Dynamic Programming

Recursion with memory

Recursion is not always Fast

Problems so far:
- Sorting
- Multiplication

Recursion is not always Fast

Problems so far:

- Sorting
- Multiplication

Property:
Same work is not (necessarily) repeated.

Recursion is not always Fast

Problems so far:
- Sorting
- Multiplication

Property:
Same work is not (necessarily) repeated.

Often in recursion:
The very same thing is computed over and over again

Recursion is not always Fast

Problems so far:
- Sorting
- Multiplication

Property:
Same work is not (necessarily) repeated.

Often in recursion:
The very same thing is computed over and over again

Dynamic Programming:
Store the results of earlier computations in a look-up table. No need to compute the same thing twice.

Recursion is not always Fast

Dynamic Programming:
Store the results of earlier computations in a look-up table. No need to compute the same thing twice.

Learning objectives:
You are able to

- apply dynamic programming to compute Fibonacci numbers.
- describe the advantage of using memoization
- describe the computation of Fibonacci numbers as a DAG

Fibonacci

Fibonacci numbers:

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

Fibonacci

Fibonacci numbers:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

Task:
Calculate F_n.

Fibonacci

Fibonacci numbers:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

Recursive algorithm:

FIBO(n)
If(n == 0 or n == 1)
  return n

Else
  return FIBO($n-1$) + FIBO($n-2$)

Fibonacci

Recursive algorithm:
FIBO(n)
If($n == 0$ or $n == 1$)
  return n

Else
  return FIBO$(n-1)$ + FIBO$(n-2)$

Runtime recurrence:
$T(n)$
$= T(n-1) + T(n-2) + O(\log n)$

Fibonacci

Recursive algorithm:
FIBO(n)
If($n == 0$ or $n == 1$)
  return n

Else
  return FIBO($n-1$) + FIBO($n-2$)

Runtime recurrence:
$T(n)$
$= T(n-1) + T(n-2) + O(\log n)$

Addition of two
$O(\log n)$ digit numbers

Fibonacci

Runtime recurrence:

$$T(n) = T(n-1) + T(n-2) + O(\log n)$$
$$\geq 2 \cdot T(n-2)$$

```
FIBO(n)
If(n == 0 or n == 1)
  return n

Else
  return FIBO(n − 1) + FIBO(n − 2)
```

Fibonacci

Runtime recurrence:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + O(\log n) \\
&\geq 2 \cdot T(n-2)
\end{aligned}
$$

