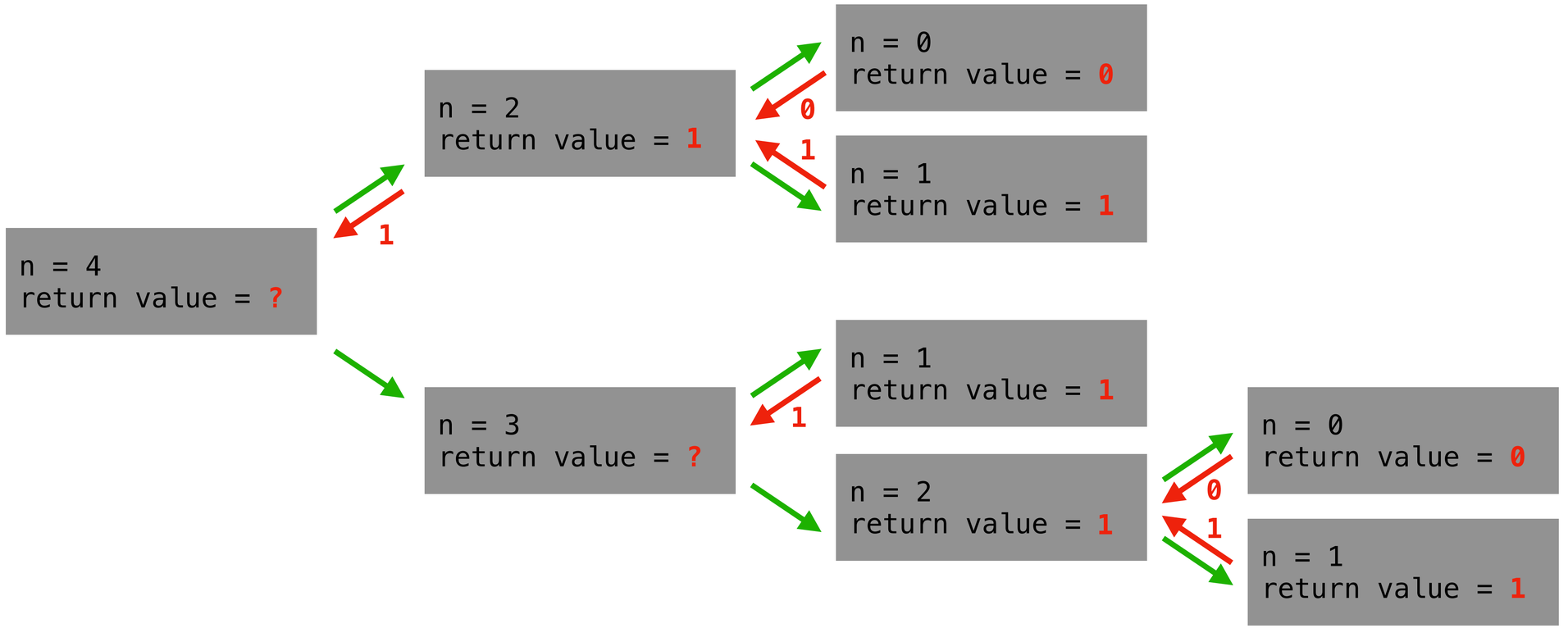
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

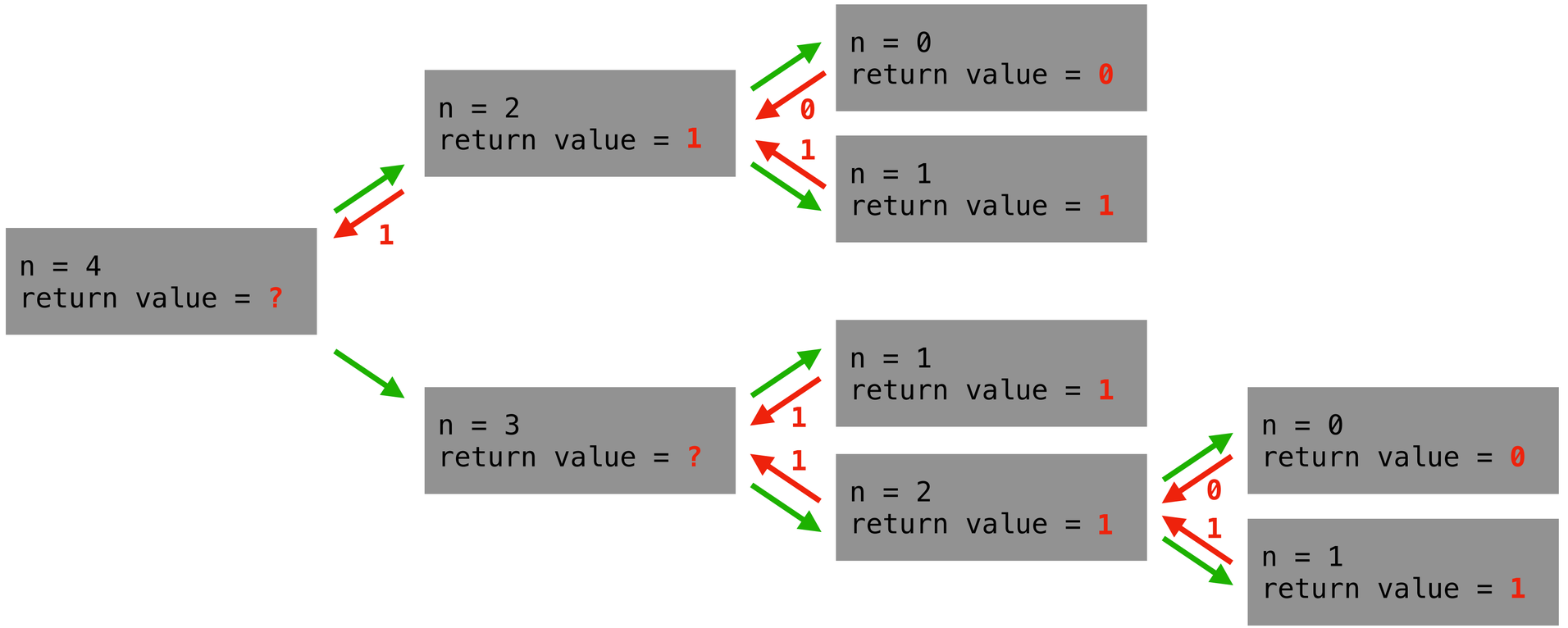
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

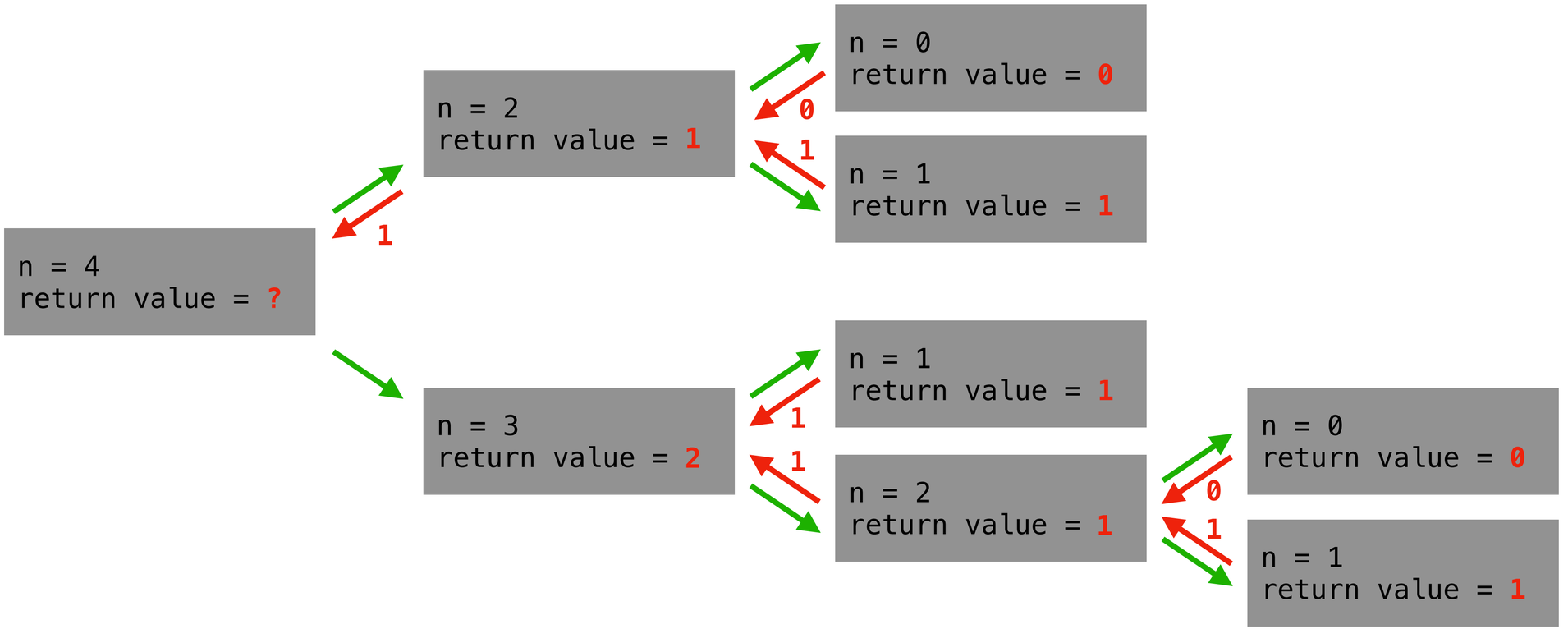
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

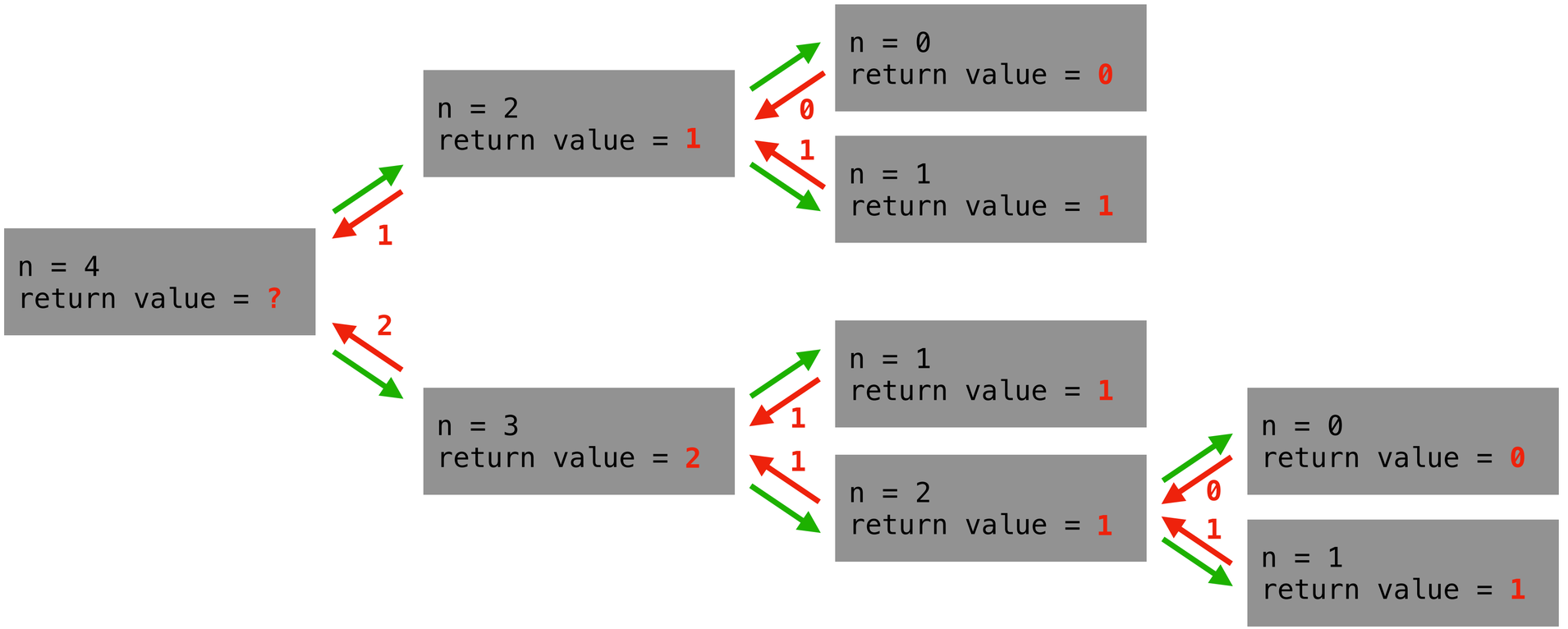
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

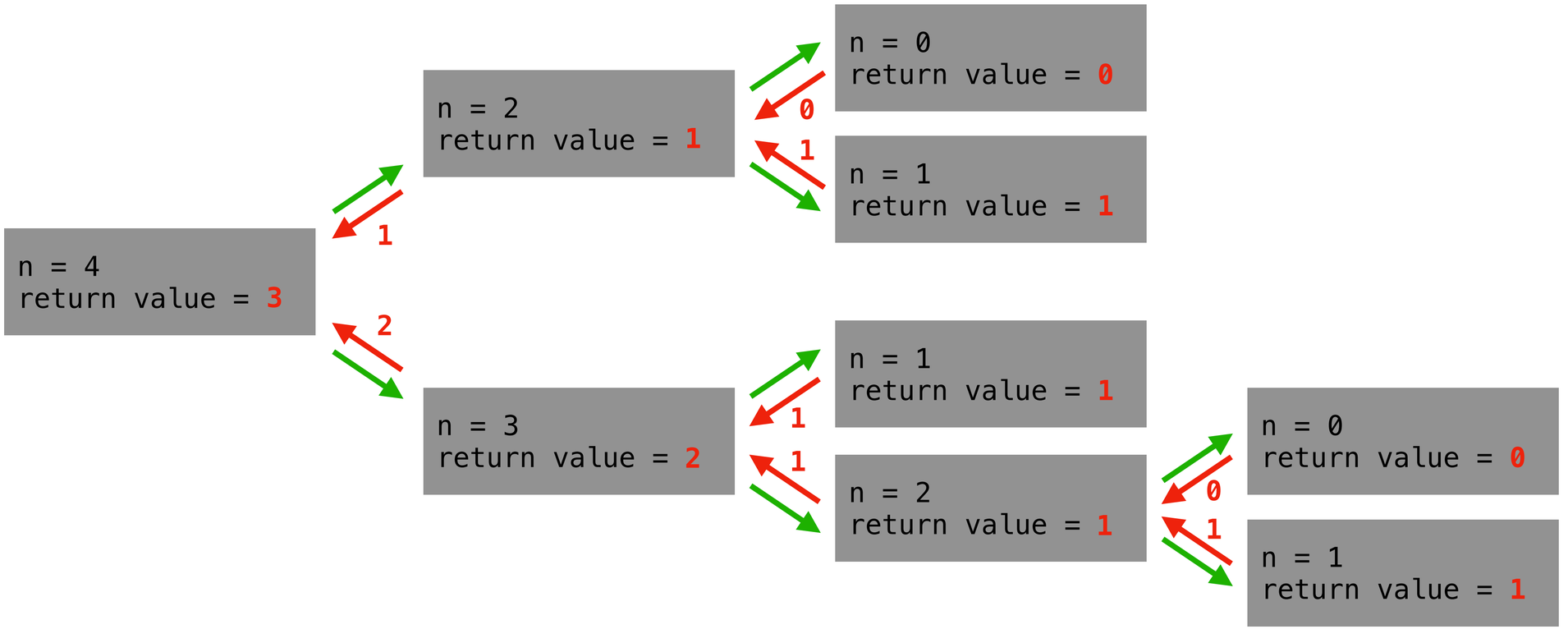
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

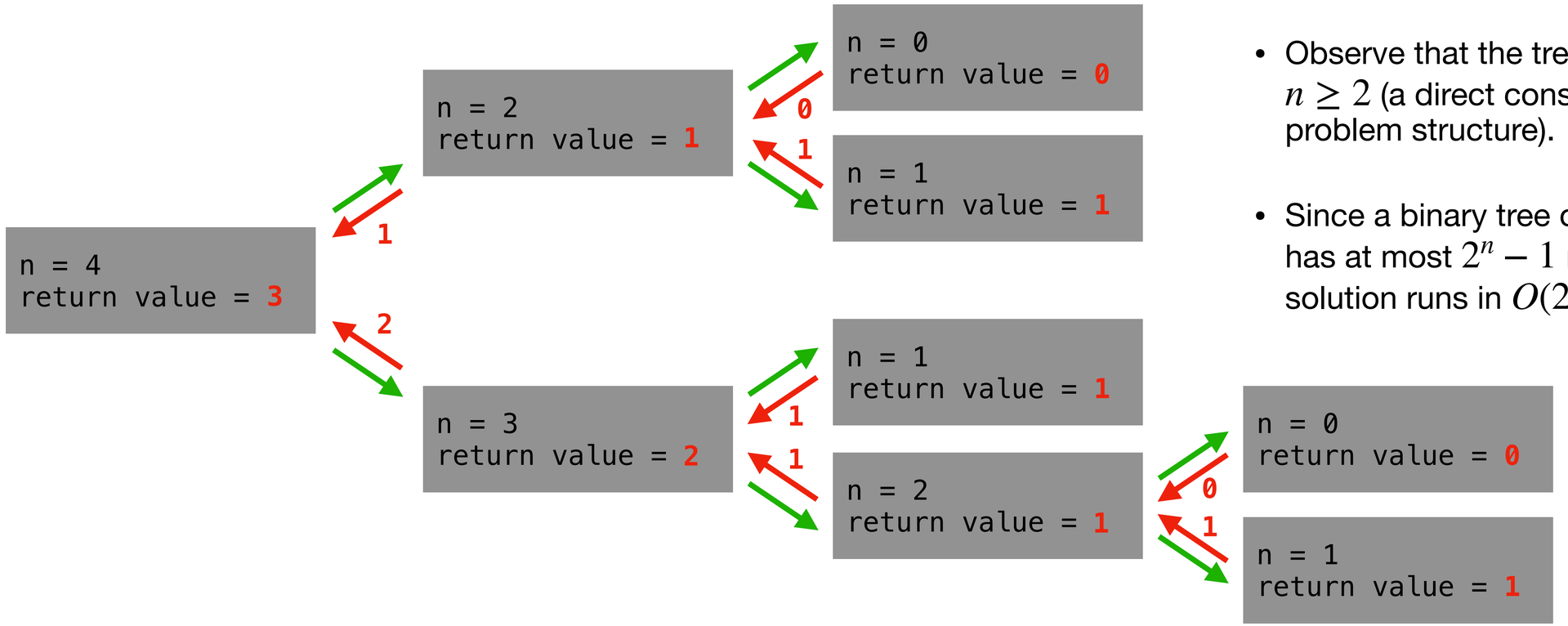
```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```



Recursion tree

Example for `fibonacci_recursive(4)`.

```c
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```

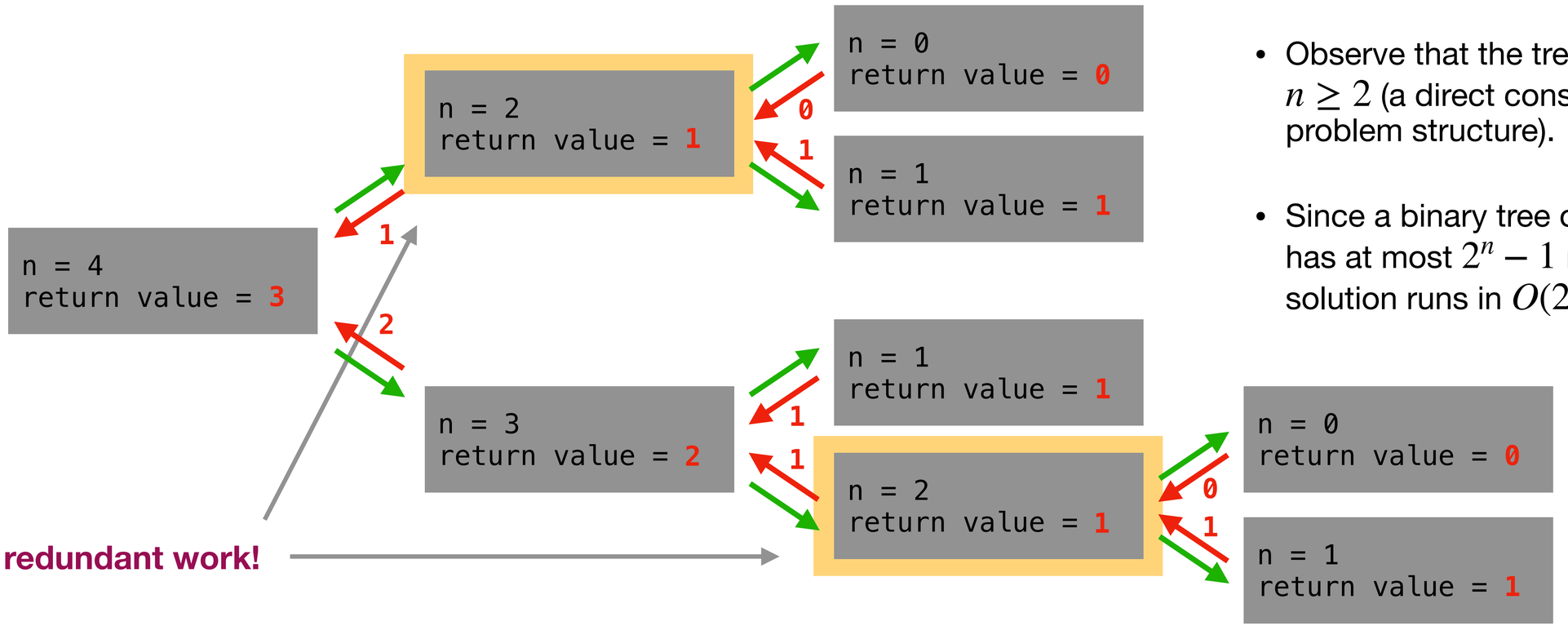


- **Depth** of a tree: num. of nodes from root to deepest leaf.
- Observe that the tree has depth n for $n \geq 2$ (a direct consequence of the problem structure).
- Since a binary tree of depth $n - 1$ has at most $2^n - 1$ nodes, the solution runs in $O(2^n)$.

Recursion tree

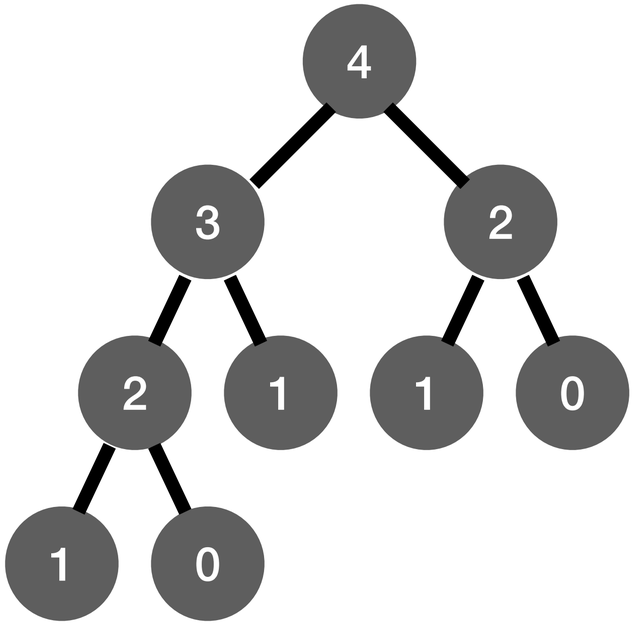
Example for `fibonacci_recursive(4)`.

```
uint64_t fibonacci_recursive(const uint64_t n) {
    if (n < 2) return n;
    return fibonacci_recursive(n - 2) + fibonacci_recursive(n - 1);
}
```

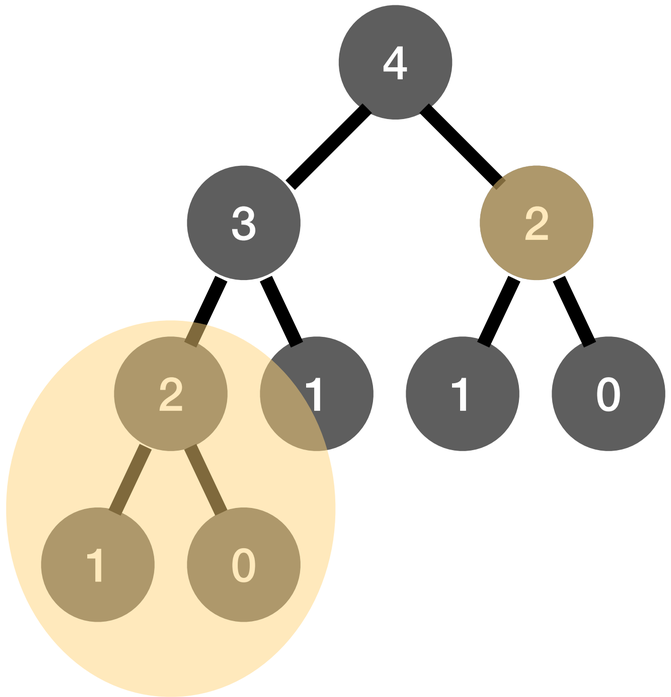


- **Depth** of a tree: num. of nodes from root to deepest leaf.
- Observe that the tree has depth n for $n \geq 2$ (a direct consequence of the problem structure).
- Since a binary tree of depth $n - 1$ has at most $2^n - 1$ nodes, the solution runs in $O(2^n)$.

