CS11313 - **Spring** 2022

Design & Analysis of Algorithms

Dynamic Programming

Ibrahim Albluwi

Motivation

Fibonacci Numbers. 0, 1, 1, 2, 3, 5, 8, 13, 21, $\underset{F_9}{34}$, $\underset{F_{10}}{55}$, $\underset{F_{11}}{89}$, …

$$F_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 1 & \text{if} \quad n = 1 \\ F_{n-1} + F_{n-2} & \text{if} \quad n > 1 \end{cases}$$

Leonardo Fibonacci
1170—1240

Motivation

Fibonacci Numbers. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

F_9 F_{10} F_{11}

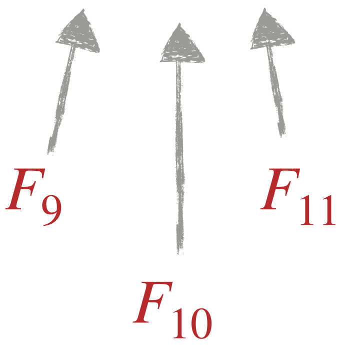
$$F_n = \begin{cases} 0 & \text{if } \ n = 0 \\ 1 & \text{if } \ n = 1 \\ F_{n-1} + F_{n-2} & \text{if } \ n > 1 \end{cases}$$

Goal. Given i compute F_i .

Leonardo Fibonacci
1170—1240

Motivation

Fibonacci Numbers. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …



F_9

F_{10}

F_{11}

$$F_n = \begin{cases} 0 & \text{if } \ n = 0 \\ 1 & \text{if } \ n = 1 \\ F_{n-1} + F_{n-2} & \text{if } \ n > 1 \end{cases}$$

Goal. Given i compute F_i .

A very simple recursive implementation:

```
FIB(n)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Leonardo Fibonacci
1170—1240

Motivation

How long does it take to compute **FIB**(100)?

A. A few seconds.

B. A few minutes.

C. A few hours.

D. A few days.

E. Armageddon!

Motivation

How long does it take to compute **`FIB`**`(100)`?

- **A.** A few seconds.
- **B.** A few minutes.
- **C.** A few hours.
- **D.** A few days.
- **E.** Armageddon!

If it takes `0.66` seconds to compute **`FIB`**`(40)`,
it takes ~`72242` years to compute **`FIB`**`(100)`.

The computer will definitely crash much sooner than that!

Motivation

```
FIB(n)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

Motivation

FIB(i)

```
if (n == 0): return 0
if (n == 1): return 1

return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

Recursion Tree

$T(n)$

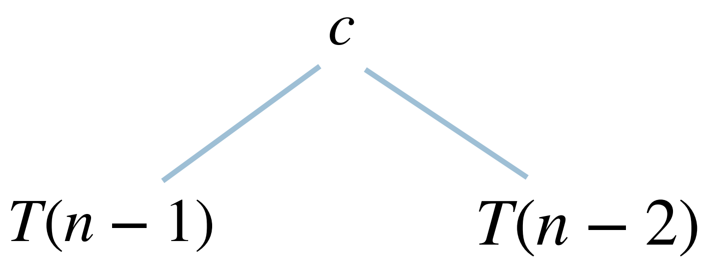
Motivation

```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

Recursion Tree



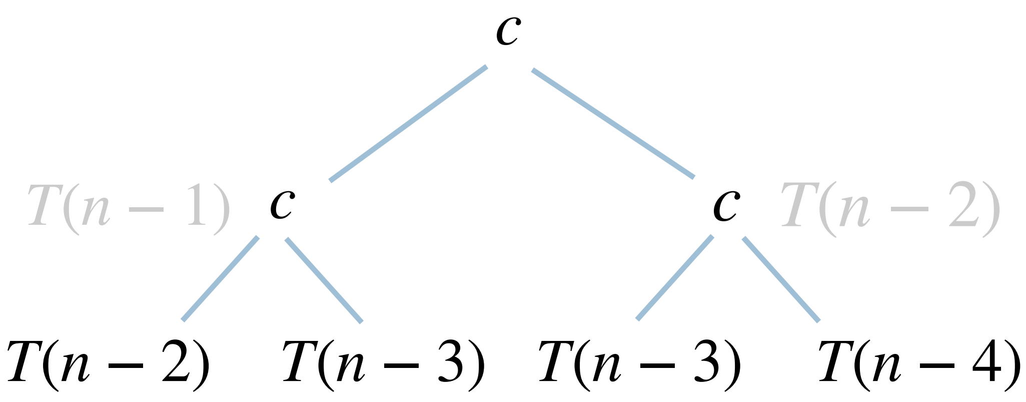
Motivation

```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

Recursion Tree

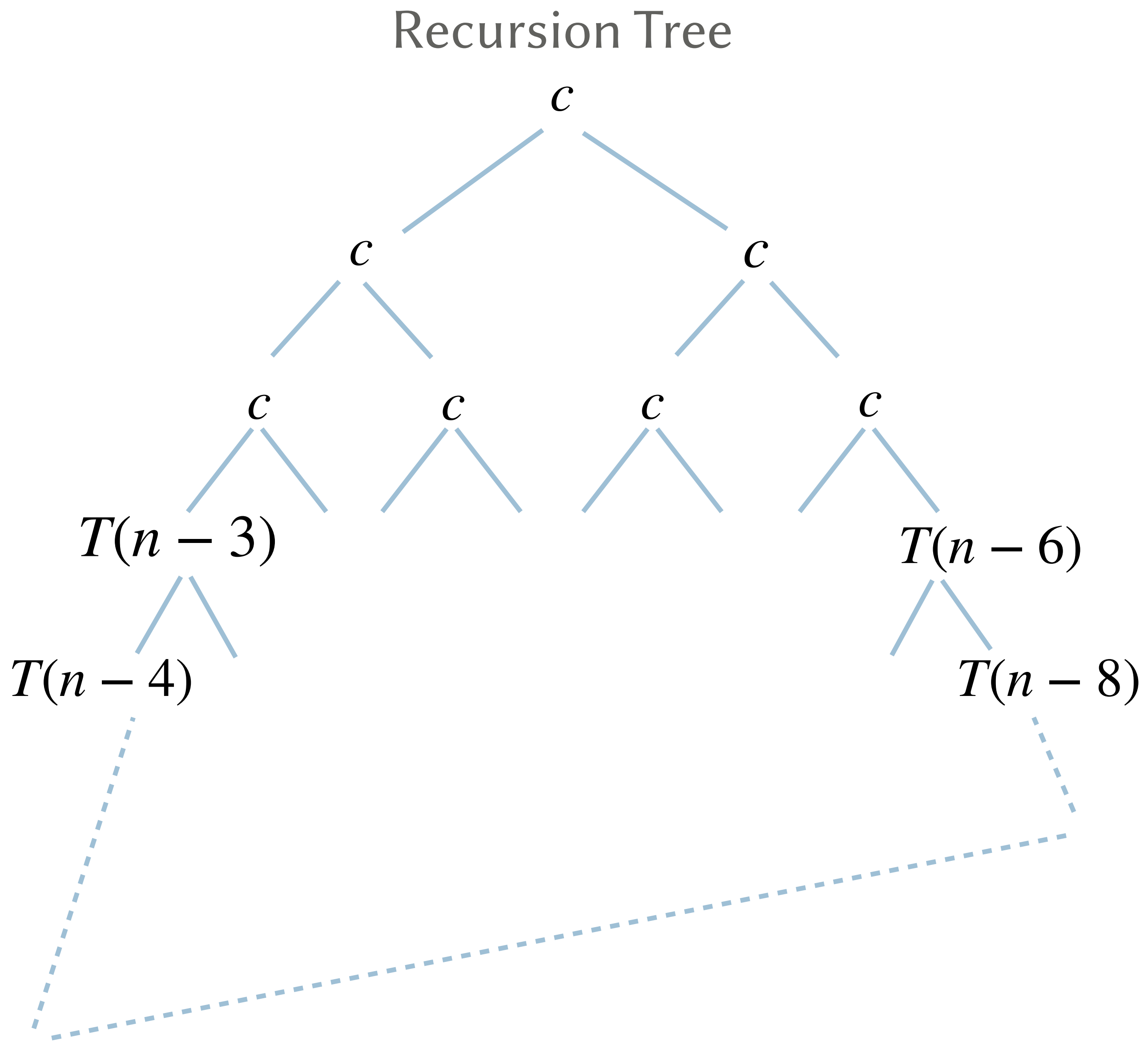


Motivation

```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$



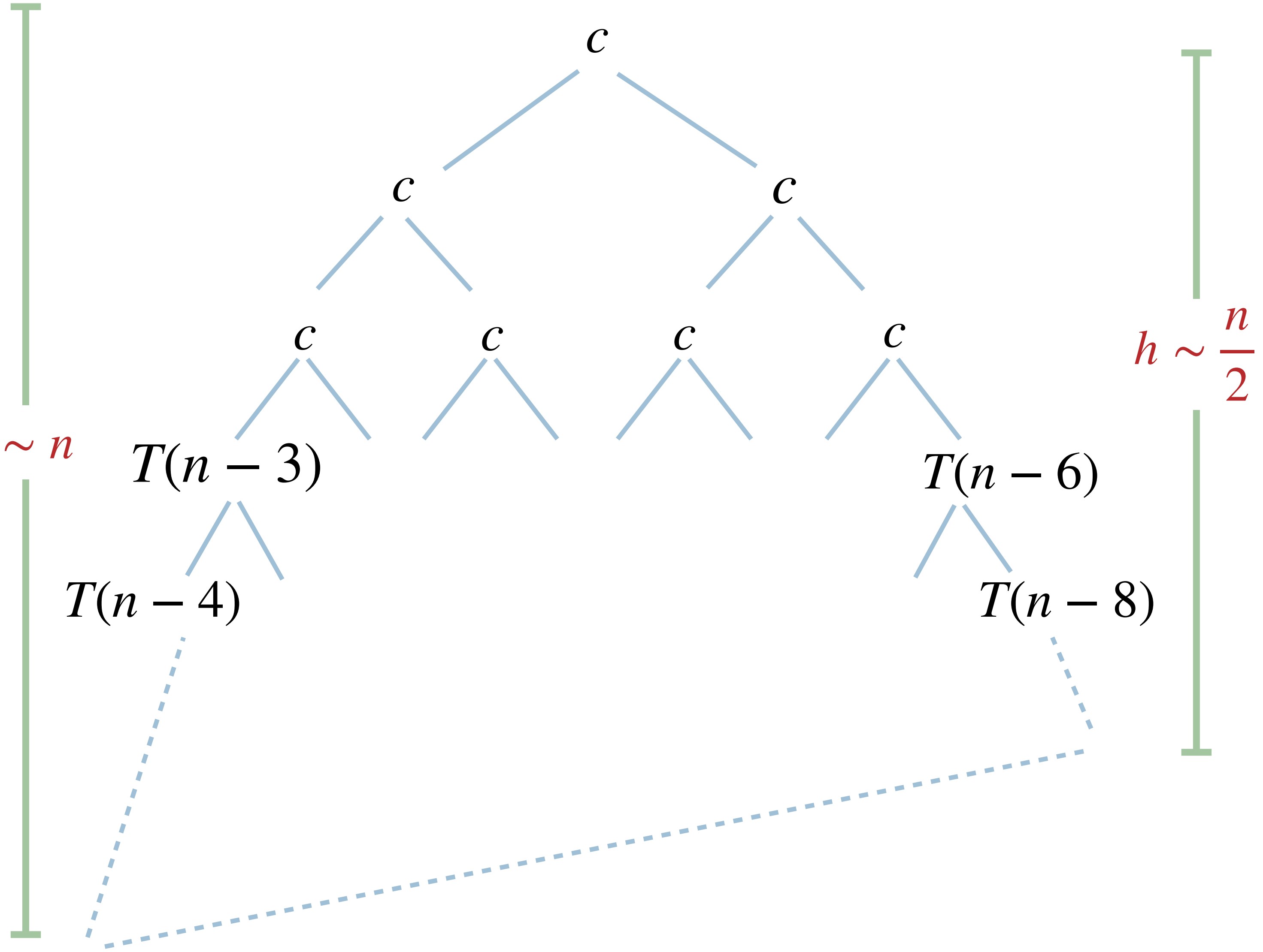
Motivation

```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

Recursion Tree



Motivation

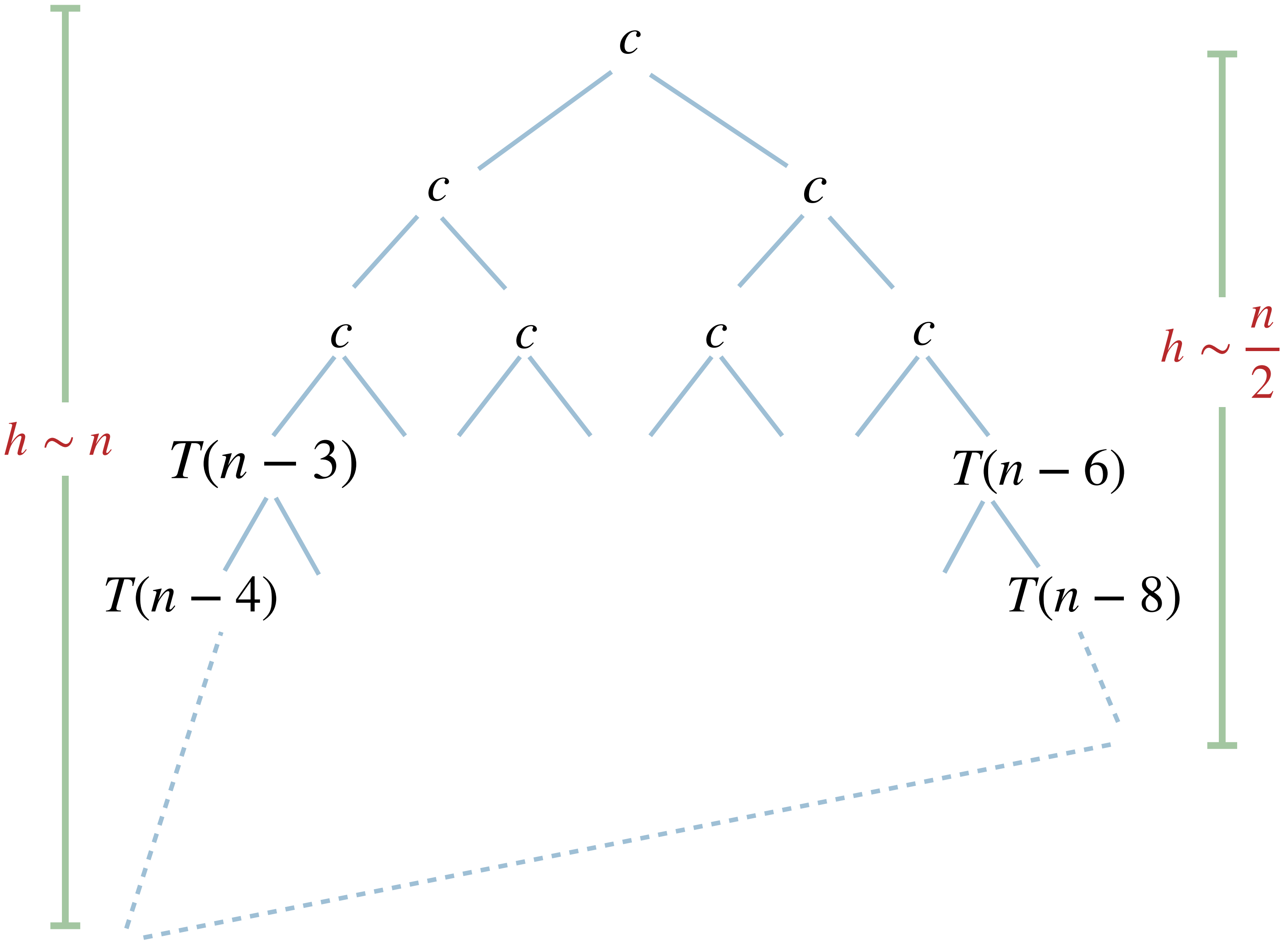
```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

- $\frac{n}{2} \leq$ number of levels $\leq n$

Recursion Tree



Motivation

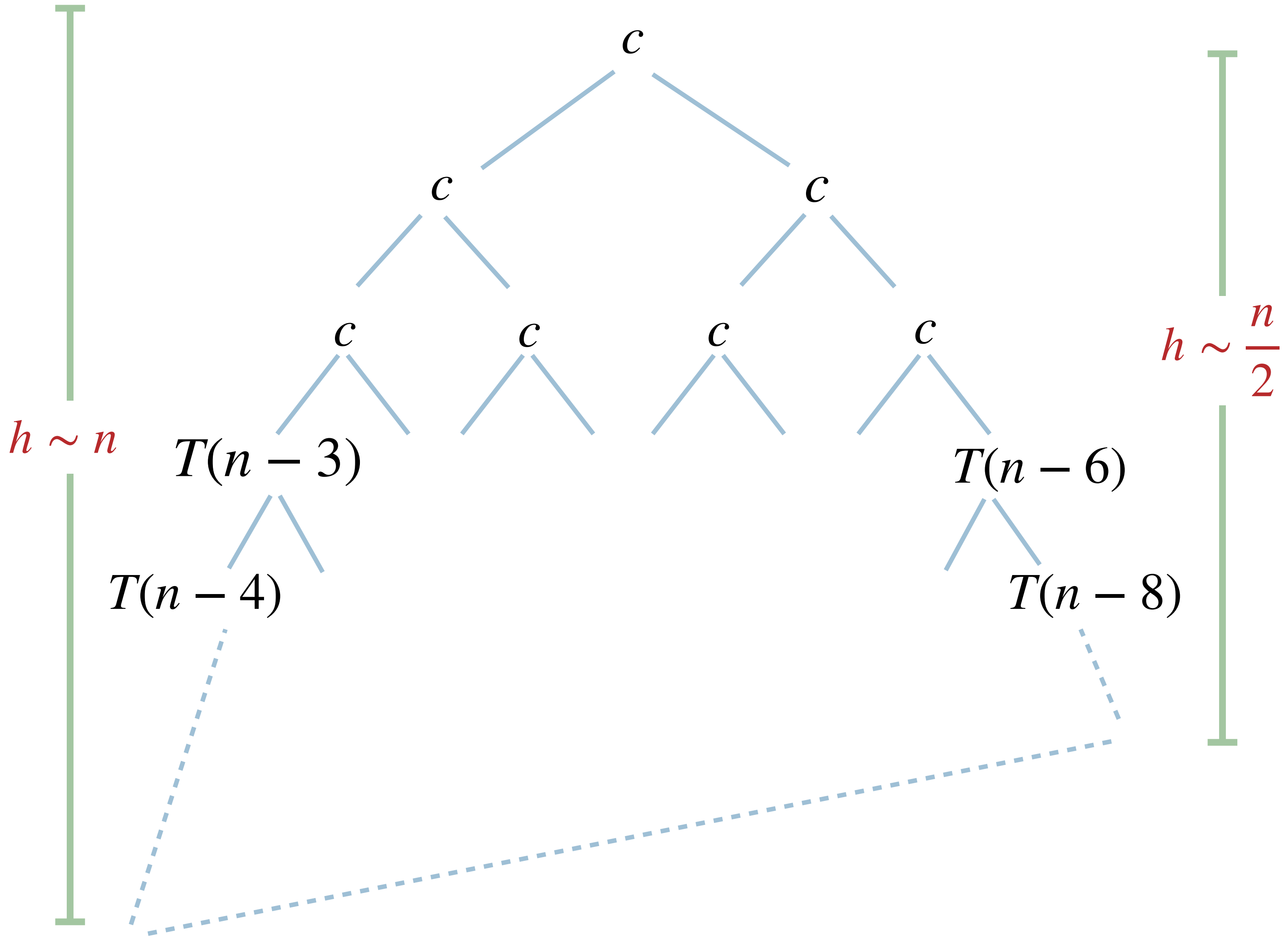
```
FIB(i)

  if (n == 0): return 0
  if (n == 1): return 1

  return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

- $\frac{n}{2} \leq$ number of levels $\leq n$
- Work at level i $\leq 2^i$

Recursion Tree



Motivation

FIB(i)

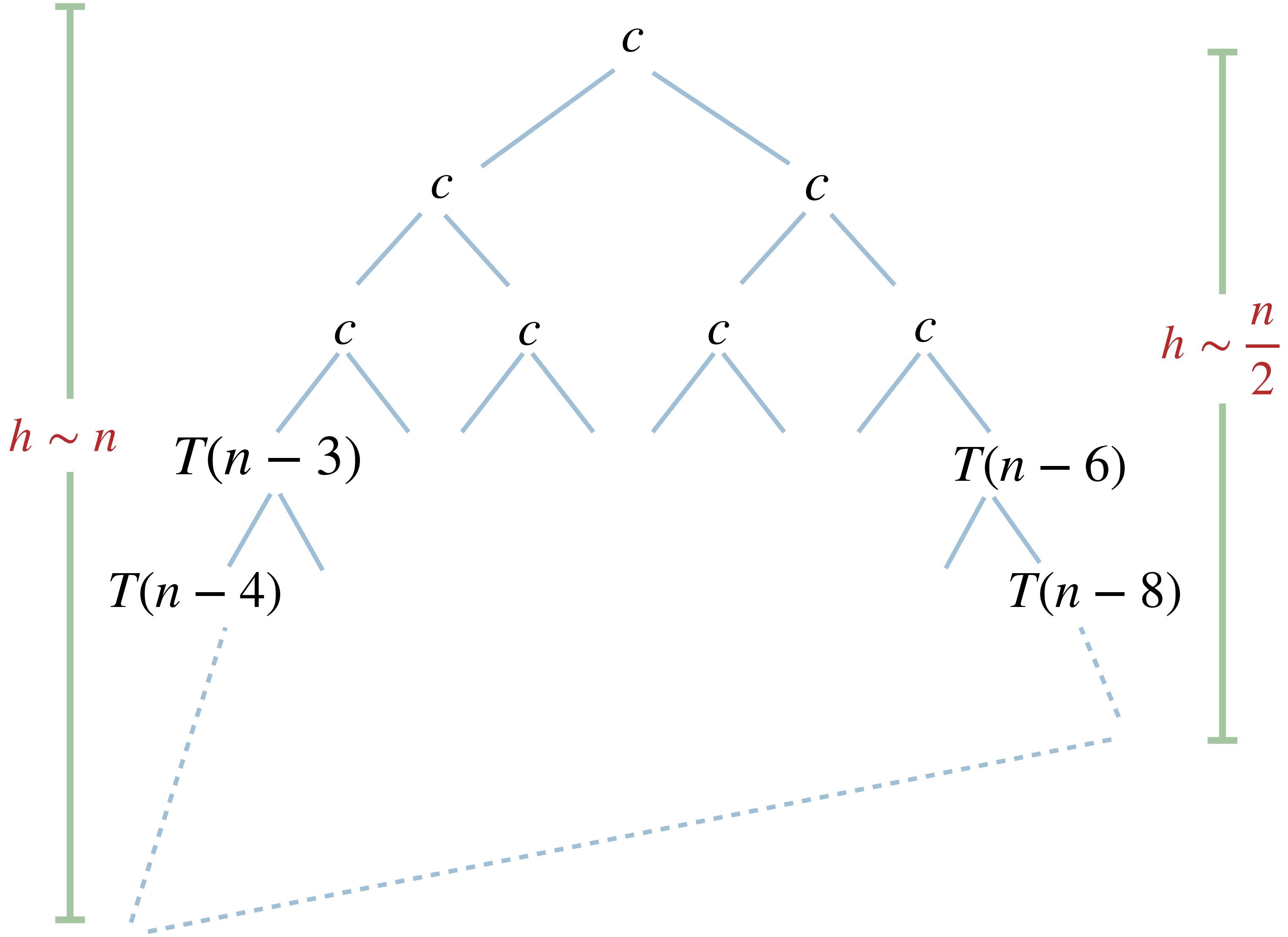
```
if (n == 0): return 0
if (n == 1): return 1