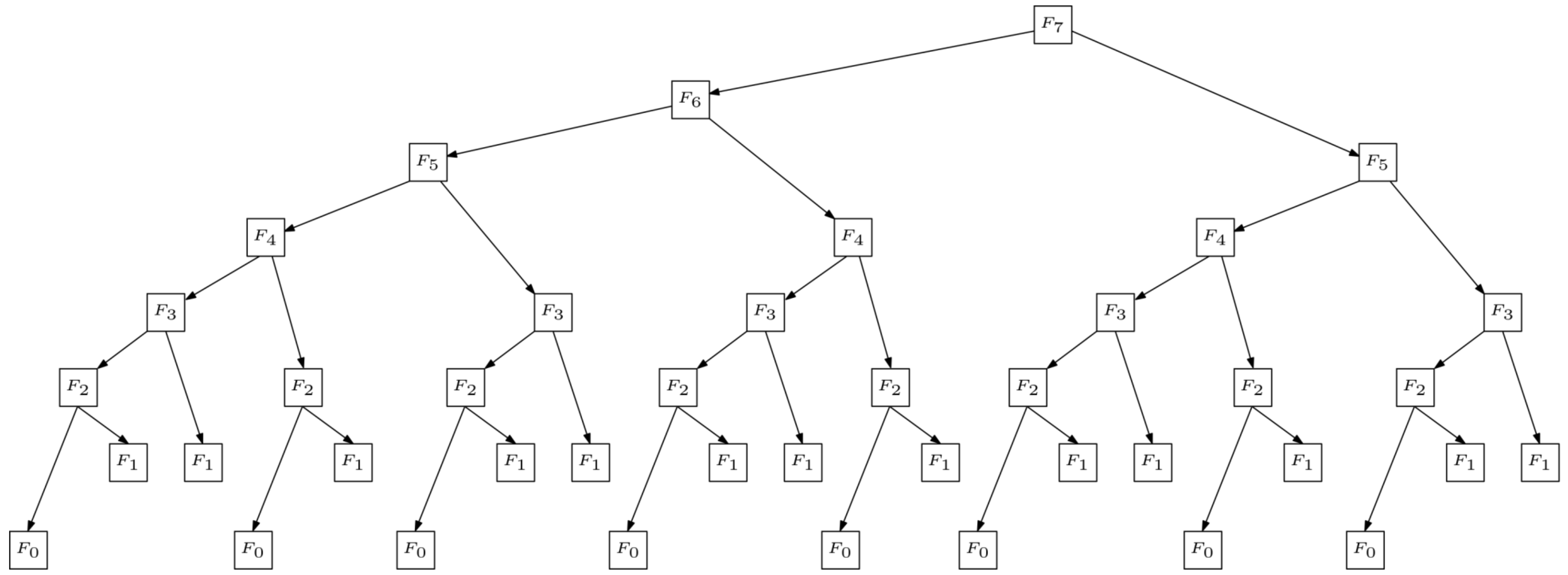
For $n \geq 2$,

$$T(n) \geq 2^{n/2}$$

😢

Fibonacci – Wasting Time

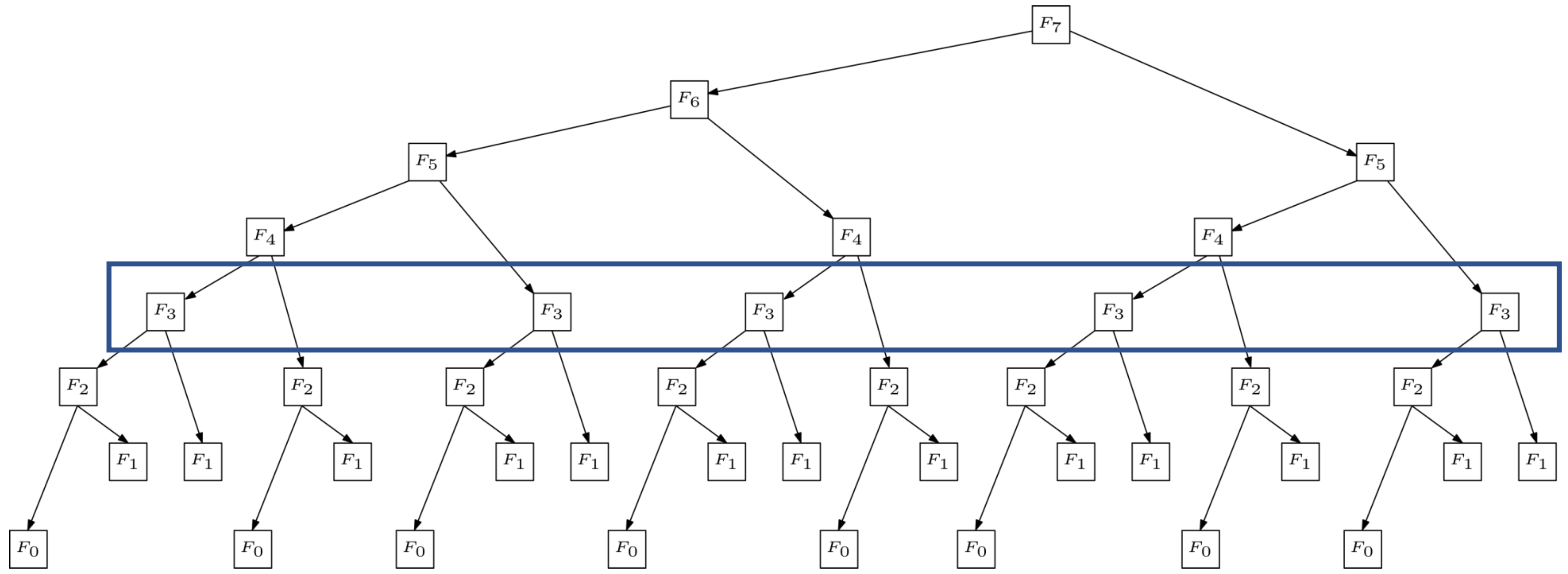
Runtime recurrence:
$T(n) = T(n-1) + T(n-2) + O(\log n)$