Recursion tree — More examples

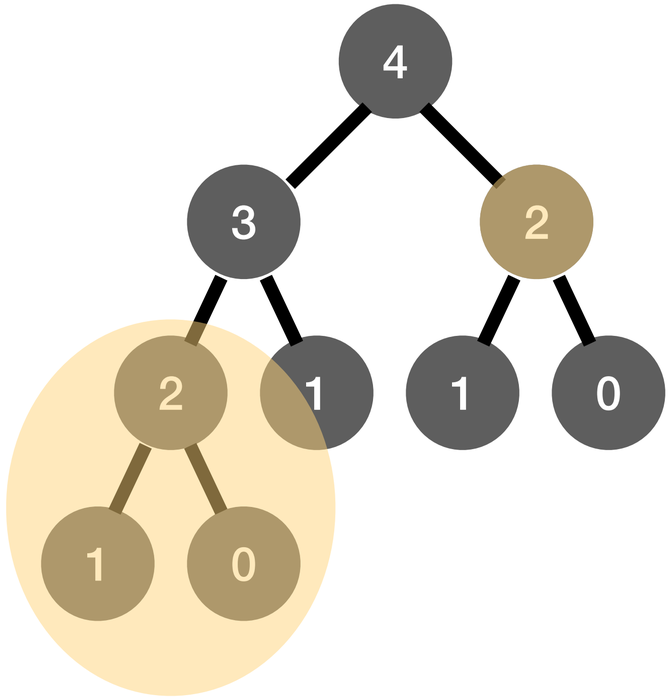


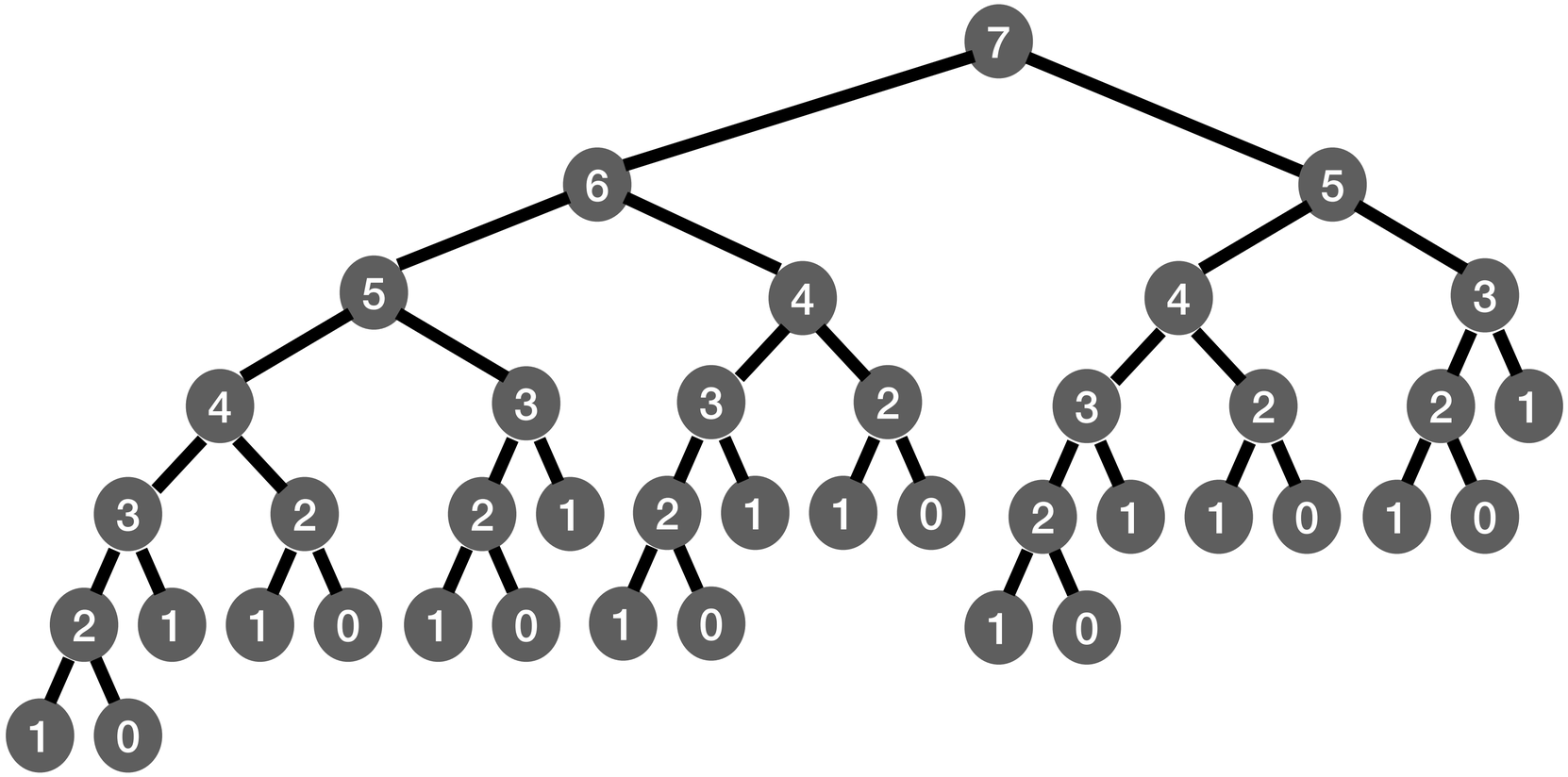
Recursion tree — More examples



redundant work!

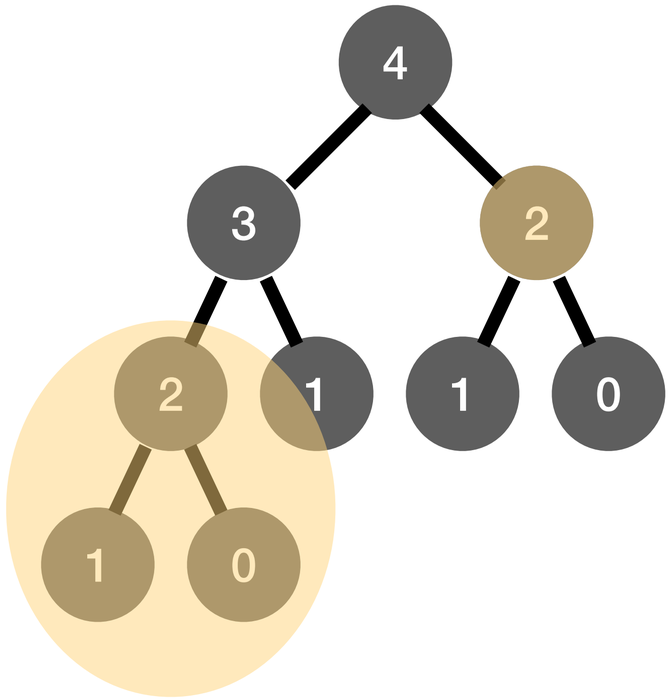
Recursion tree — More examples

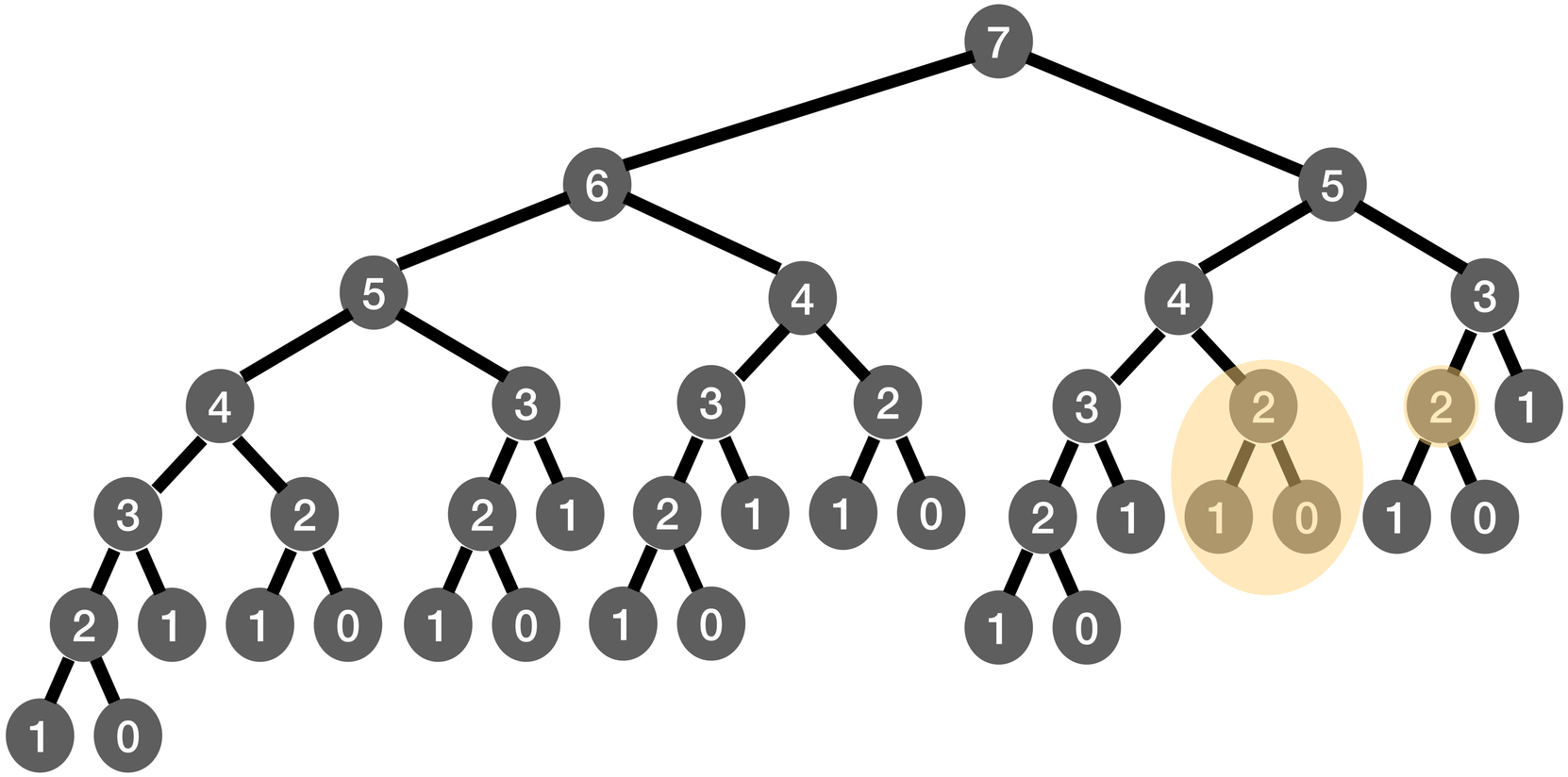




redundant work!

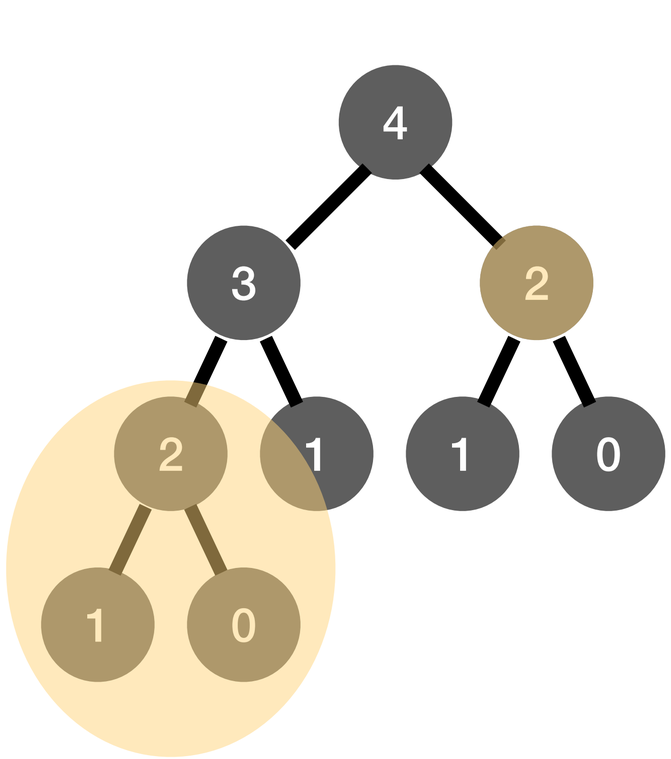
Recursion tree — More examples

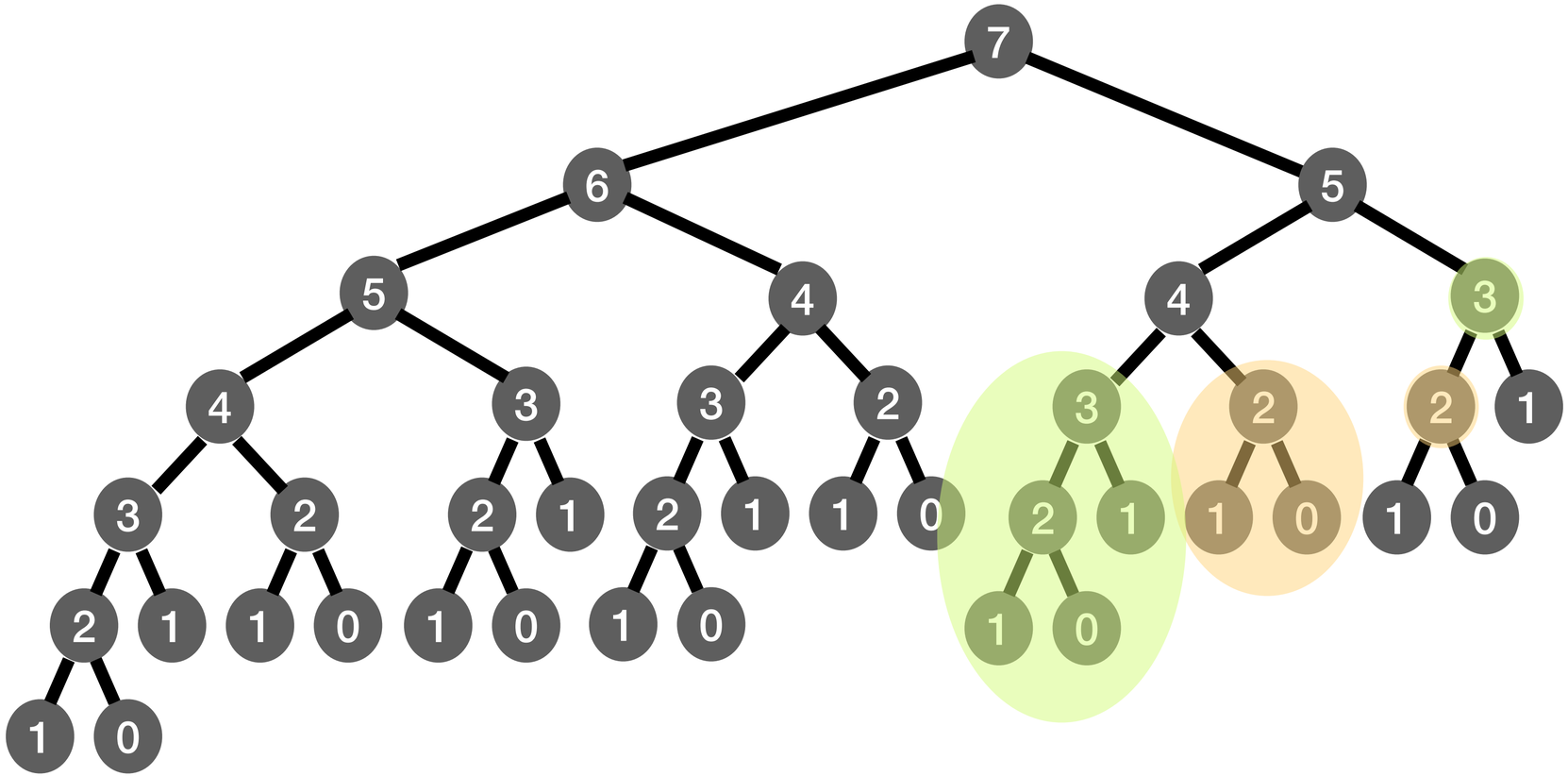




redundant work!

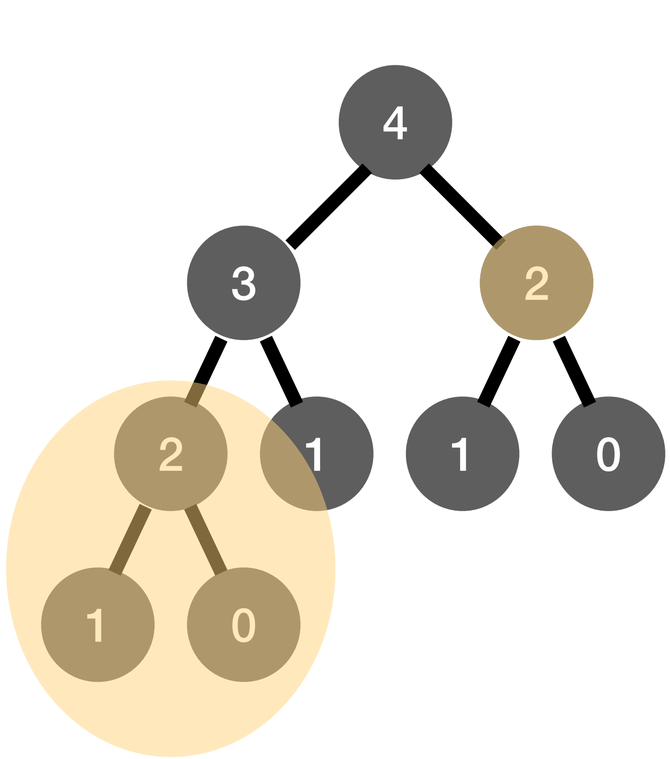
Recursion tree — More examples

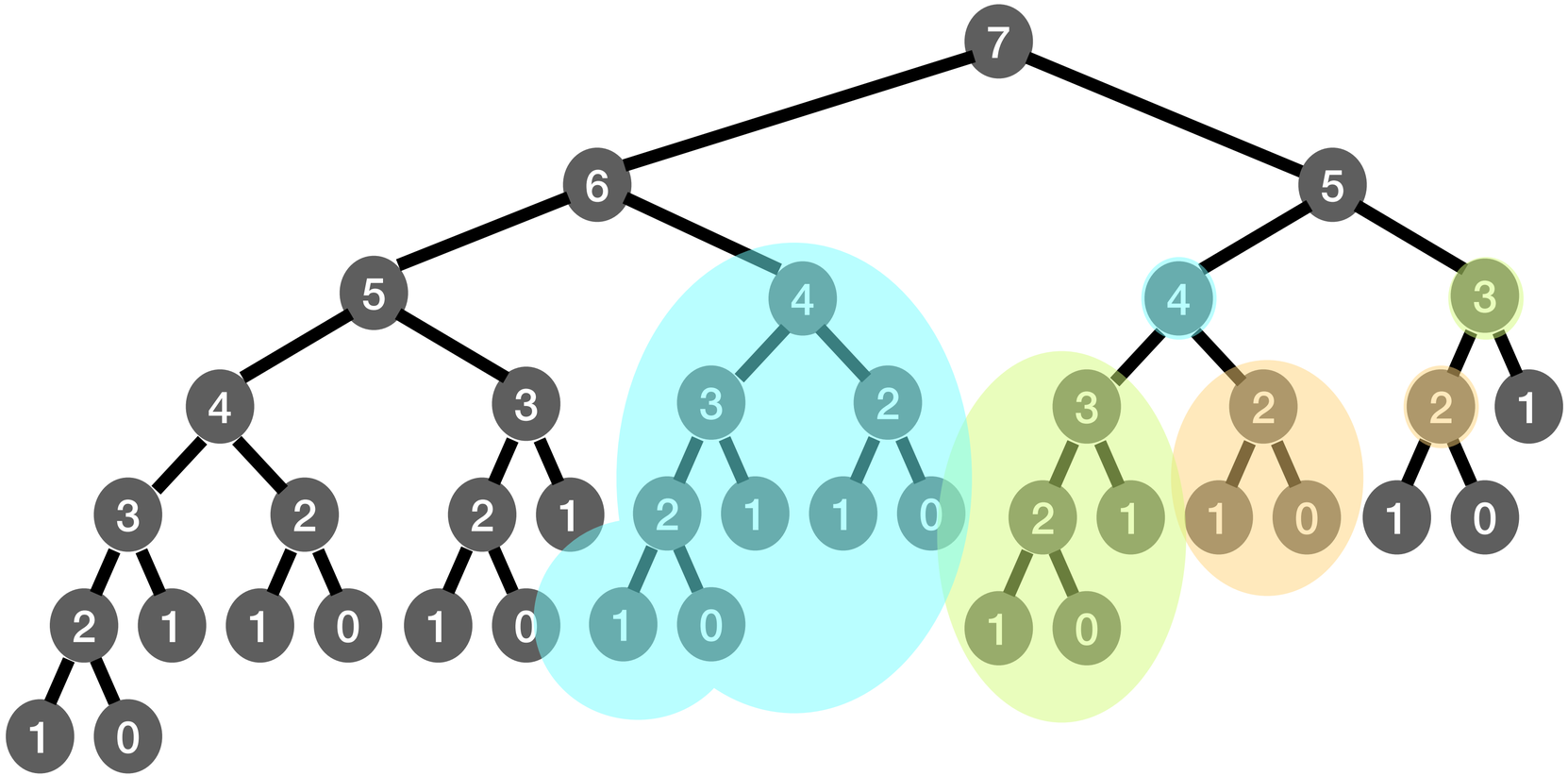




redundant work!

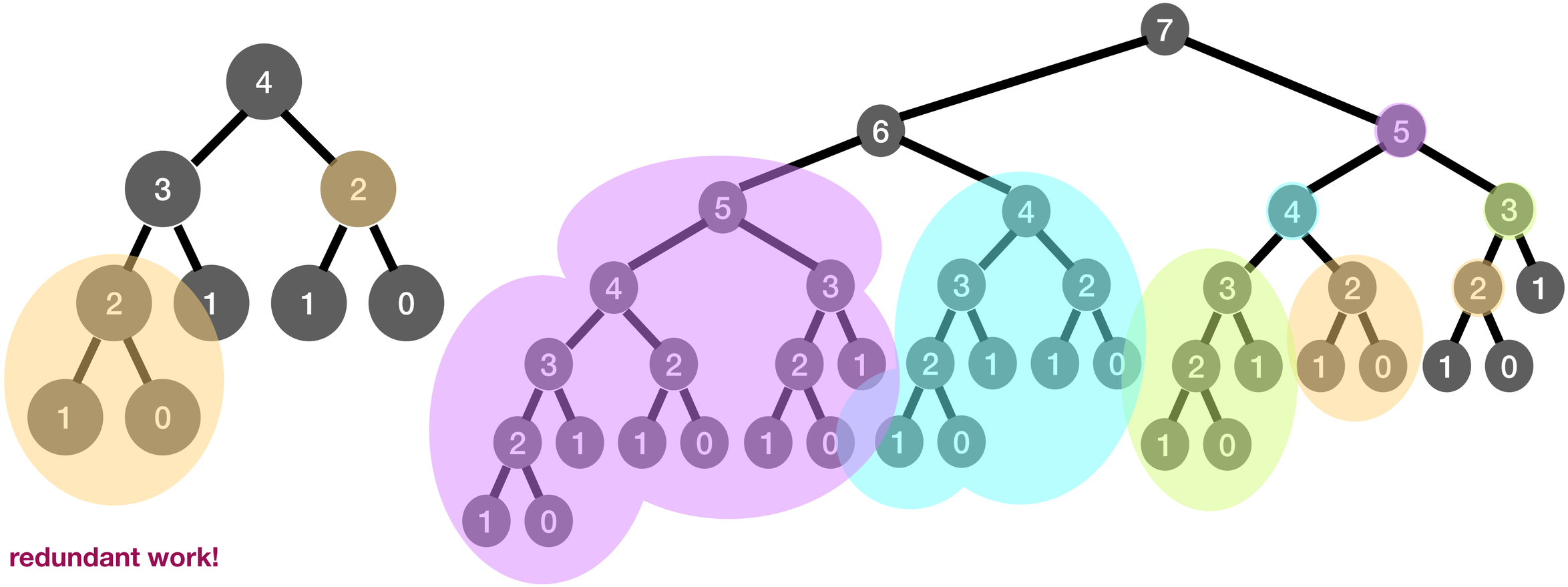
Recursion tree — More examples





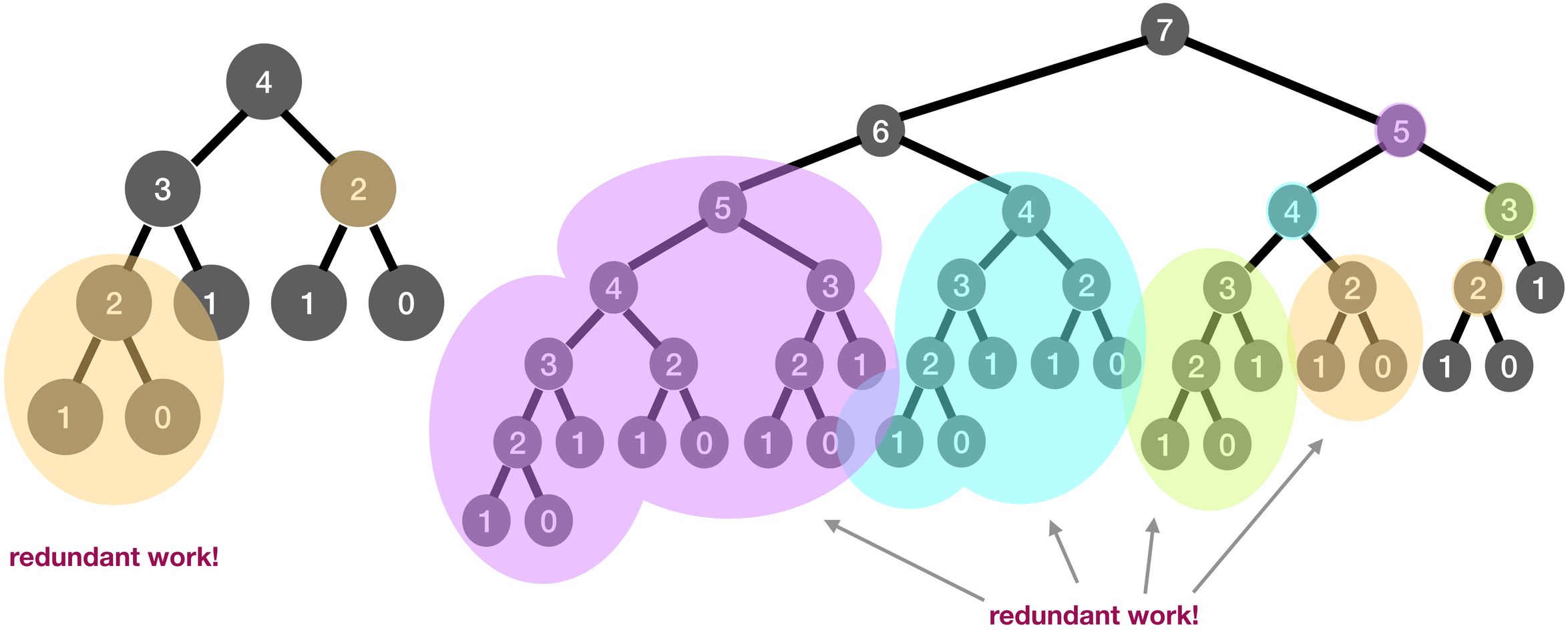
redundant work!

Recursion tree — More examples



redundant work!

Recursion tree — More examples



Memoization

- **Idea.** “Remember” (*memo*) the solution to sub-problems, i.e. store the solutions to sub-problems in an array as these are encountered.

- DP is a nice example of a **space/time trade-off**: we save time by using some space — the array holding the results to sub-problems.

```
const uint64_t INF = -1;
const uint64_t N = 100;   <-- upper bound for n
uint64_t F[N];

uint64_t fibonacci_recursive_memoized(const uint64_t n) { ...
}

int main() {
    F[0] = 0;
    F[1] = 1;
    for (uint64_t i = 2; i != N; ++i) F[i] = INF;
    const uint64_t n = 13;
    std::cout << "F_mem(" << n << ") = " << fibonacci_recursive_memoized(n)
              << std::endl;
}
```

Memoization

- **Idea.** "Remember" (*memo*) the solution to sub-problems, i.e. store the solutions to sub-problems as these are encountered in an array.

```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```

If the solution for n has been already computed, return it directly (cheap operation)!

Recursion tree — Example

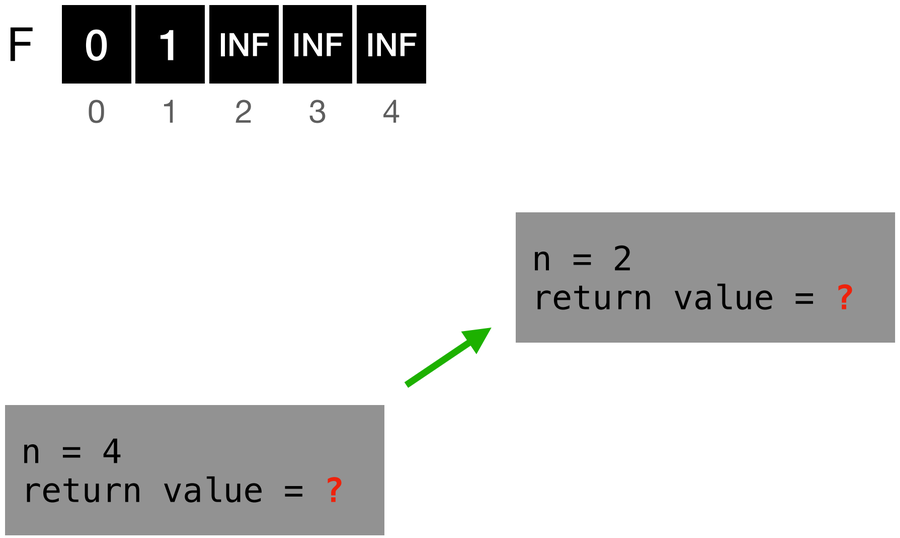
Example for `fibonacci_recursive_memoized(4)`.