return FIB(n-1) + FIB(n-2)
```

Recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$

- $\frac{n}{2} \le$ number of levels $\le n$
- Work at level i $\le 2^i$
- $T(n) = \Omega(2^{\frac{n}{2}}) = \Omega(\sqrt{2}^n)$
  $T(n) = O(2^n)$
  More precisely: $\Theta(1.618^n)$

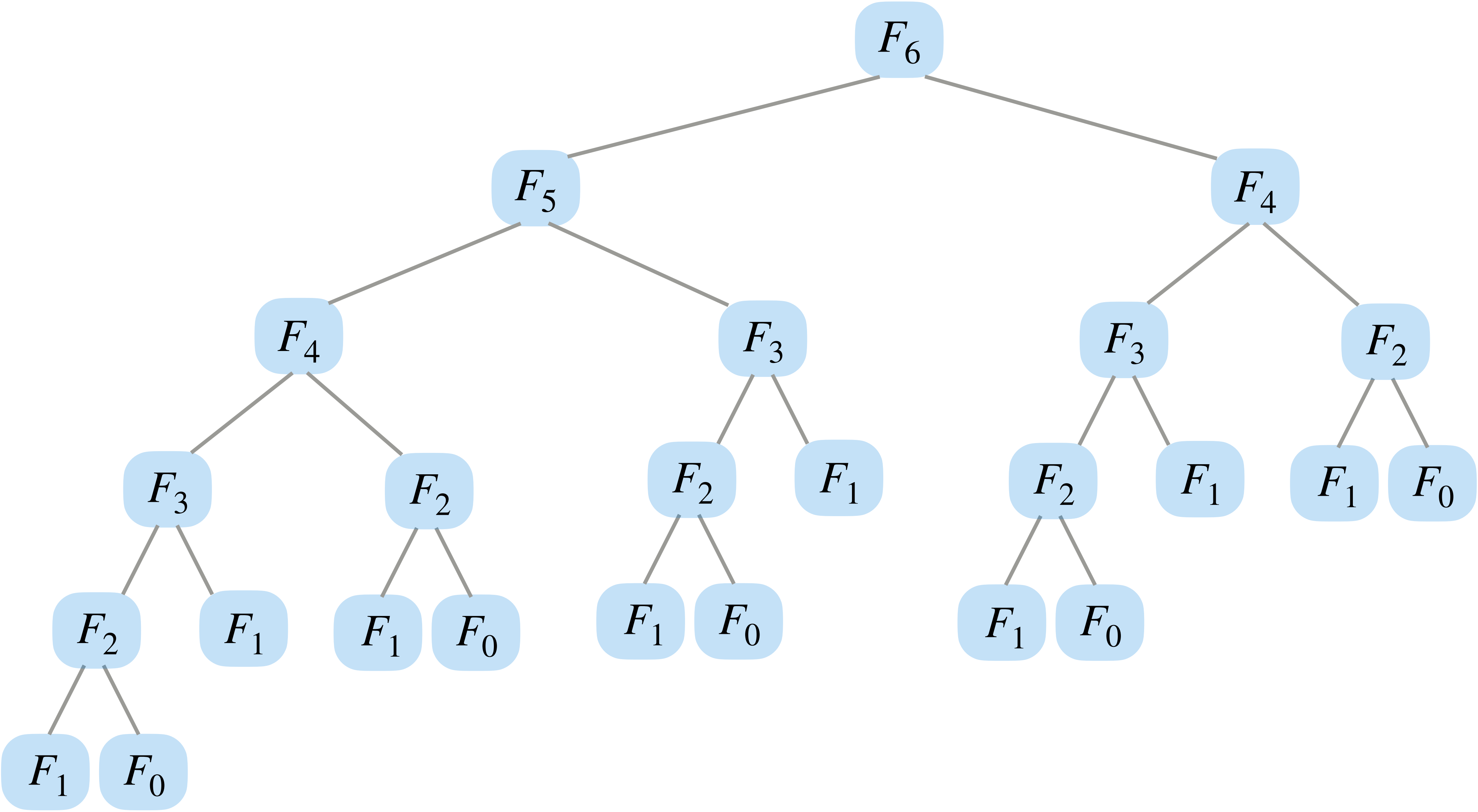
Recursion Tree



Running time is exponential!

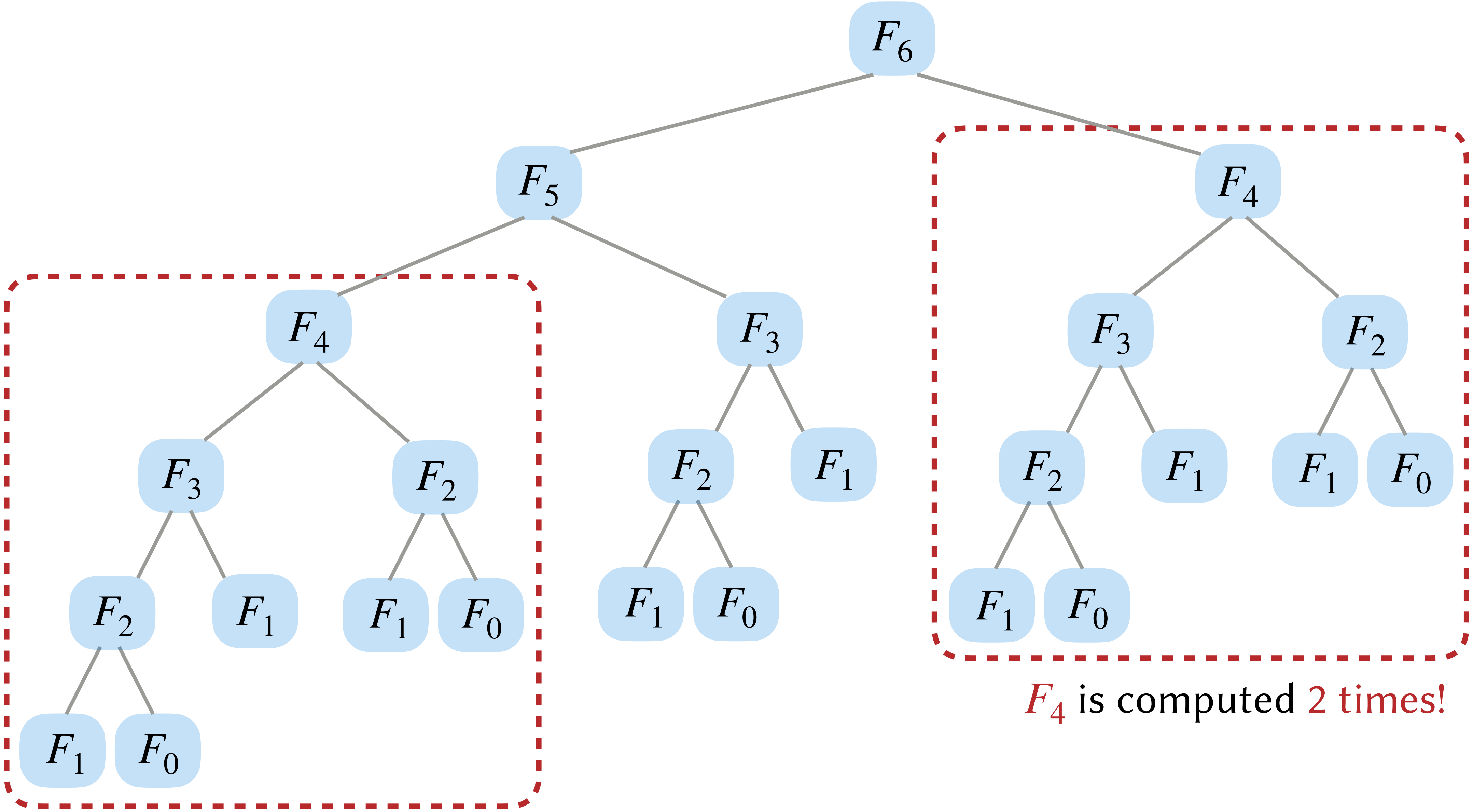
Motivation

What is the problem?



Motivation

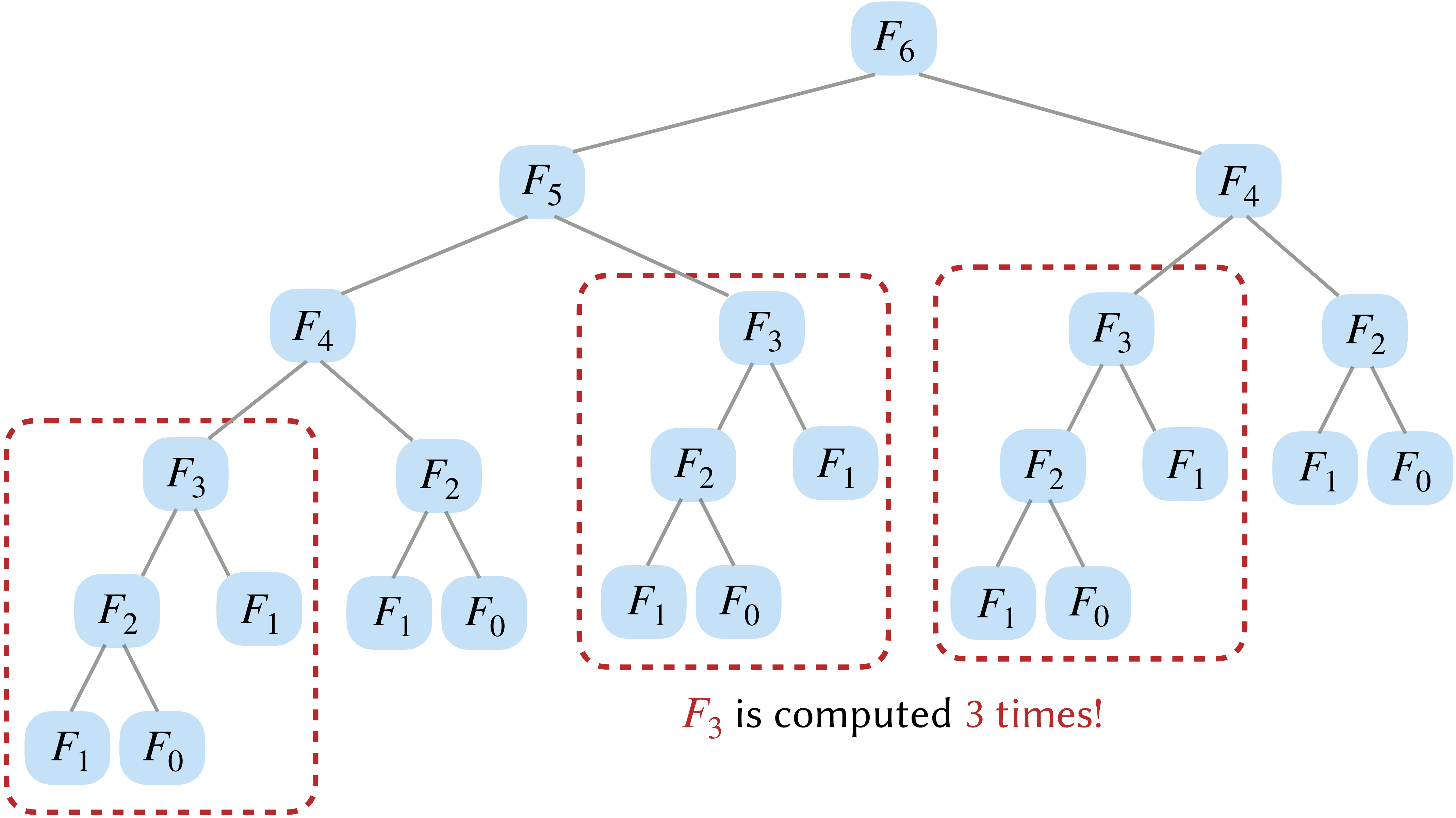
What is the problem?



F_4 is computed 2 times!

Motivation

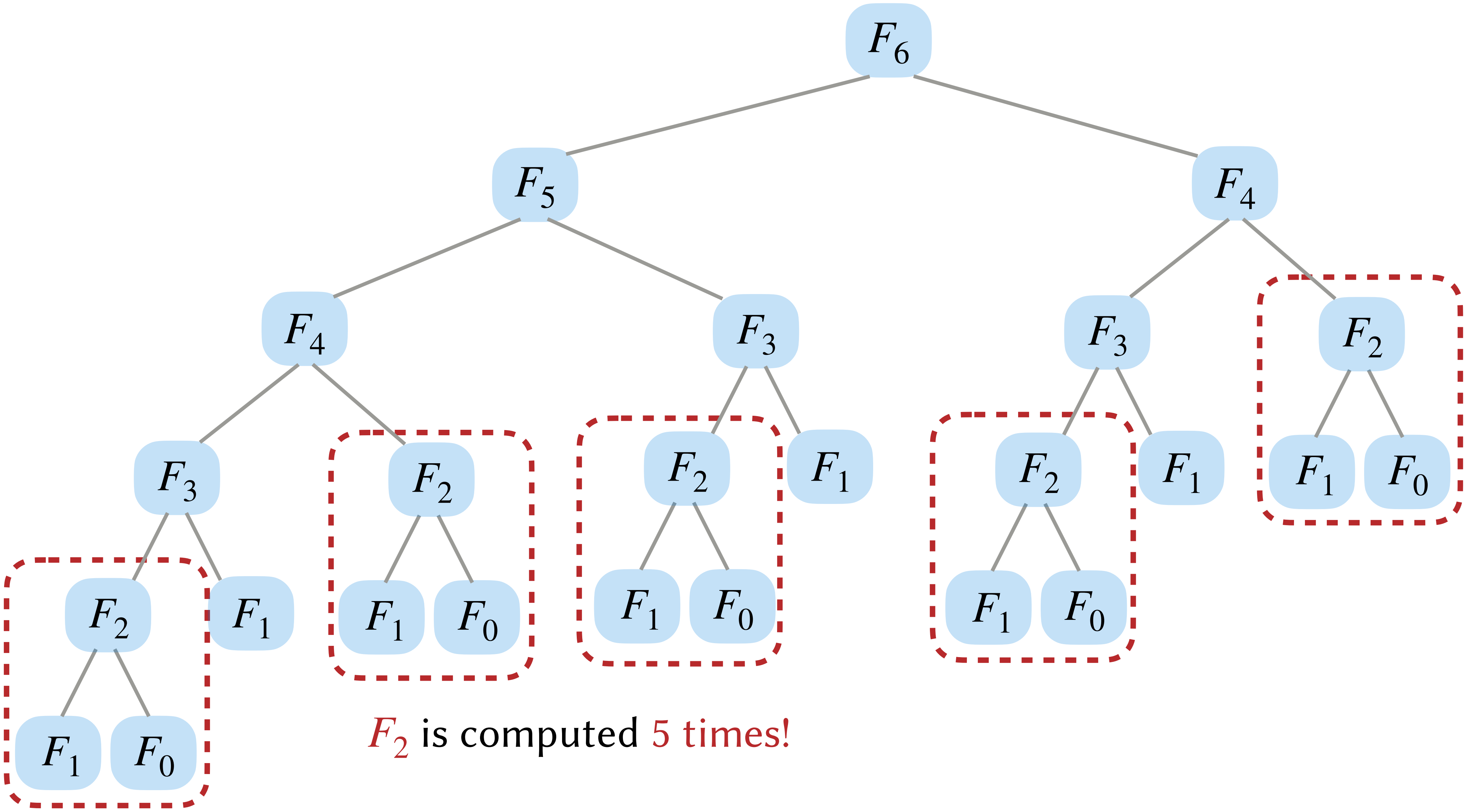
What is the problem?



F_3 is computed 3 times!

Motivation

What is the problem?

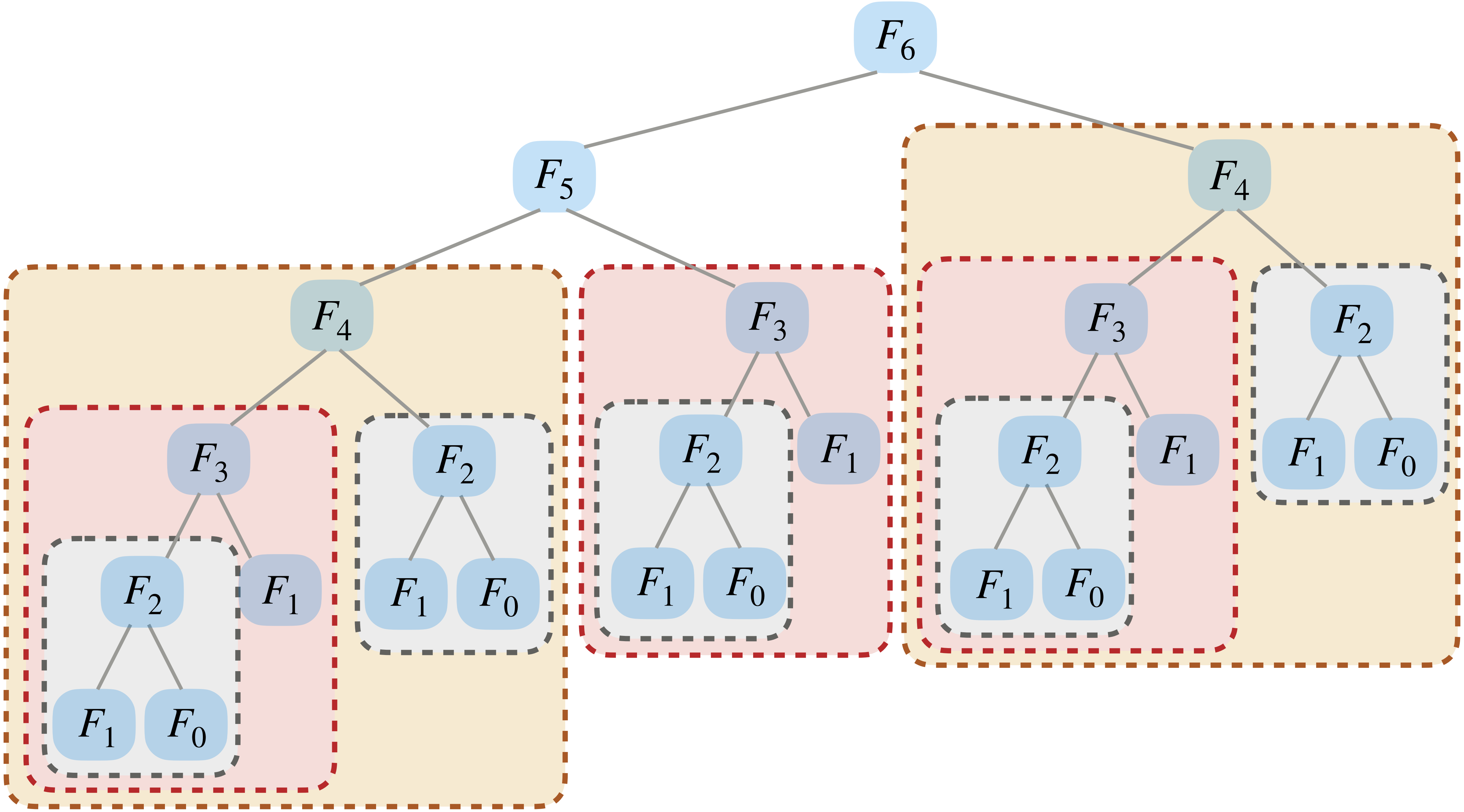


F_2 is computed 5 times!

Motivation

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

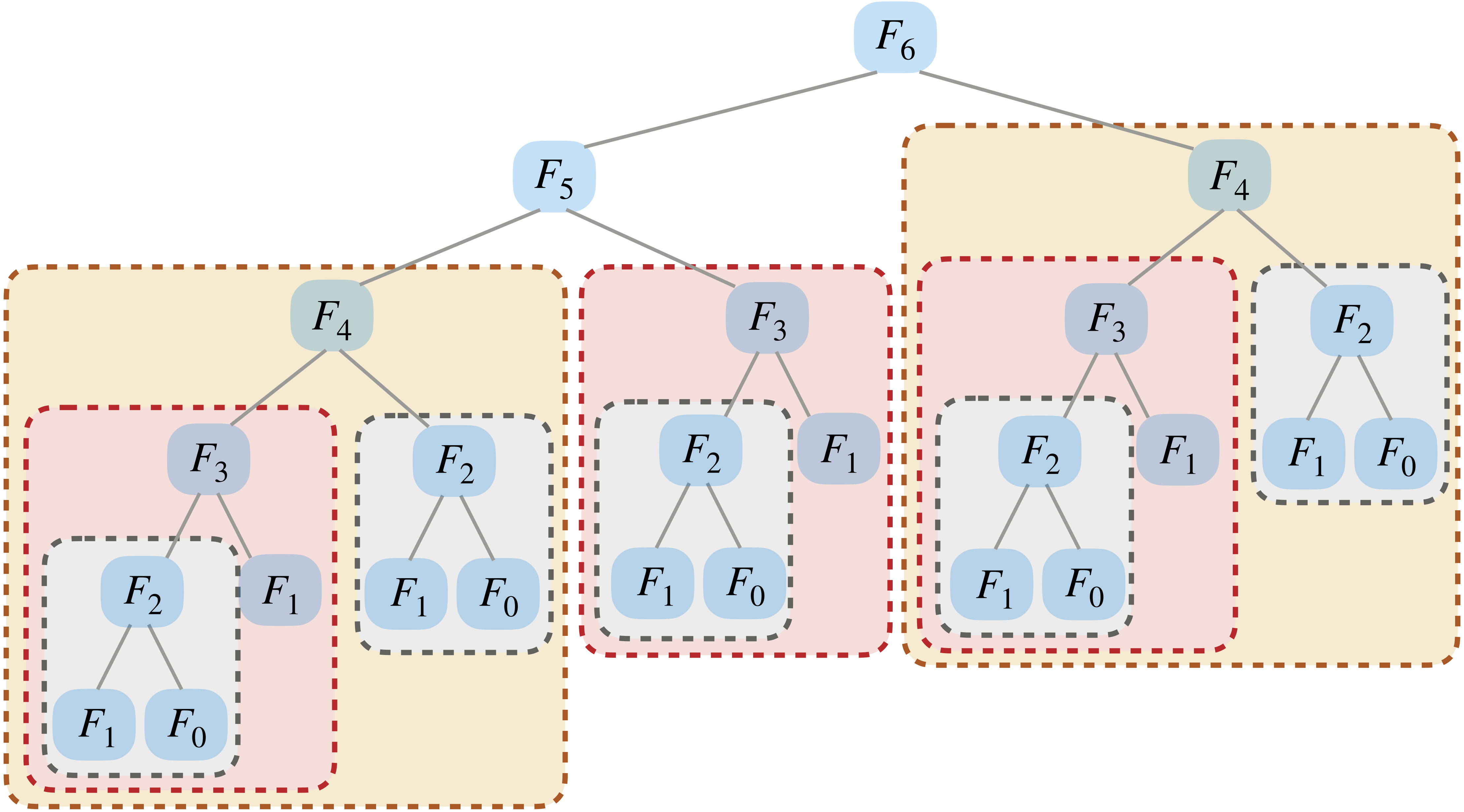


Motivation

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!



Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

Memo-ization
(keeping a memo)

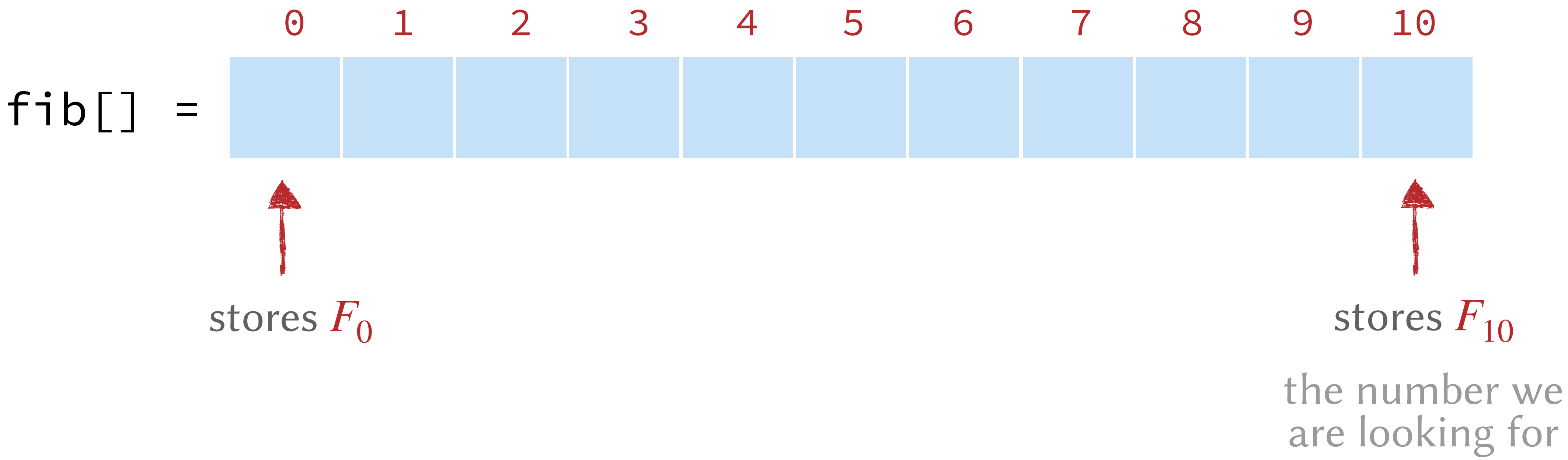
Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.



0 1 2 3 4 5 6 7 8 9 10

`fib[] =`

stores F_0

stores F_{10}

the number we are looking for

Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

	0	1	2	3	4	5	6	7	8	9	10
fib[] =	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1

base cases

unknown fibonacci numbers

```
FIB(n)

Create fib[] of size n+1
Initialize fib[] to -1
fib[0] = 0, fib[1] = 1
```

initialize the table

Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

	0	1	2	3	4	5	6	7	8	9	10
fib[] =	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1

base cases

unknown fibonacci numbers

```
FIB(n)

  Create fib[] of size n+1
  Initialize fib[] to -1
  fib[0] = 0, fib[1] = 1

  FIB(n, fib[])
  return fib[n]
```

initialize the table

fill the table

Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

	0	1	2	3	4	5	6	7	8	9	10
fib[] =	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1

base cases

unknown fibonacci numbers

```
FIB(n)

Create fib[] of size n+1
Initialize fib[] to -1
fib[0] = 0, fib[1] = 1

FIB(n, fib[])
return fib[n]
```

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]
```

fibonacci number was computed before!