We are performing the same calculations over and over again

Fibonacci – Wasting Time

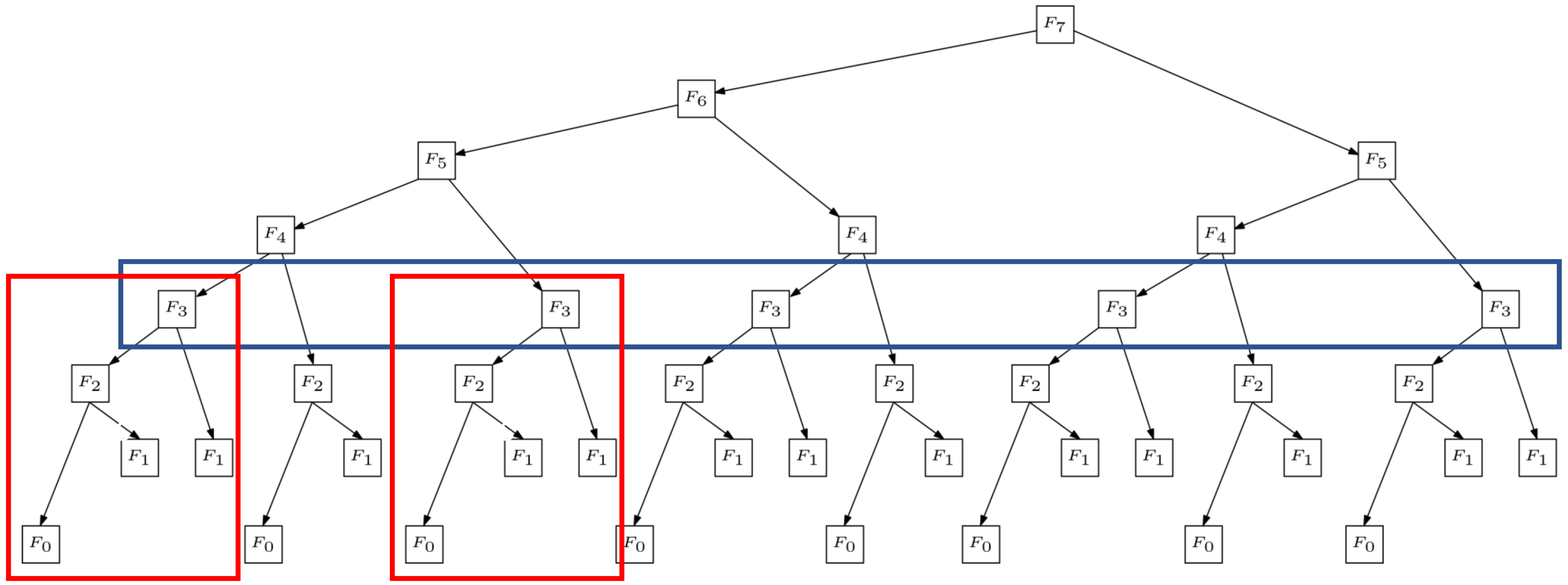
Runtime recurrence:
$T(n) = T(n-1) + T(n-2) + O(\log n)$