```
n = 4
return value = ?
```

```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```

Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

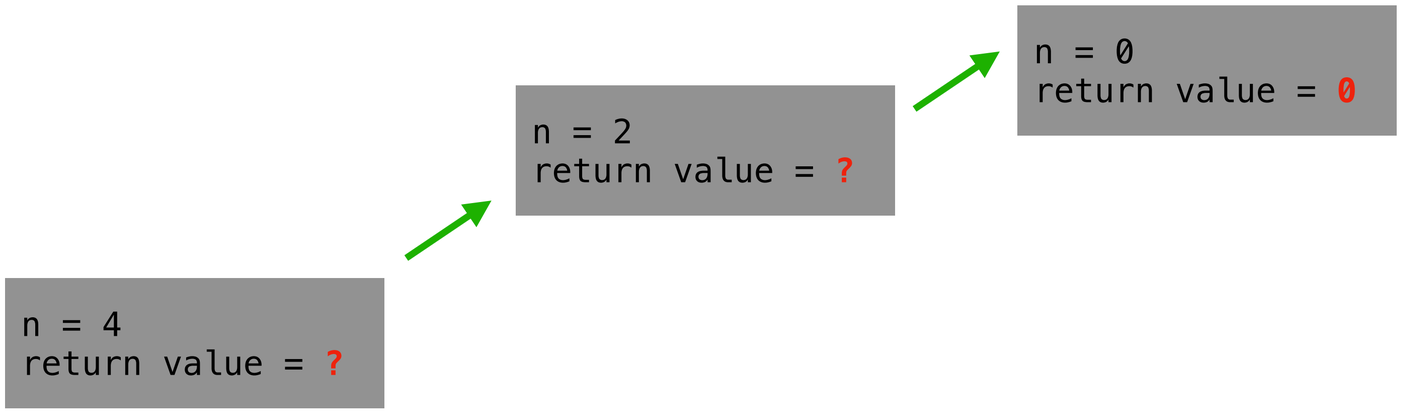



```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```

Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

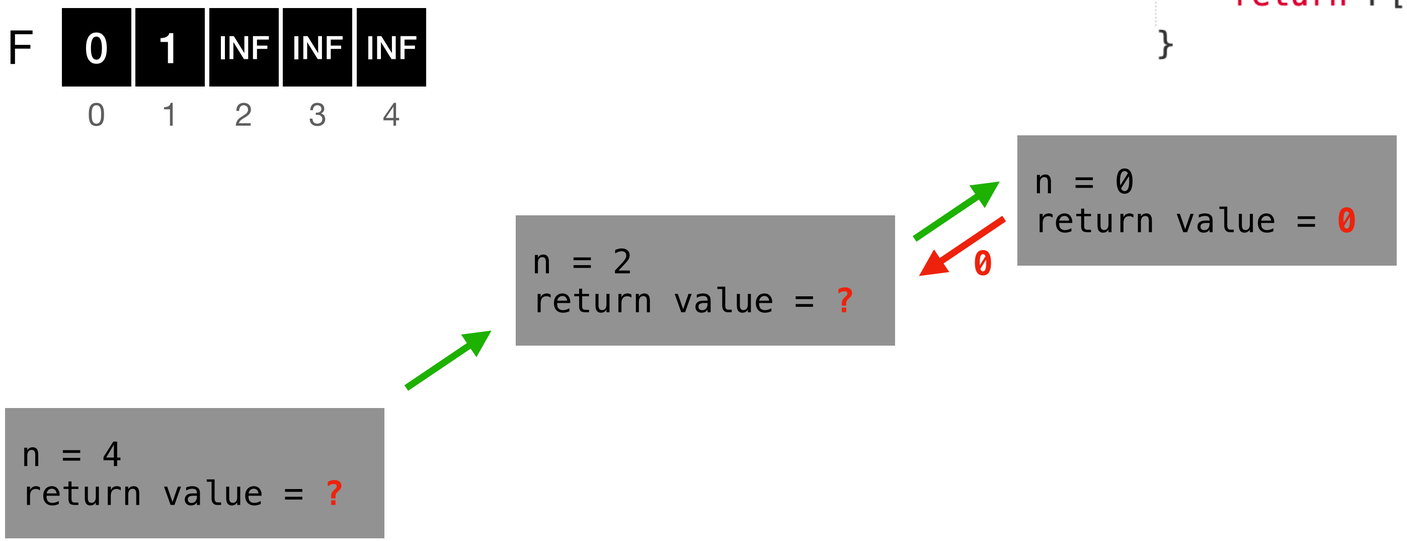

```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

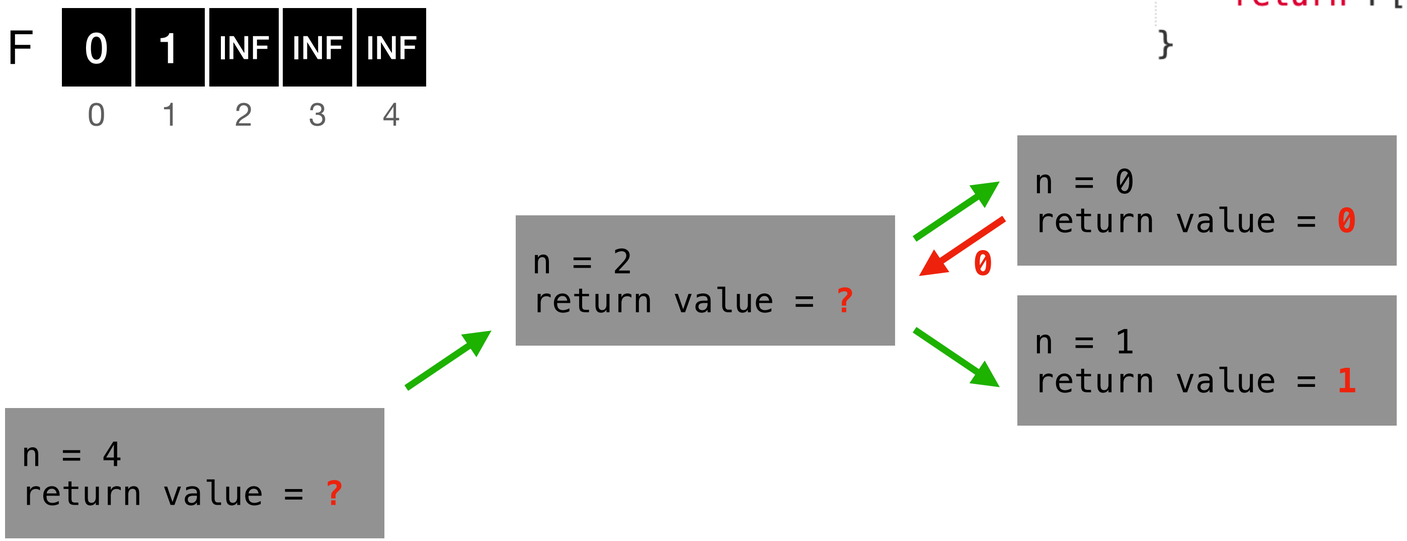
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

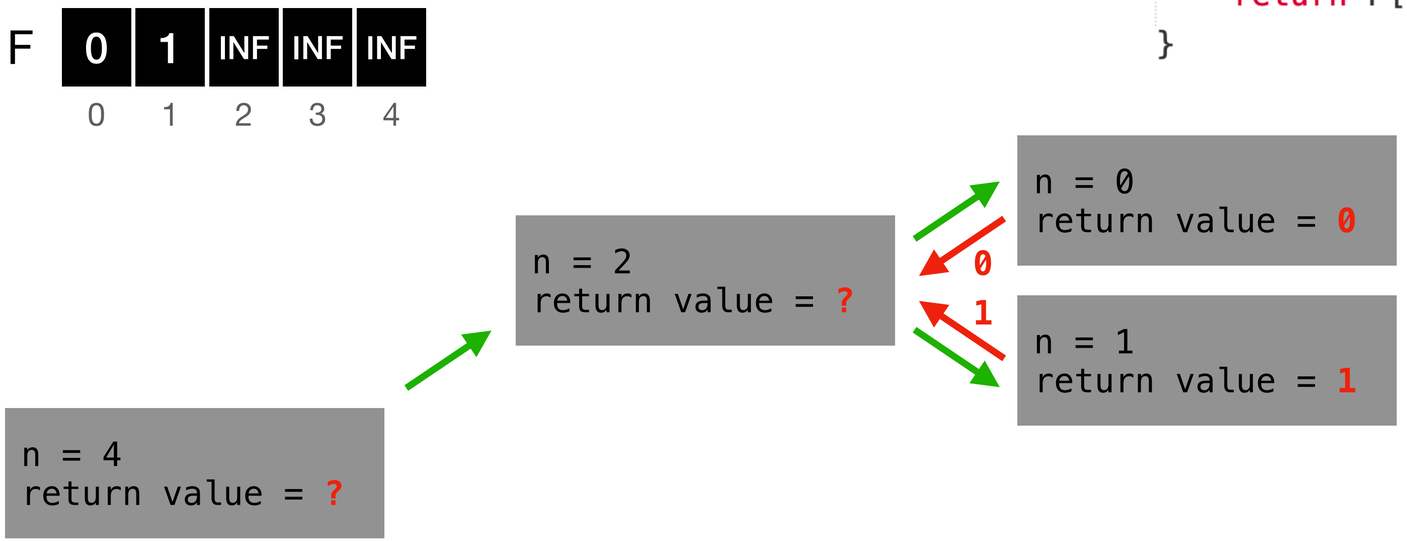
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

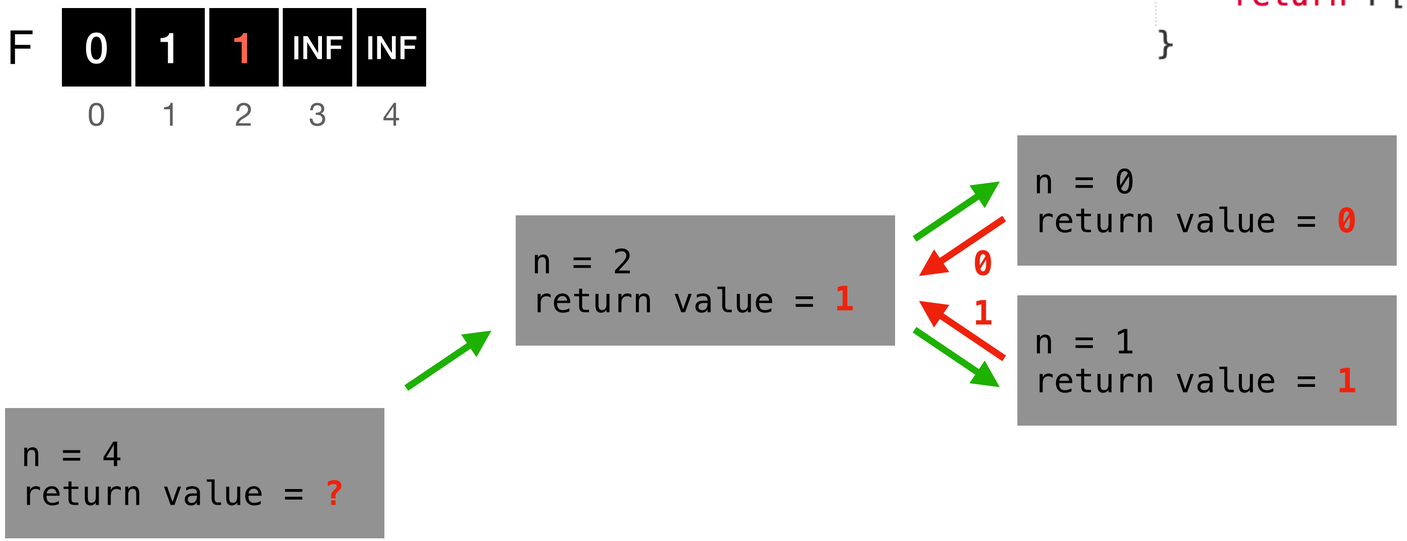
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

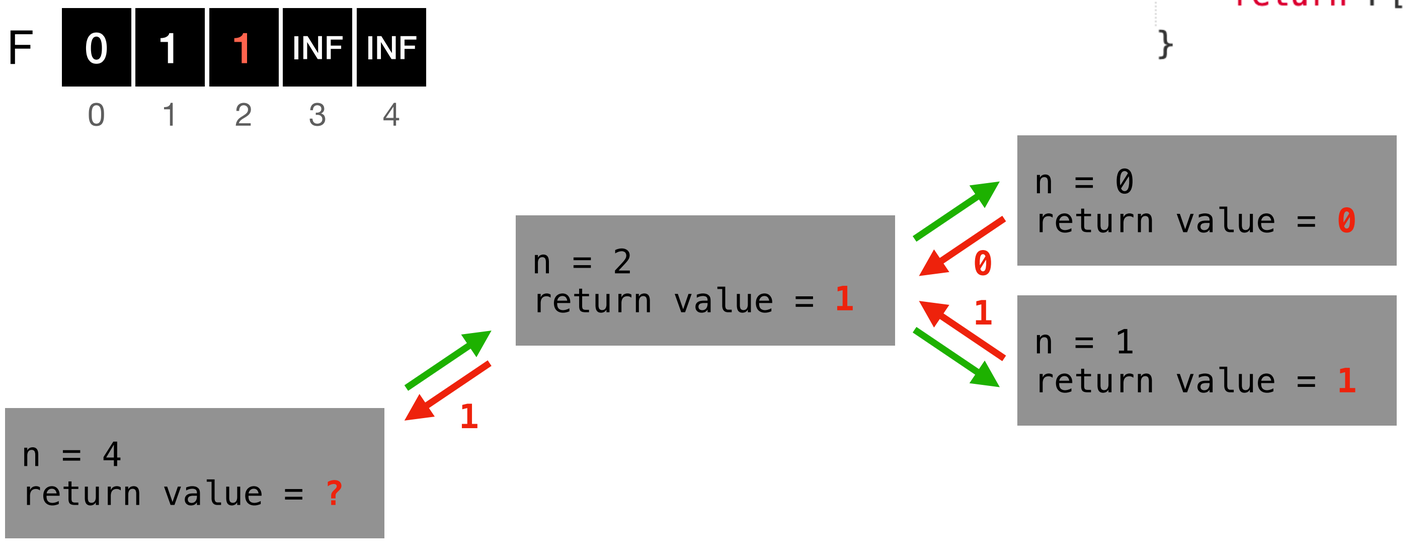
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

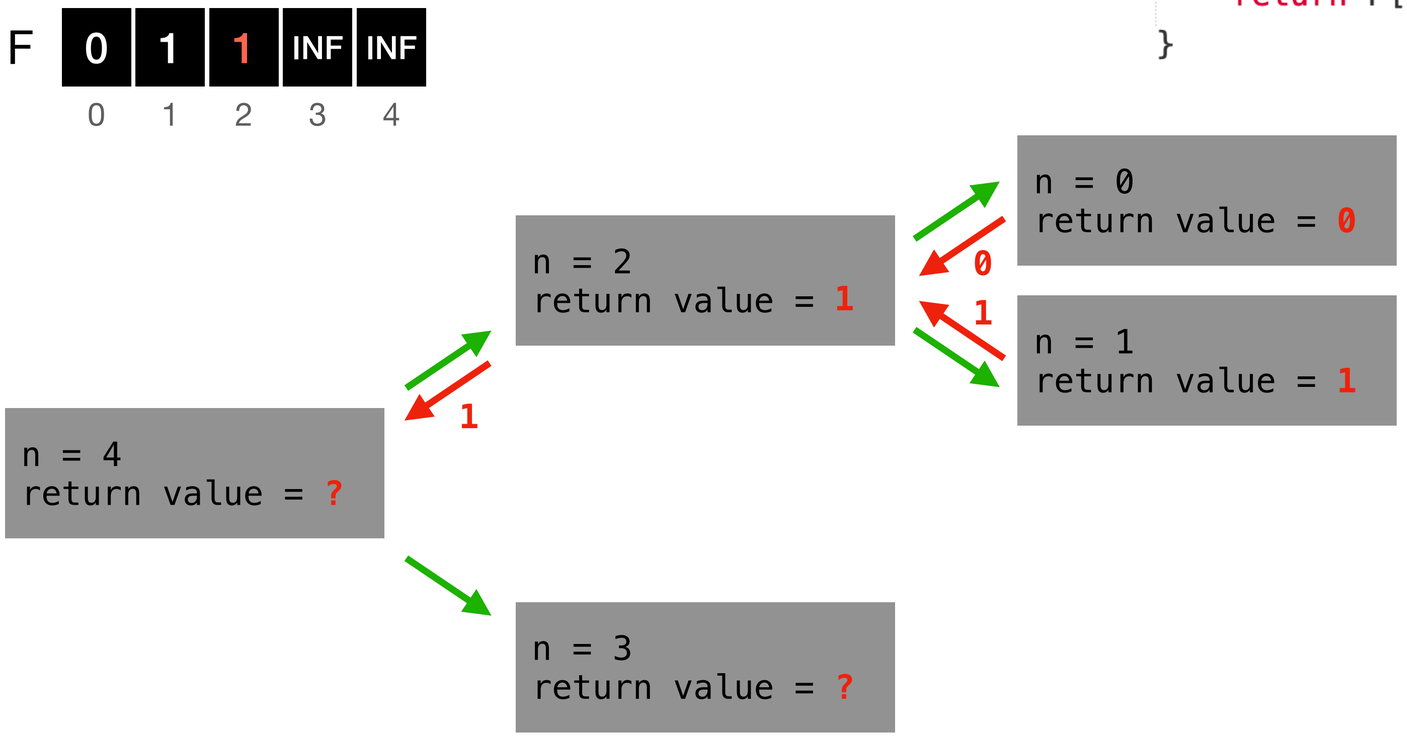
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

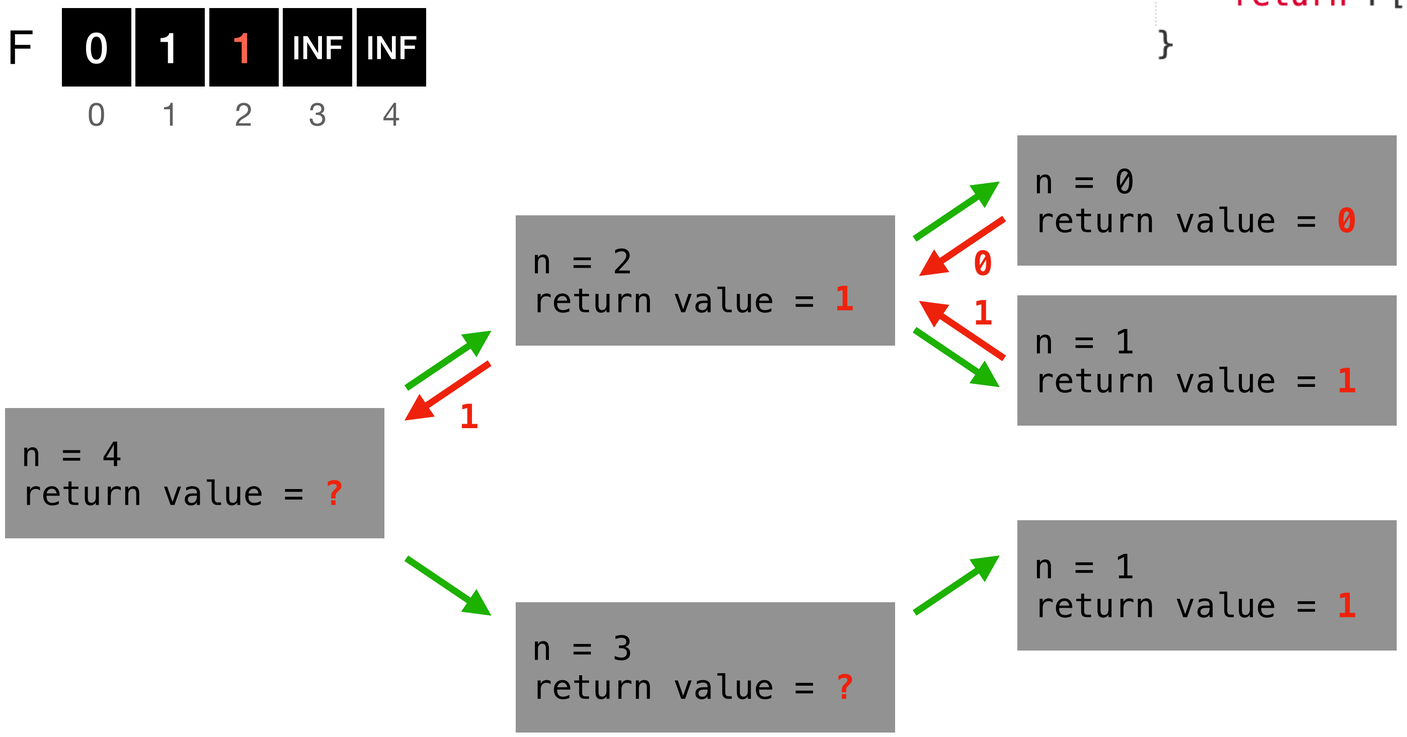
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

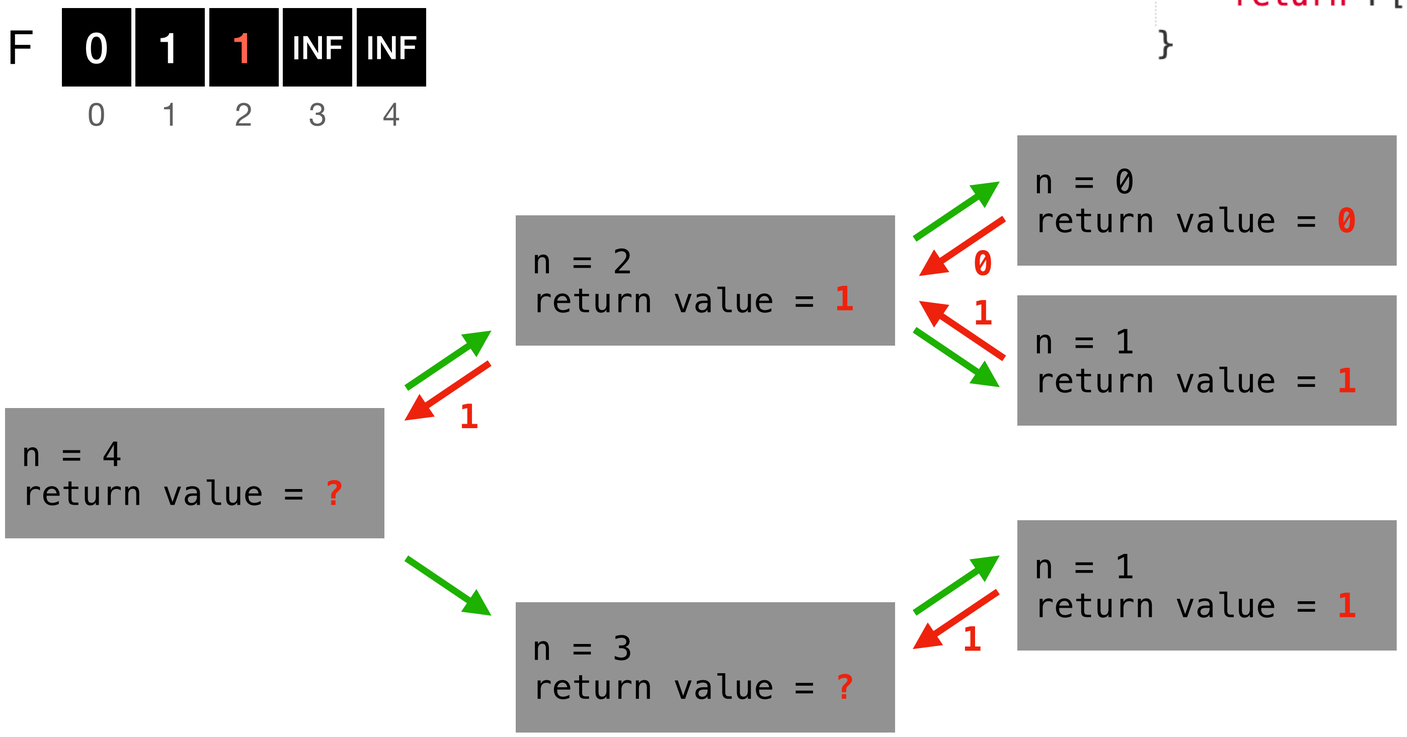
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

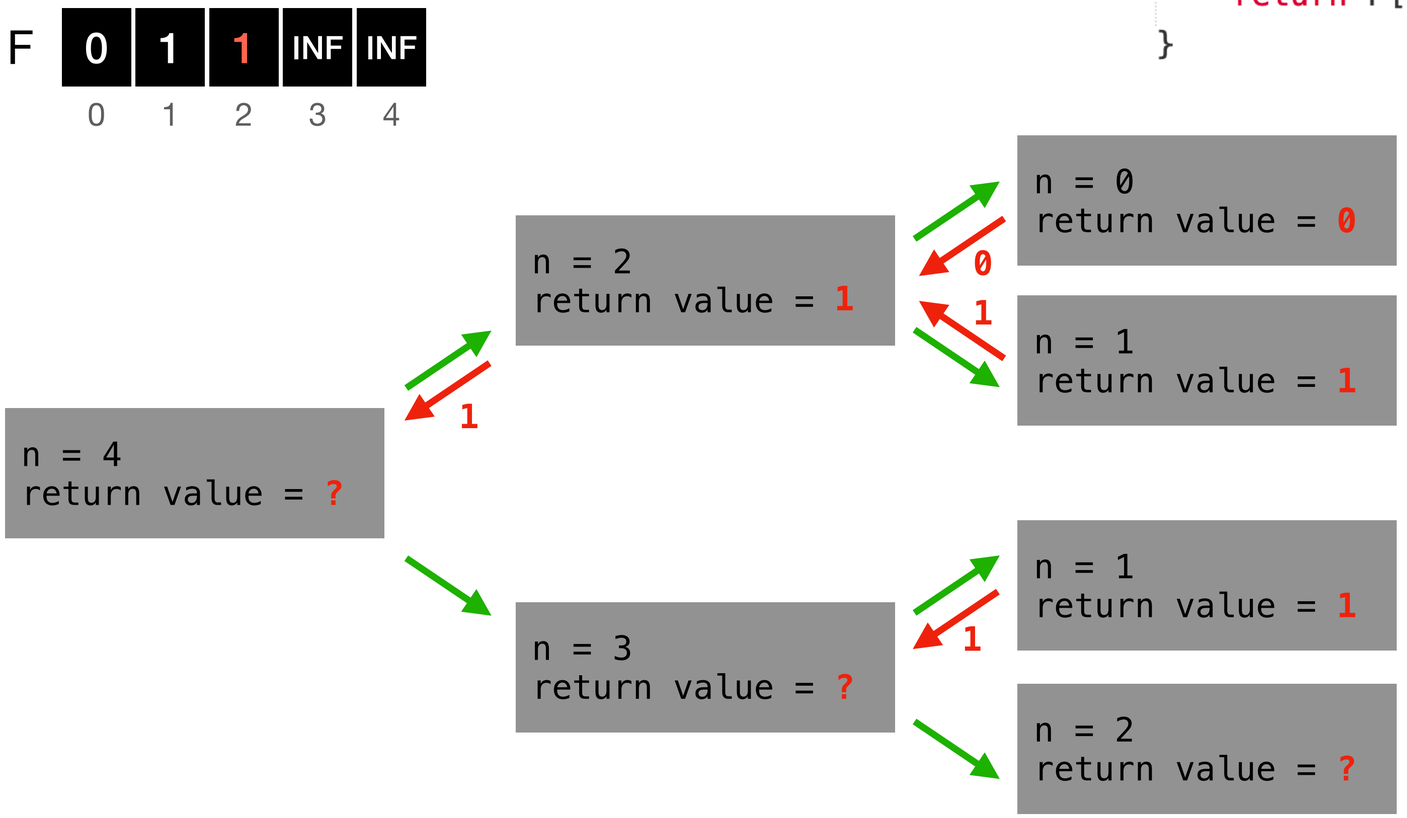
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