Note. The pseudocode assumes that changes made to `fib[]` in this function are visible in the calling function **FIB**(n)

Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

	0	1	2	3	4	5	6	7	8	9	10
fib[] =	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1

base cases

unknown fibonacci numbers

```
FIB(n)

Create fib[] of size n+1

Initialize fib[] to -1
fib[0] = 0, fib[1] = 1

FIB(n, fib[])
return fib[n]
```

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])
```

fibonacci number was *not* computed before!

Memoization

What is the problem?

- Finding F_n involves solving overlapping subproblems.
- Subproblems are recomputed multiple times.

Goal. Avoid computing the same problem twice!

Memoization. Store the result of each computation in a table. Compute only if the table does not yet have an entry for that computation.

	0	1	2	3	4	5	6	7	8	9	10
fib[] =	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1

base cases

unknown fibonacci numbers

FIB(n)

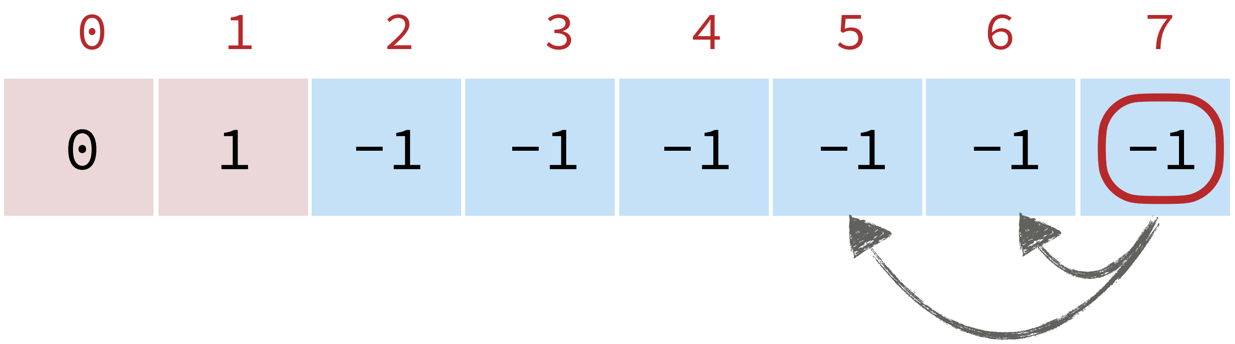
```
Create fib[] of size n+1
Initialize fib[] to -1
fib[0] = 0, fib[1] = 1

FIB(n, fib[])
return fib[n]
```

FIB(n, fib[])

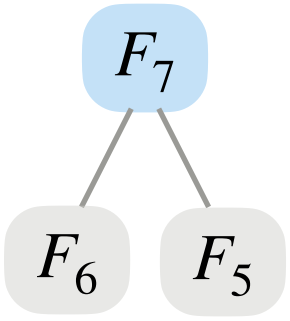
```
if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

Demo

Computing F_7 :



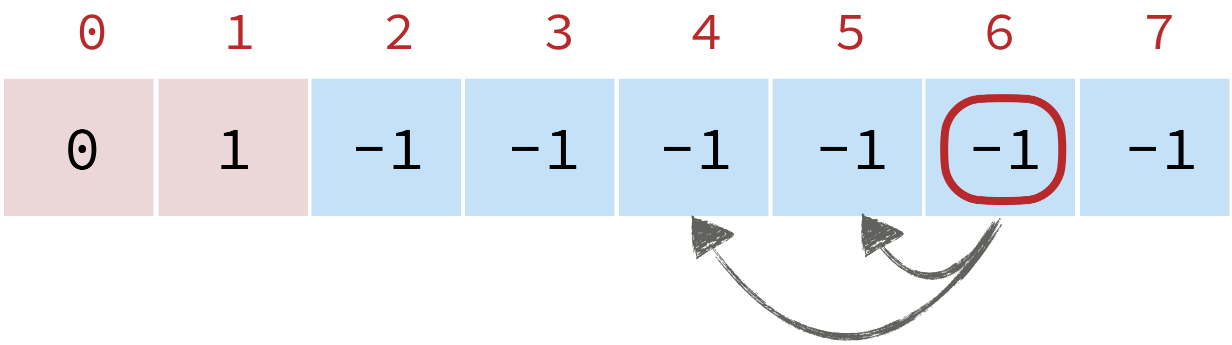
0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1



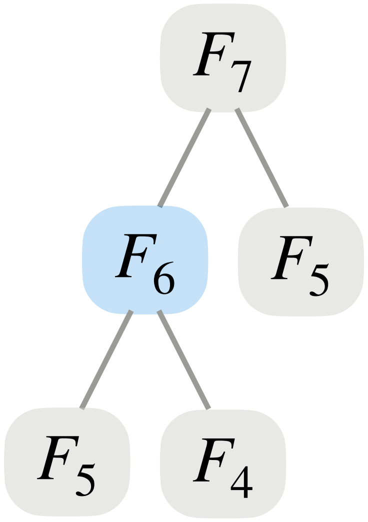
```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

Demo

Computing F_7 :



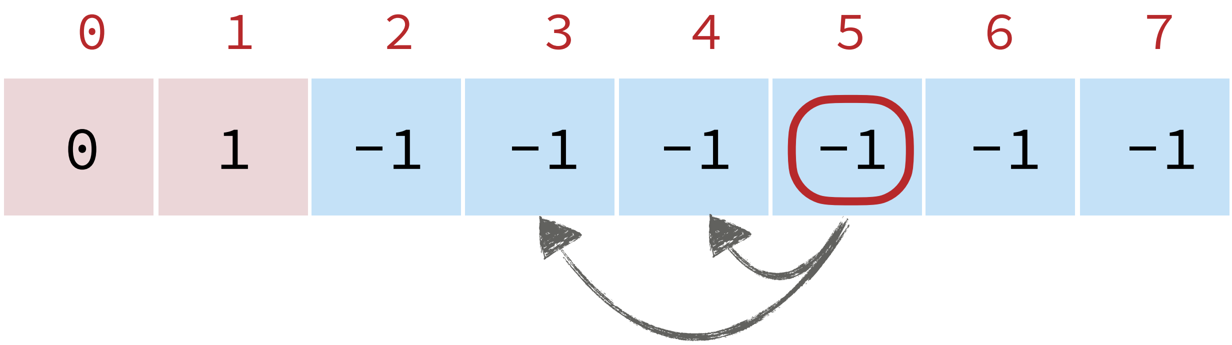
0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1



```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

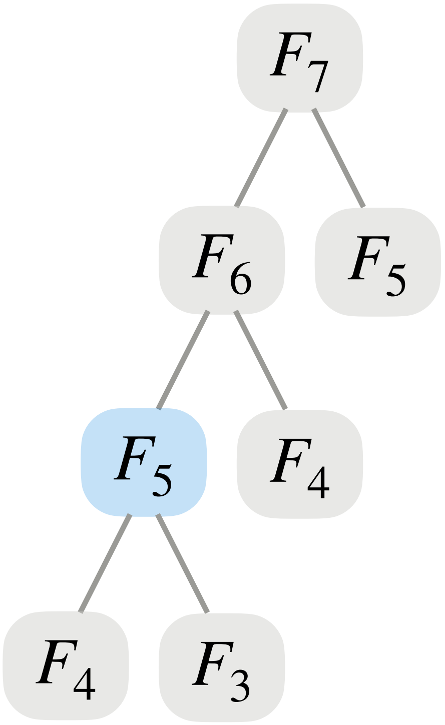
Demo

Computing F_7 :



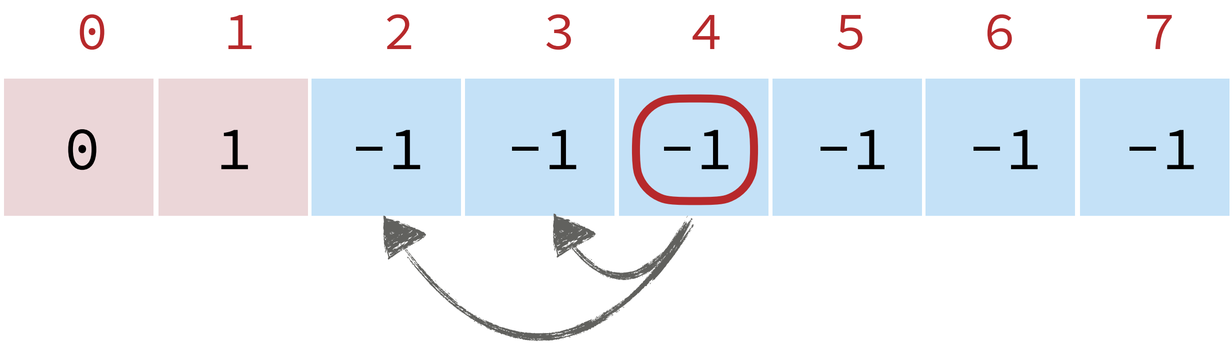
0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



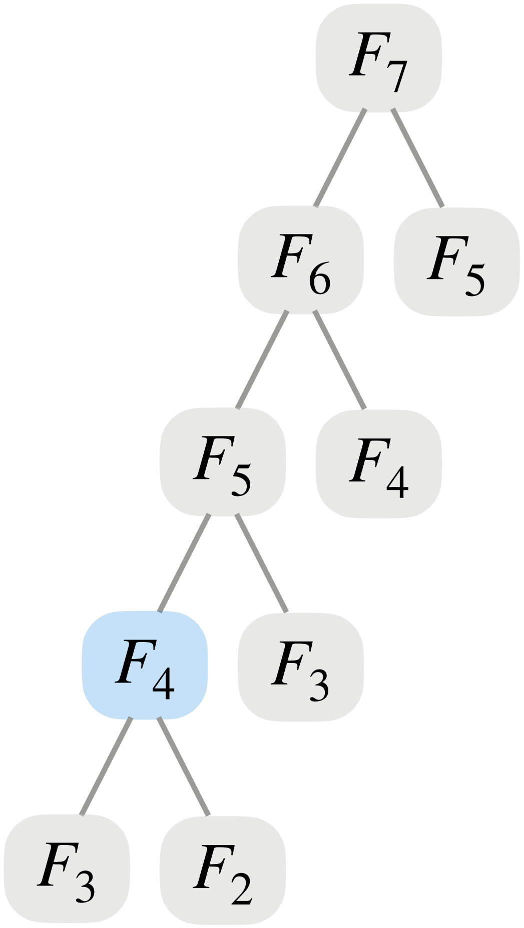
Demo

Computing F_7 :



0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

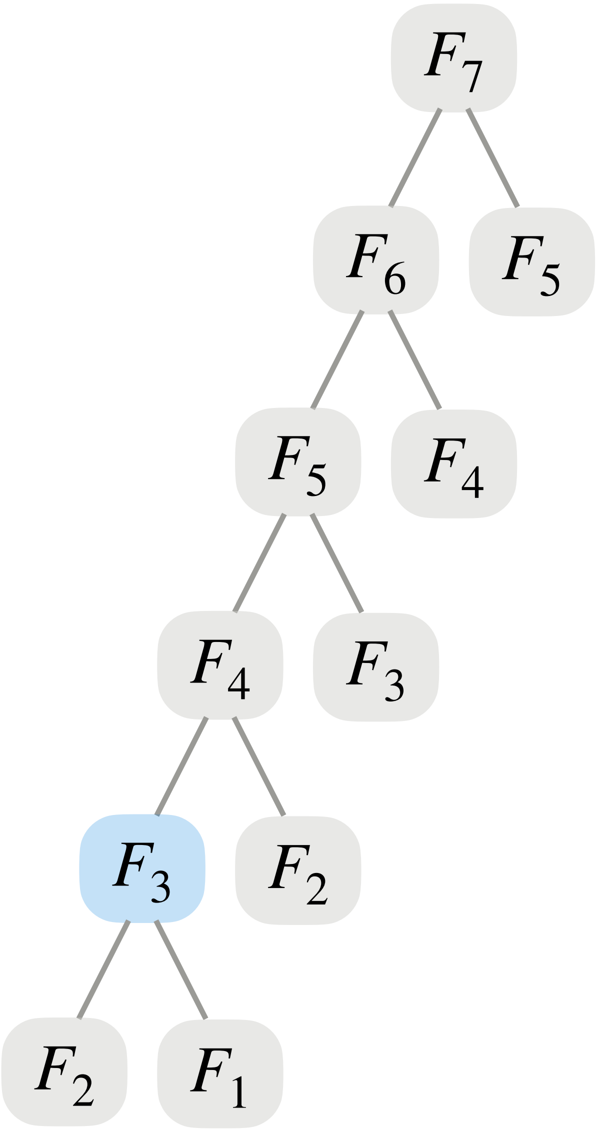


Demo

Computing F_7 :

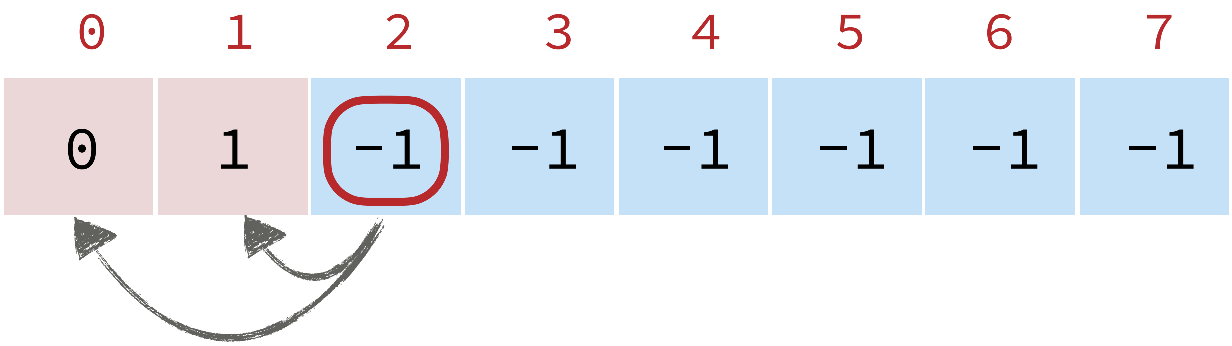
0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



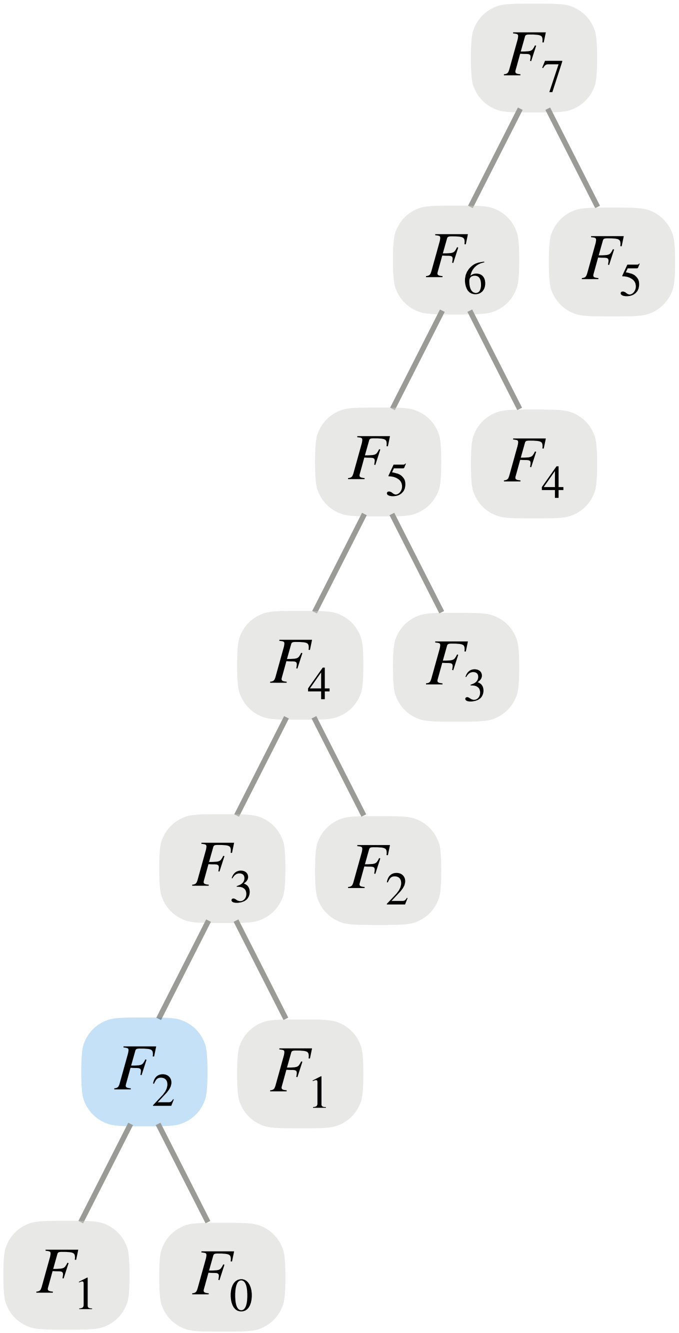
Demo

Computing F_7 :



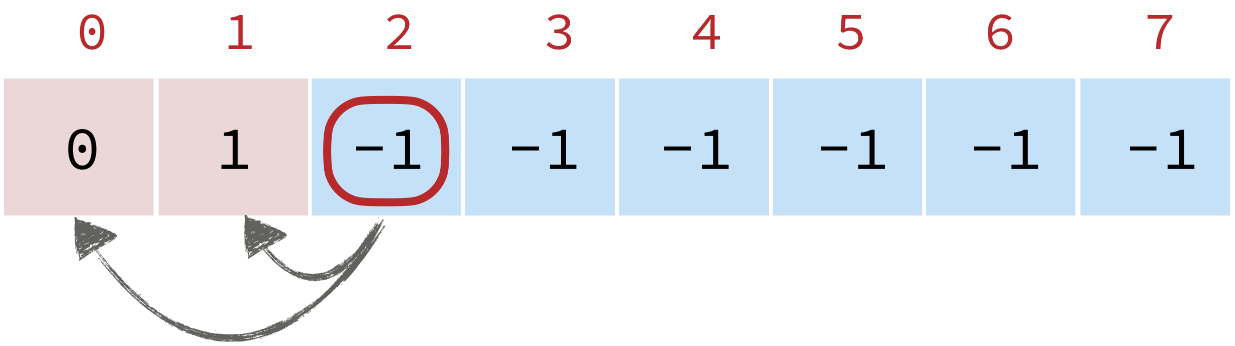
0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



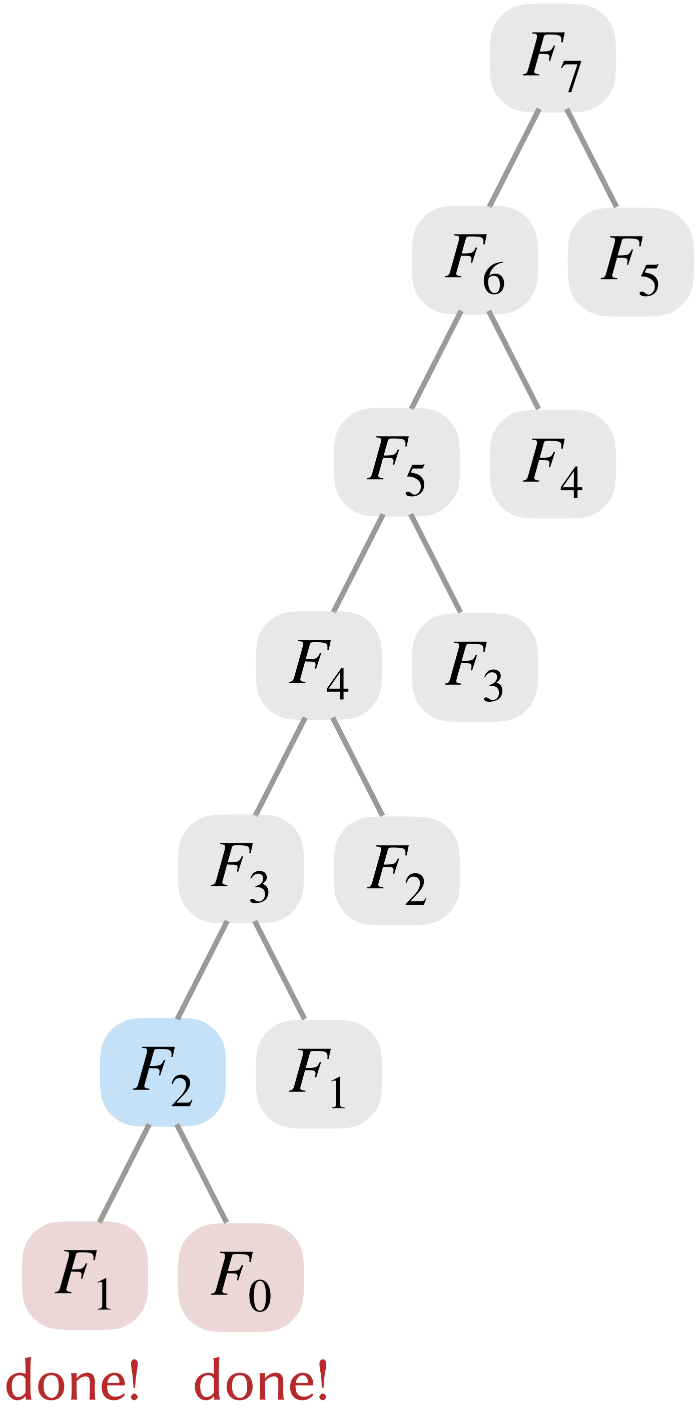
Demo

Computing F_7 :