We are performing the same calculations over and over again

Fibonacci – Wasting Time

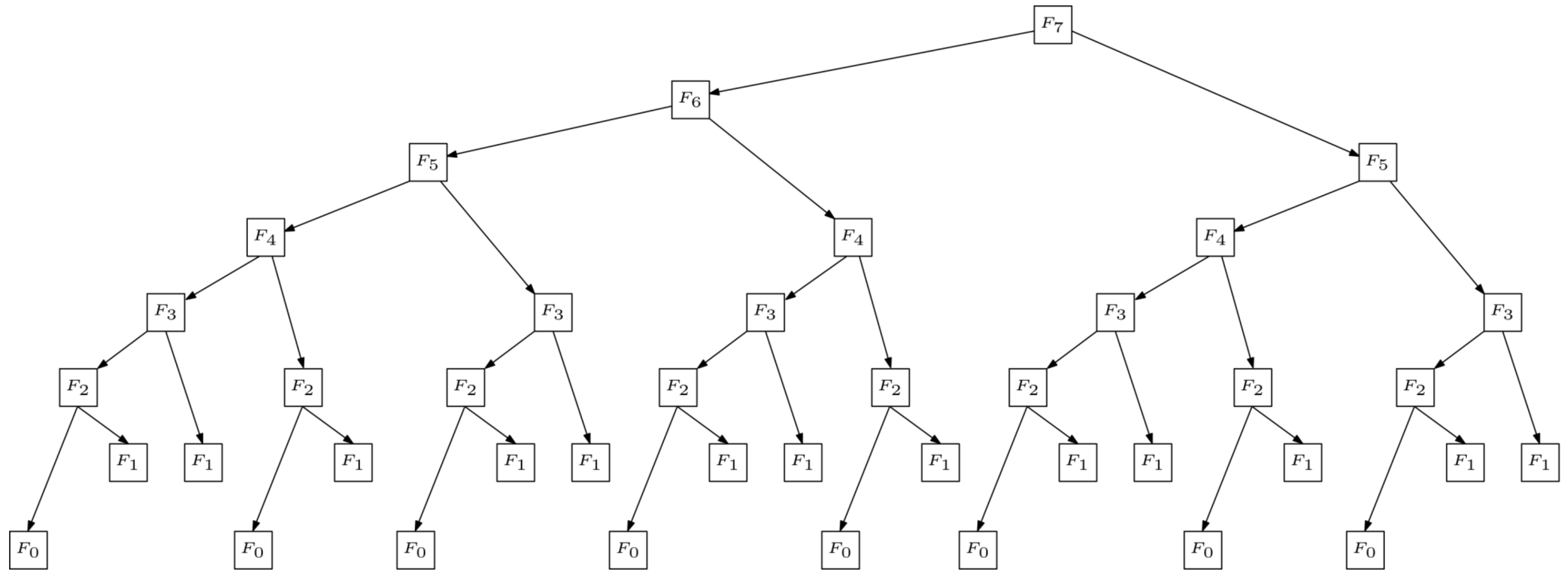
Runtime recurrence:
$T(n) = T(n-1) + T(n-2) + O(\log n)$



We are performing the same calculations over and over again.