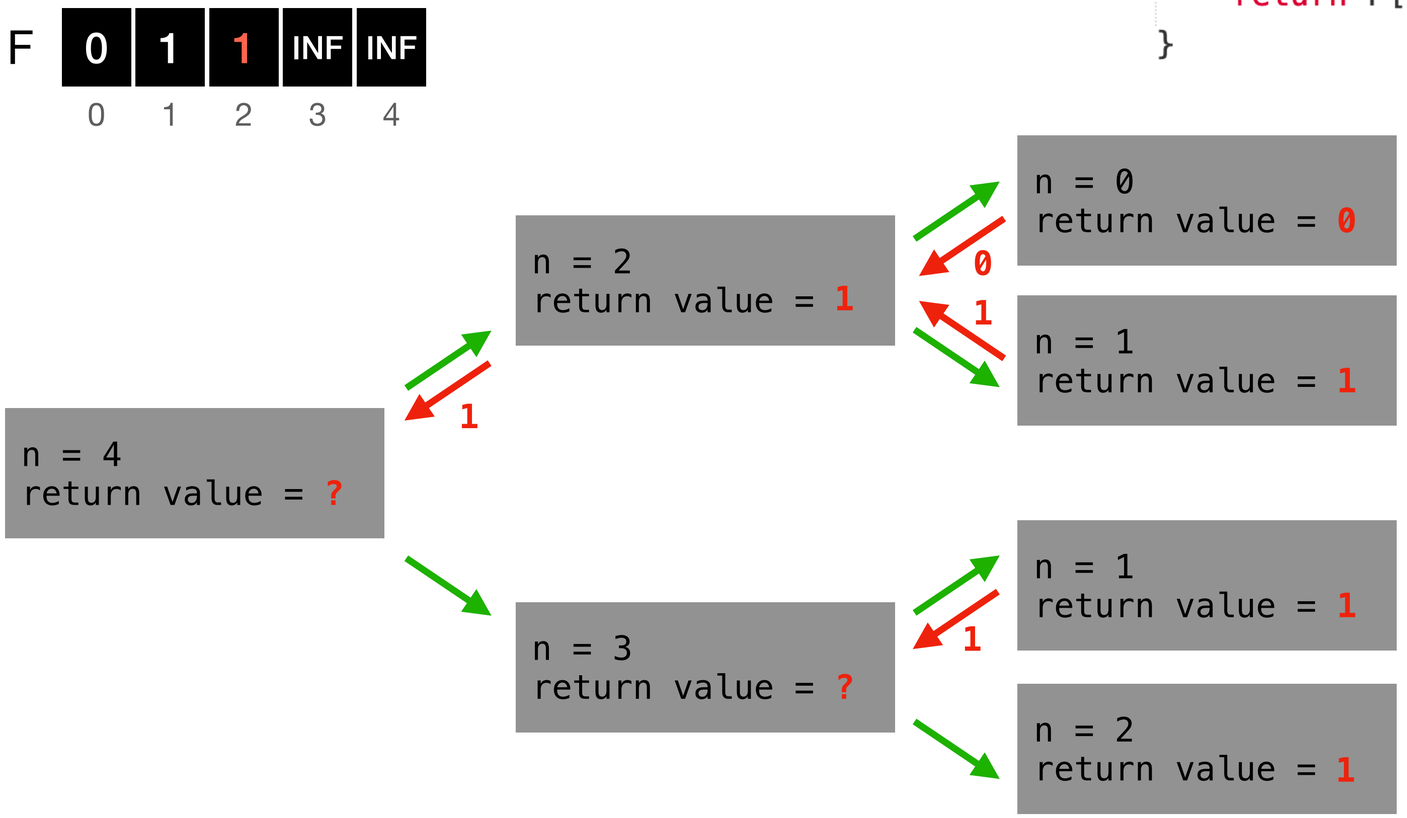
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

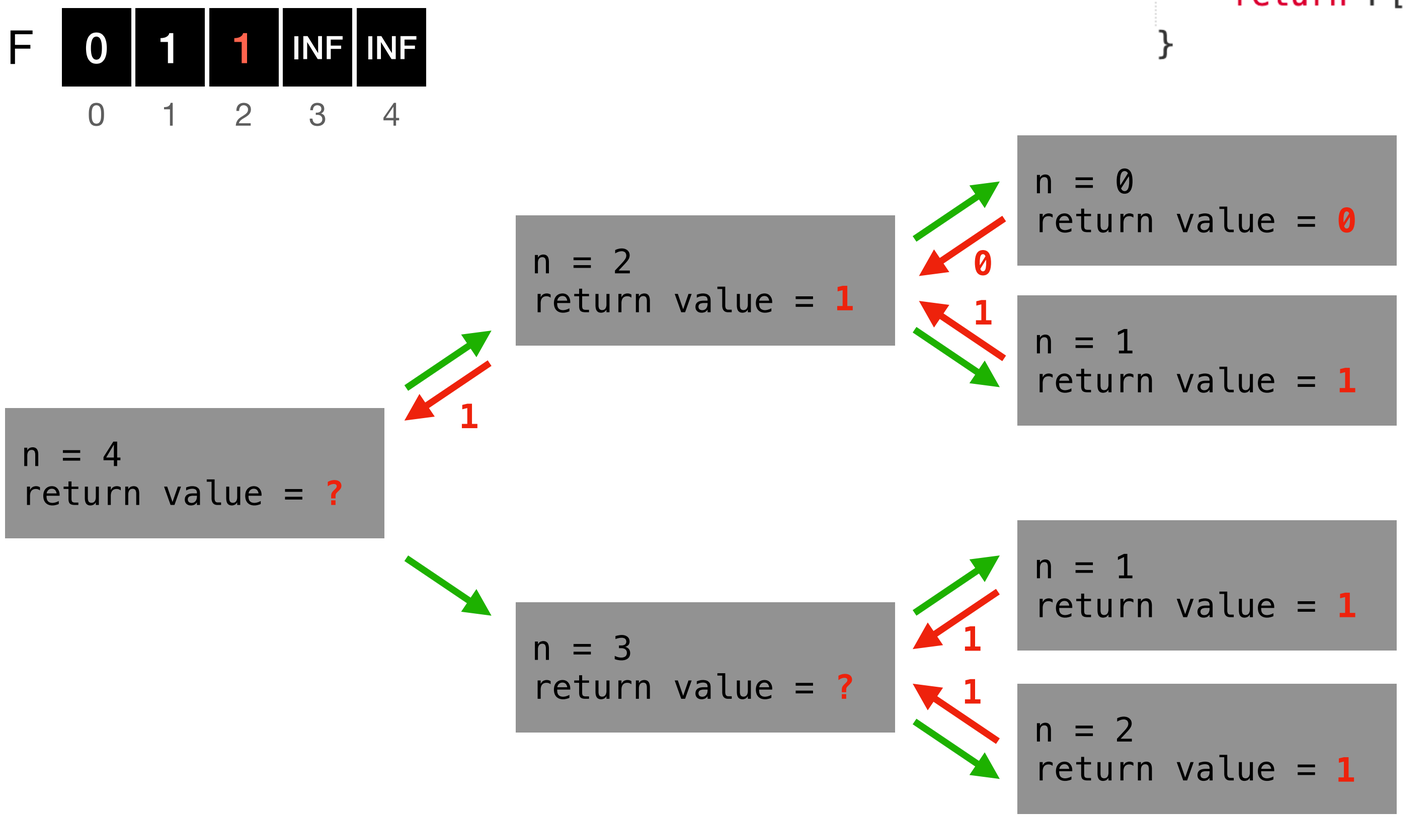
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

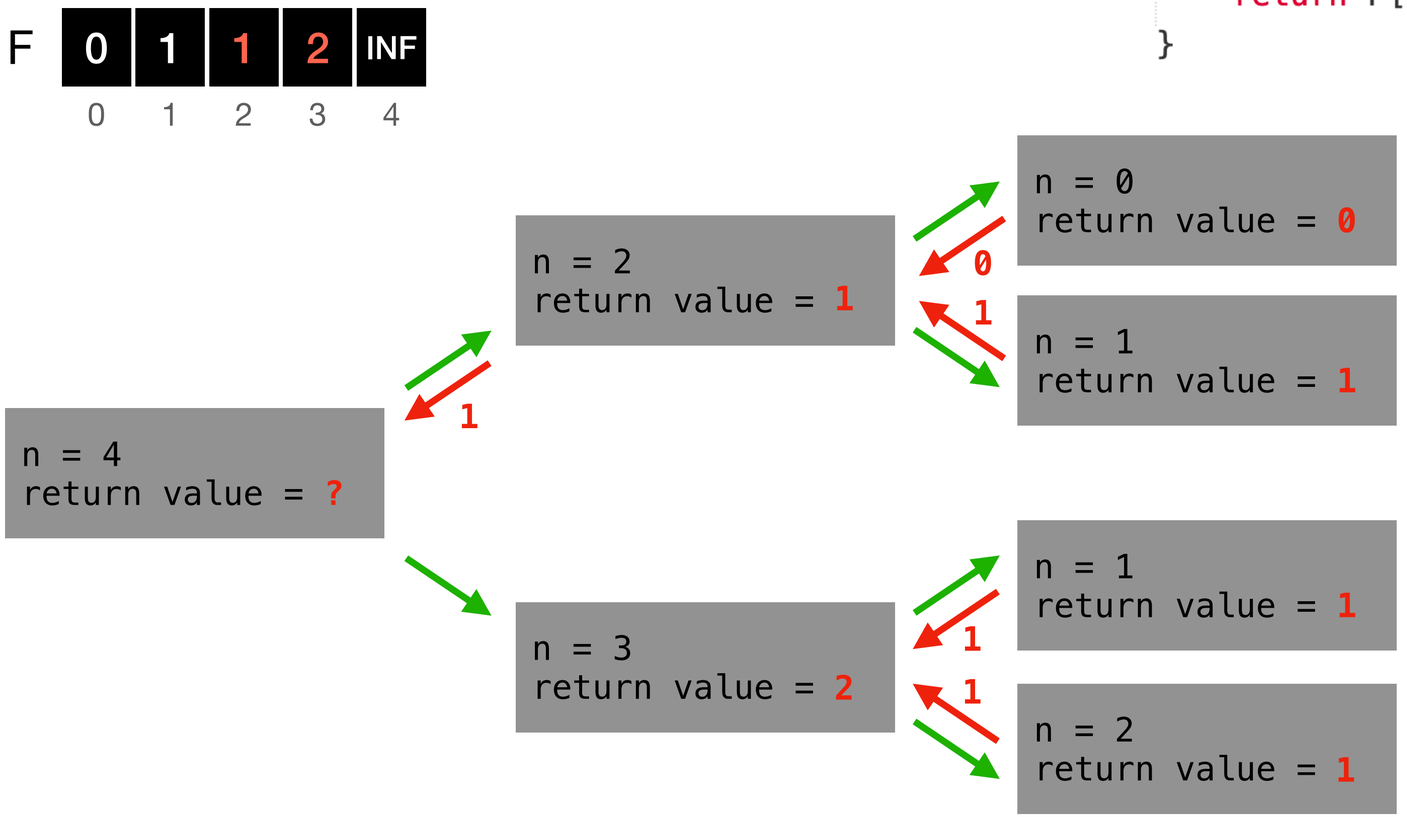
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

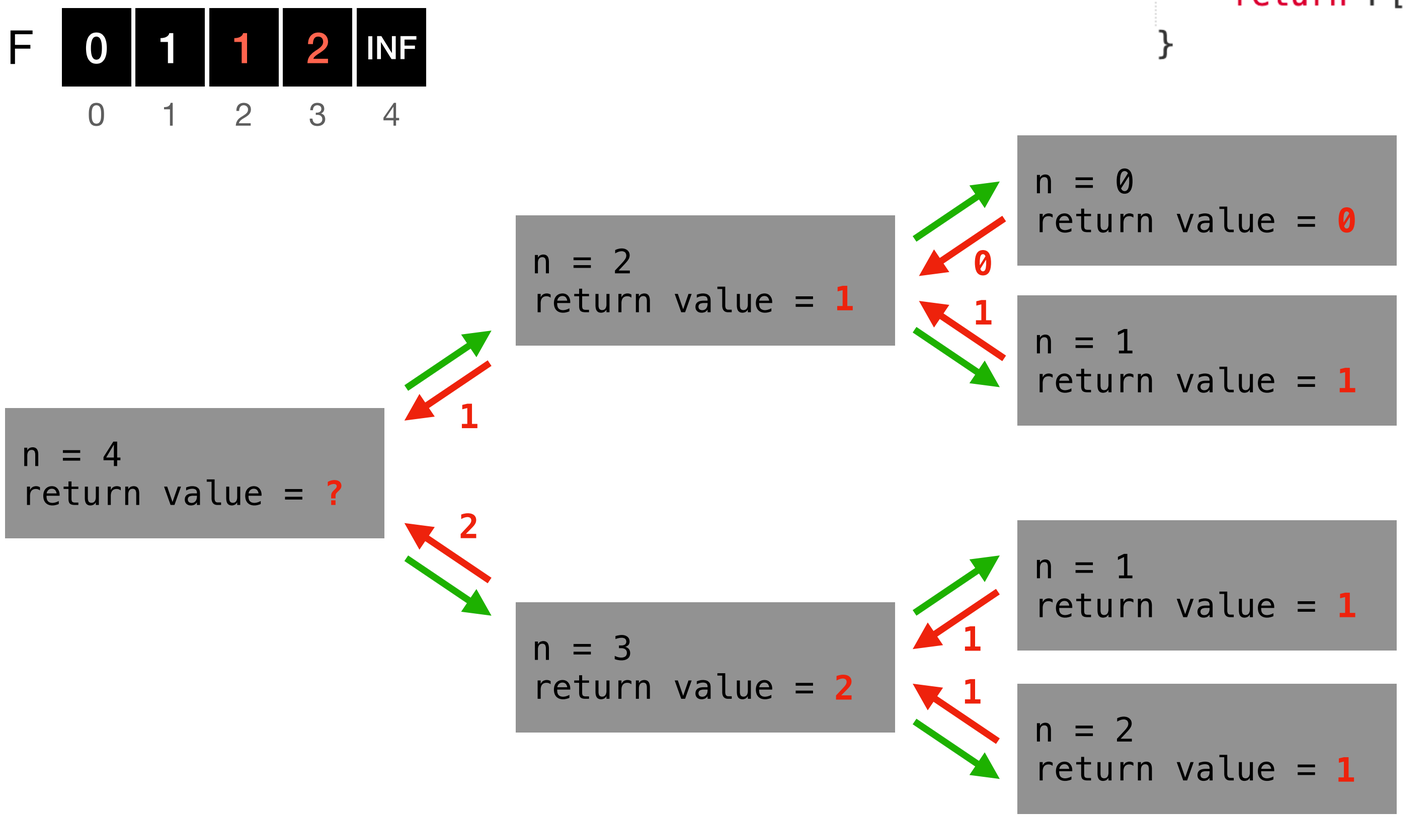
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

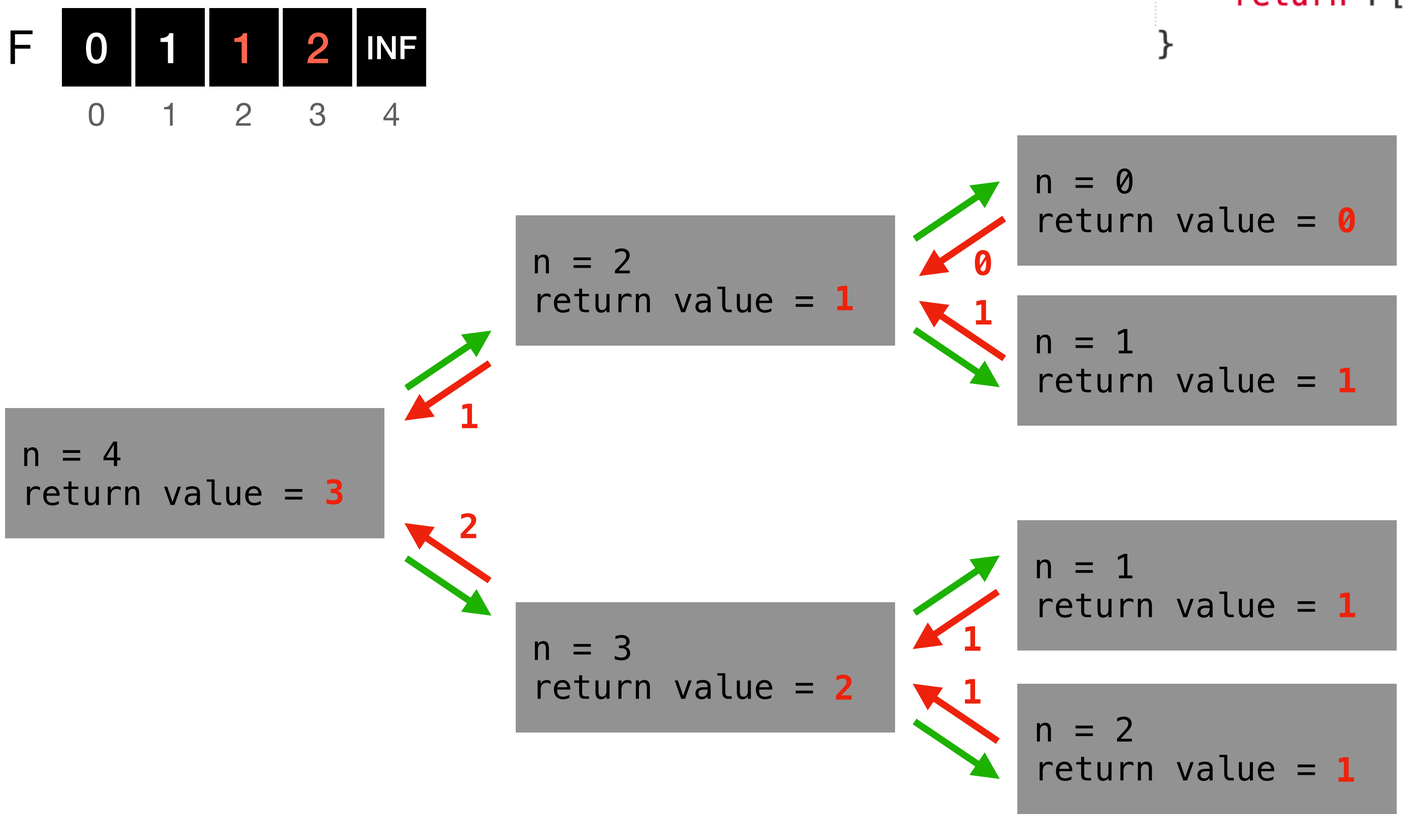
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

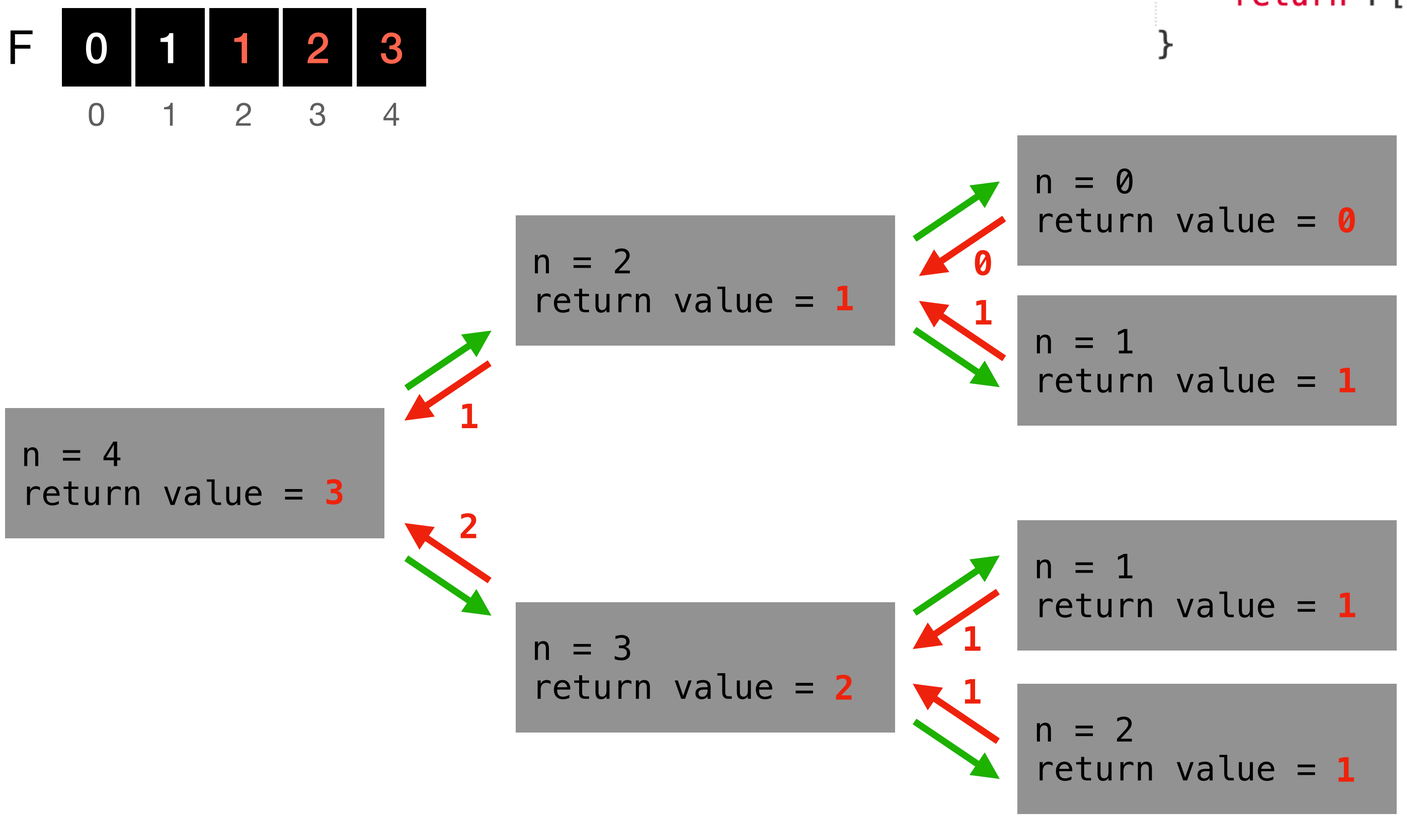
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

Example for `fibonacci_recursive_memoized(4)`.

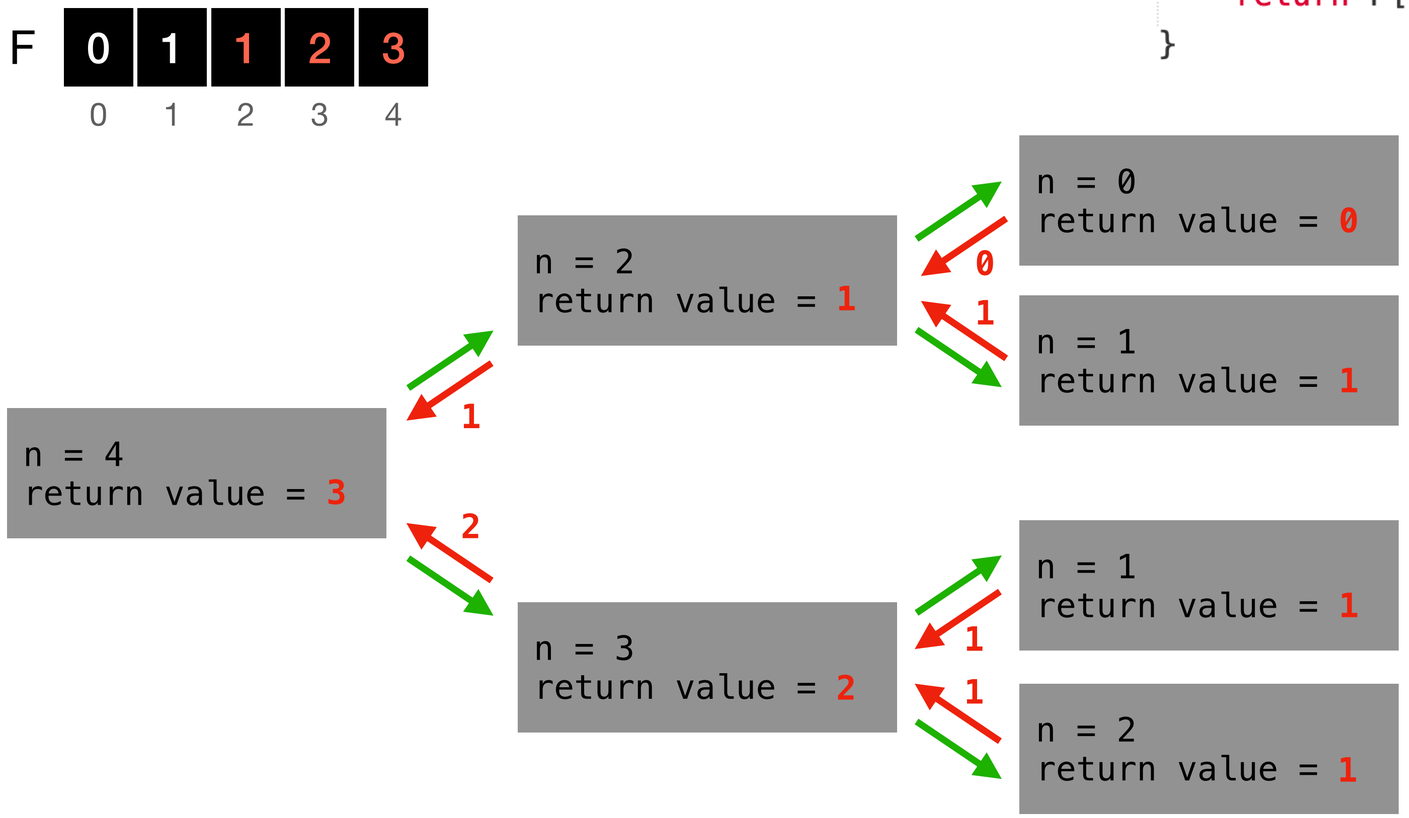
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



Recursion tree — Example

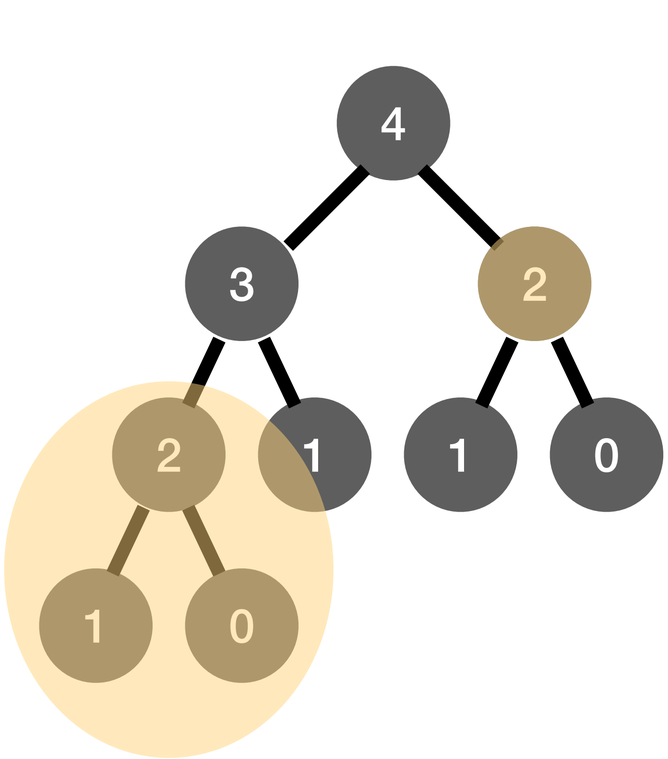
Example for `fibonacci_recursive_memoized(4)`.