0	1	2	3	4	5	6	7
0	1	-1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

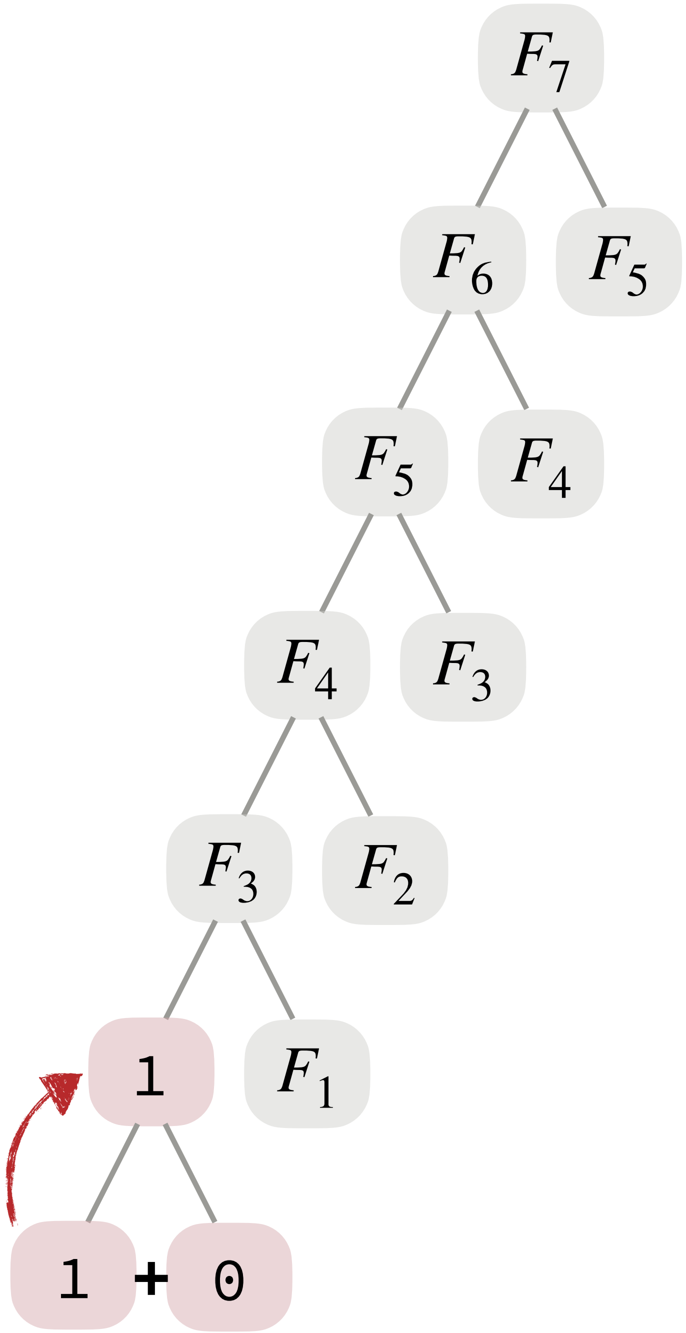


Demo

Computing F_7 :

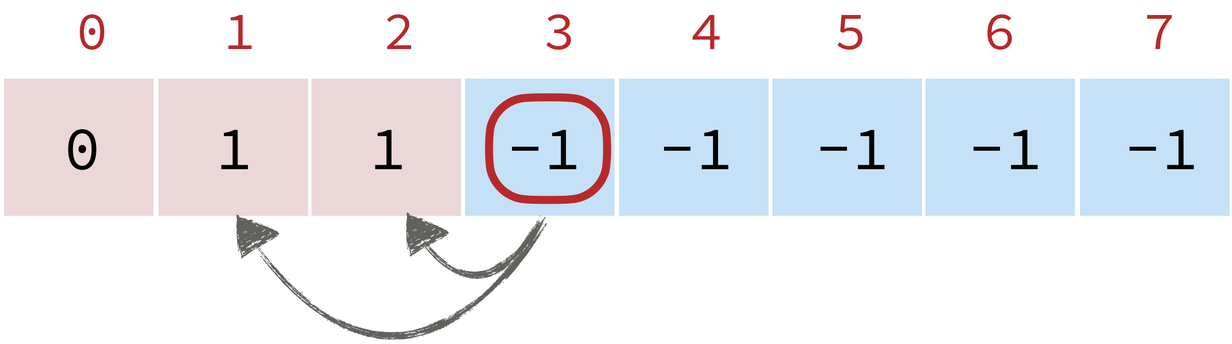
0	1	2	3	4	5	6	7
0	1	1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



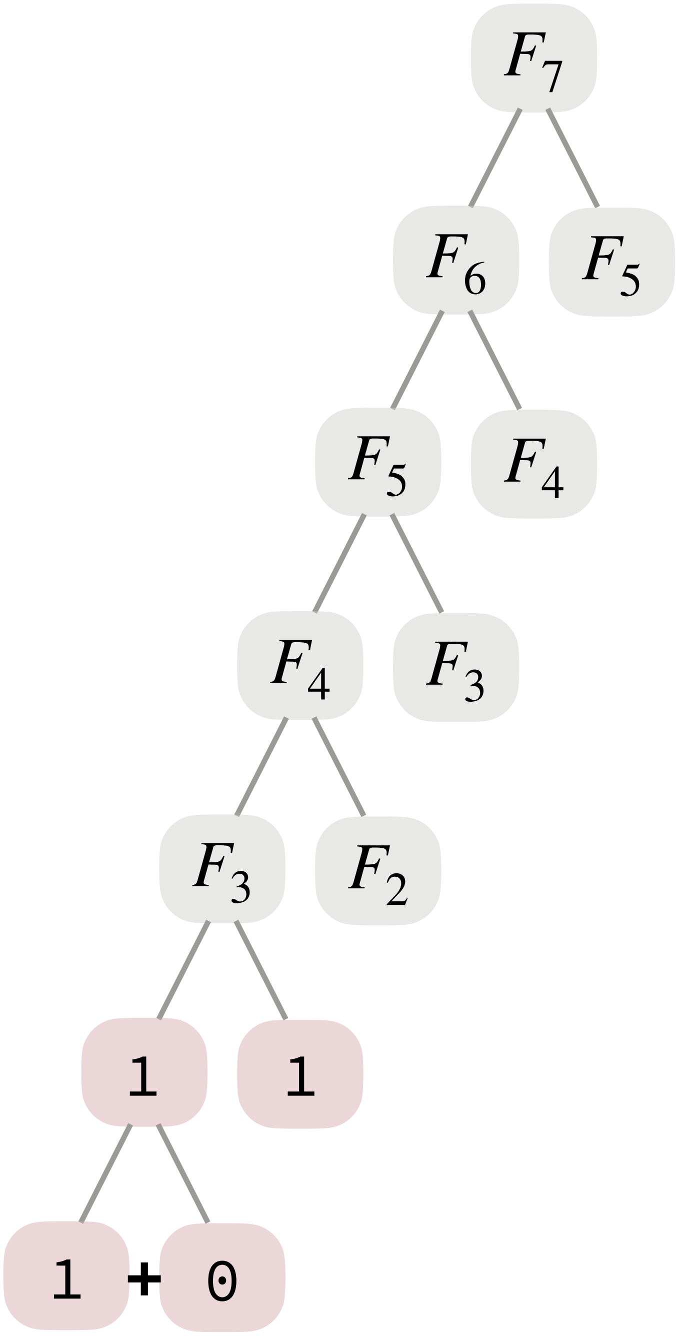
Demo

Computing F_7 :



0	1	2	3	4	5	6	7
0	1	1	-1	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

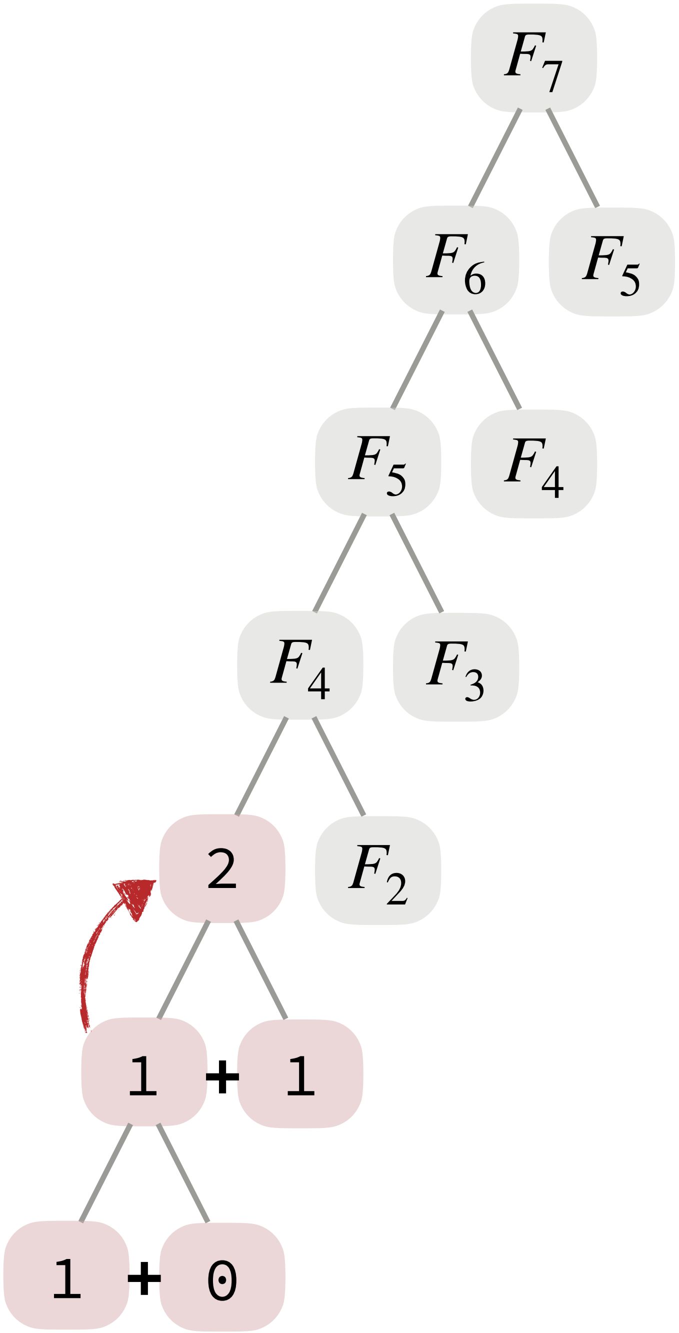


Demo

Computing F_7 :

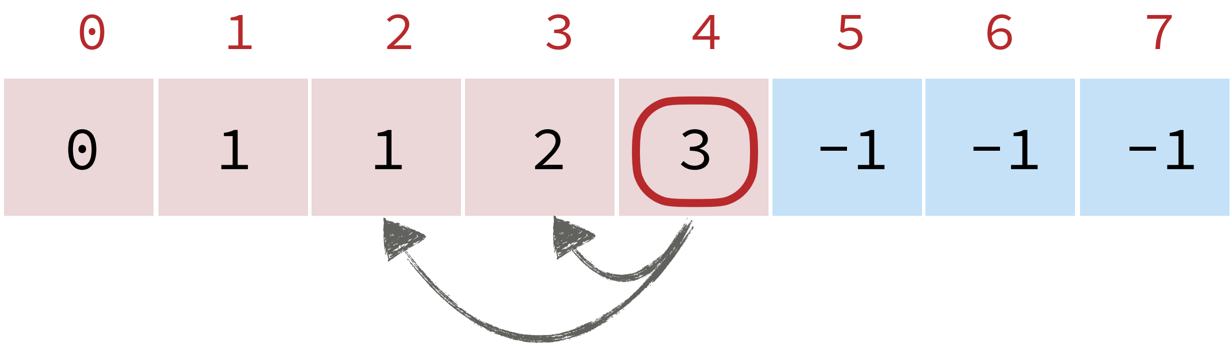
0	1	2	3	4	5	6	7
0	1	1	2	-1	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



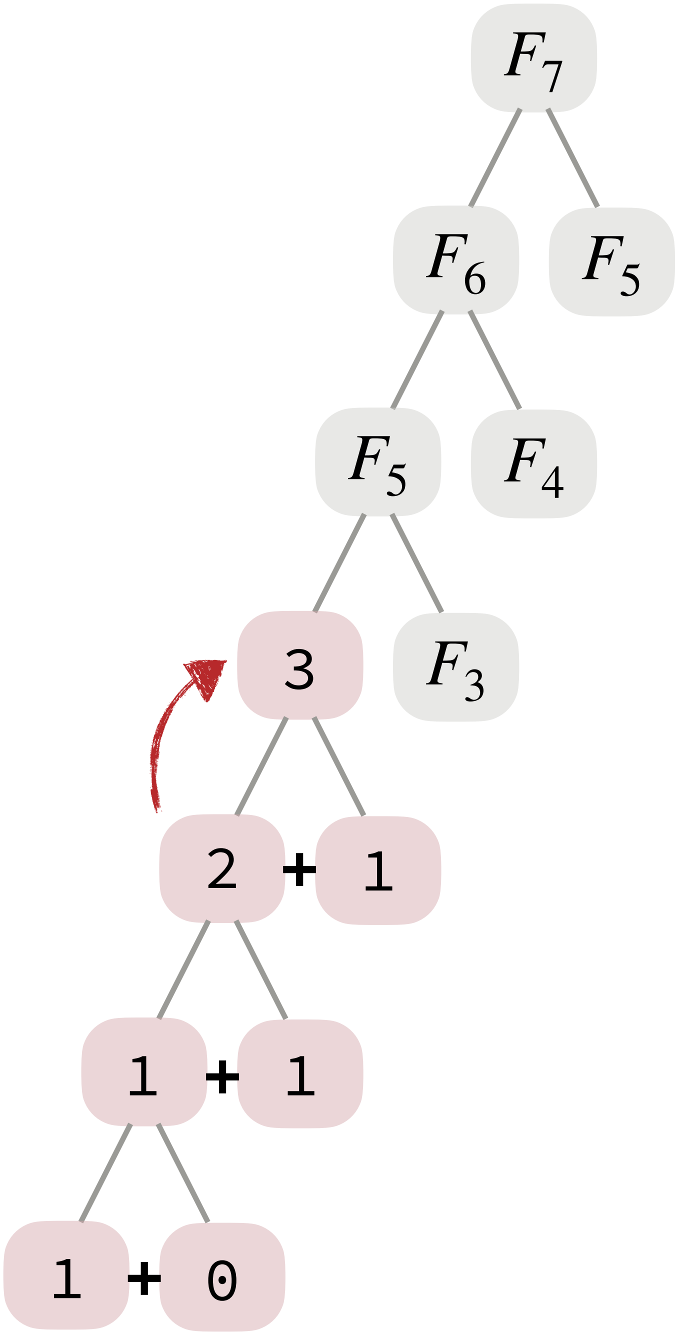
Demo

Computing F_7 :



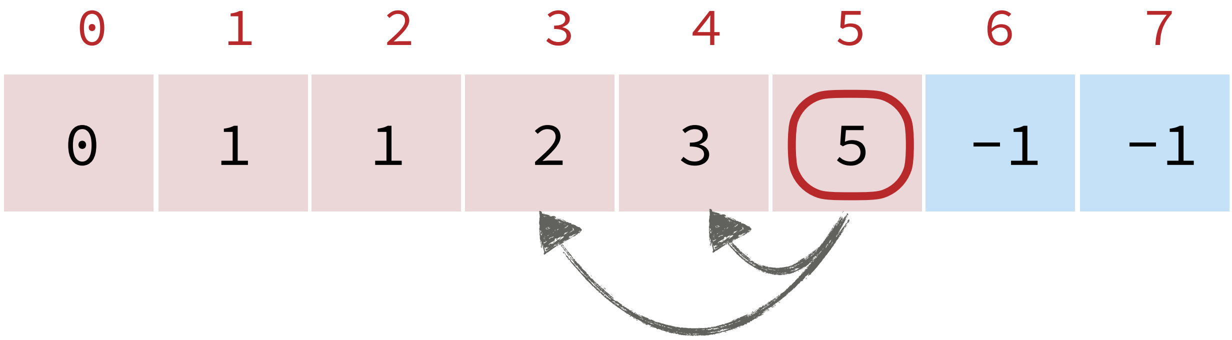
0	1	2	3	4	5	6	7
0	1	1	2	3	-1	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



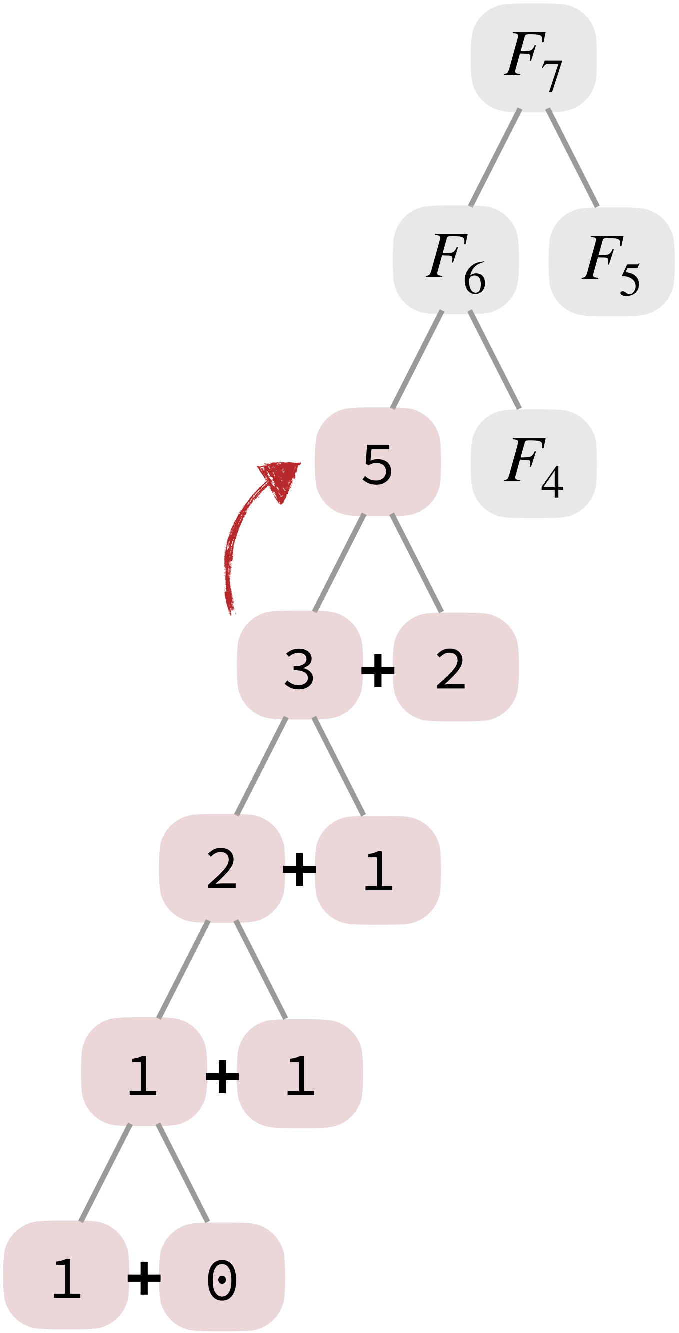
Demo

Computing F_7 :



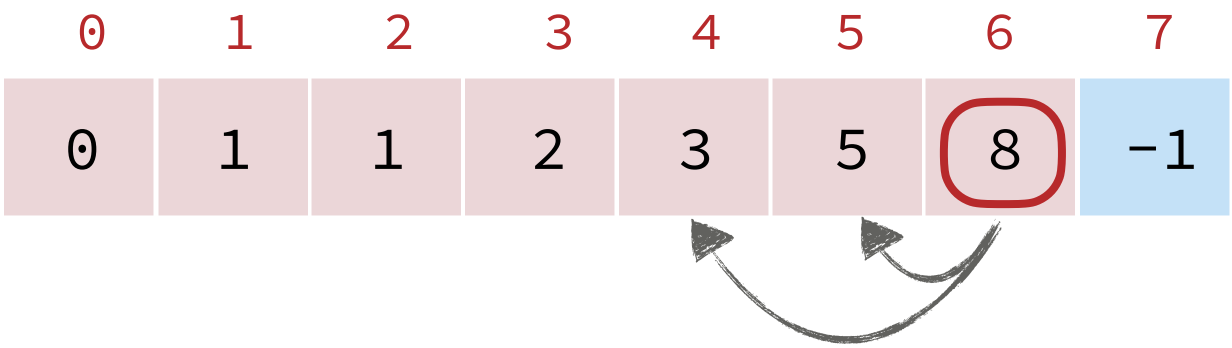
0	1	2	3	4	5	6	7
0	1	1	2	3	5	-1	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



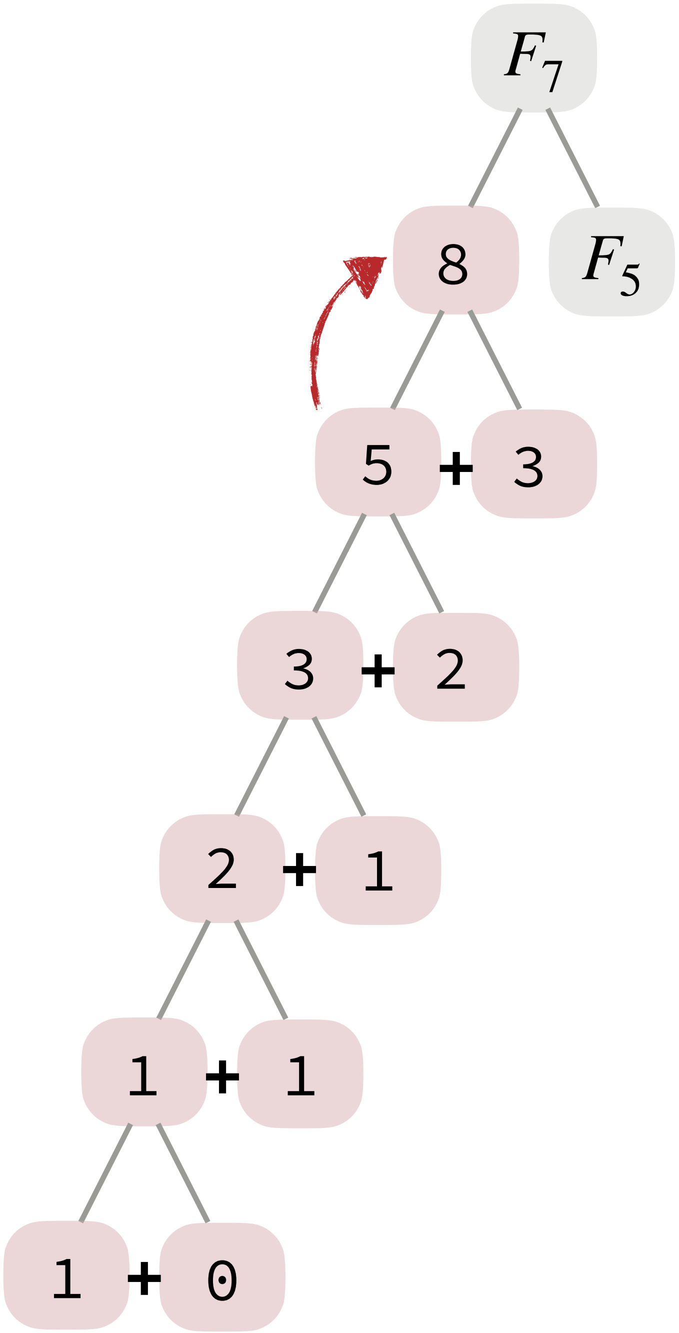
Demo

Computing F_7 :



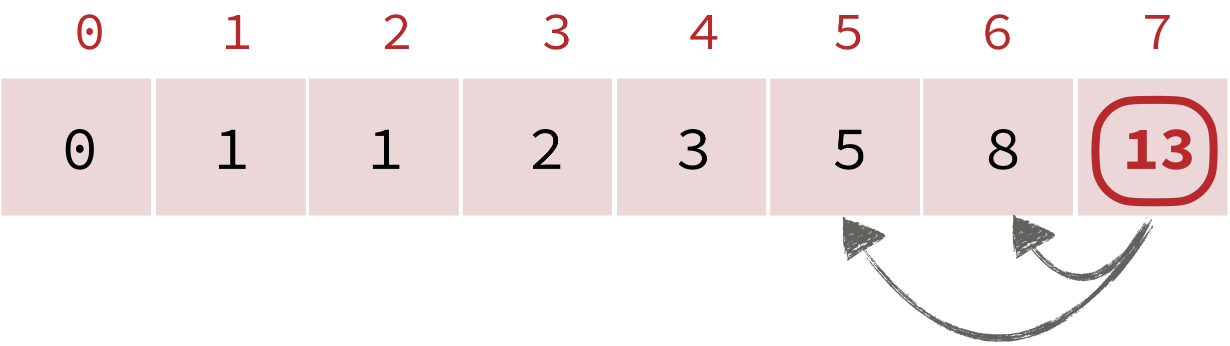
0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	-1