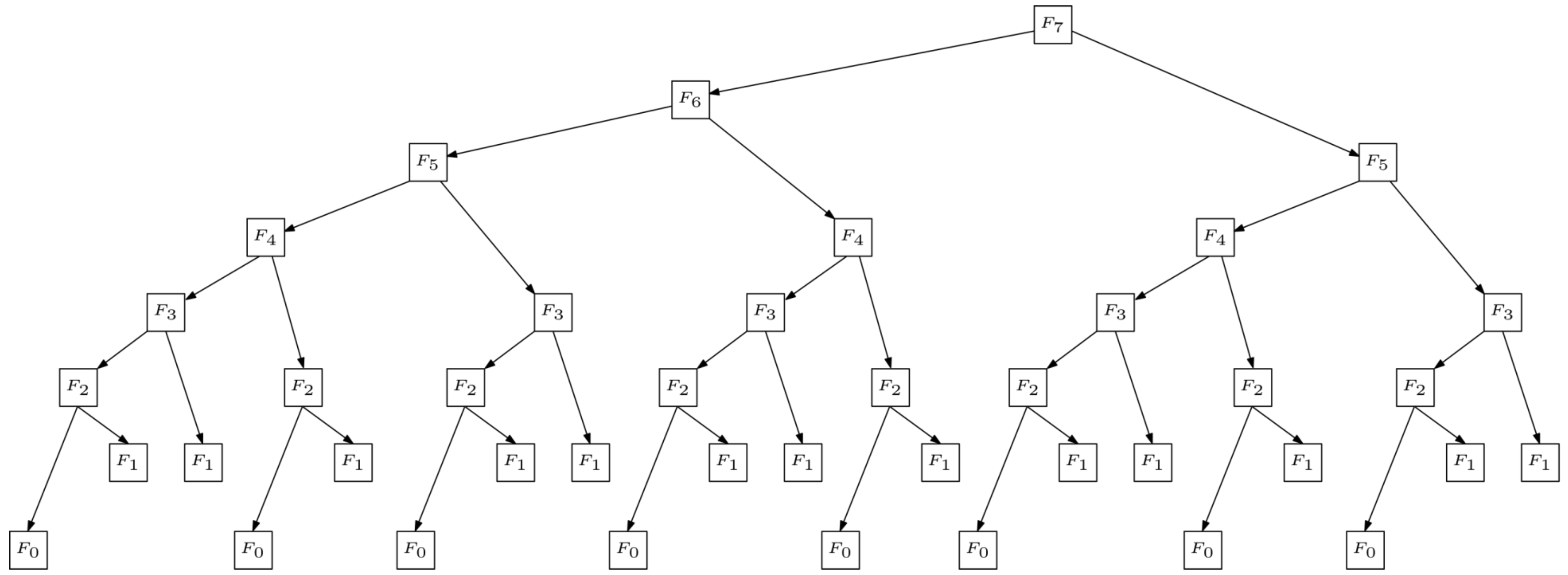
Fibonacci – Wasting Time

Dynamic Programming:
Write down the intermediate solutions.



Fibonacci – Wasting Time

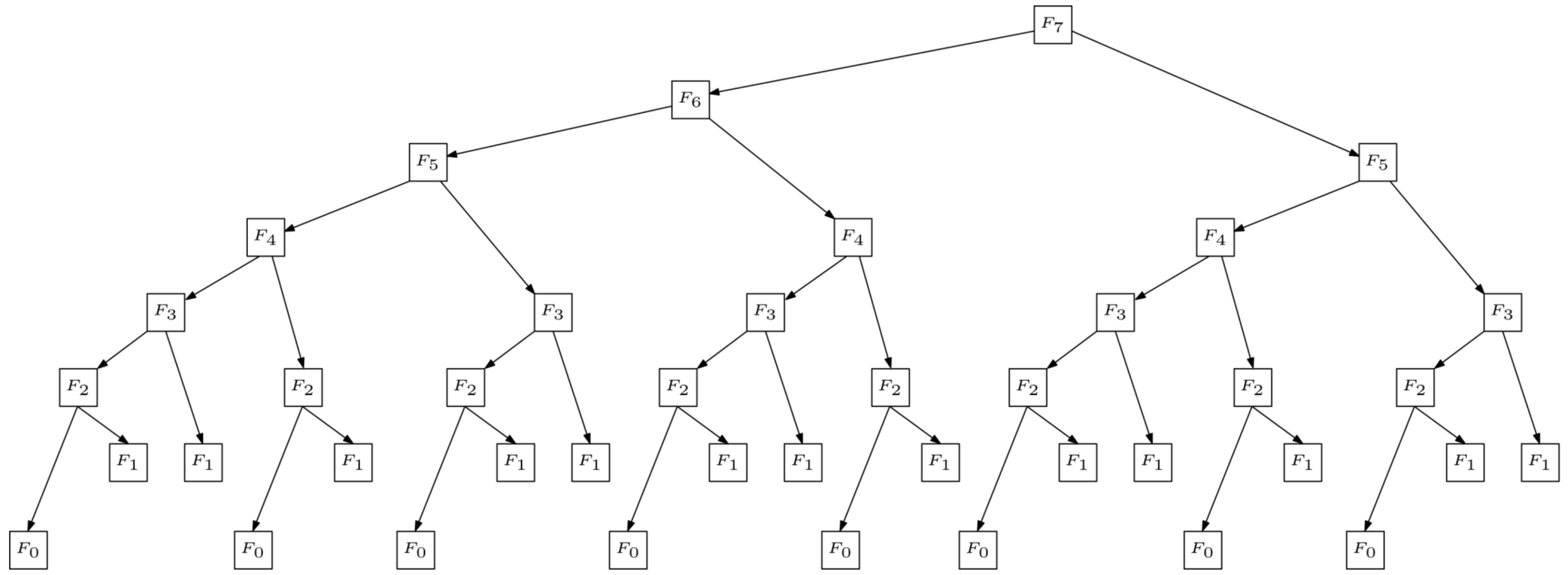
Dynamic Programming:
Write down the intermediate solutions.