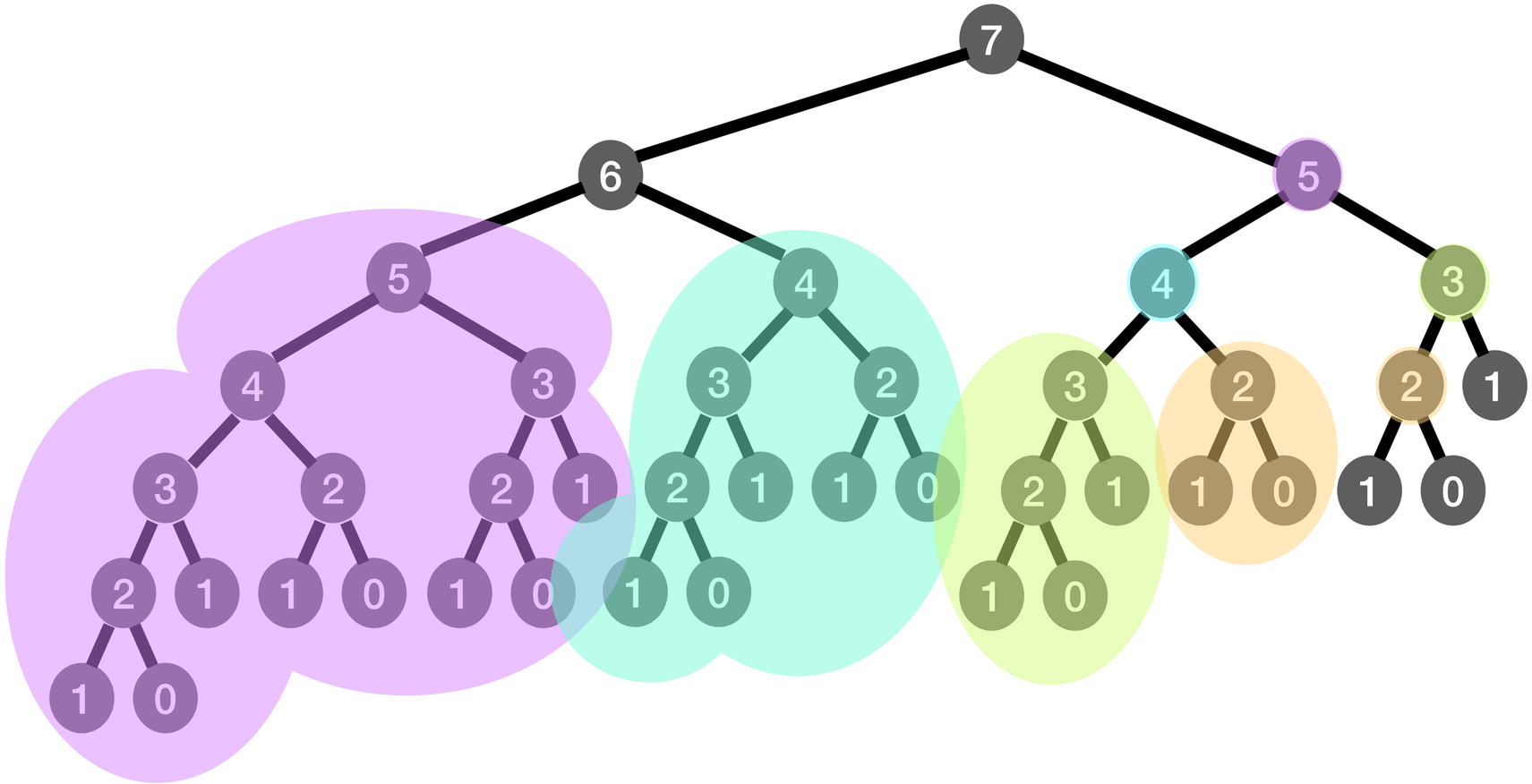
```
uint64_t fibonacci_recursive_memoized(const uint64_t n) {
    if (F[n] != INF) return F[n];
    F[n] = fibonacci_recursive_memoized(n - 2) +
           fibonacci_recursive_memoized(n - 1);
    return F[n];
}
```



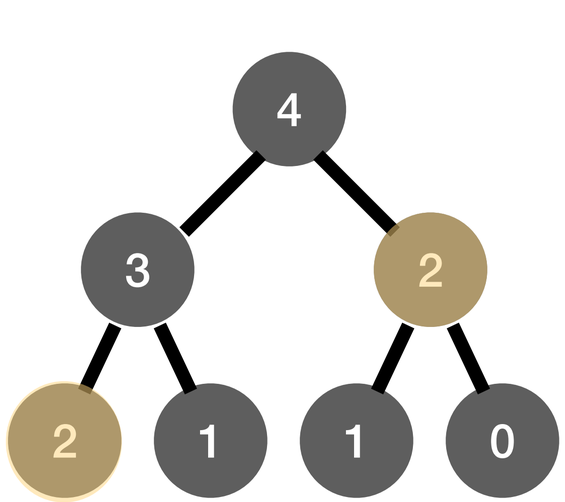
- Observe that now we solve each sub-problem **only once**, for $n = 0,\ldots,4$ (problems for $n = 0$ and $n = 1$ are already solved), spending $O(1)$ time per sub-problem (lookup in the array F).
- Hence, the solution runs in $\Theta(n)$ time **but** also consumes $\Omega(n)$ space.

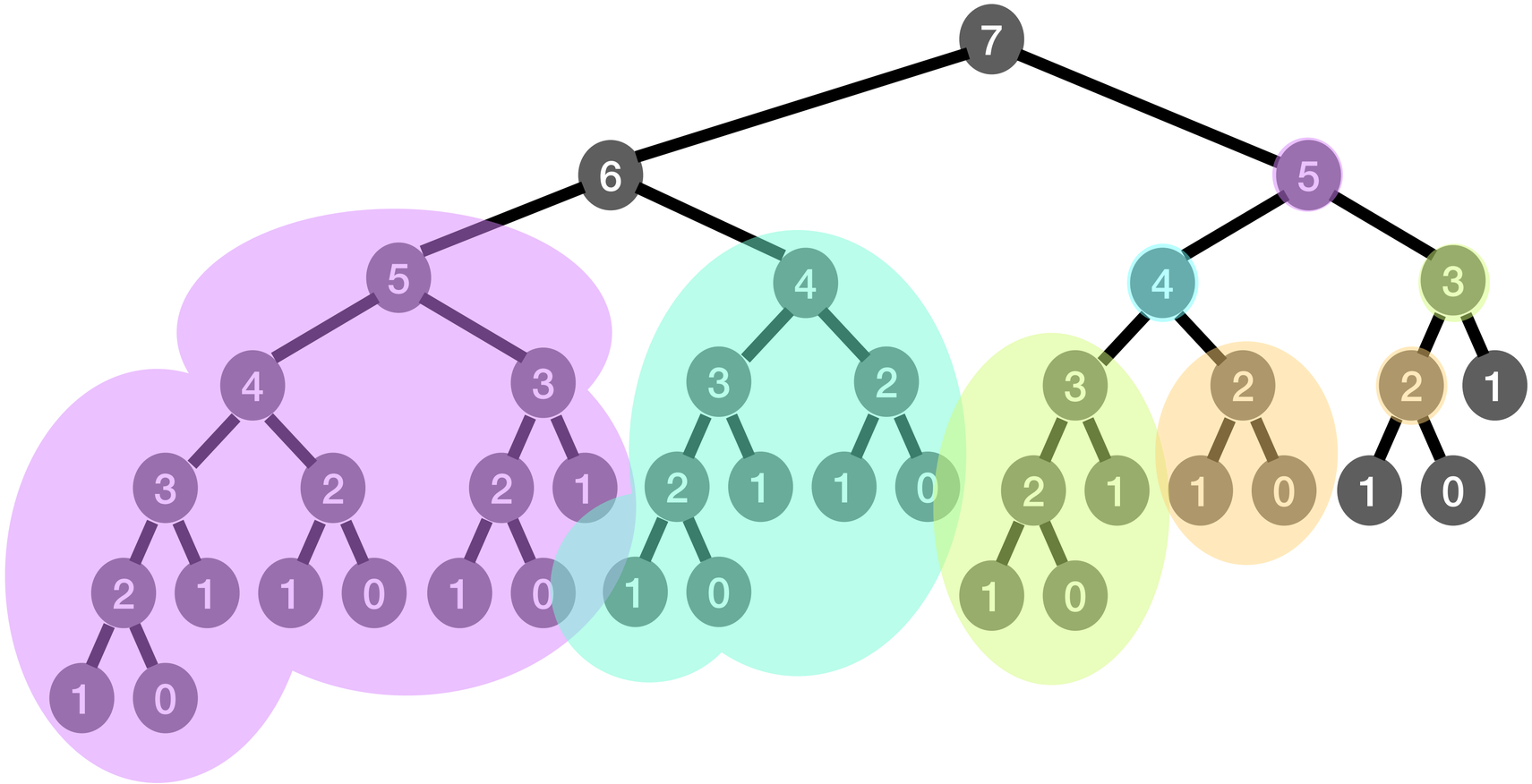
Recursion tree — More examples



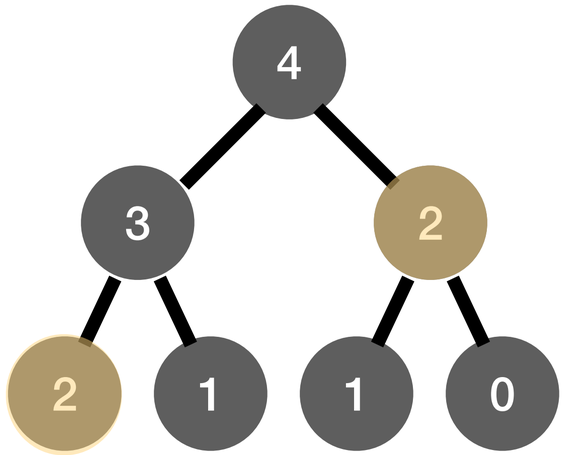


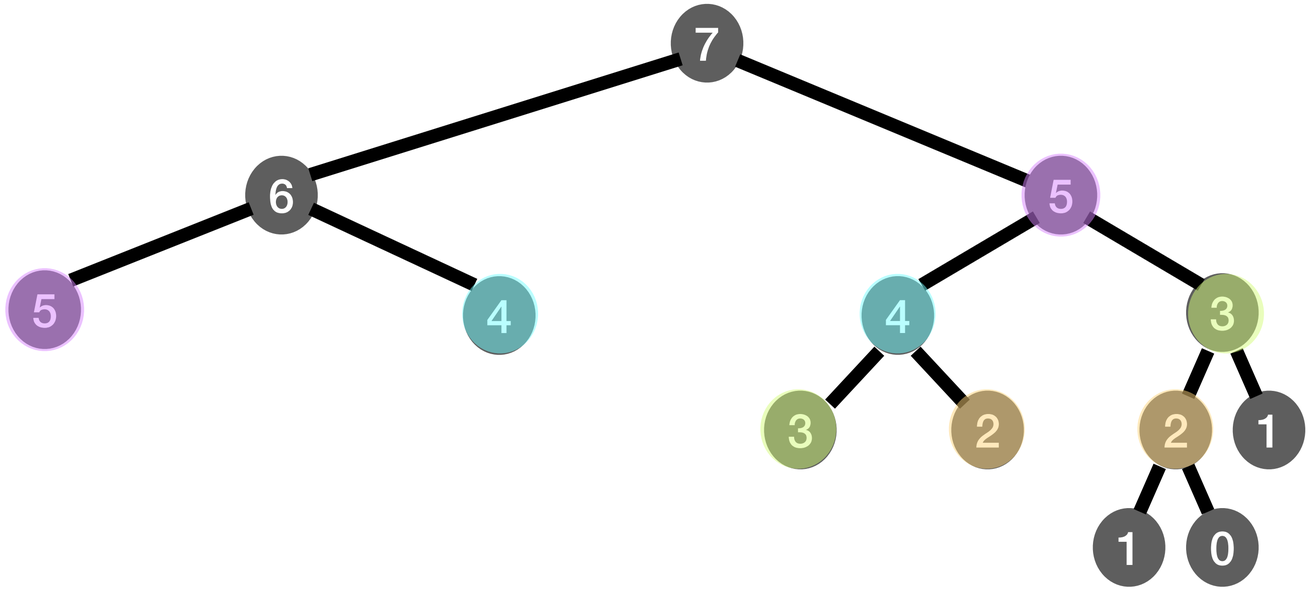
Recursion tree — More examples





Recursion tree — More examples





Coins

- **Problem.** Given a set of coin values $V = \{v_1, \ldots, v_k\}$, such that $v_i \neq v_j$ for all (i, j) and $v_i > 0$ for all i (otherwise the problem does not make sense), and a target sum of money n: **form the sum n using as few coins as possible.**

- We are going to assume: (1) integer values, (2) $v_1 = 1$ so that we can form any sum n, and (3) $k \geq 2$ (at least another coin value different from 1).

- **Example.** If the coin values are $V = \{1,3,4\}$ and $n = 10$, then the optimal solution is 3 since we need at least 3 coins to sum up to $10 = 4 + 3 + 3$. (It is easy to see we cannot do better.)

Coins

- **Problem.** Given a set of coin values $V = \{v_1, \ldots, v_k\}$, such that $v_i \neq v_j$ for all (i, j) and $v_i > 0$ for all i (otherwise the problem does not make sense), and a target sum of money n: **form the sum n using as few coins as possible.**

- We are going to assume: (1) integer values, (2) $v_1 = 1$ so that we can form any sum n, and (3) $k \geq 2$ (at least another coin value different from 1).

- **Example.** If the coin values are $V = \{1,3,4\}$ and $n = 10$, then the optimal solution is 3 since we need at least 3 coins to sum up to $10 = 4 + 3 + 3$. (It is easy to see we cannot do better.)

- Note that the greedy algorithm that **always selects the largest coin value** from the remaining sum does not always produce an optimal solution!

- **Example.** $n = 10 \rightarrow$ choose 4; $n = 6 \rightarrow$ choose 4; $n = 2 \rightarrow$ choose 1; $n = 1 \rightarrow$ choose 1. We computed a solution with 4 coins!

Coins

- Let's formulate the problem **recursively**.
- Let $coins(n)$ be the minimum number of coins required to form the sum n.

Coins

- Let's formulate the problem **recursively**.
- Let $coins(n)$ be the minimum number of coins required to form the sum n.
- Then, for our values $V = \{1,3,4\}$, it follows that

$$coins(n) = \min\{1 + coins(n-1), 1 + coins(n-3), 1 + coins(n-4)\}$$

since each time we **make a choice**: we select one coin value and reduce the problem to a smaller problem (smaller sum) with the same formulation.

- As base case, we assume that: $coins(0) = 0$ and $coins(n) = +\infty$ if $n < 0$.

Coins — Recursive solution

- Therefore we have that

$$coins(n) = \begin{cases} +\infty & n < 0 \\ 0 & n = 0 \\ \min_{v \in V} \{1 + coins(n - v)\} & n > 0 \end{cases}$$

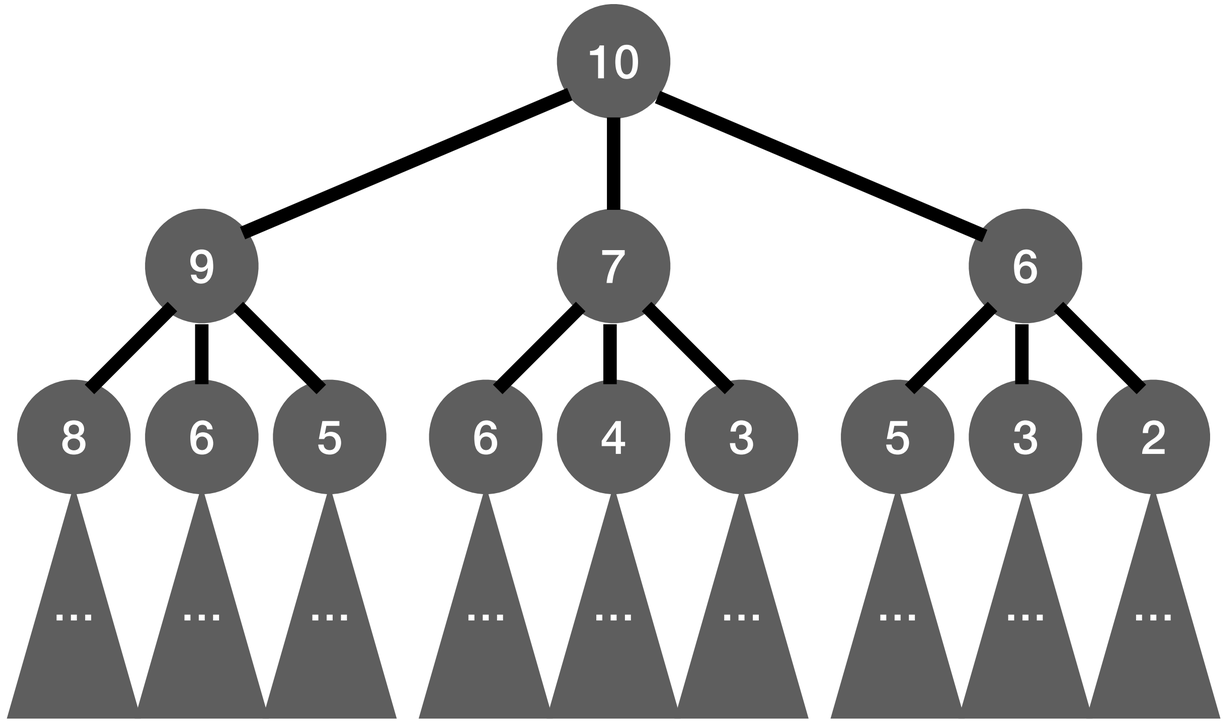
- This immediately translate to a compact solution in code.

- **Q.** Complexity?

```
11  uint64_t coins(const int64_t n) {
12      if (n == 0) return 0;
13      if (n < 0) return INF;
14      uint64_t best = INF;
15      for (uint64_t i = 0; i != k; ++i) {
16          uint64_t val = coins(n - V[i]) + 1;
17          if (val < best) best = val;
18      }
19      return best;
20  }
```

Coins — Recursion tree

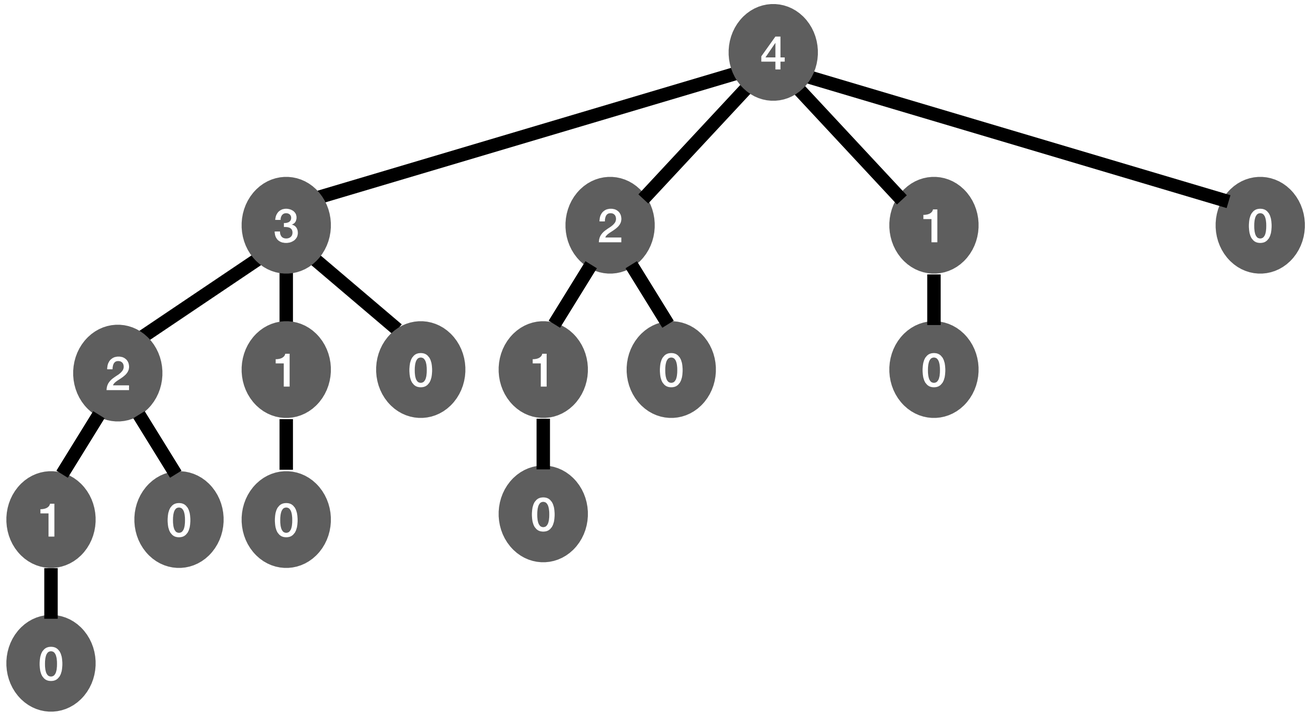
- Let's draw a picture to gain some insight, for $n = 10$ and $V = \{1,3,4\}$.
- **Q.** How many nodes do we have in the worst case?



- Note the redundant work!
- Insight: a coin value represents a "jump" towards the base case (when recursion stops).
- Hence the worst-case scenario arises when we make **as little progress as possible** towards the base case.

Coins — Recursion tree

- Worst-case scenario: $V = \{1,2,\ldots,n-1,n\}$.
- **Q.** How many nodes do we have now?



- Let $N(n)$ be the num. of nodes for the sum n.
- Thus we have
  $N(n) = 1 + \sum_{i=0}^{n-1} N(i), N(0) = 1.$

Coins — Recursion tree

- We have $N(n) = 1 + \sum_{i=0}^{n-1} N(i)$, $N(0) = 1$.
- **Claim.** $N(n) = 2N(n-1), n \geq 1$.
- In fact: $N(n) = 1 + N(0) + N(1) + \cdots + N(n-2) + N(n-1)$
  $= N(n-1) + N(n-1) = 2N(n-1)$. ■

Coins — Recursion tree

- We have $N(n) = 1 + \sum_{i=0}^{n-1} N(i)$, $N(0) = 1$.
- **Claim.** $N(n) = 2N(n-1), n \geq 1$.
- In fact: $N(n) = 1 + N(0) + N(1) + \cdots + N(n-2) + N(n-1)$
  $= N(n-1) + N(n-1) = 2N(n-1)$. ■
- So now:
  $N(n) = 1 + N(0) + N(1) + N(2) + \cdots + N(n-1)$
  $= 1 + (1) + (2N(0)) + (2N(1)) + \cdots + (2N(n-2))$ ← For the claim above
  $= 1 + (1) + (2) + (4) + (8) + \cdots + (2^{n-1})$
  $= 1 + (2^n - 1) = 2^n$. ■

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

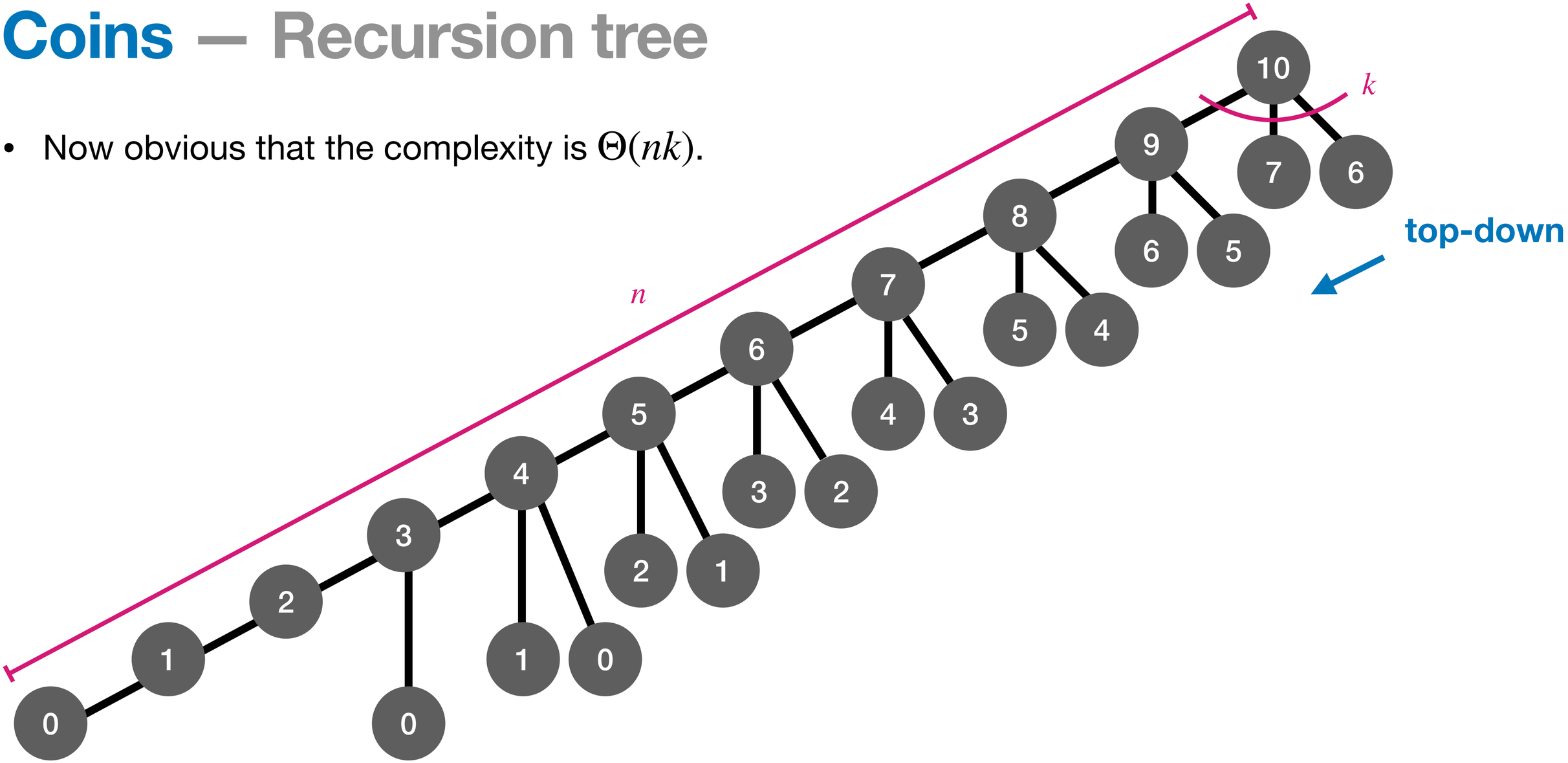
Coins — Recursive solution + memoization