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



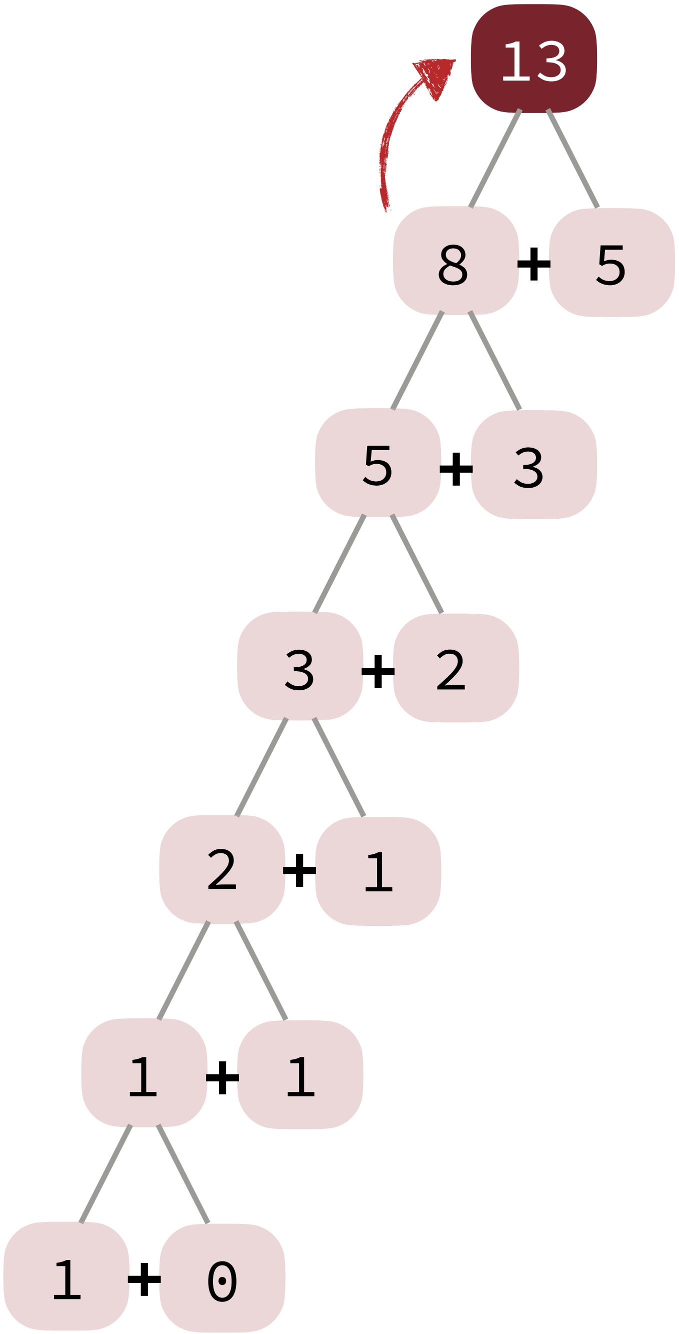
Demo

Computing F_7 :



0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	13

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```



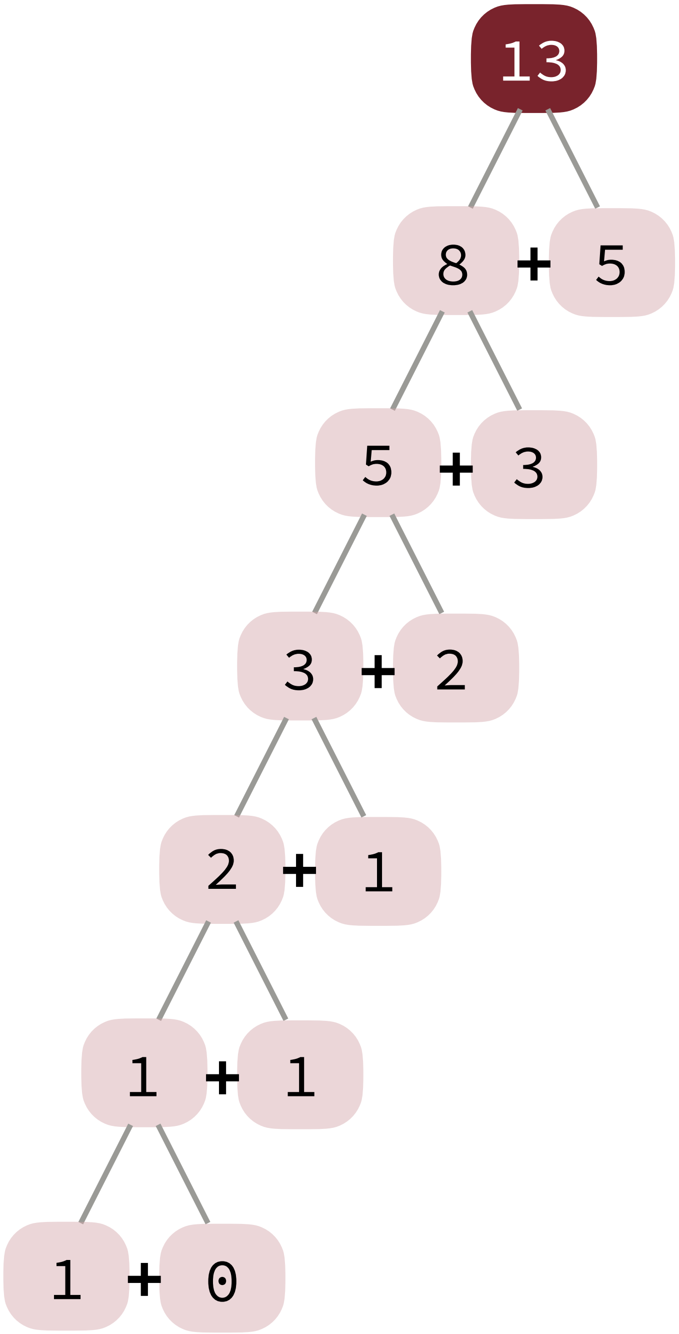
Demo

Computing F_7 :

0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	**13**

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

Running Time.



Demo

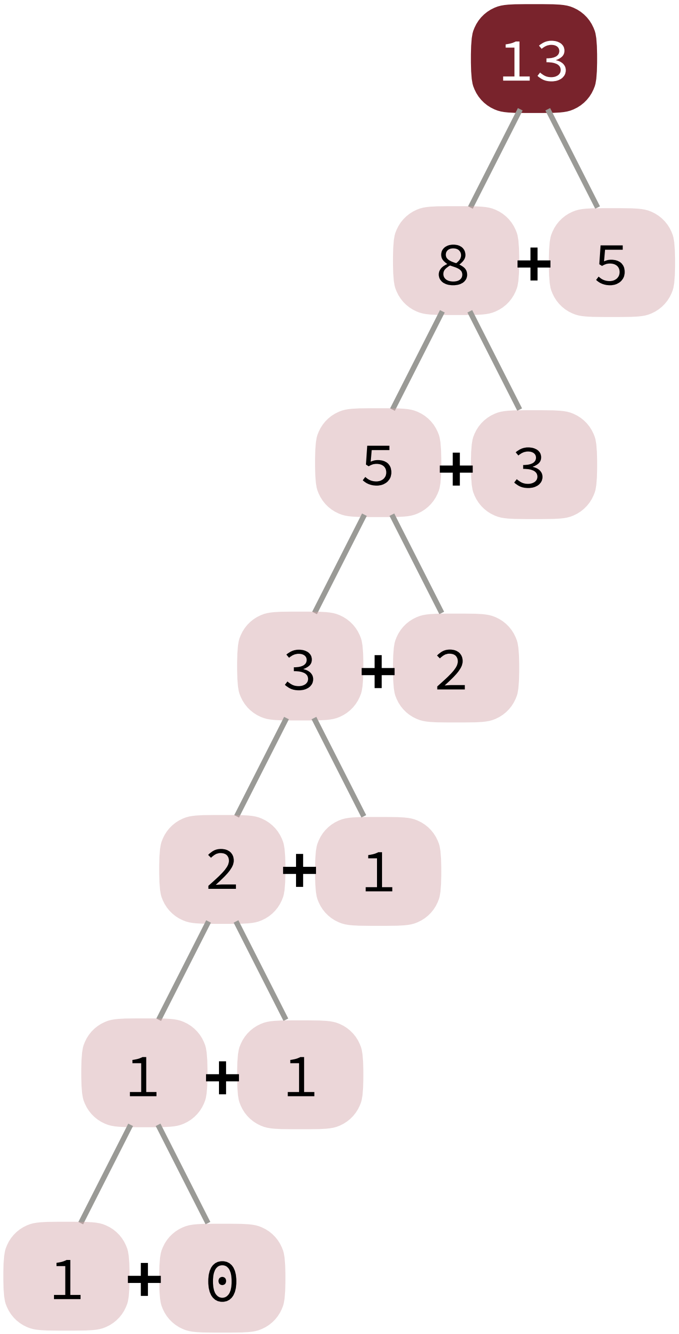
Computing F_7 :

0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	**13**

```
FIB(n, fib[])

if (fib[n] != -1): return fib[n]

fib[n] = FIB(n-1, fib[]) +
         FIB(n-2, fib[])

return fib[n]
```

Running Time.

$\Theta(n)$: $n + 1$ problems
each computed only once.



Demo

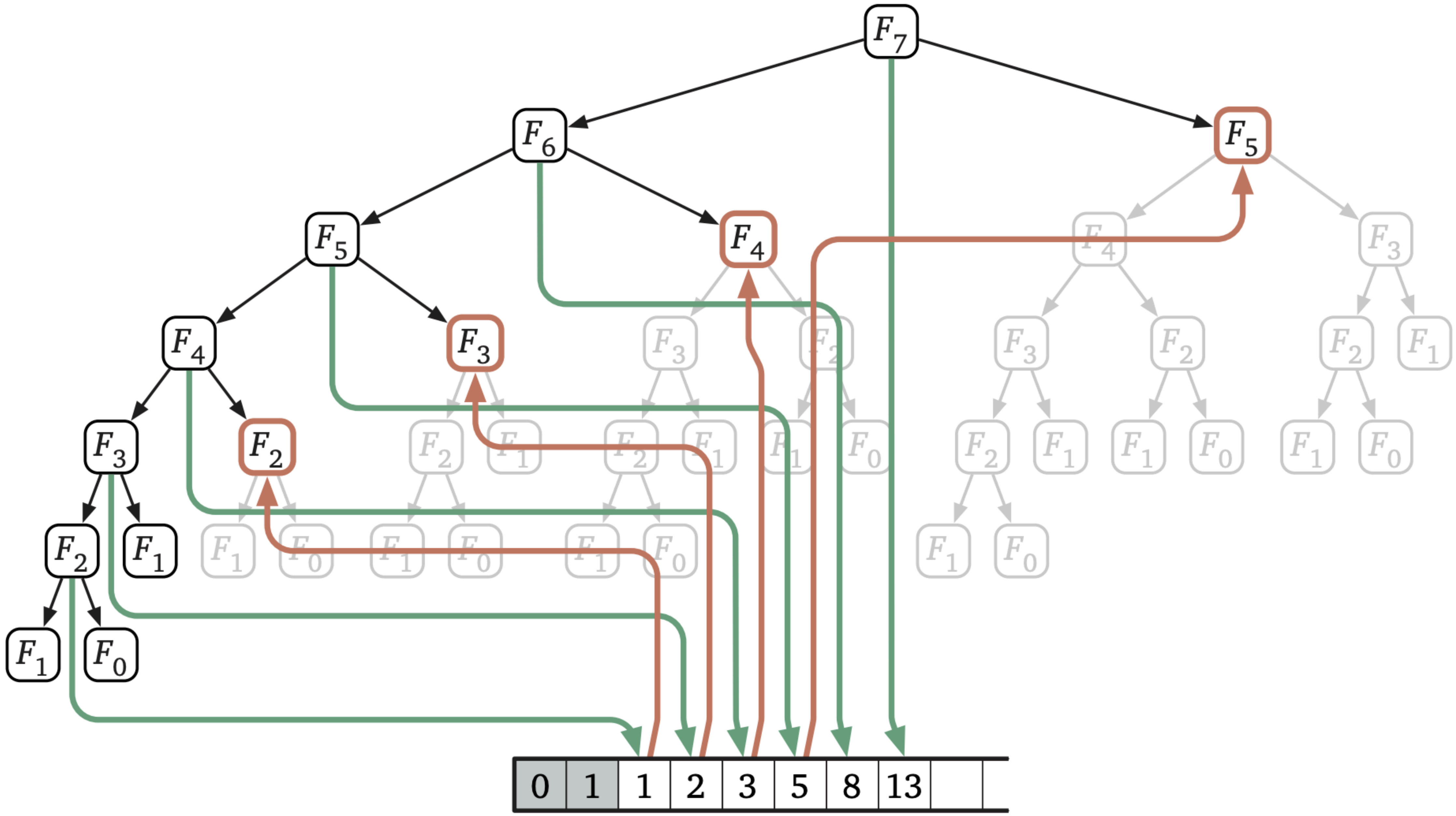


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Figure from Algorithms by Jeff Erickson

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!

0	1	2	3	4	5	6	7
0	1						

```
FIB(n)

  Create fib[] of size i+1
  fib[0] = 0, fib[1] = 1
```

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!

0	1	2	3	4	5	6	7
0	1						

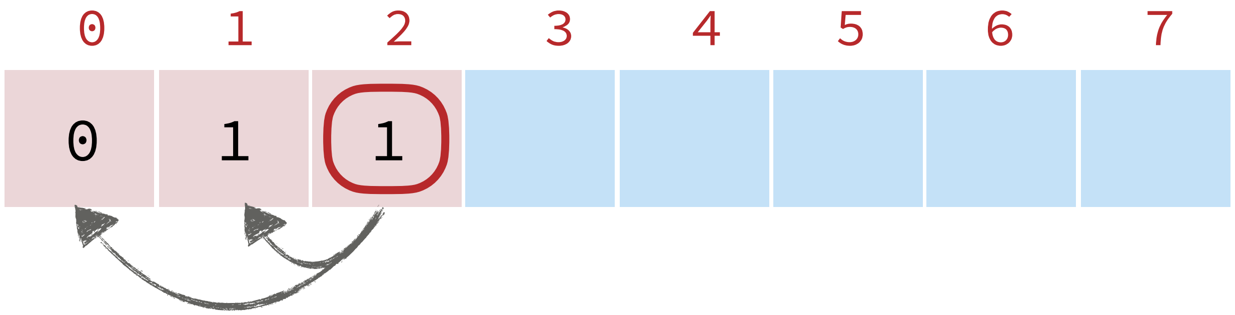
```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems
and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1					

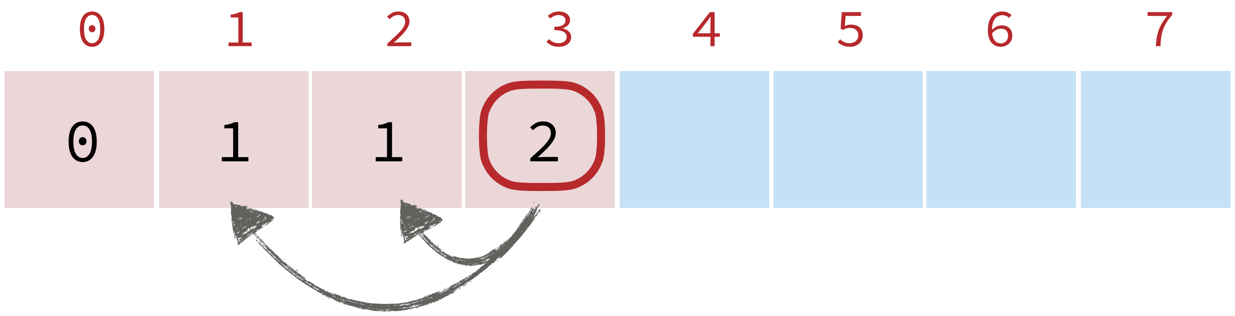
```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1	2				

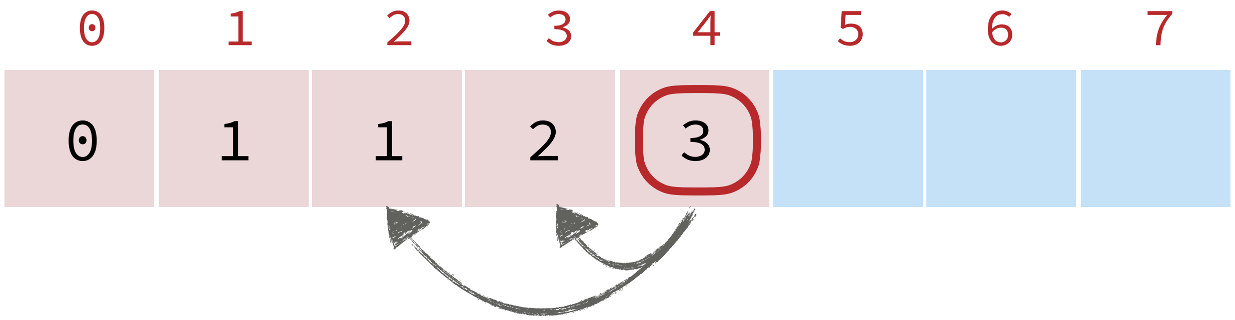
```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems
and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1	2	3			

FIB(n)

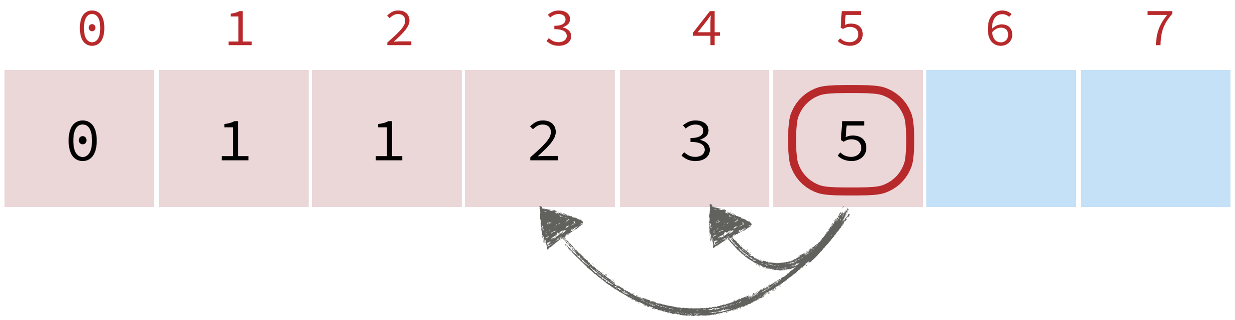
```
Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1	2	3	5		

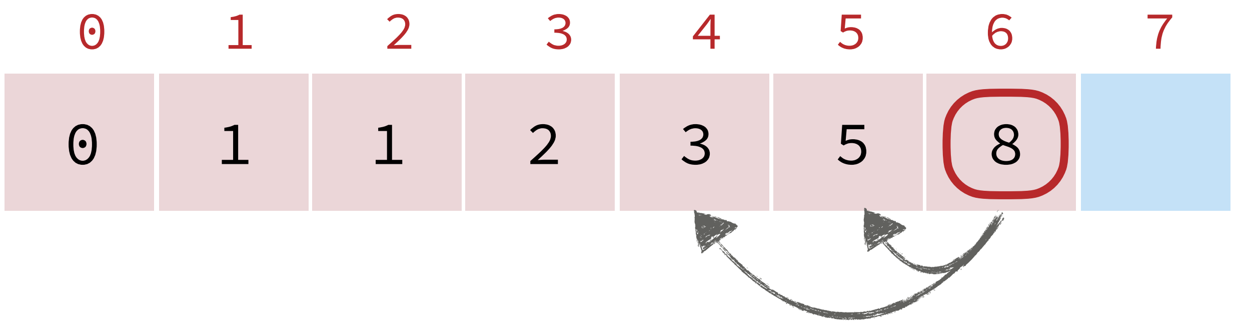
```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems
and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	

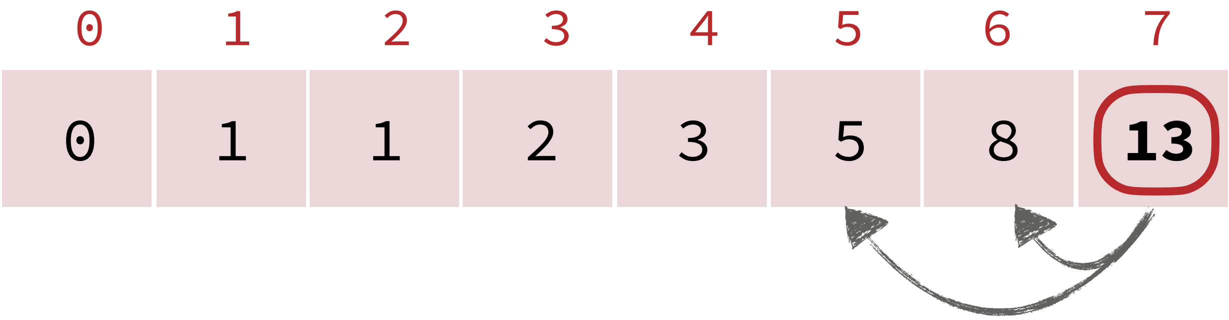
```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems
and move up to larger ones

Bottom-up Approach

Note. We know that larger subproblems depend on smaller subproblems.

Implication. Solve smaller subproblems *before* larger ones!



0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	**13**

solution found!

```
FIB(n)

Create fib[] of size i+1
fib[0] = 0, fib[1] = 1

for (j = 2 ⟶ n):
    fib[j] = fib[j-1] + fib[j-2]

return fib[n]
```

start from smaller problems
and move up to larger ones

Space Improvement

optional

The bottom-up solution uses $\Theta(n)$ extra space. Can we reduce it to $\Theta(1)$?

Space Improvement

optional

The bottom-up solution uses $\Theta(n)$ extra space. Can we reduce it to $\Theta(1)$?

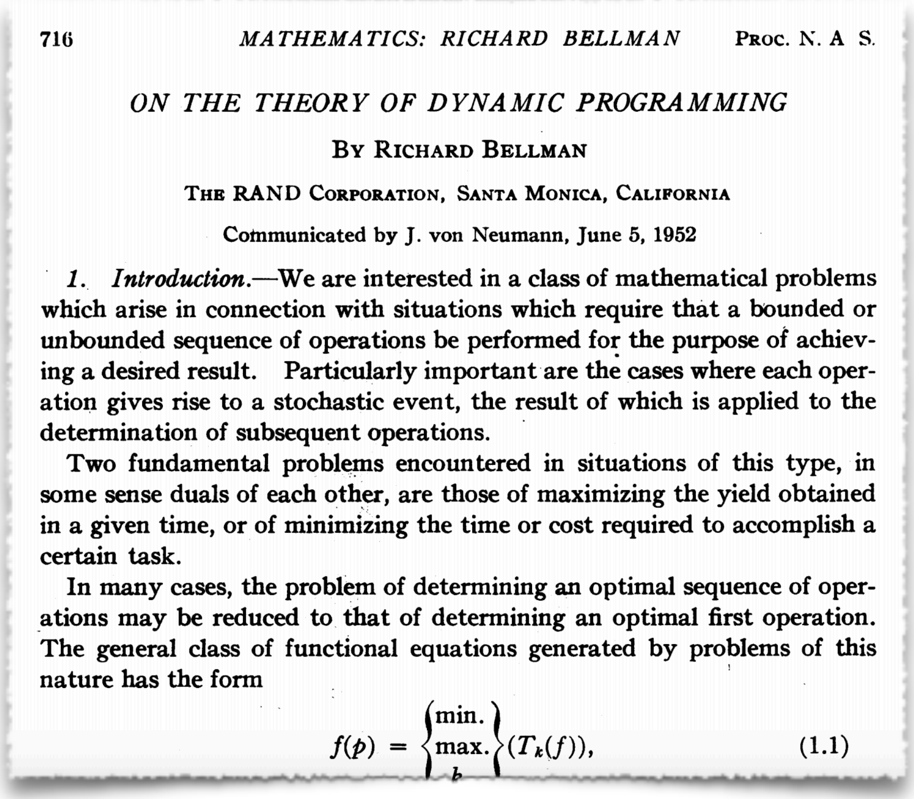
Answer.

```
FIB(i, fib[])

  f = 1, g = 0

  for j=2 ⟶ i:
      f = f + g
      g = f − g

  return f
```

Dynamic Programming

Dynamic Programming

- Introduced by Richard Bellman in 1952.