Array $F[n]$ of integers for intermediate solutions.

Fibonacci – Wasting Time

Dynamic Programming:
Write down the intermediate solutions.

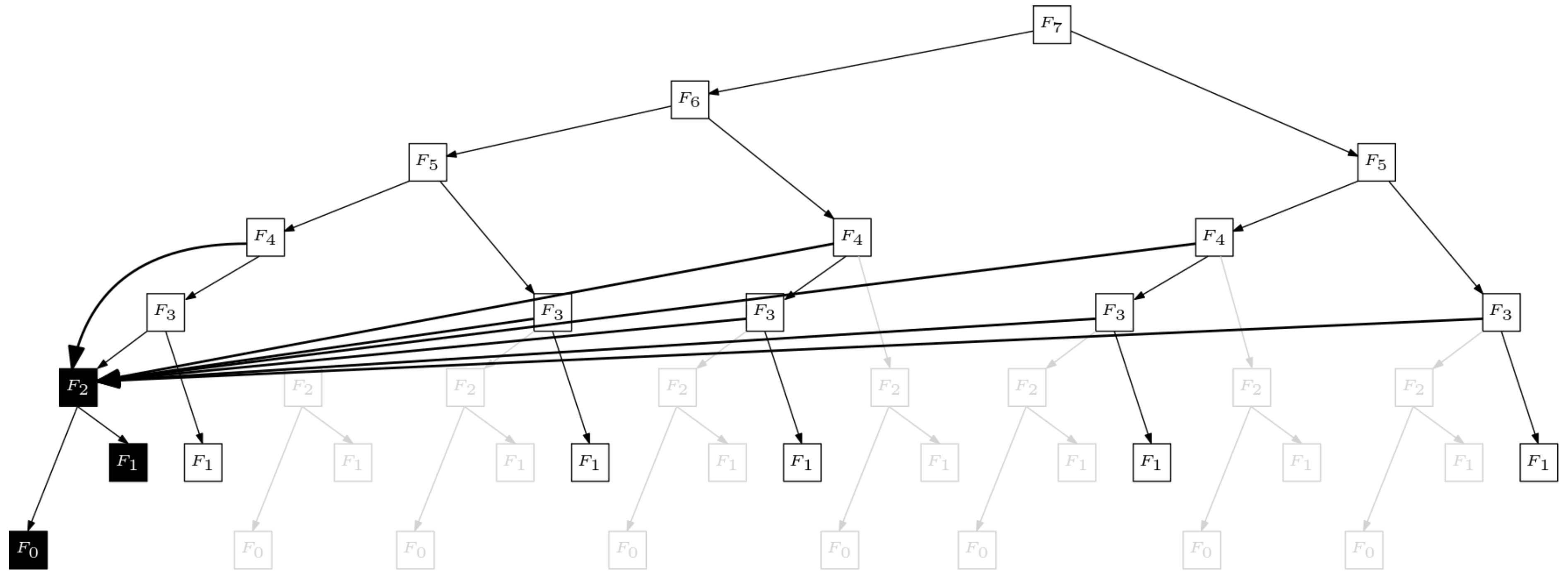


Array $F[n]$ of integers for intermediate solutions.

Fibo(n):