- Again, the idea is simple: use an array, `C`, where `C[n]` stores the result of $coins(n)$.
- Let `C[0]=0` and `C[i]=INF` for $1 \leq i < N$.
- **Top-down**, recursive, solution.
- We now solve each sub-problem once for any $n - V[i]$, so the complexity is $\Theta(kn)$.

```
25  uint64_t coins_rec_memoized(const int64_t n) {
26      if (n < 0) return INF;
27      if (C[n] != INF) return C[n];
28      uint64_t best = INF;
29      for (uint64_t i = 0; i != k; ++i) {
30          uint64_t val = coins_rec_memoized(n - V[i]) + 1;
31          if (val < best) best = val;
32      }
33      C[n] = best;
34      return C[n];
35  }
```

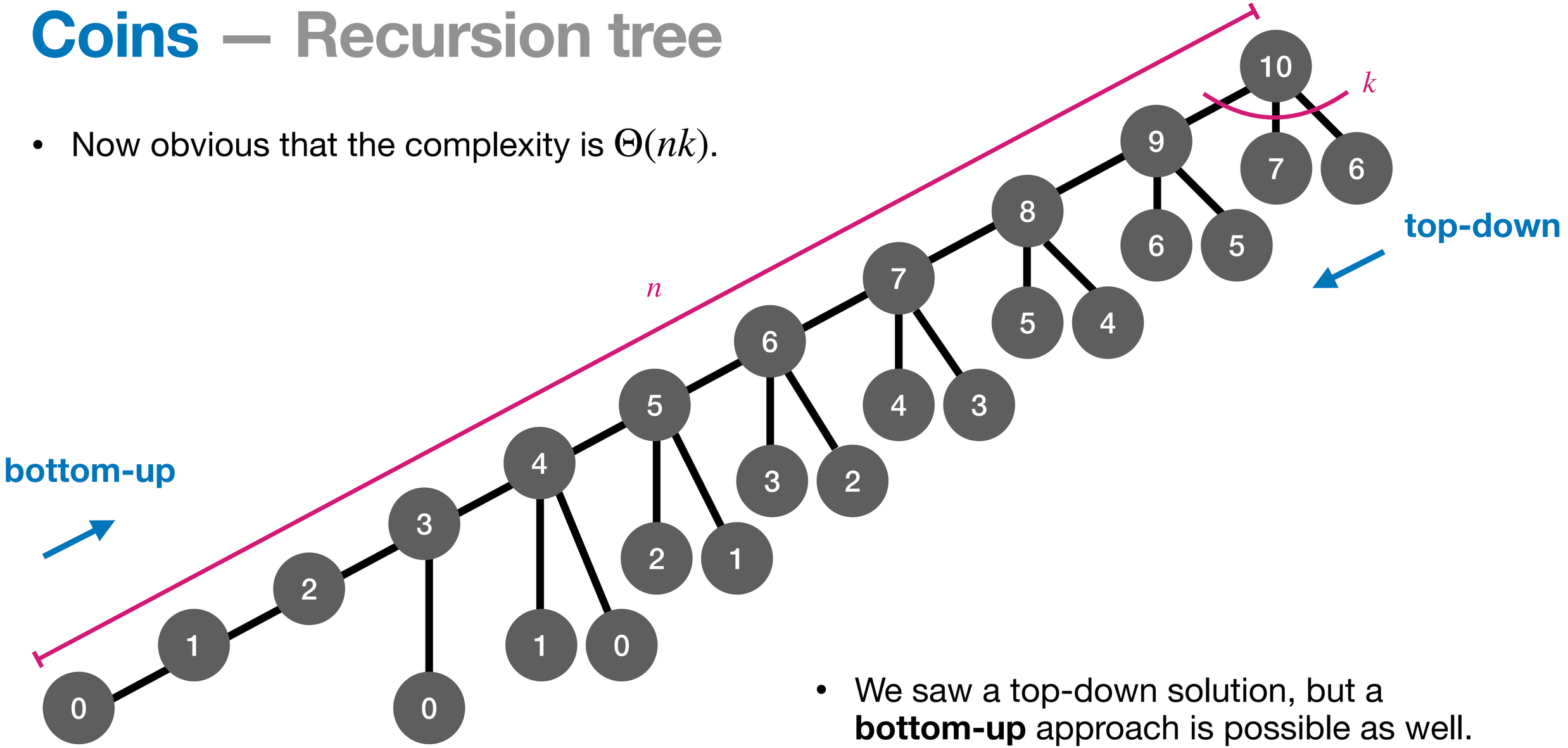
Coins — Recursion tree

- Now obvious that the complexity is $\Theta(nk)$.



Coins — Recursion tree

- Now obvious that the complexity is $\Theta(nk)$.



- We saw a top-down solution, but a **bottom-up** approach is possible as well.

Coins — Iterative solution + memoization

- We can also develop a bottom-up, iterative, solution.
- This solution has the **same asymptotic complexity** as the previous one (recursive) but **better constant factors, hence it is practically faster**.

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

Coins — Example

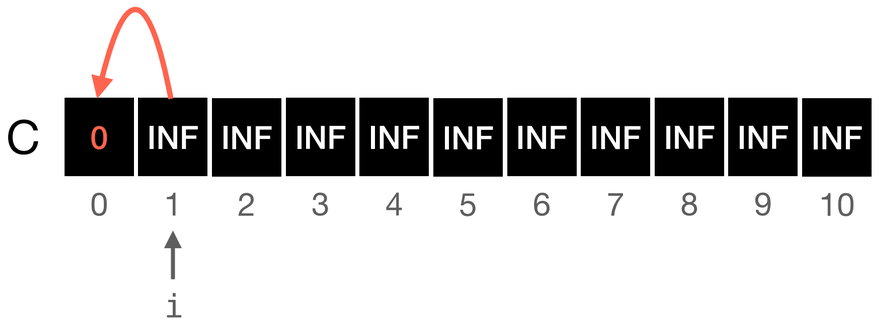
```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.

Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

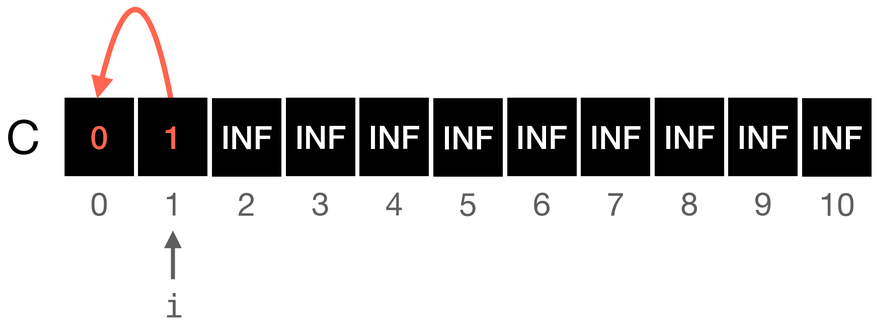
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

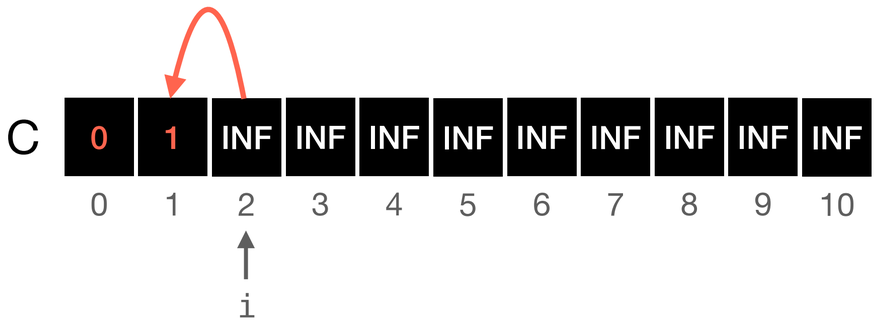
```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.

Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

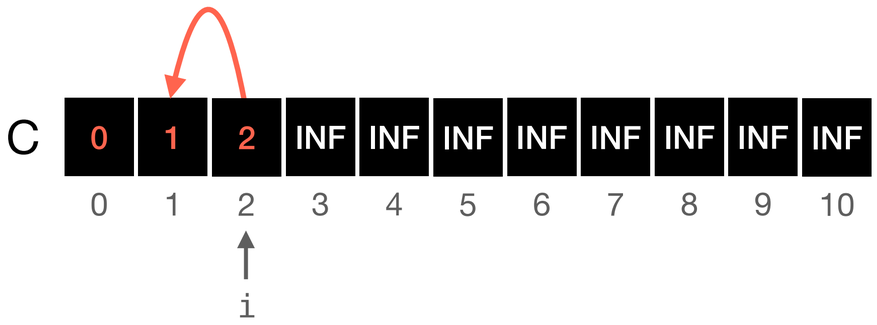
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

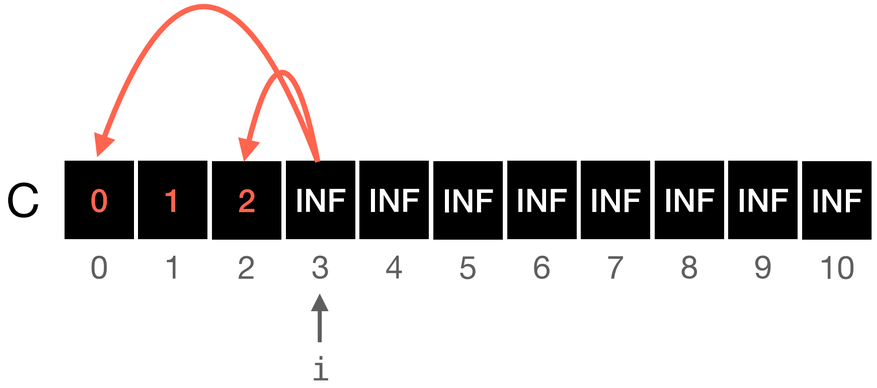
```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.

Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

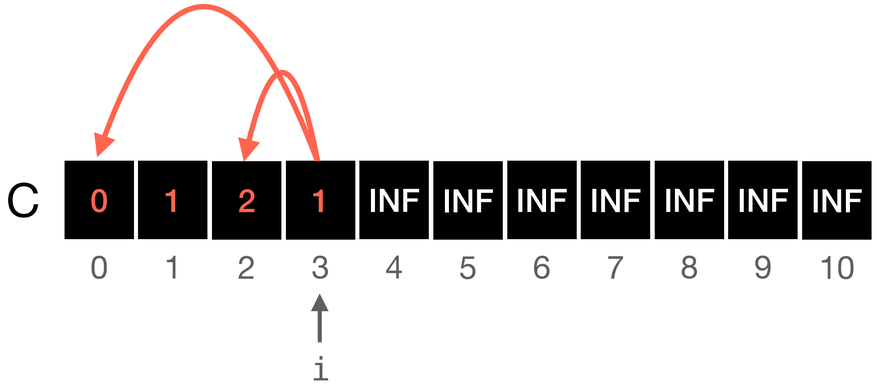
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

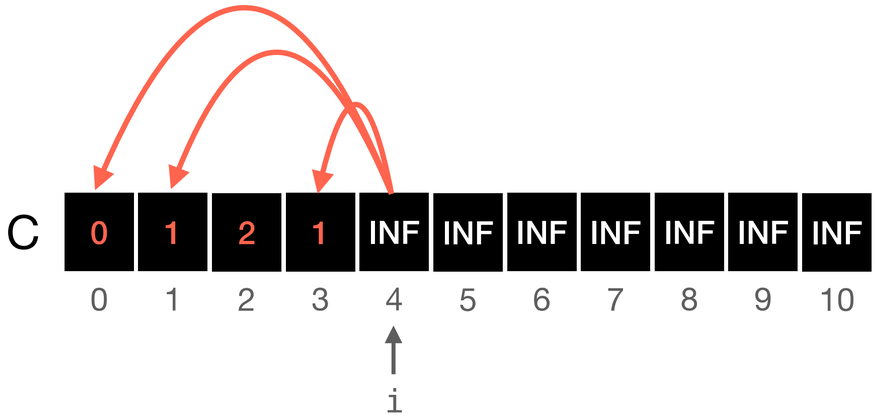
```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.

Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

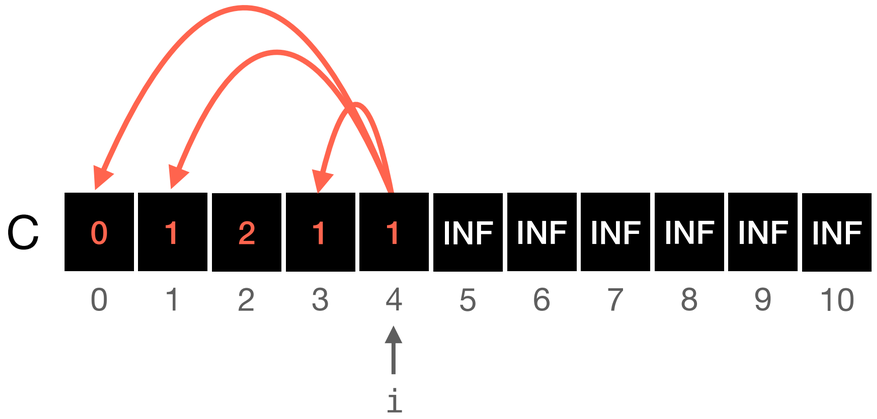
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

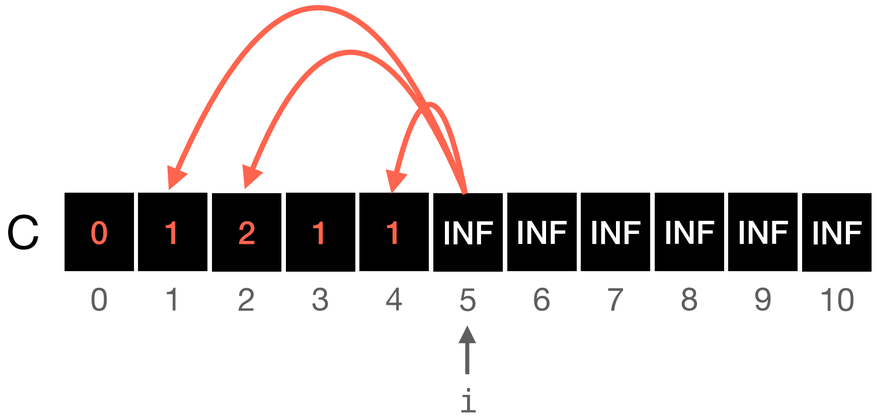
```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.

Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

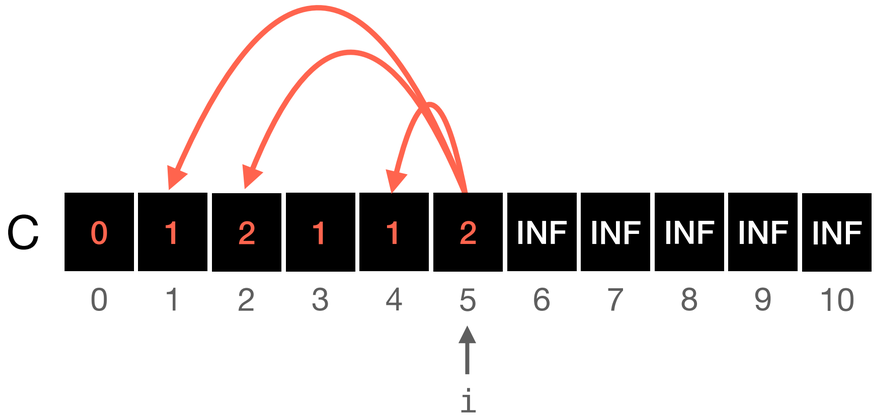
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

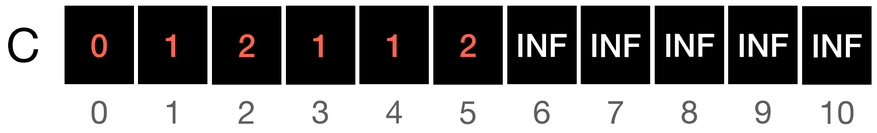
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

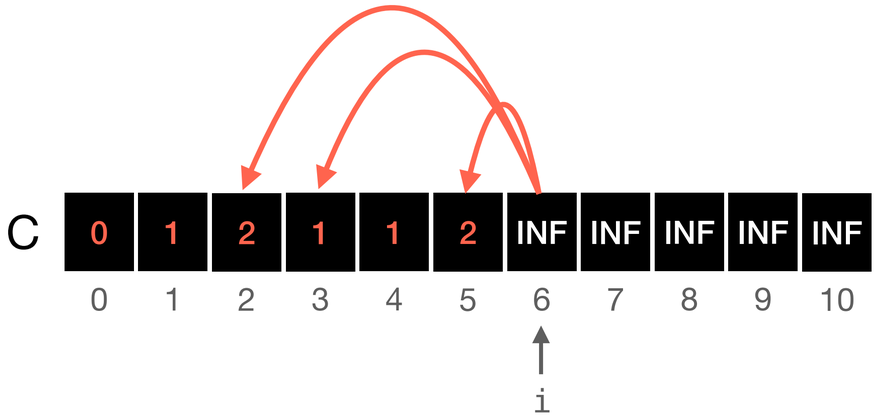
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

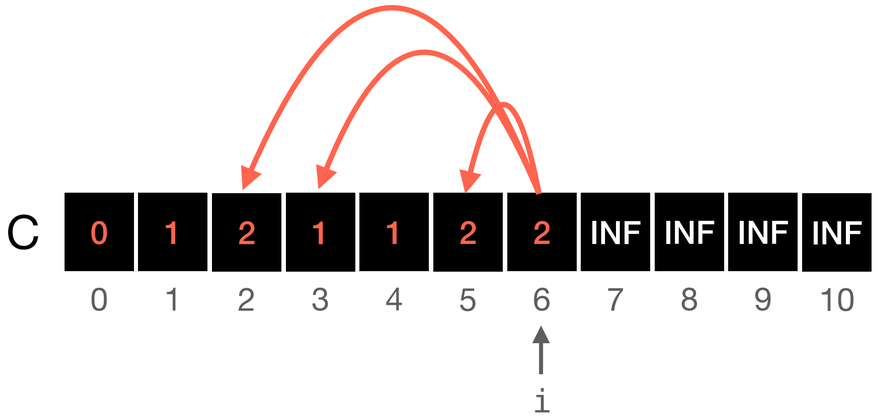
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

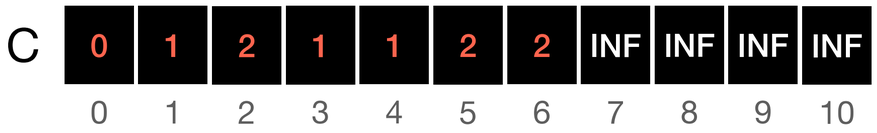
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

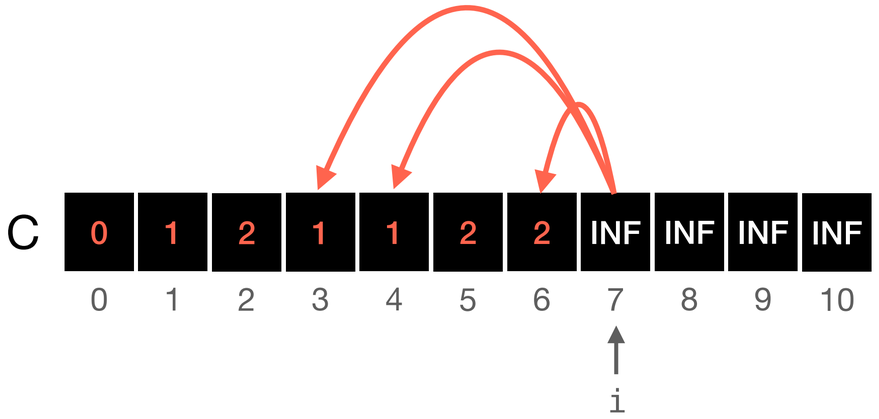
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

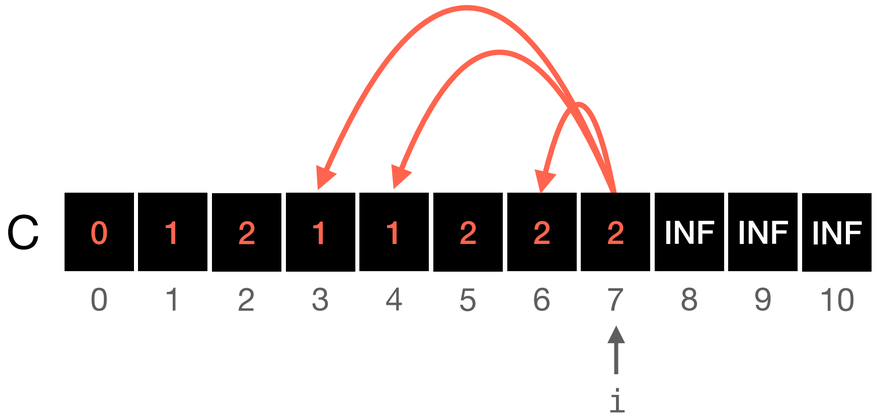
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

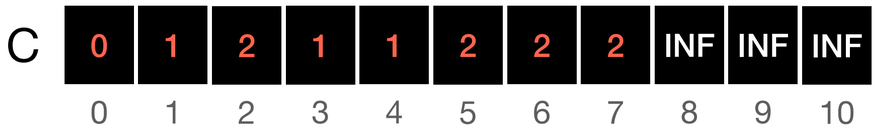
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

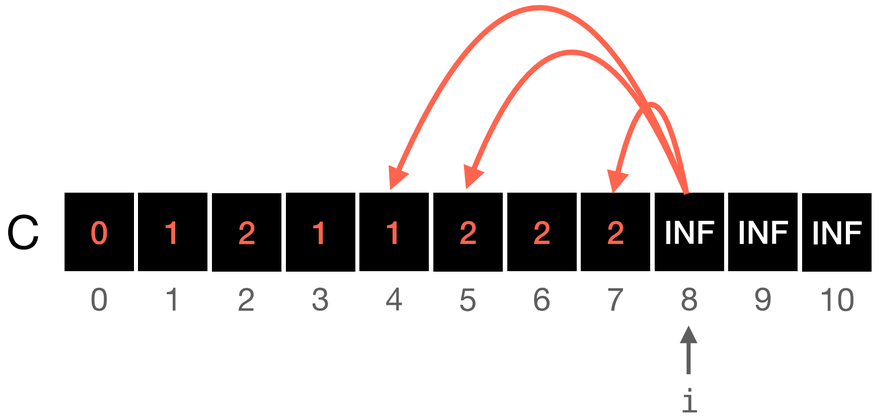
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

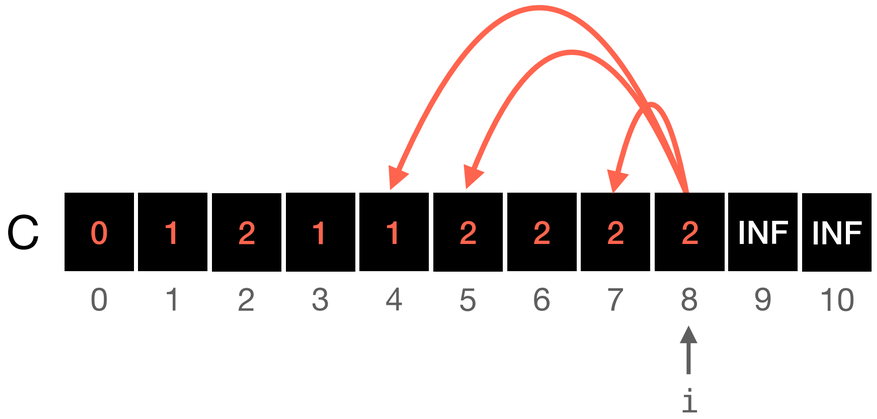
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

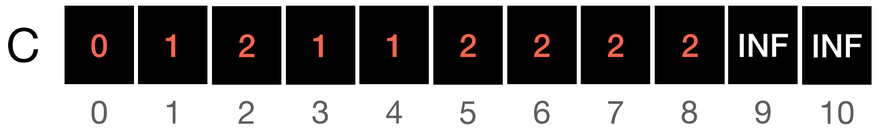
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

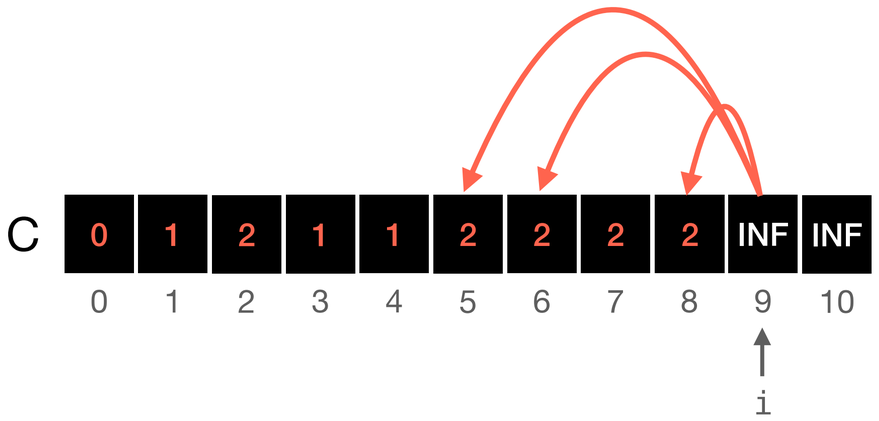
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

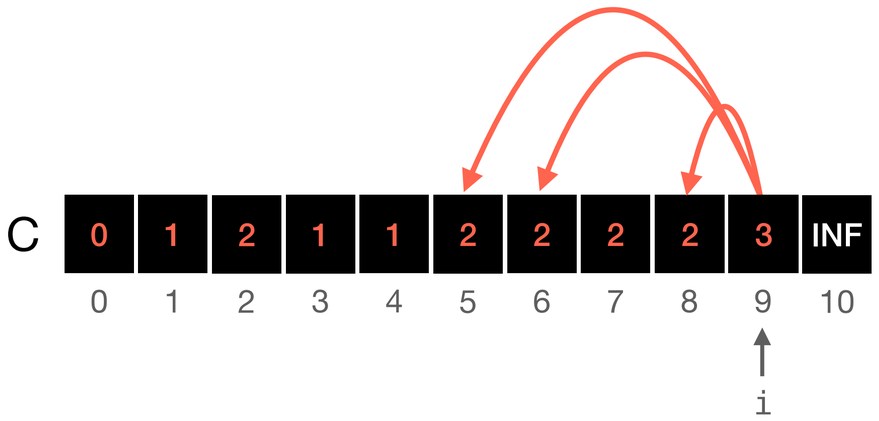
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

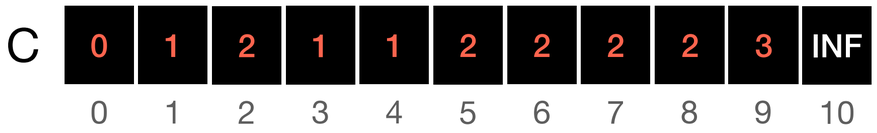
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

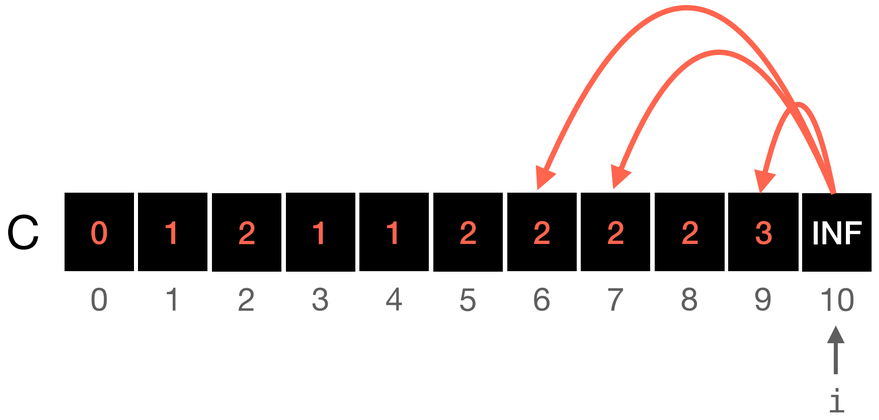
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

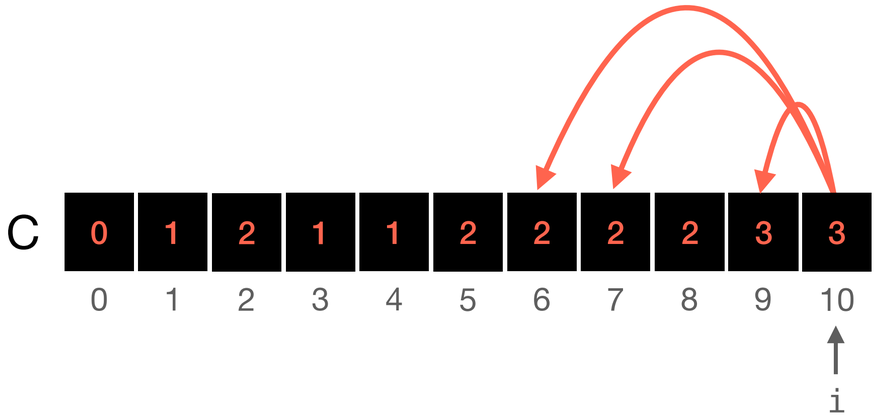
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

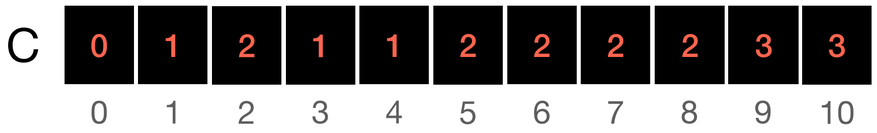
- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Example

```
39  uint64_t coins_itr_memoized(const uint64_t n) {
40      for (uint64_t i = 0; i <= n; ++i) {
41          for (uint64_t j = 0; j != k; ++j) {
42              if (i >= V[j]) C[i] = std::min(C[i], C[i - V[j]] + 1);
43          }
44      }
45      return C[n];
46  }
```

- Let's run an example for $n = 10$ and $V = \{1,3,4\}$.



Coins — Constructing the solution

- So far, we have computed the **value of the solution**, i.e. the minimum number of coins that sum up to n, but we do not know **which values** contribute to the sum.
- But a little modification to the code we have written, allows us to do so.
- The idea is general: during the process of solving the sub-problems, store some extra information (another array!) to allow quick computation of the solution.

Coins — Constructing the solution

- So far, we have computed the **value of the solution**, i.e. the minimum number of coins that sum up to n, but we do not know **which values** contribute to the sum.
- But a little modification to the code we have written, allows us to do so.
- The idea is general: during the process of solving the sub-problems, store some extra information (another array!) to allow quick computation of the solution.