716 *MATHEMATICS: RICHARD BELLMAN* PROC. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

BY RICHARD BELLMAN

THE RAND CORPORATION, SANTA MONICA, CALIFORNIA

Communicated by J. von Neumann, June 5, 1952

1. *Introduction.*—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

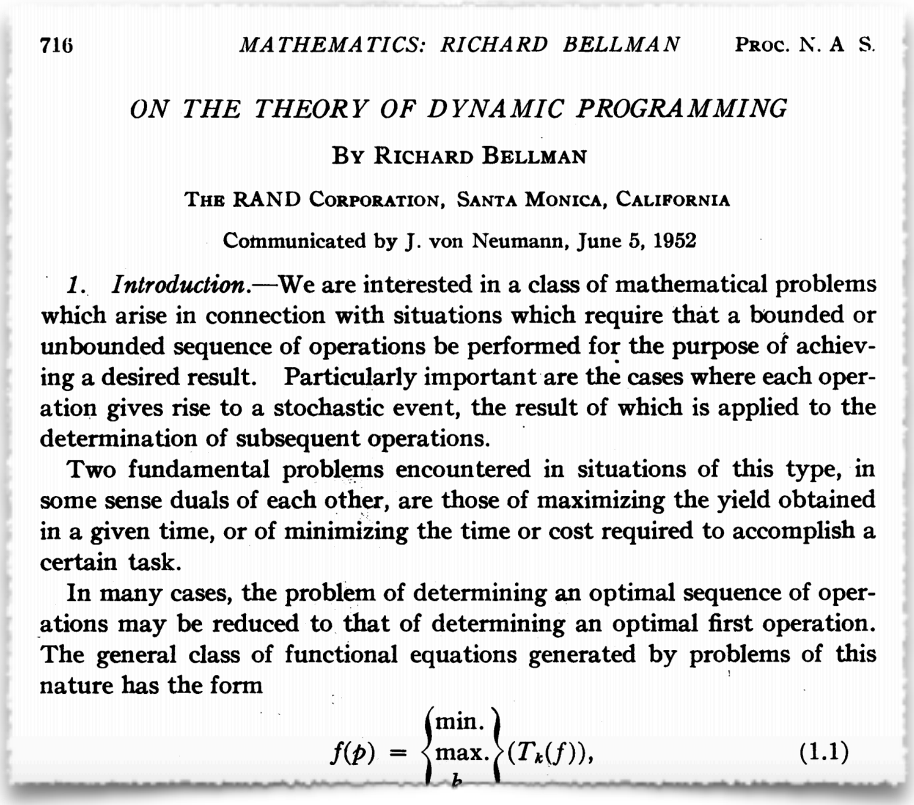
Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

$$f(p) = \begin{Bmatrix}\text{min.}\\ \text{max.}\end{Bmatrix}_{k} (T_k(f)), \qquad (1.1)$$

Dynamic Programming

- Introduced by Richard Bellman in 1952.
- Typically used for solving *optimization problems.*
- In an optimization problem, we aim at optimizing a certain value (e.g. find the shortest path, the maximum return, the minimum number of hours, etc.)



716 *MATHEMATICS: RICHARD BELLMAN* PROC. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

BY RICHARD BELLMAN

THE RAND CORPORATION, SANTA MONICA, CALIFORNIA

Communicated by J. von Neumann, June 5, 1952

1. *Introduction.*—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

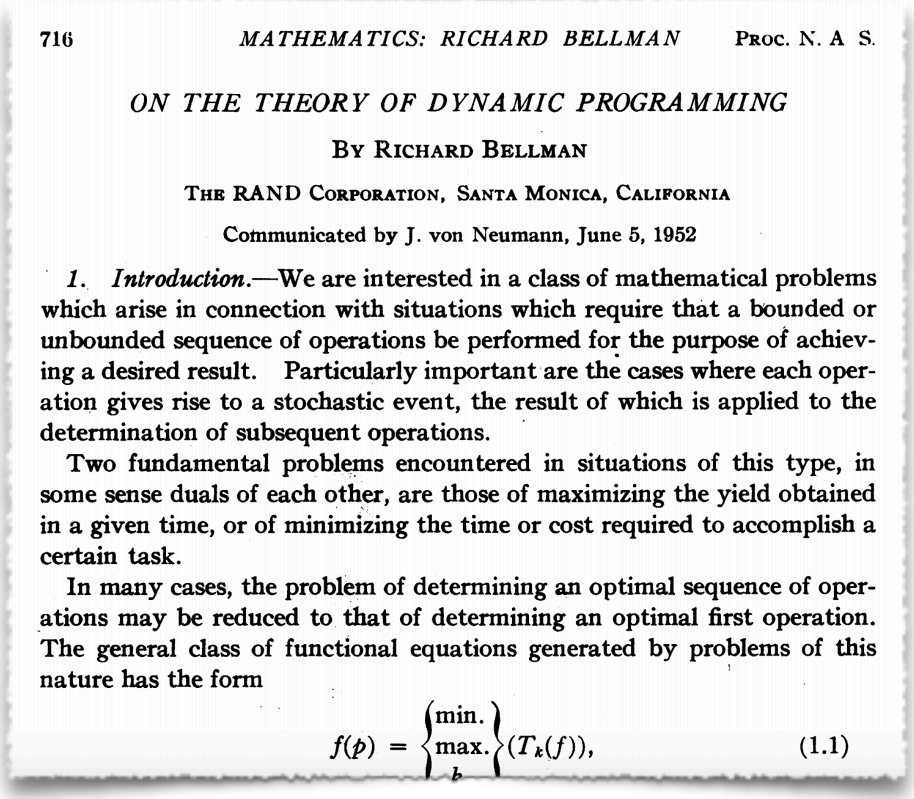
In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

$$f(p) = \begin{Bmatrix} \text{min.} \\ \text{max.}_{k} \end{Bmatrix} (T_k(f)), \qquad (1.1)$$

Dynamic Programming

- Introduced by Richard Bellman in 1952.
- Typically used for solving *optimization problems.*
- In an optimization problem, we aim at optimizing a certain value (e.g. find the shortest path, the maximum return, the minimum number of hours, etc.)

Fibonacci is *not* an example of an optimization problem



716 *MATHEMATICS: RICHARD BELLMAN* PROC. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

BY RICHARD BELLMAN

THE RAND CORPORATION, SANTA MONICA, CALIFORNIA

Communicated by J. von Neumann, June 5, 1952

1. *Introduction.*—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

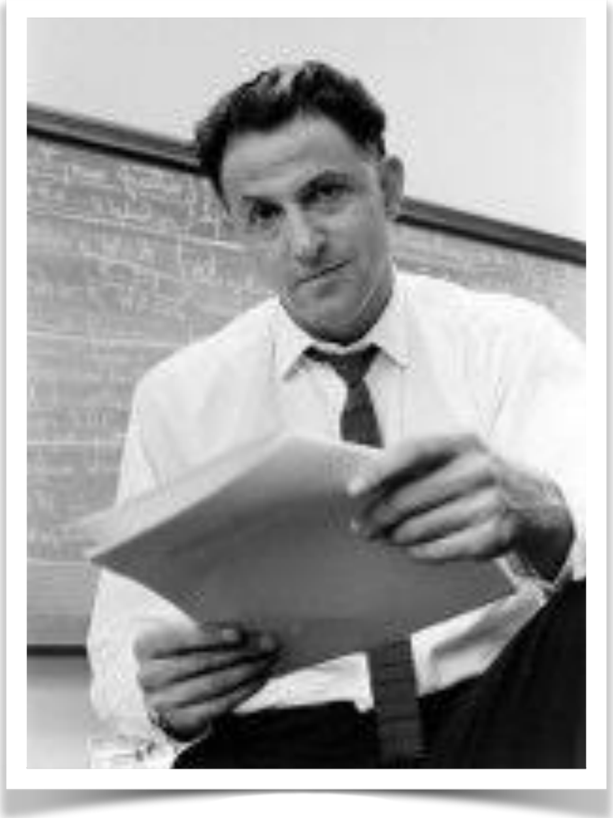
Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

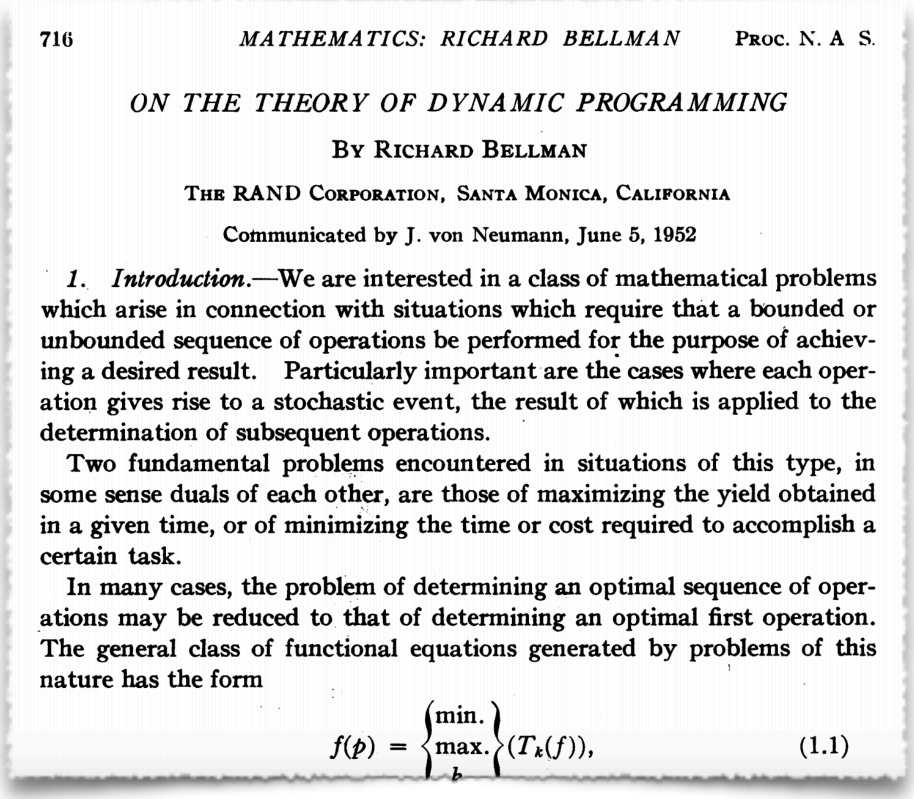
In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

$$f(p) = \begin{Bmatrix}\text{min.}\\ \text{max.}\end{Bmatrix}_{k} (T_k(f)), \qquad (1.1)$$

Dynamic Programming

- Introduced by Richard Bellman in 1952.
- Typically used for solving *optimization problems.*
- In an optimization problem, we aim at optimizing a certain value (e.g. find the shortest path, the maximum return, the minimum number of hours, etc.)
- **Main Steps:**
  - Identify the optimal substructure in the problem.
    Identify how the optimal solution for smaller problems can be used to find the optimal solution for larger problems.





716 *MATHEMATICS: RICHARD BELLMAN* Proc. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

By Richard Bellman

The RAND Corporation, Santa Monica, California

Communicated by J. von Neumann, June 5, 1952

1. *Introduction.*—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

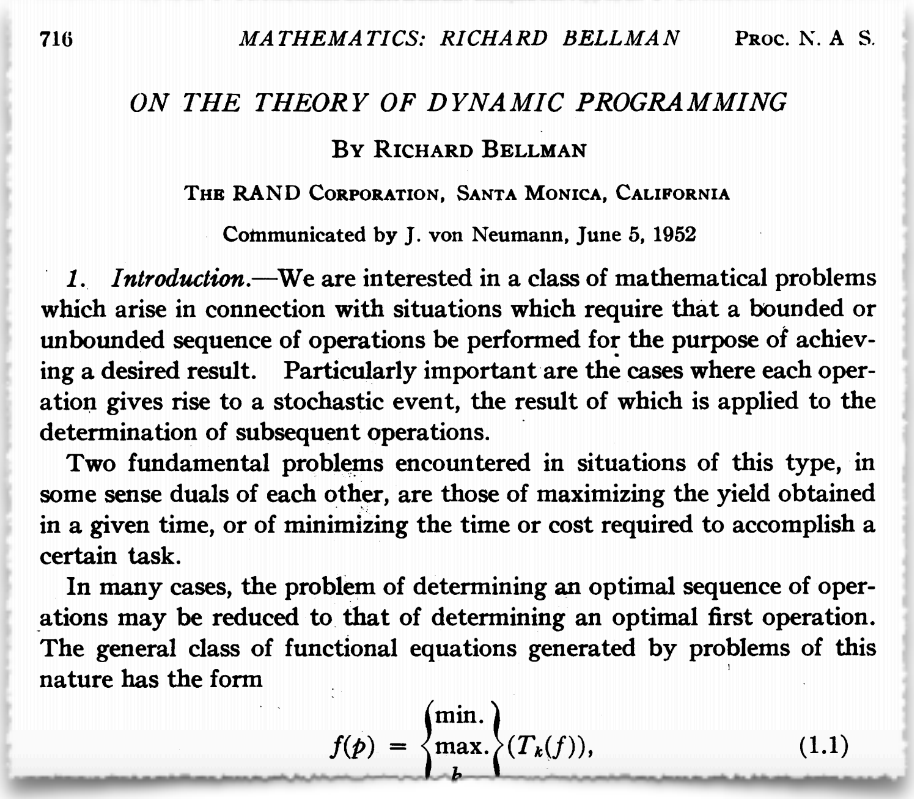
Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

$$f(p) = \left\{\begin{matrix}\text{min.}\\ \text{max.}\end{matrix}\right\}_k (T_k(f)), \qquad (1.1)$$

Dynamic Programming

- Introduced by Richard Bellman in 1952.
- Typically used for solving *optimization problems.*
- In an optimization problem, we aim at optimizing a certain value (e.g. find the shortest path, the maximum return, the minimum number of hours, etc.)
- **Main Steps:**
  - Identify the optimal substructure in the problem.
    Identify how the optimal solution for smaller problems can be used to find the optimal solution for larger problems.
  - Store the results for each solved subproblem to avoid recomputing it again.



716 *MATHEMATICS: RICHARD BELLMAN* PROC. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

BY RICHARD BELLMAN

THE RAND CORPORATION, SANTA MONICA, CALIFORNIA

Communicated by J. von Neumann, June 5, 1952

1. *Introduction.*—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

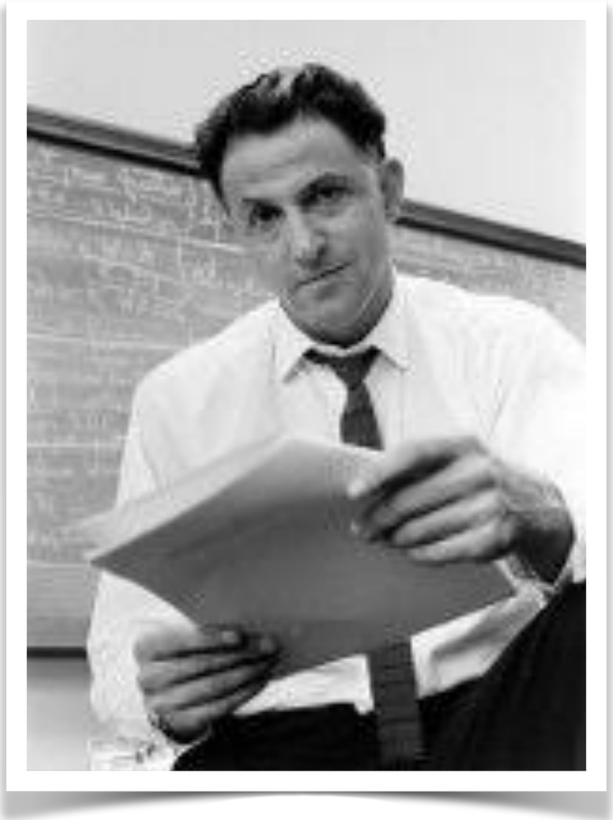
In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

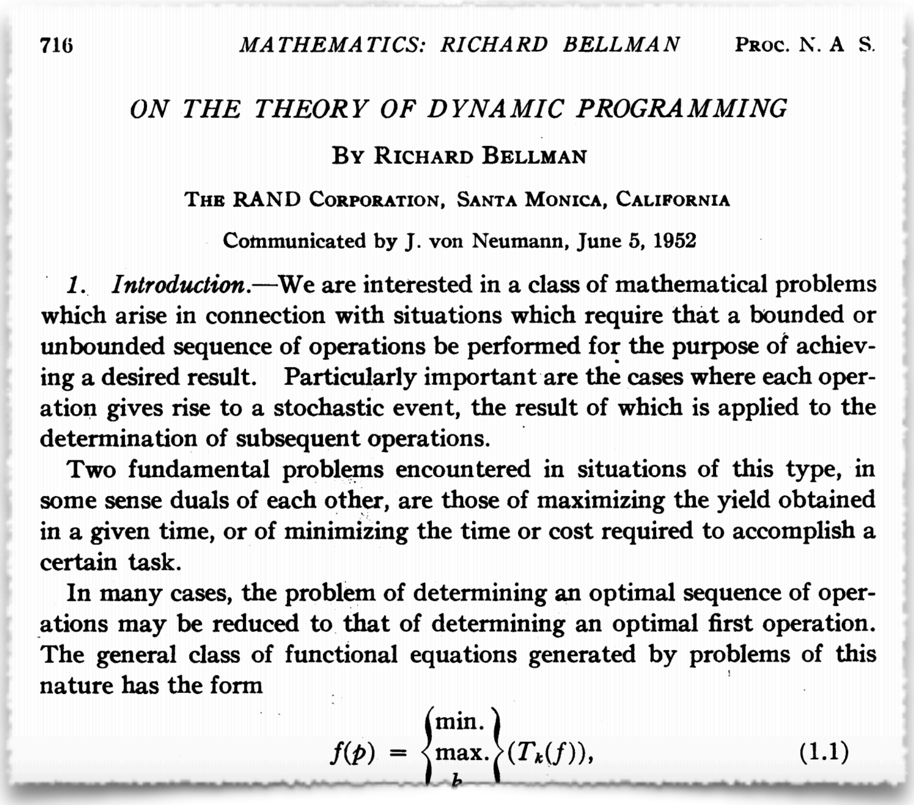
$$f(p) = \left\{\begin{matrix}\text{min.}\\ \text{max.}\end{matrix}\right\}_k (T_k(f)), \qquad (1.1)$$

Dynamic Programming

- Introduced by Richard Bellman in 1952.
- Typically used for solving *optimization problems.*
- In an optimization problem, we aim at optimizing a certain value (e.g. find the shortest path, the maximum return, the minimum number of hours, etc.)
- **Main Steps:**
  - Identify the optimal substructure in the problem.
    Identify how the optimal solution for smaller problems can be used to find the optimal solution for larger problems.
  - Store the results for each solved subproblem to avoid recomputing it again.

If there are no overlapping subproblems, the solution becomes a normal divide-and-conquer solution.





716 *MATHEMATICS: RICHARD BELLMAN* Proc. N. A. S.

ON THE THEORY OF DYNAMIC PROGRAMMING

By Richard Bellman

The RAND Corporation, Santa Monica, California

Communicated by J. von Neumann, June 5, 1952

1. Introduction.—We are interested in a class of mathematical problems which arise in connection with situations which require that a bounded or unbounded sequence of operations be performed for the purpose of achieving a desired result. Particularly important are the cases where each operation gives rise to a stochastic event, the result of which is applied to the determination of subsequent operations.

Two fundamental problems encountered in situations of this type, in some sense duals of each other, are those of maximizing the yield obtained in a given time, or of minimizing the time or cost required to accomplish a certain task.

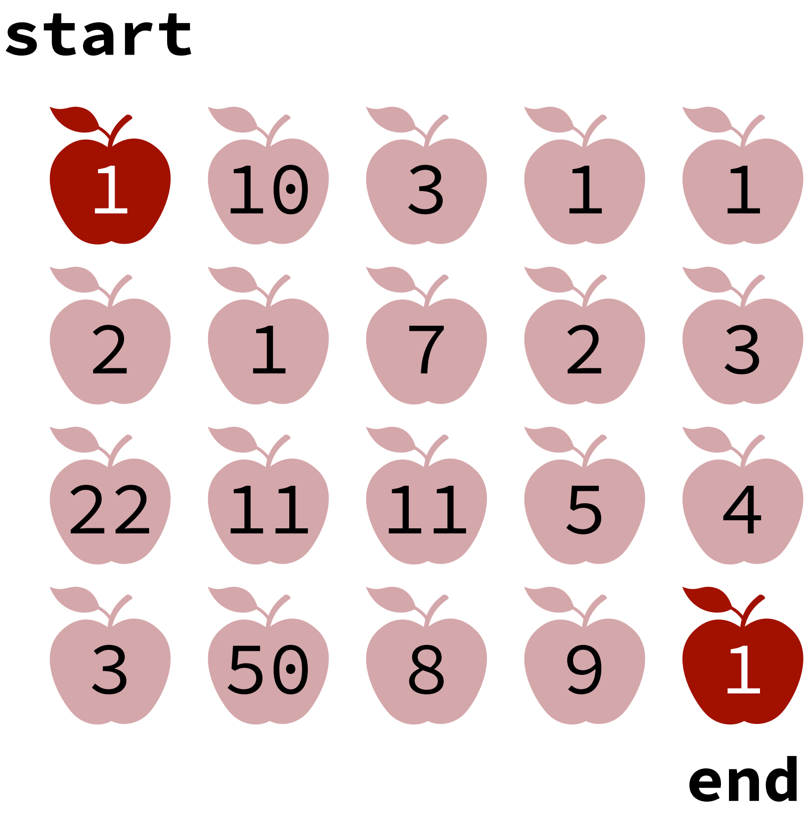
In many cases, the problem of determining an optimal sequence of operations may be reduced to that of determining an optimal first operation. The general class of functional equations generated by problems of this nature has the form

$$f(p) = \begin{Bmatrix} \text{min.} \\ \text{max.} \end{Bmatrix}_k (T_k(f)), \qquad (1.1)$$

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.



start

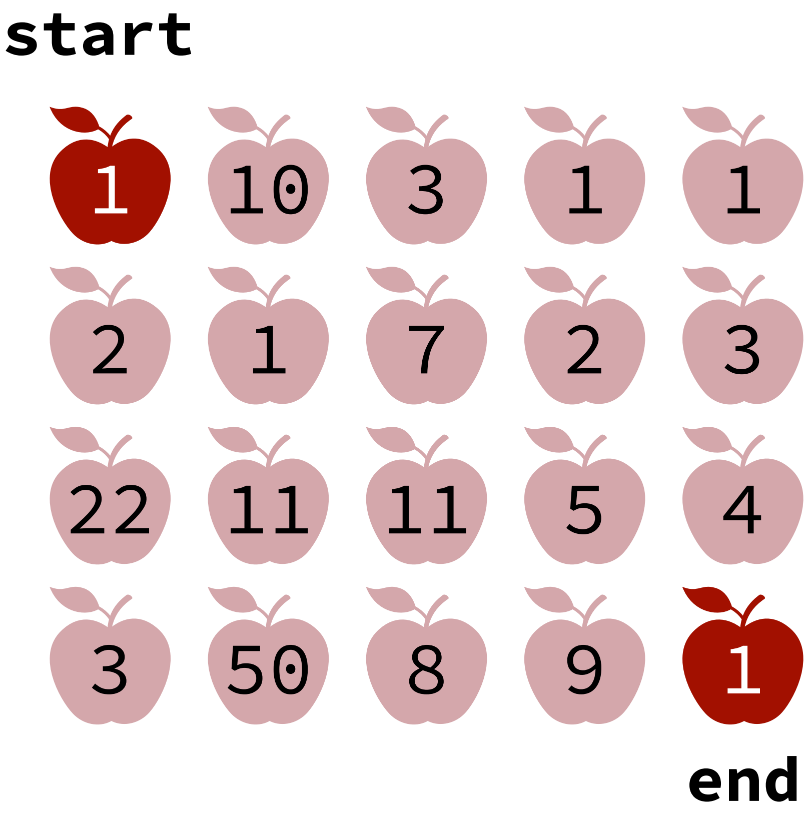
1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number
  of apples at cell `[i][j]`.