If($n == 0$ or $n == 1$)
  return n
If($F[n]$ undefined)
  $F[n] := \text{Fibo}(n-1) + \text{Fibo}(n-2)$
Return $F[n]$

Fibonacci – Wasting Time

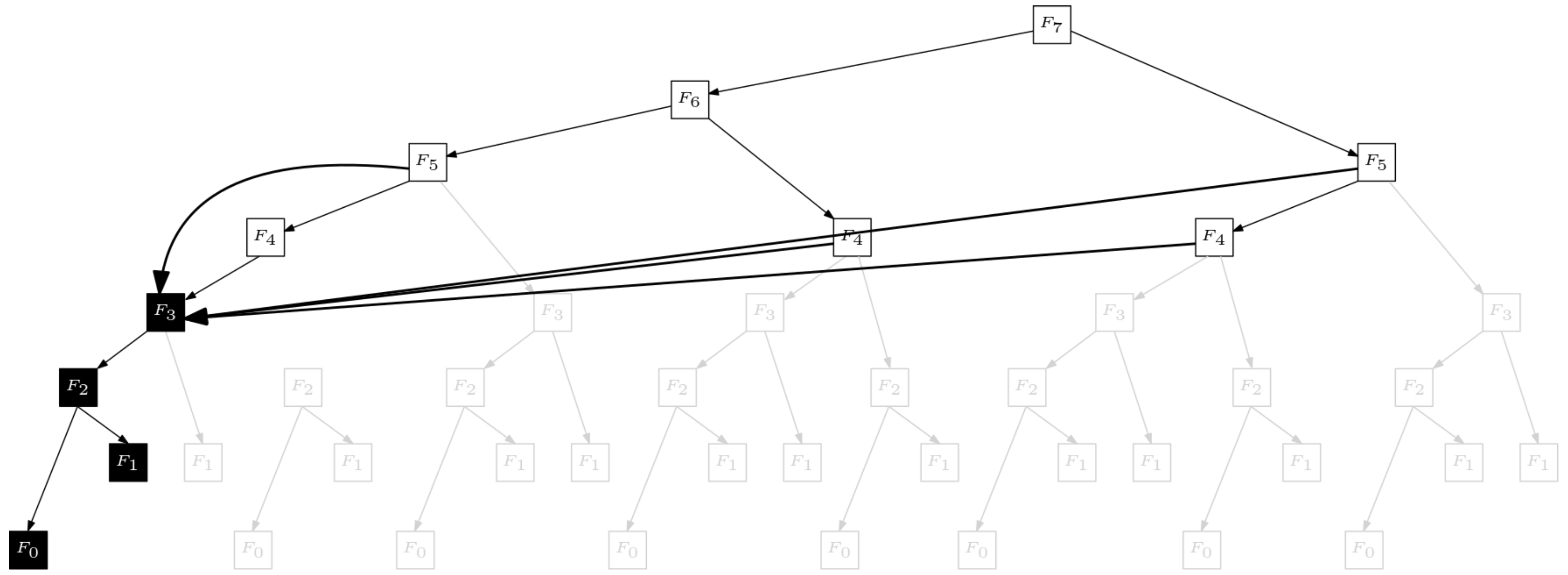
Dynamic Programming:
Write down the intermediate solutions.