```
50  uint64_t coins_itr_memoized_with_sol(const uint64_t n) {
51      for (uint64_t i = 0; i <= n; ++i) {
52          for (uint64_t j = 0; j != k; ++j) {
53              if (i >= V[j] and C[i - V[j]] + 1 < C[i]) {
54                  C[i] = C[i - V[j]] + 1;
55                  S[i] = V[j];
56              }
57          }
58      }
59      return C[n];
60  }
```

use another array `S[0..n+1]` and
record at `S[i]` the first coin in the sum `i`

Coins — Printing the solution

- From a sum of n, print the first coin value, that is $S[n]$. Then you reduce the problem of printing the first value from the sum $n - S[n]$, which is $S[n - S[n]]$, etc.
- In code:

```
108     uint64_t i = n;
109     while (i != 0) {
110         std::cout << S[i] << ' ';
111         i -= S[i];
112     }
113     std::cout << std::endl;
```

Rod cutting

- You are given a rod of length n and a list of prices $[p_1, \ldots, p_n]$ where p_i represents the price of a rod of length i.

- **Problem.** How should you cut a rod of length n into smaller pieces so that you can earn the maximum amount by selling the pieces?

Rod cutting

- You are given a rod of length n and a list of prices $[p_1, \ldots, p_n]$ where p_i represents the price of a rod of length i.

- **Problem.** How should you cut a rod of length n into smaller pieces so that you can earn the maximum amount by selling the pieces?

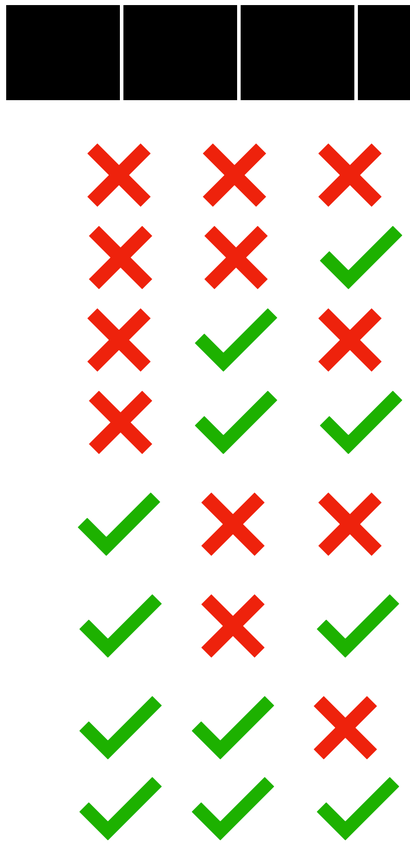
- Example for $n = 4$. Suppose we are given the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

We can cut the rod into 1 | 3 to earn a total of $p_1 + p_3 = 9$ (same as not cutting the rod at all because $p_4 = 9$), but the optimal solution is cutting it into 2 | 2, to earn $p_2 + p_2 = 10$.

Rod cutting

- In general, there are 2^{n-1} possible cuts (assuming $p_1 = 1$ is always available), because for each position $i = 1, \ldots, n-1$ we have the choice of cutting or not cutting.



$n = 4$

Rod cutting

- Let's formulate the problem **recursively**.
- Let $R(n)$ be the maximum revenue we can get from a rod of length n.
- **Q.** If you cut the rod of length n at i, i.e. you obtain the two shorter rods of length i and $n - i$ respectively, how can you express $R(n)$ in terms of $R(i)$ and $R(n - i)$?

Rod cutting

- Let's formulate the problem **recursively**.
- Let $R(n)$ be the maximum revenue we can get from a rod of length n.
- **Q.** If you cut the rod of length n at i, i.e. you obtain the two shorter rods of length i and $n - i$ respectively, how can you express $R(n)$ in terms of $R(i)$ and $R(n - i)$?

- **A.** Clearly:

$$R(n) = \max\{p_n, R(1) + R(n-1), R(2) + R(n-2), \ldots, R(n-1) + R(1)\}.$$

Rod cutting

- Hence we can write $R(n)$ as

$$R(n) = \max\{p_n, R(1) + R(n-1), R(2) + R(n-2), \ldots, R(n-1) + R(1)\},$$

  which we can further simplify into

$$R(n) = \max_{1 \le i \le n} \{p_i + R(n-i)\} \text{ for } n > 0 \text{ and } R(0) = 0$$

  (a rod of length 0, does not give any revenue), by fixing one cut at i and letting only the right rod be further splittable.

- Similarly to the “coins” problem: we make a choice (a cut) and recurse on a smaller sub-problem.

Rod cutting — Recursive solution

- Now that we have this problem formulation:

$$R(n) = \max_{1 \le i \le n} \{p_i + R(n-i)\} \text{ for } n > 0 \text{ and } R(0) = 0,$$

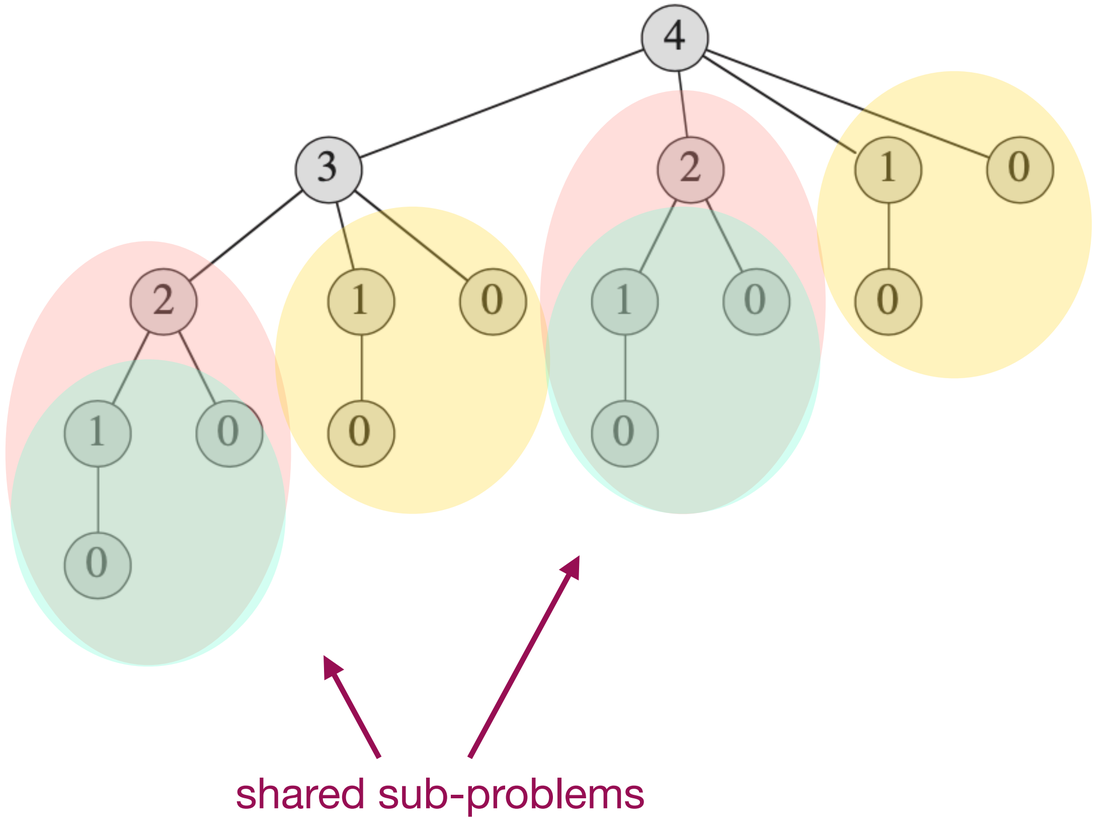
it is easy to translate it to code.

```
7   const uint64_t n = 10;
8   uint64_t P[n + 1];  // P[i] := p_i
9
10  uint64_t rod_cutting(const uint64_t n) {
11      uint64_t best = 0;
12      for (uint64_t i = 1; i <= n; ++i) {
13          best = std::max(best, P[i] + rod_cutting(n - i));
14      }
15      return best;
16  }
```

- Note that the base case for $n = 0$ (we always return 0) is guaranteed even without an explicit test.

Rod cutting — Recursion tree

- Recursion tree for `rod_cutting(4)`.
- There are a total of 2^n nodes in the tree (same number of nodes as for the “coins” problem in the worst-case), hence the total running time is $\Theta(2^n)$.



Rod cutting — Recursive solution + memoization

```
18  uint64_t R[n + 1];  // one extra dummy value
19
20  uint64_t rod_cutting_rec_memoized(const uint64_t n) {
21      if (R[n] > 0) return R[n];
22      uint64_t best = 0;
23      for (uint64_t i = 1; i <= n; ++i) {
24          best = std::max(best, P[i] + rod_cutting_rec_memoized(n - i));
25      }
26      R[n] = best;
27      return best;
28  }
```

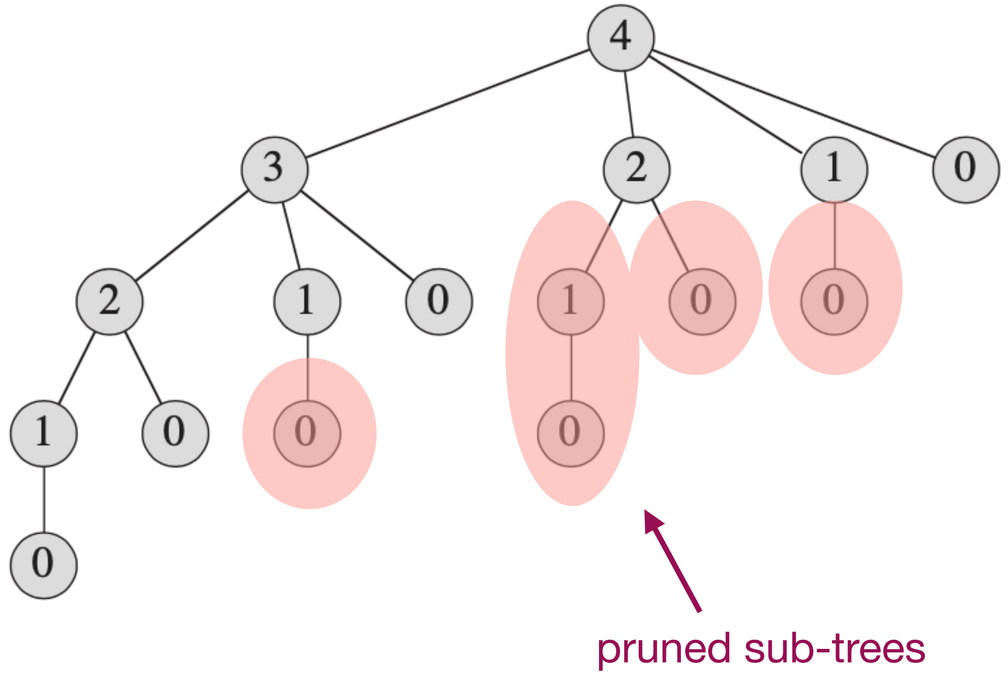
- **Q.** Complexity?
- **A.** Let’s draw the recursion tree for this version.

Rod cutting — Recursion tree

- Recursion tree for `rod_cutting_rec_memoized(4)`.
- There are a total of

$$1 + \sum_{i=1}^{n} i = 1 + \frac{n(n+1)}{2}$$

nodes in the tree, hence the total running time is $\Theta(n^2)$.



Rod cutting — Iterative solution + memoization

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

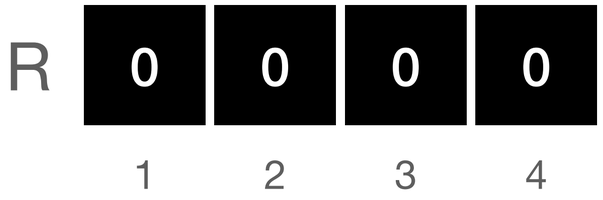
- **Q.** Complexity? Now easy to see that the complexity is $\Theta(n^2)$ because of the nested for-loop.

Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

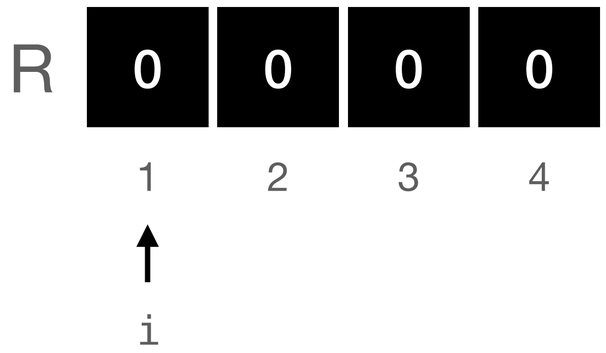


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

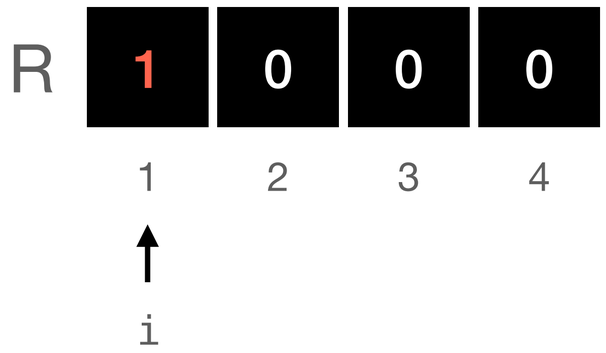


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

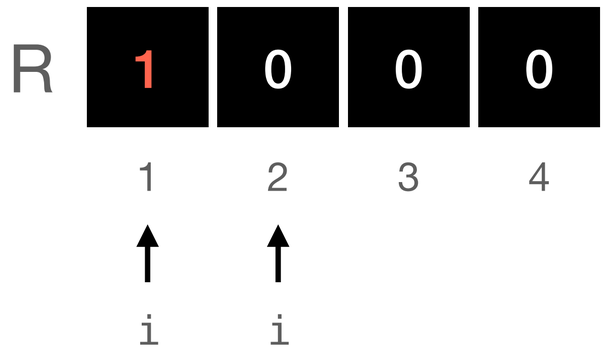


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

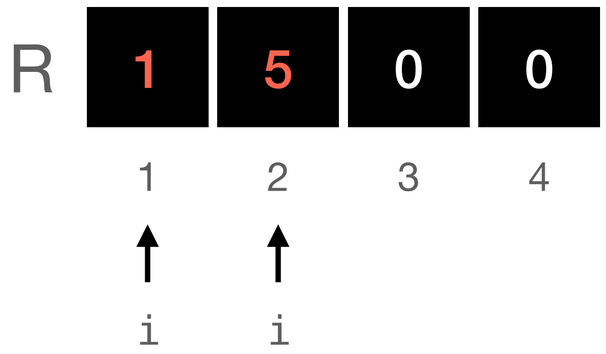


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

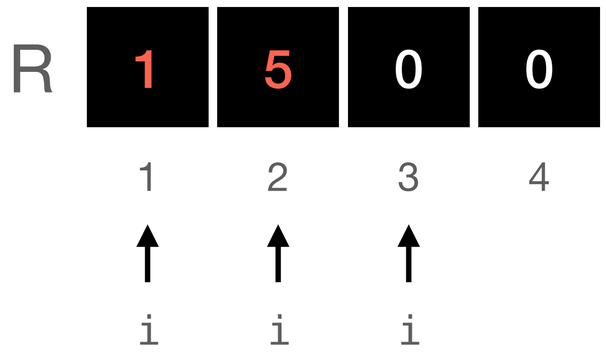


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

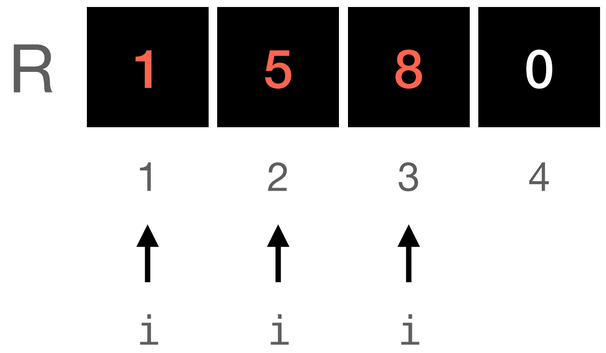


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

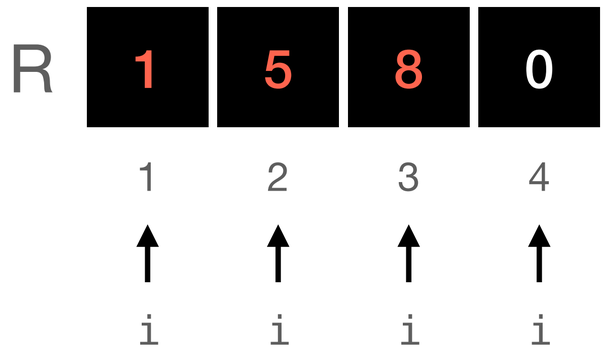


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```

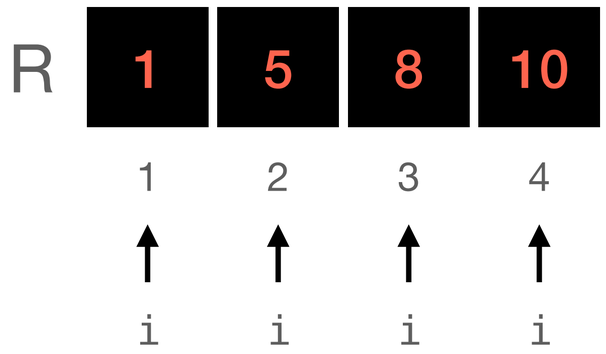


Rod cutting — Example

- Example for $n = 4$ and the following price table.

length i	1	2	3	4
price p_i	1	5	8	9

```
30  uint64_t rod_cutting_itr_memoized(const uint64_t n) {
31      for (uint64_t i = 1; i <= n; ++i) {
32          uint64_t best = 0;
33          for (uint64_t j = 1; j <= i; ++j) {
34              best = std::max(best, P[j] + R[i - j]);
35          }
36          R[i] = best;
37      }
38      return R[n];
39  }
```



Rod cutting — constructing and printing the solution

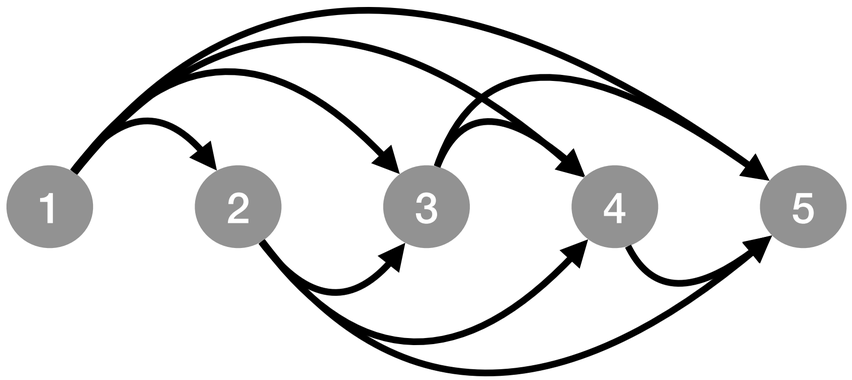
- As for the coins problem, use an extra array `S[1..n]`, and let `S[i]=j` if the best cut for a rod of length i is j.

```
41  uint64_t S[n + 1];  // one extra dummy value
42
43  uint64_t rod_cutting_itr_memoized_with_sol(const uint64_t n) {
44      for (uint64_t i = 1; i <= n; ++i) {
45          uint64_t best_rev = 0;
46          uint64_t best_cut = 0;
47          for (uint64_t j = 1; j <= i; ++j) {
48              uint64_t rev = P[j] + R[i - j];
49              if (rev > best_rev) {
50                  best_rev = rev;
51                  best_cut = j;
52              }
53          }
54          R[i] = best_rev;
55          S[i] = best_cut;
56      }
57      return R[n];
58  }
```

```
131     uint64_t i = n;
132     while (i != 0) {
133         std::cout << S[i] << ' ';
134         i -= S[i];
135     }
136     std::cout << std::endl;
```

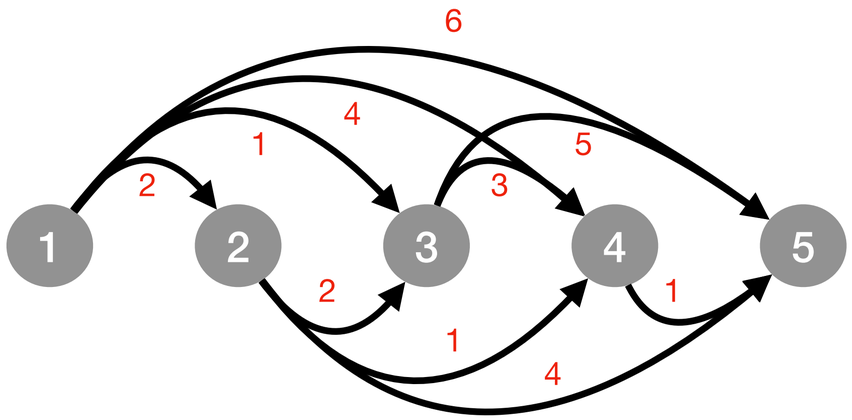
Exercise: Shortest path on a DAG

- A directed acyclic graph (DAG) is a graph with n nodes and a direct edge (j, i) for every $1 \leq j < i, i = 2, \ldots, n$. Hence there are $n(n-1)/2 = \Theta(n^2)$ edges.



Exercise: Shortest path on a DAG

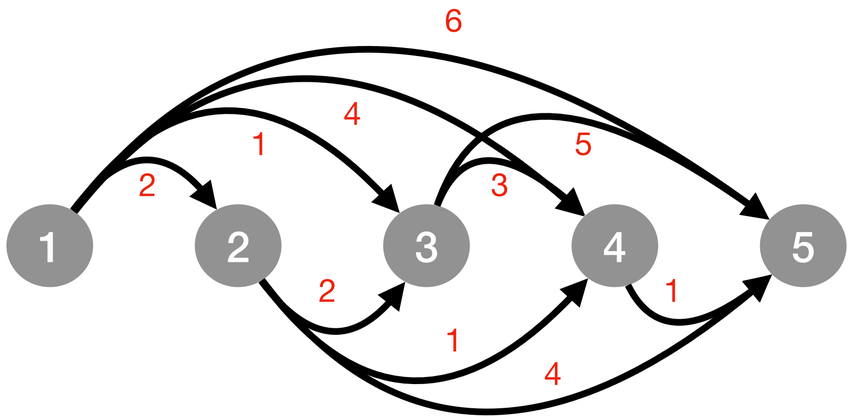
- A directed acyclic graph (DAG) is a graph with n nodes and a direct edge (j, i) for every $1 \leq j < i$, $i = 2, \ldots, n$. Hence there are $n(n-1)/2 = \Theta(n^2)$ edges.



- Let's now add weights to the edges: each edge (j, i) has a weight $w(j, i) > 0$.
- Let the **cost of a path** be the **sum of the weights of the edges it traverses**.

Exercise: Shortest path on a DAG

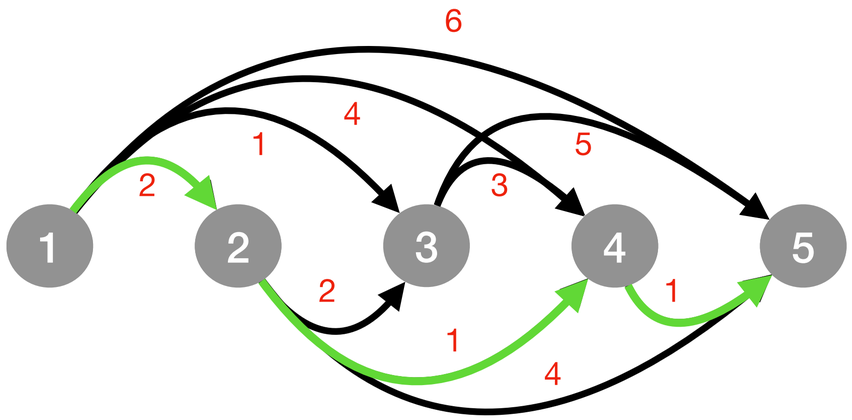
- A directed acyclic graph (DAG) is a graph with n nodes and a direct edge (j, i) for every $1 \leq j < i$, $i = 2, \ldots, n$. Hence there are $n(n-1)/2 = \Theta(n^2)$ edges.



- Let's now add weights to the edges: each edge (j, i) has a weight $w(j, i) > 0$.
- Let the **cost of a path** be the **sum of the weights of the edges it traverses**.
- **Problem.** How to get from node 1 to node n with a path of **minimum** cost?

Exercise: Shortest path on a DAG

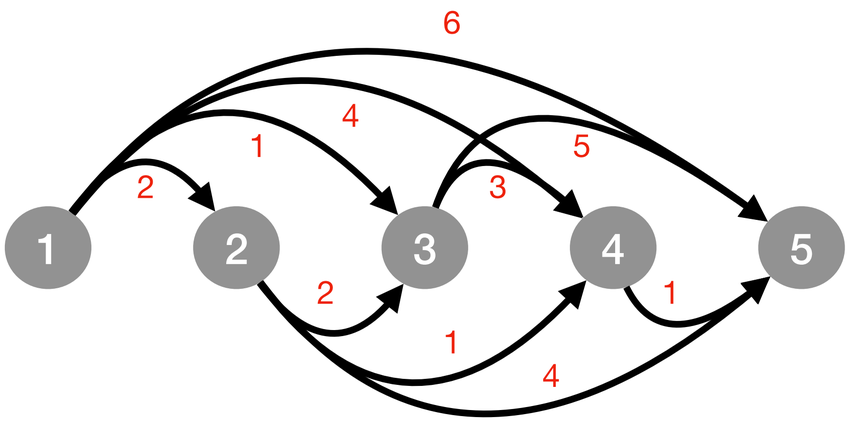
- A directed acyclic graph (DAG) is a graph with n nodes and a direct edge (j, i) for every $1 \leq j < i$, $i = 2, \ldots, n$. Hence there are $n(n-1)/2 = \Theta(n^2)$ edges.



- Let's now add weights to the edges: each edge (j, i) has a weight $w(j, i) > 0$.
- Let the **cost of a path** be the **sum of the weights of the edges it traverses**.
- **Problem.** How to get from node 1 to node n with a path of **minimum** cost?

Plan

1. Formulate the problem recursively.
2. Write a (recursive) solution for the problem. Complexity?
3. Use memoization (still with recursion) to obtain a faster solution. Complexity?
4. Bonus: write the iterative version with memoization.



1. Formulate the problem recursively

- Let $C(n)$ be the cost of the shortest path from node 1 to node n.
- Then

$$C(n) = \min\{C(1) + w(1,n), C(2) + w(2,n), \ldots, C(n-1) + w(n-1,n)\}, \text{ with } C(1) = 0.$$

That is:

$$C(n) = \min_{1 \le i < n} \{C(i) + w(i,n)\}, \text{ with } C(1) = 0.$$

2. Recursive solution