start

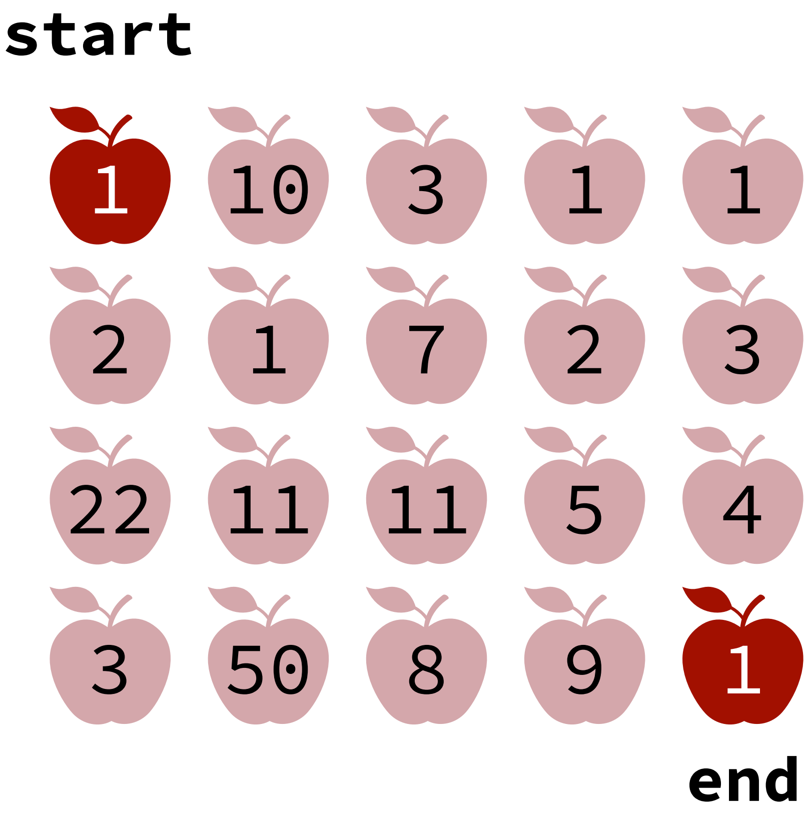
1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

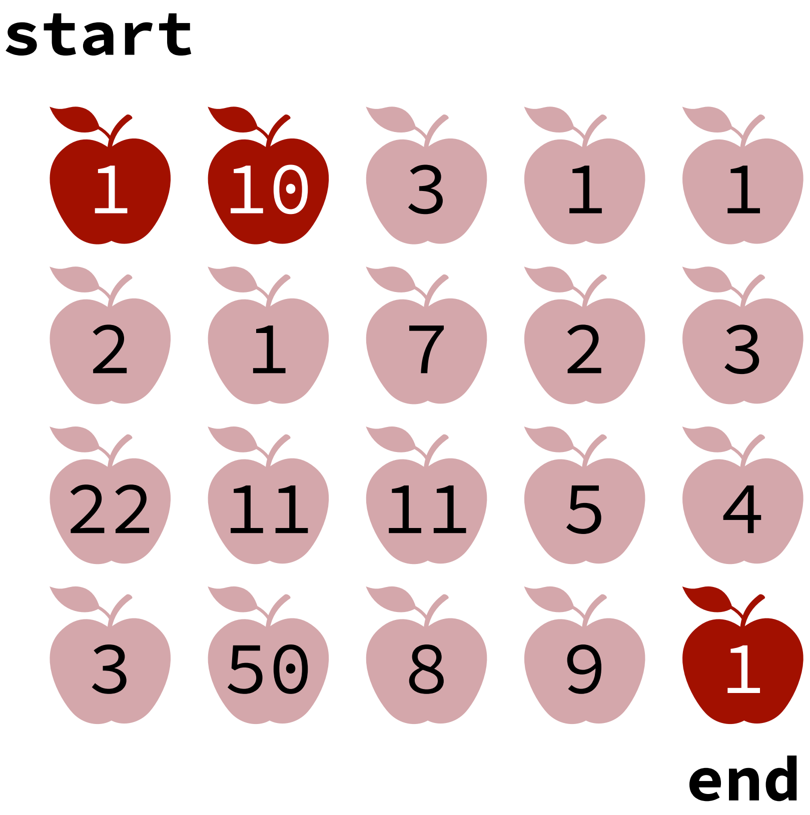
Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

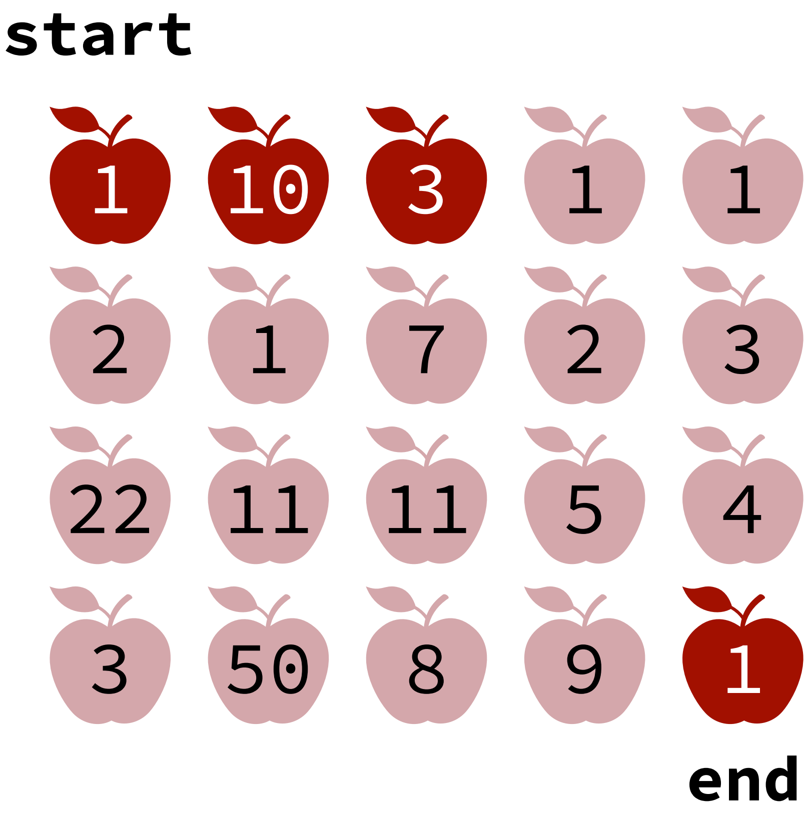
Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number
  of apples at cell `[i][j]`.



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

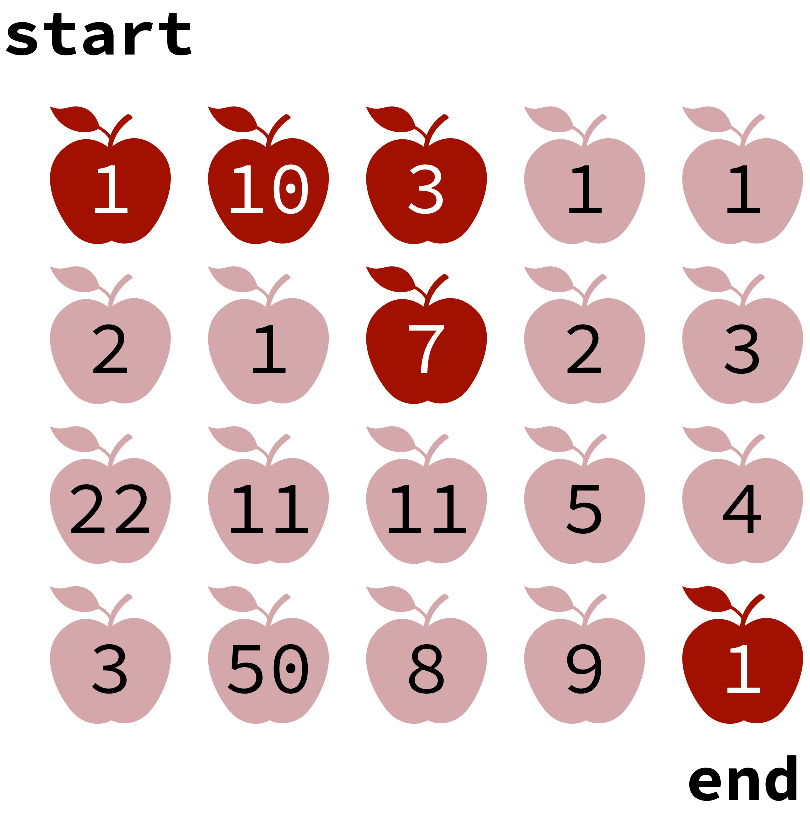
Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

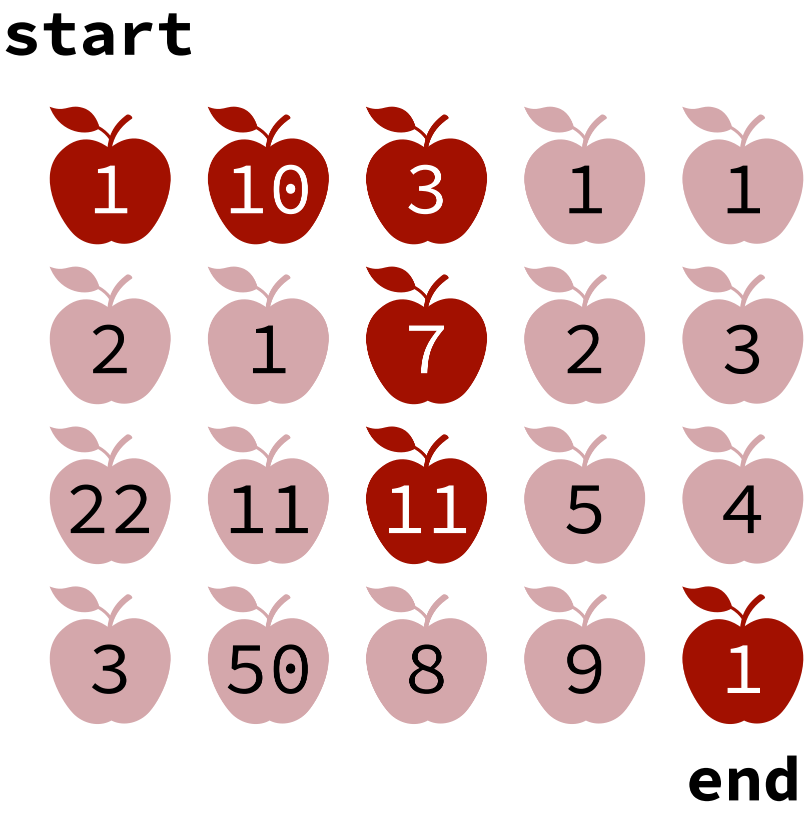
Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

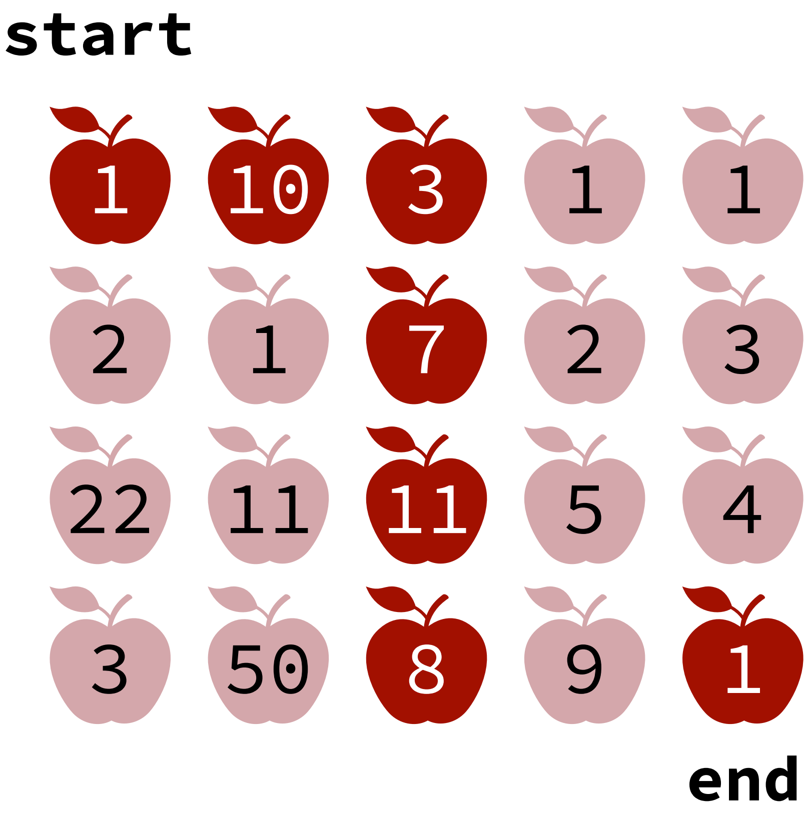
Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Solution # 1. Repeat the following until the goal is reached.

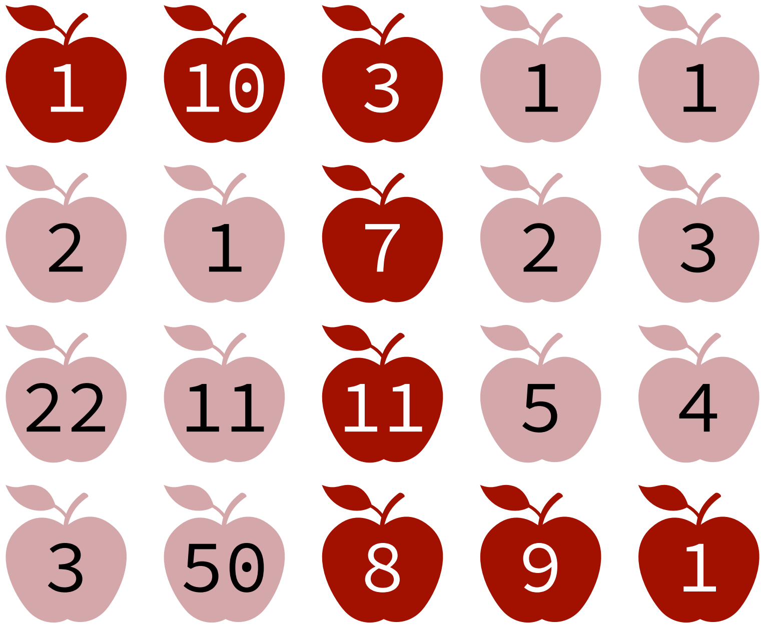
```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Total = 50

Solution # 1. Repeat the following until the goal is reached.

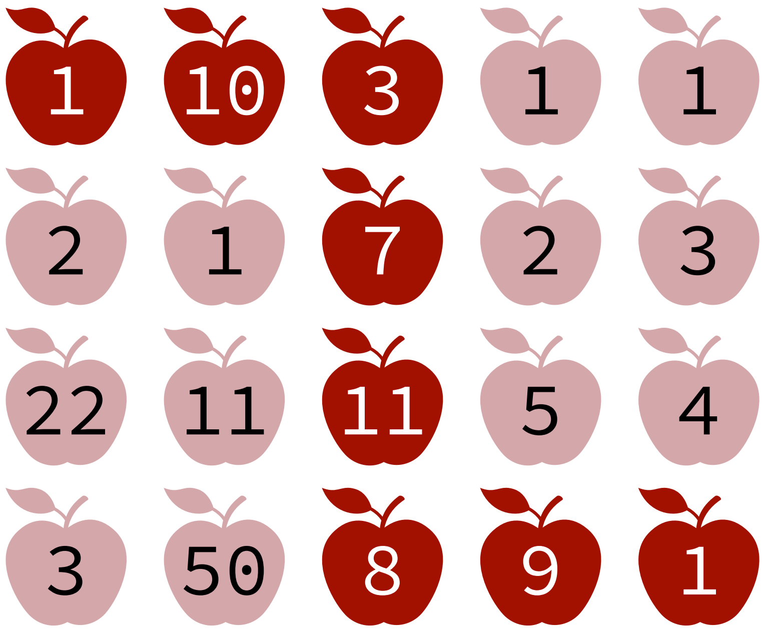
```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

Example: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Total = 50

Solution # 1. Repeat the following until the goal is reached.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

🤔 **Can we do better?**

Example: Collecting Apples

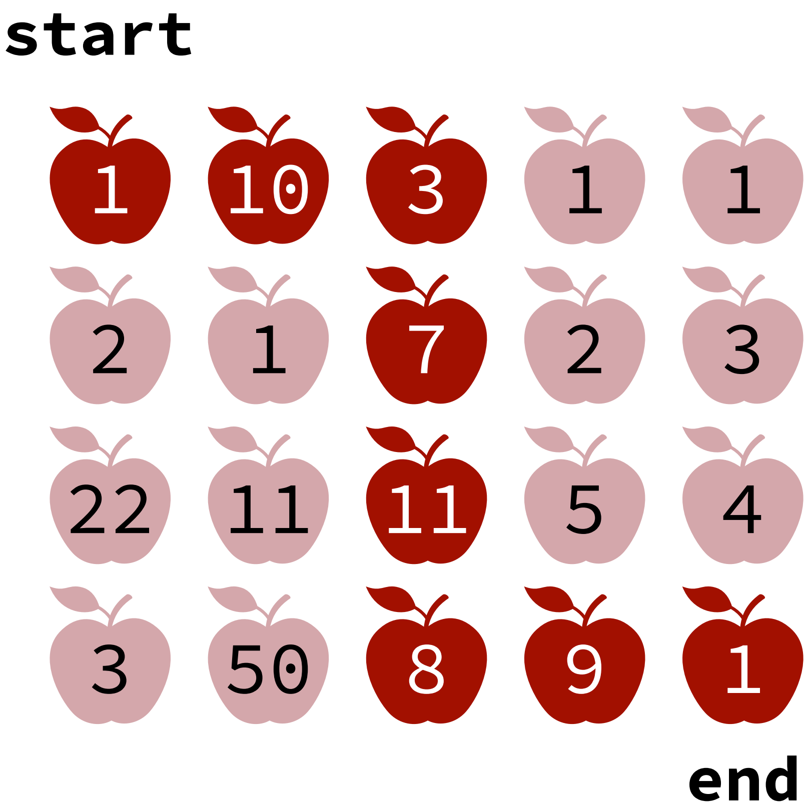
Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

Solution # 1.

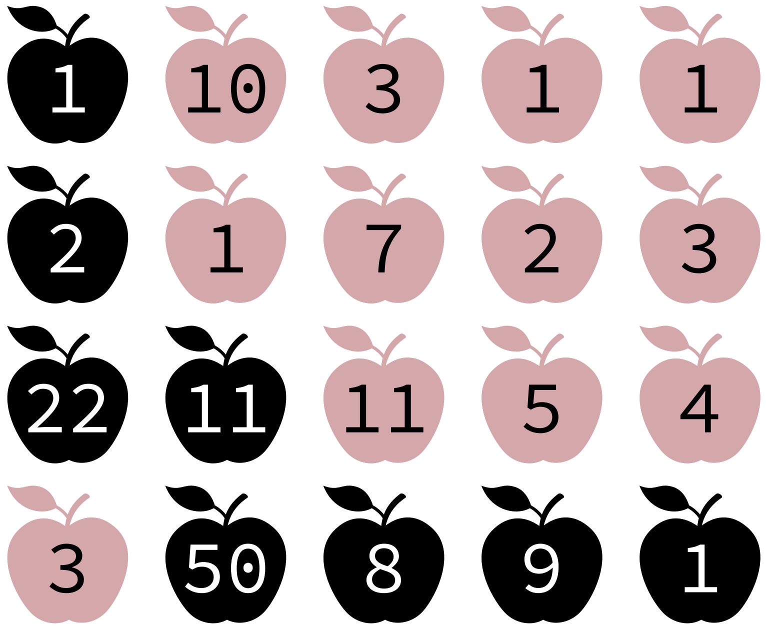
```
if apples[i+1][j] > apples[i][j+1]:
      go down.
else go right.
```

FAIL



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Total = 50



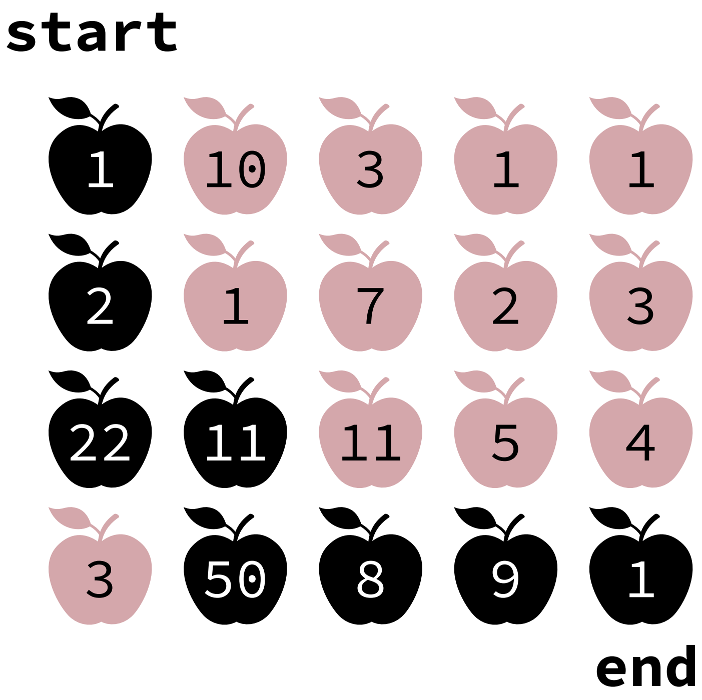
1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Total = 104

Example: Collecting Apples

Let:

- **`max_apples`**`(i, j)` = maximum number of apples that can be collected from `[0][0]` to `[i][j]`

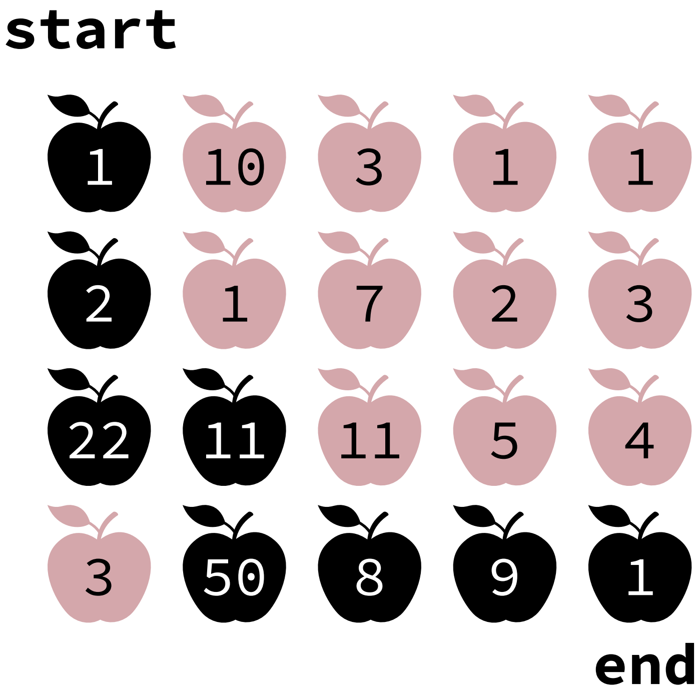


1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Let:

- **`max_apples`**`(i, j)` = maximum number of apples that can be collected from `[0][0]` to `[i][j]`
- **`max_apples`**`(N-1, M-1)` = The problem to be solved.



start

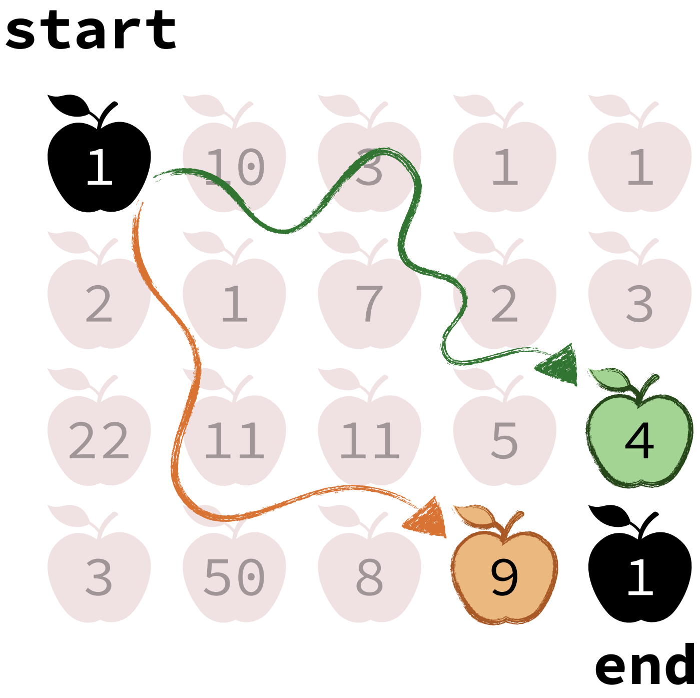
1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Let:

- **`max_apples`**`(i, j)` = maximum number of apples that can be collected from `[0][0]` to `[i][j]`
- **`max_apples`**`(N-1, M-1)` = The problem to be solved.