Memoization:
Once we know F_2, no need to evaluate it again. The grayed-out parts are never computed

Fibonacci – Wasting Time

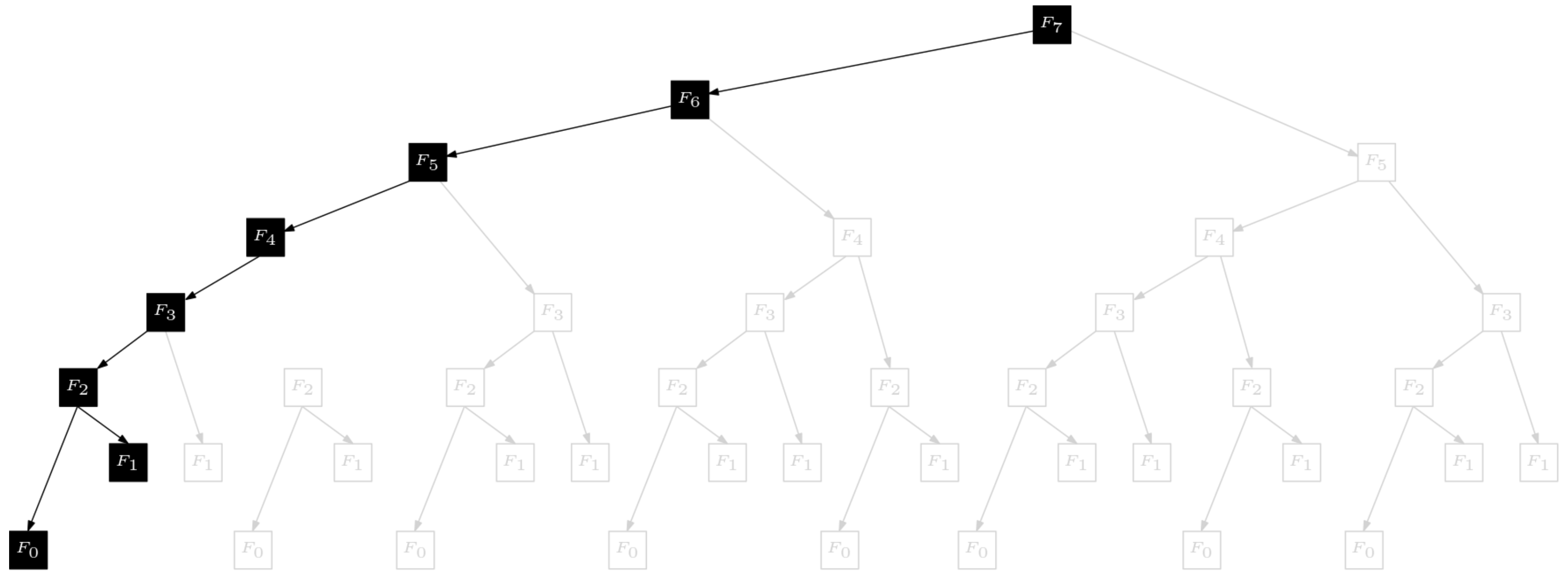
Dynamic Programming:
Write down the intermediate solutions.



Same for F_3.

Fibonacci – Wasting Time

Dynamic Programming:
Write down the intermediate solutions.



Only the black nodes are evaluated. Other values are just fetched from memory

Computation as a Directed Acyclic Graph

Think of the computation as a DAG.

It is always possible to evaluate a sink using one addition.

Runtime: $O(n)$ additions

Computation as a Directed Acyclic Graph

Think of the computation as a DAG.

It is always possible to evaluate a sink using one addition.

Runtime: $O(n)$ additions
Caveat: Integer sizes are large

Dynamic Programming

In a recursion tree, every branch is evaluated

Dynamic Programming:
Smart recursion

Represent the recursion tree as a DAG

Works beyond Fibonacci numbers.