```
7   const uint64_t INF = -1;
8   const uint64_t N = 20;
9   uint64_t W[N + 1][N + 1];  // W[i][j] := weight of edge(i,j)
10
11  uint64_t shortest_path(const uint64_t n) {
12      if (n == 1) return 0;
13      uint64_t best = INF;
14      for (uint64_t i = 1; i != n; ++i) {
15          uint64_t cost = shortest_path(i) + W[i][n];
16          if (cost < best) best = cost;
17      }
18      return best;
19  }
```

- **Q.** Complexity?

2. Recursive solution

```
7   const uint64_t INF = -1;
8   const uint64_t N = 20;
9   uint64_t W[N + 1][N + 1];  // W[i][j] := weight of edge(i,j)
10
11  uint64_t shortest_path(const uint64_t n) {
12      if (n == 1) return 0;
13      uint64_t best = INF;
14      for (uint64_t i = 1; i != n; ++i) {
15          uint64_t cost = shortest_path(i) + W[i][n];
16          if (cost < best) best = cost;
17      }
18      return best;
19  }
```

- **Q.** Complexity?
- **A.** Draw the tree of sub-problems to see it is identical to that of the rod-cutting problem. Observe that there are 2^n nodes in the tree, hence the solution runs in $\Theta(2^n)$.

3. Use memoization to speed up the solution

```
21  uint64_t C[N + 1];  // C[i] := cost of shortest path from node 1 to node i
22
23  uint64_t shortest_path_rec_memoized(const uint64_t n) {
24      if (C[n] != INF) return C[n];
25      uint64_t best = INF;
26      for (uint64_t i = 1; i != n; ++i) {
27          uint64_t cost = shortest_path_rec_memoized(i) + W[i][n];
28          if (cost < best) best = cost;
29      }
30      C[n] = best;
31      return best;
32  }
```

- **Q.** Complexity?

3. Use memoization to speed up the solution

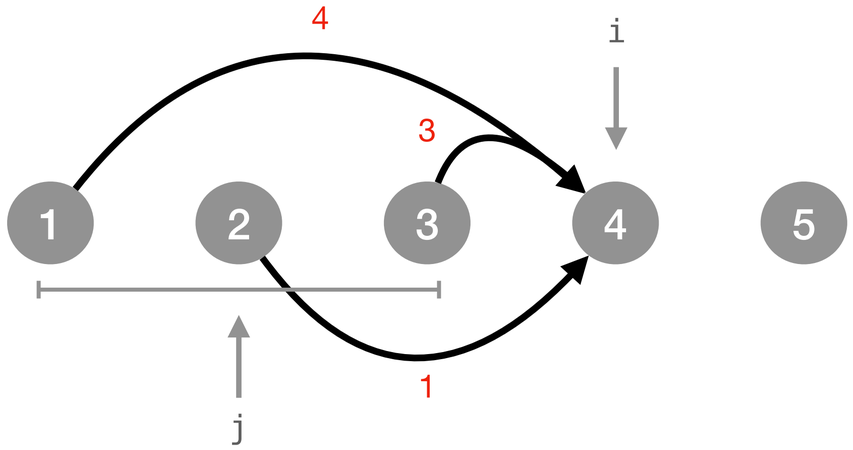
```
21  uint64_t C[N + 1];  // C[i] := cost of shortest path from node 1 to node i
22
23  uint64_t shortest_path_rec_memoized(const uint64_t n) {
24      if (C[n] != INF) return C[n];
25      uint64_t best = INF;
26      for (uint64_t i = 1; i != n; ++i) {
27          uint64_t cost = shortest_path_rec_memoized(i) + W[i][n];
28          if (cost < best) best = cost;
29      }
30      C[n] = best;
31      return best;
32  }
```

- **Q.** Complexity?
- **A.** We traverse each edge once, hence the solution runs in $\Theta(n^2)$.

4. Bonus: iterative solution

- Fill the array `C` in order: compute `C[i]` looking at the previously computed `C[j]` for `j < i`.

```
34  uint64_t shortest_path_itr_memoized(const uint64_t n) {
35      for (uint64_t i = 1; i <= n; ++i) {
36          for (uint64_t j = 1; j < i; ++j) {
37              C[i] = std::min(C[i], C[j] + W[j][i]);
38          }
39      }
40      return C[n];
41  }
```



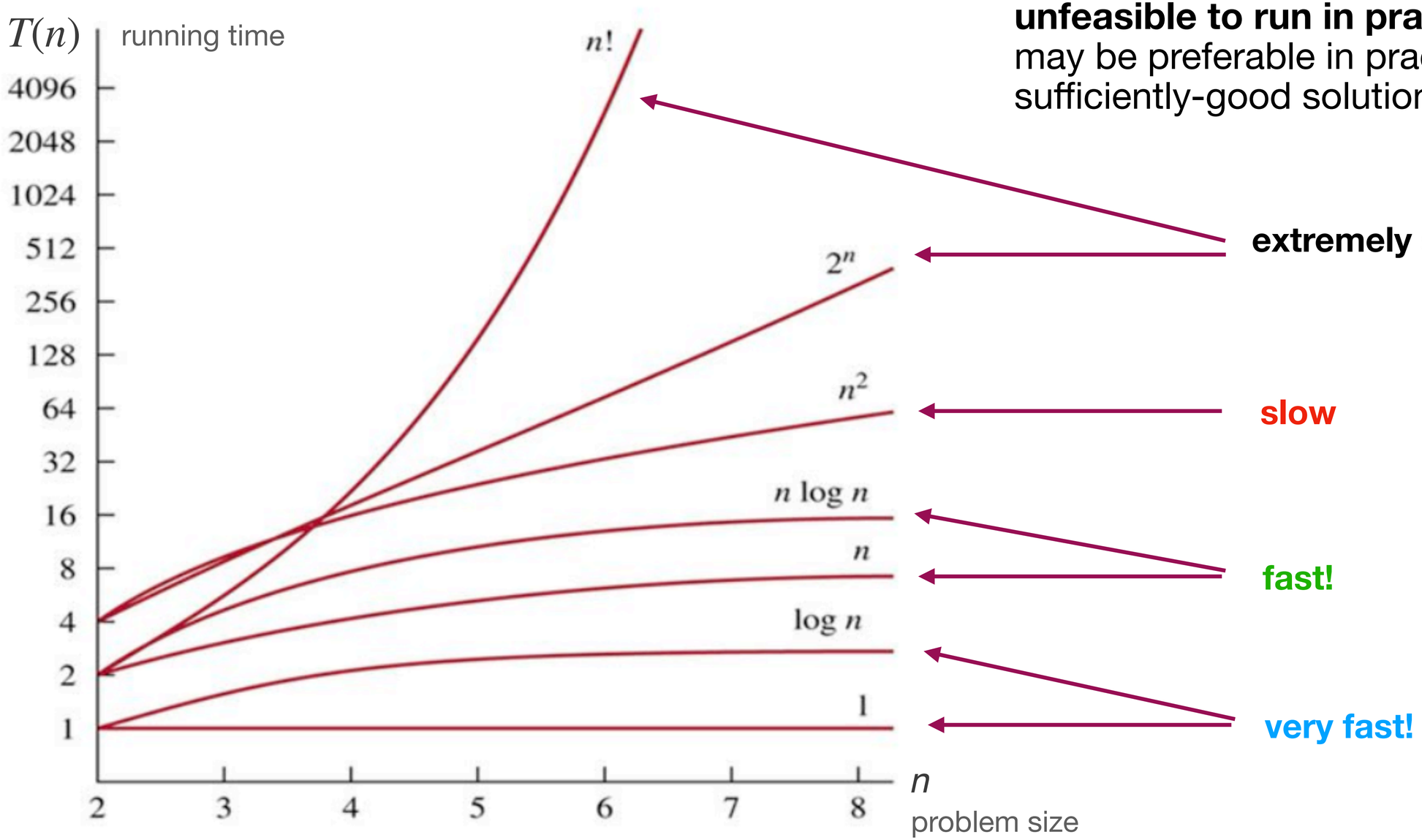
Recap

Problem	Naive	DP
Fibonacci	$O(2^n)$	$\Theta(n)$
Coins	$O(2^n)$	$\Theta(kn)$
Rod cutting	$\Theta(2^n)$	$\Theta(n^2)$
Shortest path in a DAG	$\Theta(2^n)$	$\Theta(n^2)$

- In all these examples, DP reduces the complexity of the naive algorithm **from exponential to polynomial**.

Growth of running time

- **A quadratic solution might still be unfeasible to run in practice!** Heuristics may be preferable in practice if a sufficiently-good solution is enough.



What we have done so far

- In our examples so far, we have followed a common pattern to arrive a DP solution:

  1. Understand the **form of an optimal solution**;

  2. Write the form of an optimal solution **recursively** in terms of optimal solutions to smaller problems;

  3. Solve the recurrence, usually in a top-down fashion (iterative, bottom up, can be even faster in practice).

- It turns out there are **specific properties** a problem must have for DP to apply.

So when is DP applicable?

- Two key properties a problem must have to be solvable by a DP solution:

So when is DP applicable?

- Two key properties a problem must have to be solvable by a DP solution:

    1. **Optimal sub-structure.** We can express the form of an optimal solution as composition of the optimal solutions to sub-problems.

So when is DP applicable?

- Two key properties a problem must have to be solvable by a DP solution:

    1. **Optimal sub-structure.** We can express the form of an optimal solution as composition of the optimal solutions to sub-problems.

    2. **Sub-problems overlap.** The problem generates the same, “small”, set of different sub-problems rather than always generating new ones.

So when is DP applicable?

- Two key properties a problem must have to be solvable by a DP solution:

    1. **Optimal sub-structure.** We can express the form of an optimal solution as composition of the optimal solutions to sub-problems.

    2. **Sub-problems overlap.** The problem generates the same, “small”, set of different sub-problems rather than always generating new ones.

- If a problem enjoys these two properties, we have a good clue that DP might apply.
- If so, we can express the solution recursively (see previous slides) and the implementation is relatively easy (but mind the base case of recursion).
- **The difficult part is to get the problem characterisation right!**

So when is DP applicable?

- Note that **both properties** are necessary:

    1. If the **optimal sub-structure** property does not hold, then we cannot describe an optimal solution recursively in terms of sub-problems.

    2. If **sub-problems do not overlap** but instead we always generate “new” problems, then it does not make sense to store their solutions in an array since they will not be used.

Optimal sub-structure

- To show that solutions to sub-problems within an optimal solution to the problem must themselves be optimal, we can use a ***cut-and-paste*** technique.

- Essentially, a **proof by contradiction**: we assume the negated statement is true but arrive at a logical contradiction.

- **Statement.** Suppose the cost of any solution S decomposes as

$$cost(S) = c_0 + cost(S_1) + \cdots + cost(S_k),$$

  where where $S_1, \ldots, S_k$ are its sub-solutions and c_0 is the (fixed) cost of combining them. If a solution S^* is optimal, then the solutions to the sub-problems, $S_1^*, \ldots, S_k^*$, must all be optimal.

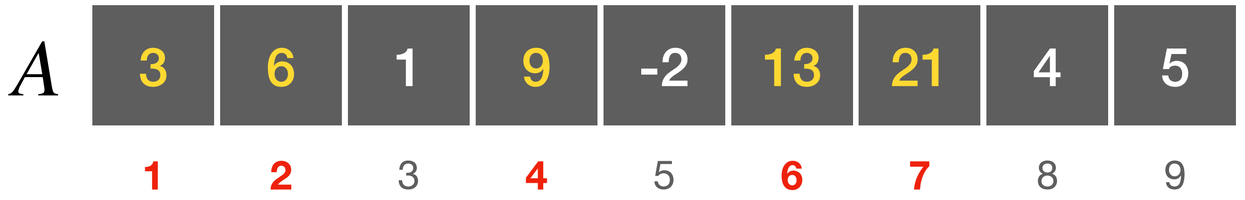
- *Proof sketch*. By contradiction. Assume the negated statement is true: S^* is optimal but there is at least one S_i^* that is **not** optimal. So there must exist a solution S_i of smaller cost, i.e., $cost(S_i) < cost(S_i^*)$. Then "cut" S_i^* and **replace** it with S_i: now we have that $cost(S) < cost(S^*)$, which **contradicts** the assumption that S^* is optimal. We conclude that all $S_1^*, \ldots, S_k^*$ must therefore be optimal too. ■

Longest increasing subsequence (LIS)

- Given an array $A[1..n]$, an increasing subsequence of A is any list of indexes $i_1, \ldots, i_k$ such that $i_1 < i_2 < \cdots < i_k$ and $A[i_1] < A[i_2] < \cdots < A[i_k]$, for some $k \geq 1$.
- **Problem.** Determine the length of the longest increasing subsequence (LIS) of A.
- Without loss of generality, assume A is an integer array.

Longest increasing subsequence (LIS)

- Given an array $A[1..n]$, an increasing subsequence of A is any list of indexes $i_1, \ldots, i_k$ such that $i_1 < i_2 < \cdots < i_k$ and $A[i_1] < A[i_2] < \cdots < A[i_k]$, for some $k \geq 1$.
- **Problem.** Determine the length of the longest increasing subsequence (LIS) of A.
- Without loss of generality, assume A is an integer array.
- Example.

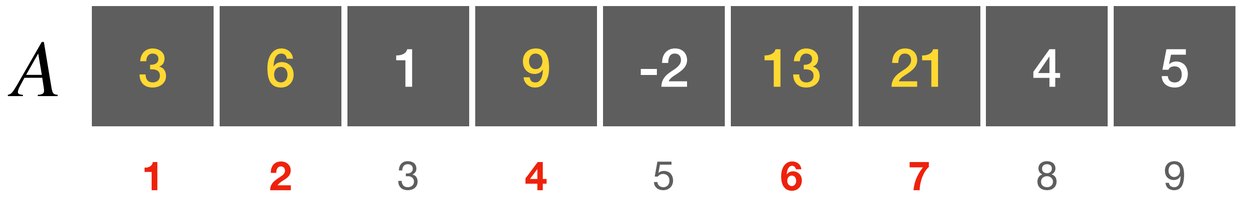


Longest increasing subsequence (LIS)

- Let's write the form of the optimal solution recursively.
- Let $len(i)$ be the length of the longest increasing subsequence **ending** at position i, for $1 \leq i \leq n$.

Longest increasing subsequence (LIS)

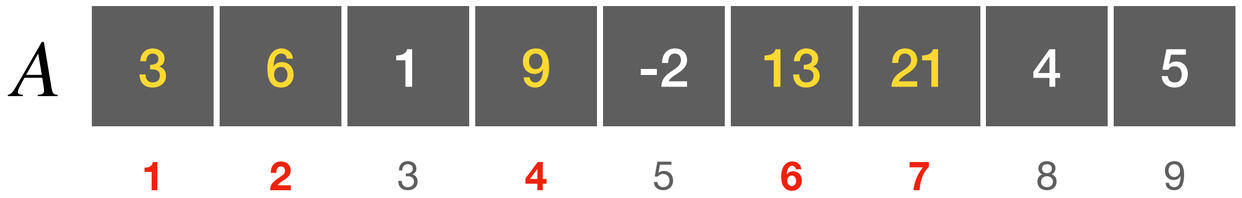
- Let's write the form of the optimal solution recursively.
- Let $len(i)$ be the length of the longest increasing subsequence **ending** at position i, for $1 \leq i \leq n$.
- Example: $len(1) = 1, len(2) = 2, len(3) = 1, len(4) = 3, len(5) = 1,$
  $len(6) = 4, len(7) = 5, len(8) = 2, len(9) = 3.$



- **Note**: $len(i)$ is the length of an IS that includes $A[i]$, **not** the LIS of the prefix $A[1..i]$.

Longest increasing subsequence (LIS)

- Let's write the form of the optimal solution recursively.
- Let $len(i)$ be the length of the longest increasing subsequence **ending** at position i, for $1 \leq i \leq n$.
- Example: $len(1) = 1, len(2) = 2, len(3) = 1, len(4) = 3, len(5) = 1,$
  $len(6) = 4, len(7) = 5, len(8) = 2, len(9) = 3.$



- **Note**: $len(i)$ is the length of an IS that includes $A[i]$, **not** the LIS of the prefix $A[1..i]$.
- The solution is therefore $\max_{1 \leq i \leq n} \{len(i)\}$.

Longest increasing subsequence (LIS)

$S(3) = \emptyset$

$S(4) = \{1,2,3\}$

$S(8) = \{1,3,5\}$

Longest increasing subsequence (LIS)

- We need to express $len(i)$ in terms of $len(j)$ for some $j < i$.
- Let $S(i) := \{1 \leq j < i \,|\, A[i] > A[j]\}$.
- Example.

$S(3) = \emptyset$

$S(4) = \{1,2,3\}$

$S(8) = \{1,3,5\}$

Longest increasing subsequence (LIS)

- We need to express $len(i)$ in terms of $len(j)$ for some $j < i$.
- Let $S(i) := \{1 \leq j < i \,|\, A[i] > A[j]\}$.
- Example.

$S(3) = \emptyset$

$S(4) = \{1,2,3\}$

$S(8) = \{1,3,5\}$

Longest increasing subsequence (LIS)

- We need to express $len(i)$ in terms of $len(j)$ for some $j < i$.
- Let $S(i) := \{1 \leq j < i \,|\, A[i] > A[j]\}$.
- Example.

$S(3) = \emptyset$

$S(4) = \{1,2,3\}$

$S(8) = \{1,3,5\}$

- We can write $len(i)$ as

$$len(i) = \begin{cases} \max_{j \in S(i)} \{len(j) + 1\} & S(i) \neq \emptyset \\ 1 & \text{otherwise} \end{cases}.$$

LIS — Recursive solution

```
16  uint64_t len(uint64_t i) {
17      uint64_t best = 1;
18      for (uint64_t j = 0; j < i; ++j) {
19          uint64_t l = len(j);
20          if (A[i] > A[j] and l + 1 > best) {
21              best = l + 1;
22          }
23      }
24      return best;
25  }
```

```
53      uint64_t best = 0;
54      for (uint64_t i = 0; i != n; ++i) {
55          uint64_t l = len(i);
56          std::cout << l << ' ';
57          if (l > best) best = l;
58      }
```

- Note that the base case for `i=0` is handled correctly (even without an explicit test) because we always return 1.

- The complexity is $\Theta(2^n)$ because there are 2^n nodes in the recursion tree (as for the rod cutting problem).

LIS — Recursive solution + memoization

```
27  uint64_t L[n];
28
29  uint64_t len_rec_memoized(uint64_t i) {
30      if (L[i] > 0) return L[i];
31      uint64_t best = 1;
32      for (uint64_t j = 0; j < i; ++j) {
33          uint64_t l = len_rec_memoized(j);
34          if (A[i] > A[j] and l + 1 > best) best = l + 1;
35      }
36      L[i] = best;
37      return best;
38  }
```

```
71      L[0] = 1;
72      for (uint64_t i = 1; i != n; ++i) L[i] = 0;
73      len_rec_memoized(n - 1);
74      uint64_t best = *std::max_element(std::begin(L), std::end(L));
```

- Now we prune sub-trees in the recursion tree and the complexity is quadratic, $\Theta(n^2)$.

LIS — Iterative solution + memoization

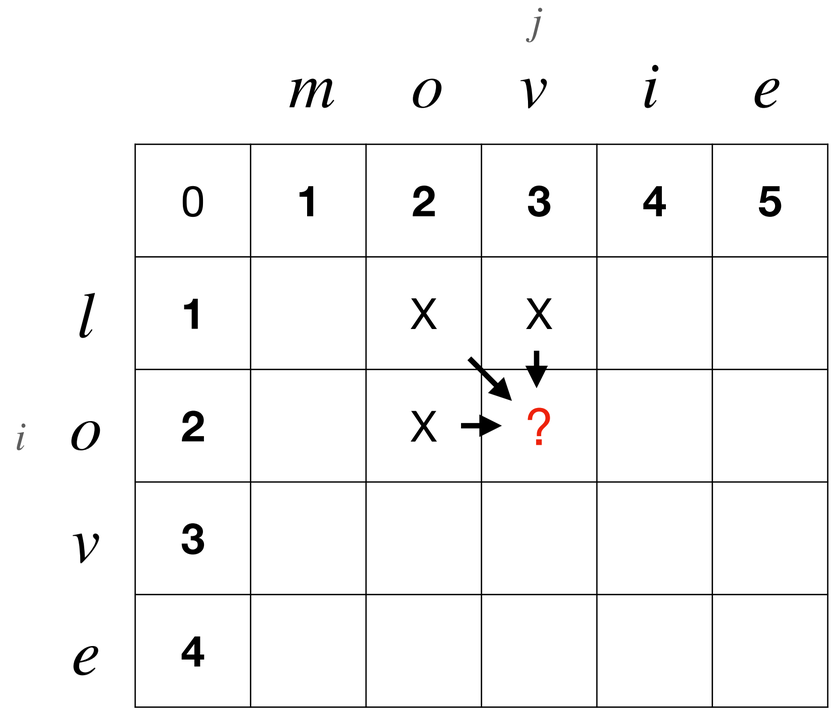
```
40  uint64_t len_itr_memoized(const uint64_t n) {
41      uint64_t max = 0;
42      for (uint64_t i = 0; i != n; ++i) {
43          uint64_t best = 1;
44          for (uint64_t j = 0; j < i; ++j) {
45              uint64_t l = L[j];
46              if (A[i] > A[j] and l + 1 > best) best = l + 1;
47          }
48          L[i] = best;
49          max = std::max(max, best);
50      }
51      return max;
52  }
```

Edit distance

- The **edit distance** (or *Levenshtein* distance) between two strings $P[1..n]$ and $Q[1..m]$ is the minimum number of edit operations needed to transform P into Q.
- We consider the following edit operations, for any position in a string:
  - **insert** a character;
  - **delete** a character;
  - **substitute** a character.
- Example with $P = love$ and $Q = movie$. The edit distance is 2 because we substitute l with m (at position 1) and insert i (at position 4).
- **Problem.** Given P and Q, calculate their edit distance.

Edit distance

- Let $ed(i, j)$ be the edit distance between $P[1..i]$ and $Q[1..j]$.
- Our goal is to compute $ed(n, m)$. To do so with DP, we have to write $ed(i, j)$ in terms of a recurrence relation. Let's consider the example for $P = love$ and $Q = movie$.
- We want to compute the value of $ed(i, j)$, knowing the **past** choices:
  - $ed(i-1, j)$
  - $ed(i, j-1)$
  - $ed(i-1, j-1)$



Edit distance — Cases

- The cost $ed(i,j)$ must therefore be the minimum between the following costs:

$$\begin{array}{l} Q[1..j] = mov \\ P[1..i] = lov \end{array} \quad \begin{array}{c} j-1 \\ \downarrow \\ \\ i \end{array}$$

Cost is $ed(i,j-1)+1$ (**insert** a character at the end of P)

$$\begin{array}{l} Q[1..j] = mov \\ P[1..i] = lo \end{array} \quad \begin{array}{c} j \\ \\ \uparrow \\ i-1 \end{array}$$

Cost is $ed(i-1,j)+1$ (**remove** the last character of P)

$$\begin{array}{l} Q[1..j] = mov \\ P[1..i] = lov \end{array} \quad \begin{array}{c} j-1 \\ \downarrow \\ \\ i-1 \end{array}$$

Cost is $ed(i-1,j-1)+cost(i,j)$, where $cost(i,j)$ is 0 if $P[i] = Q[j]$ or 1 otherwise (**substitution**).

Edit distance

- Filling the table is clearly done in $\Theta(nm)$ time (and space).

```
16      ed[0][0] = 0;
17      for (uint64_t i = 1; i <= n; ++i) ed[i][0] = i;
18      for (uint64_t j = 1; j <= m; ++j) ed[0][j] = j;
19
20      for (uint64_t i = 1; i <= n; ++i) {
21          for (uint64_t j = 1; j <= m; ++j) {
22              ed[i][j] = std::min({
23                  ed[i - 1][j] + 1,
24                  ed[i][j - 1] + 1,
25                  ed[i - 1][j - 1] + (P[i - 1] != Q[j - 1] ? 1 : 0)
26              });
27          }
28      }
```

i-th and j-th caracters of P and Q
(indexes in strings start from 0)

		m	o	v	i	e
	0	**1**	**2**	**3**	**4**	**5**
l	**1**	1	2	3	4	5
o	**2**	2	1	2	3	4
v	**3**	3	2	1	2	3
e	**4**	4	3	2	2	2

References

- *Competitive Programmer's Handbook* (Chapter 7) - A. Laaksonen, 2018
  https://cses.fi/book/book.pdf

- *Introduction to Algorithms* (Third ed., Chapter 15) - T. H. Cormen at al., 2009