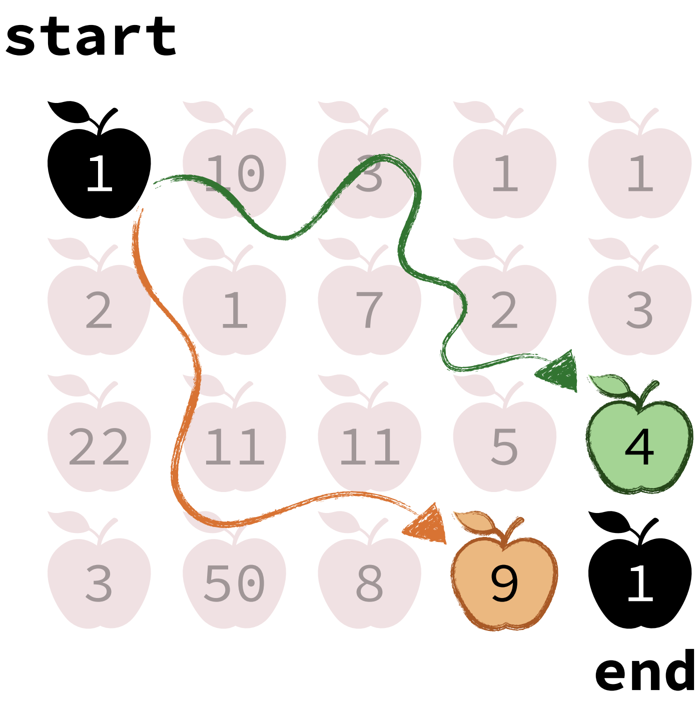


1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Observations.

- The path to the final cell can only come from the cell *above* or the cell to its *left*.
- If we know the best solution to these two cells, we know the best solution to the final cell!

Example: Collecting Apples



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

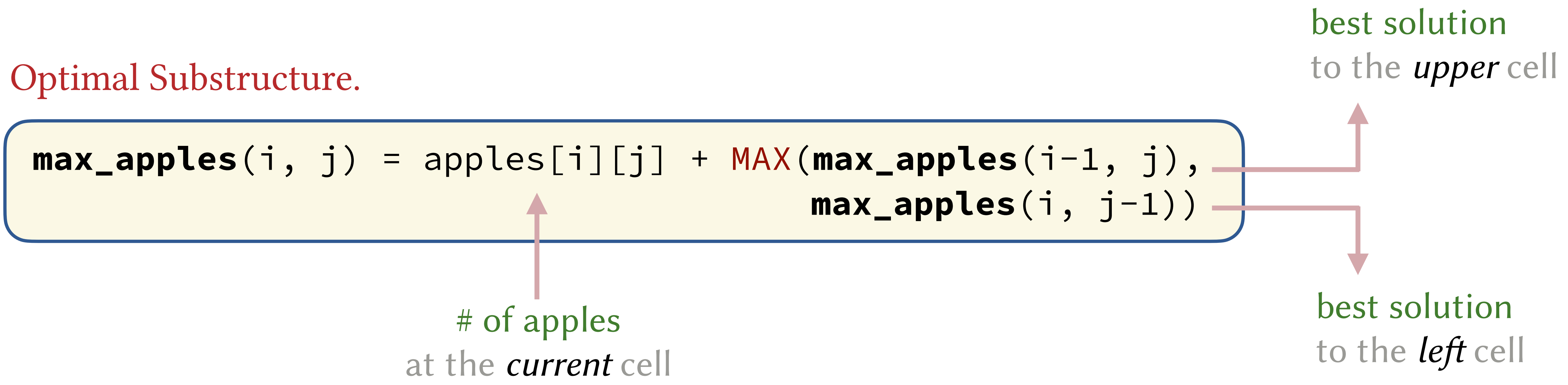
Let:

- **`max_apples`**`(i, j)` = maximum number of apples that can be collected from `[0][0]` to `[i][j]`
- **`max_apples`**`(N-1, M-1)` = The problem to be solved.

Observations.

- The path to the final cell can only come from the cell *above* or the cell to its *left*.
- If we know the best solution to these two cells, we know the best solution to the final cell!

Optimal Substructure.



```
max_apples(i, j) = apples[i][j] + MAX(max_apples(i-1, j),
                                      max_apples(i, j-1))
```

best solution
to the *upper* cell

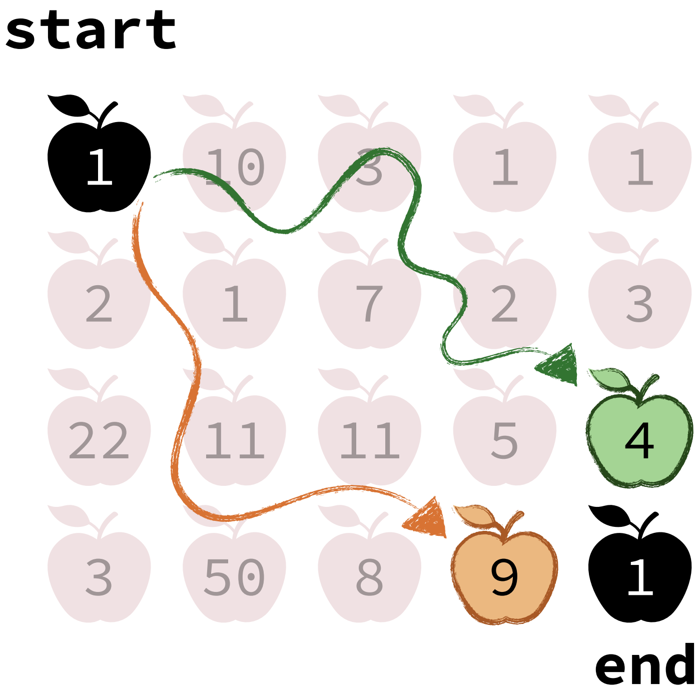
# of apples
at the *current* cell

best solution
to the *left* cell

Example: Collecting Apples

Optimal Substructure.

```
max_apples(i, j) = apples[i][j] +
                   MAX(max_apples(i-1, j),
                       max_apples(i, j-1))
```



Recursive Solution

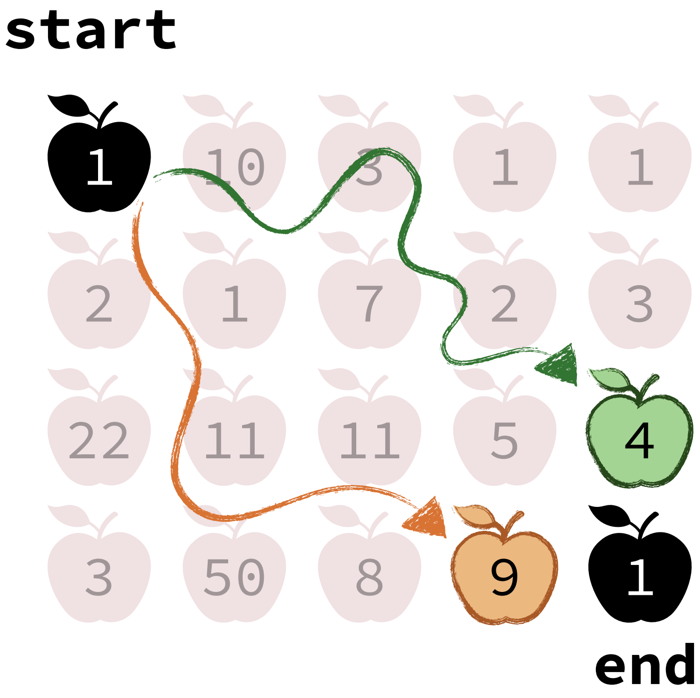
```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]
```

base case

Example: Collecting Apples

Optimal Substructure.

```
max_apples(i, j) = apples[i][j] +
                   MAX(max_apples(i-1, j),
                       max_apples(i, j-1))
```



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)
```

guard against
corner cases

Example: Collecting Apples

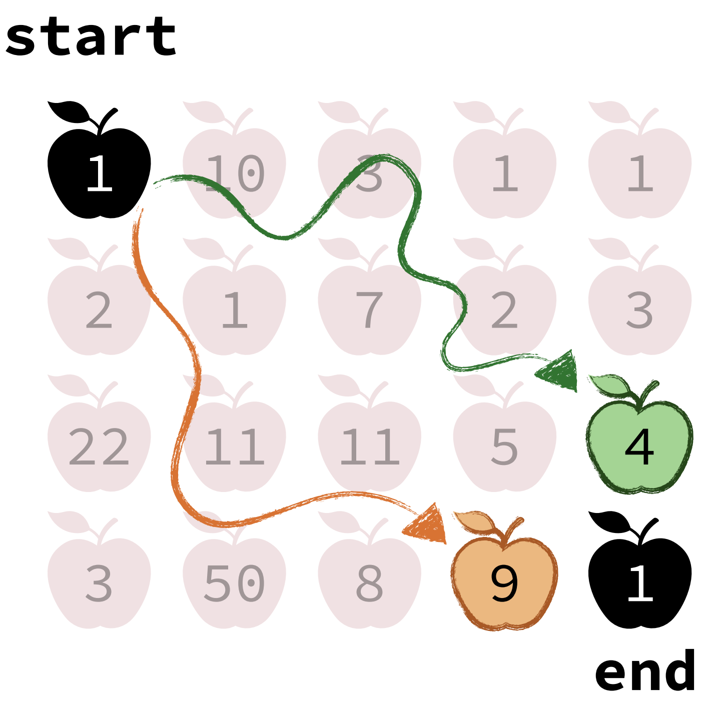
Optimal Substructure.

```
max_apples(i, j) = apples[i][j] +
                   MAX(max_apples(i-1, j),
                       max_apples(i, j-1))
```

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)
```

Recursively solve the needed subproblems



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

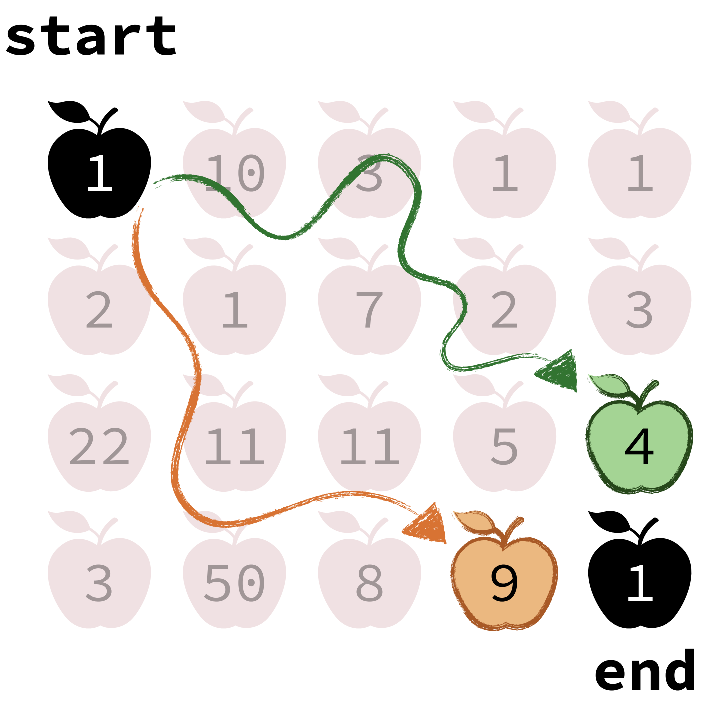
Optimal Substructure.

```
max_apples(i, j) = apples[i][j] +
                   MAX(max_apples(i-1, j),
                       max_apples(i, j-1))
```

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```

Combine the results of the two subproblems



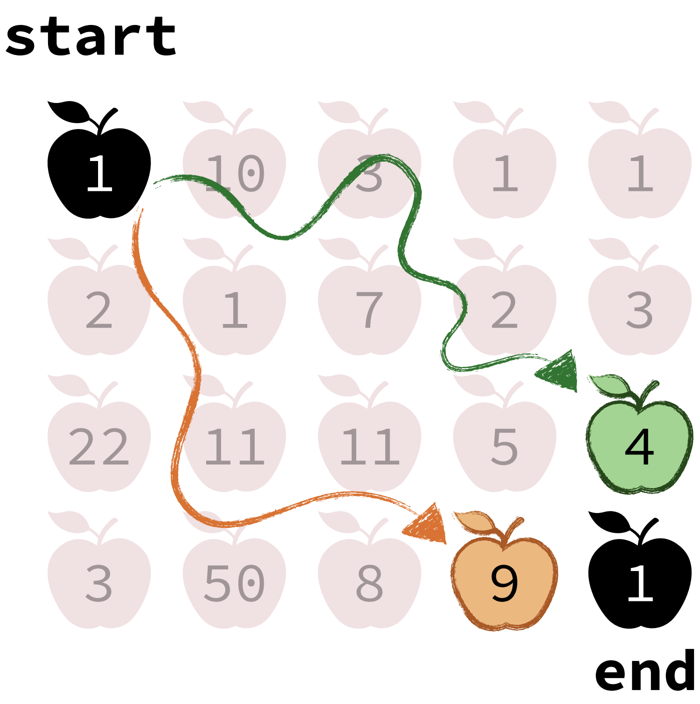
Example: Collecting Apples

Optimal Substructure.

```
max_apples(i, j) = apples[i][j] +
                   MAX(max_apples(i-1, j),
                       max_apples(i, j-1))
```

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```

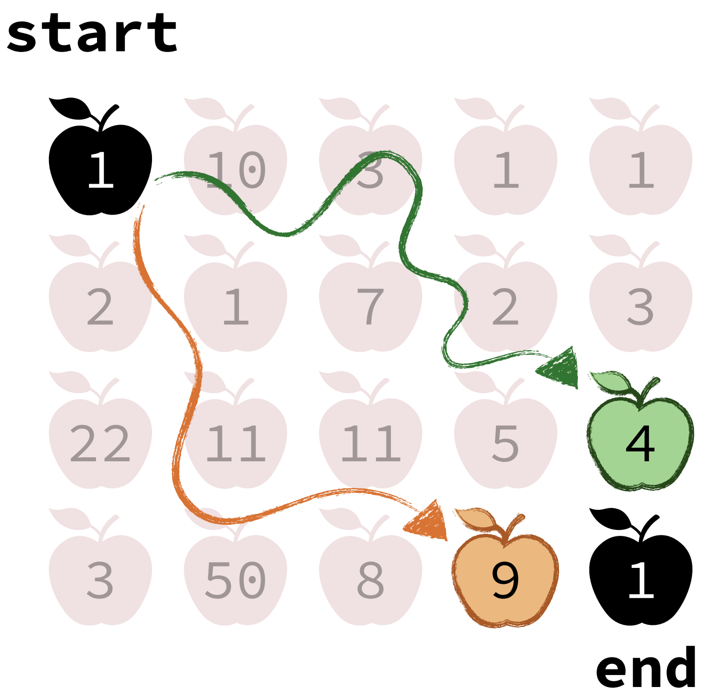


1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

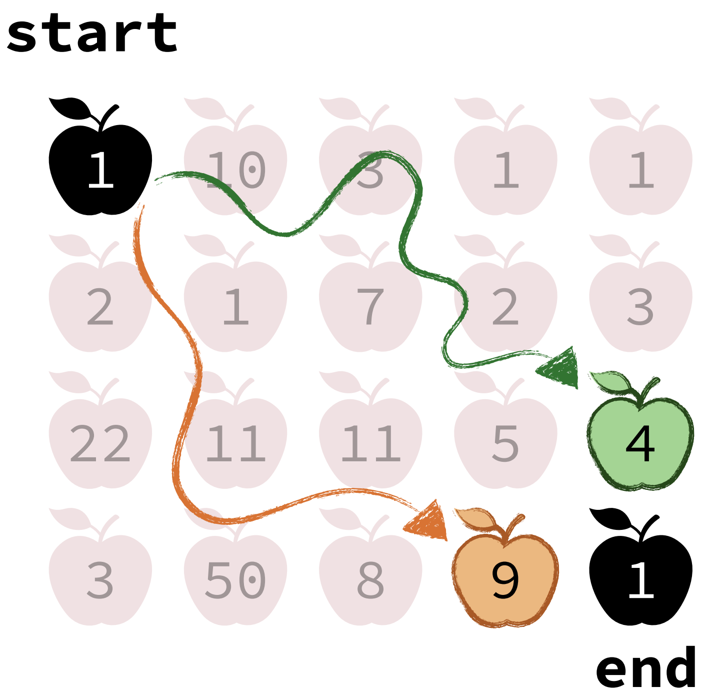
Example Trace

MAX_APPLES(5, 5)

Example: Collecting Apples

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```

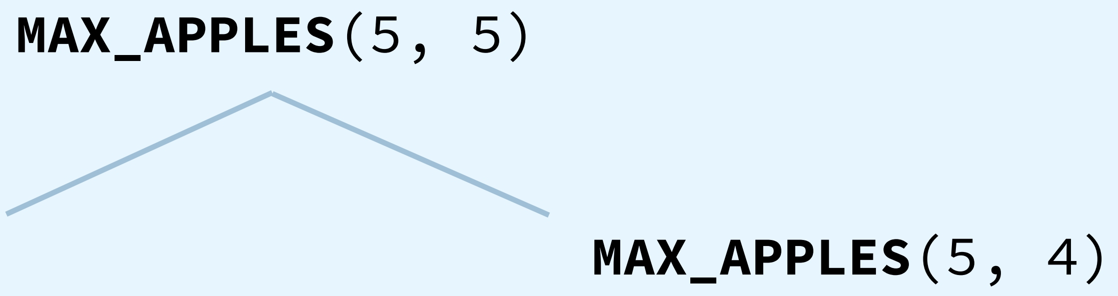


start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example Trace



MAX_APPLES(5, 5)

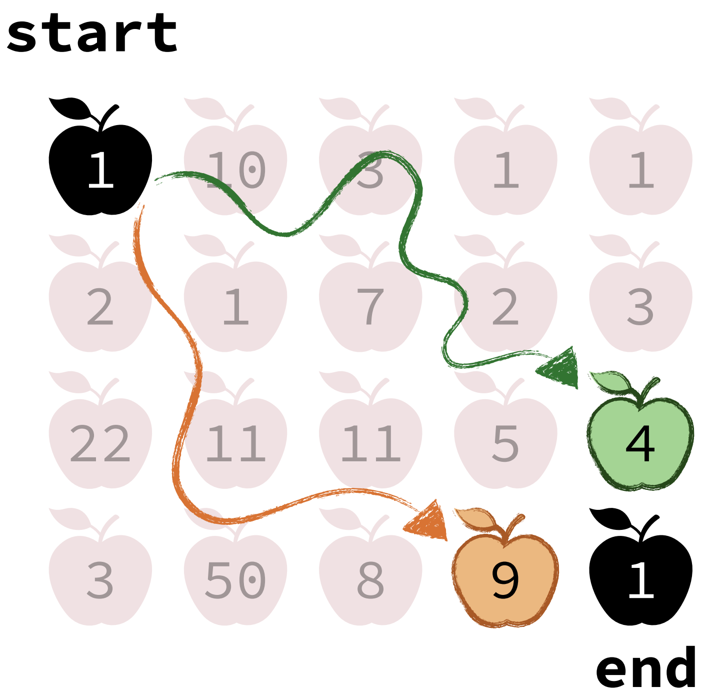
MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

Example: Collecting Apples

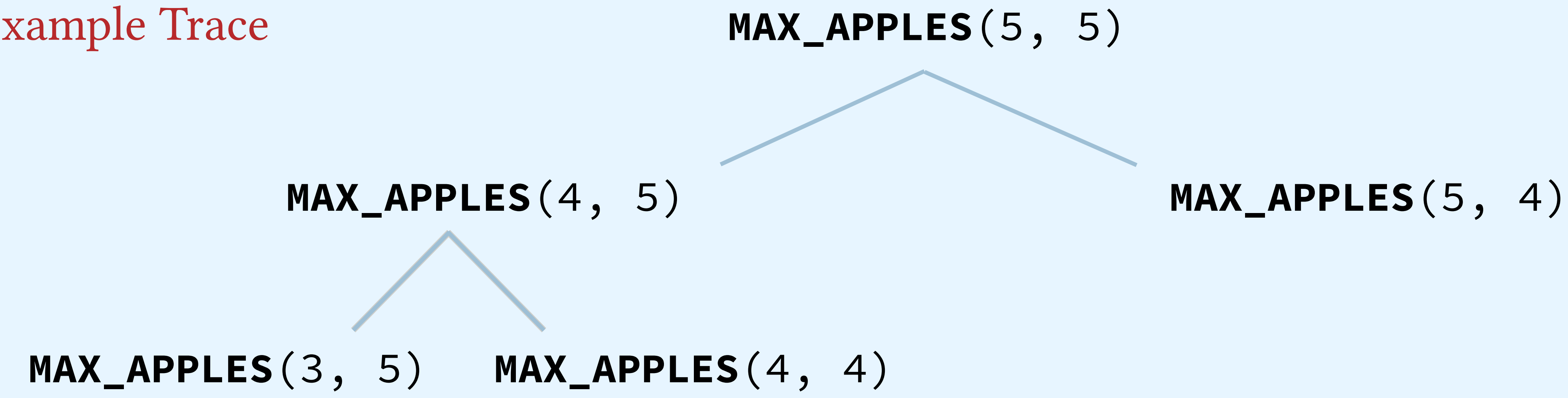
Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example Trace



MAX_APPLES(5, 5)

MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

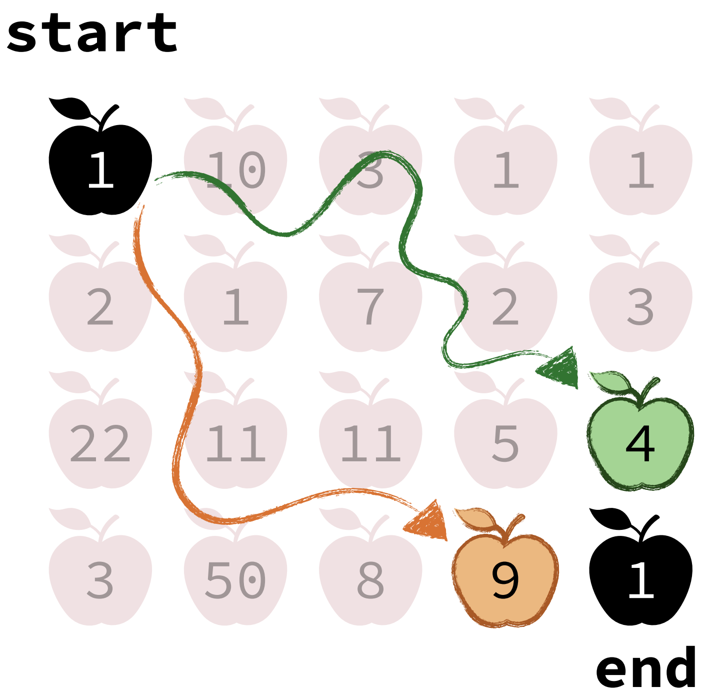
MAX_APPLES(3, 5)

MAX_APPLES(4, 4)

Example: Collecting Apples

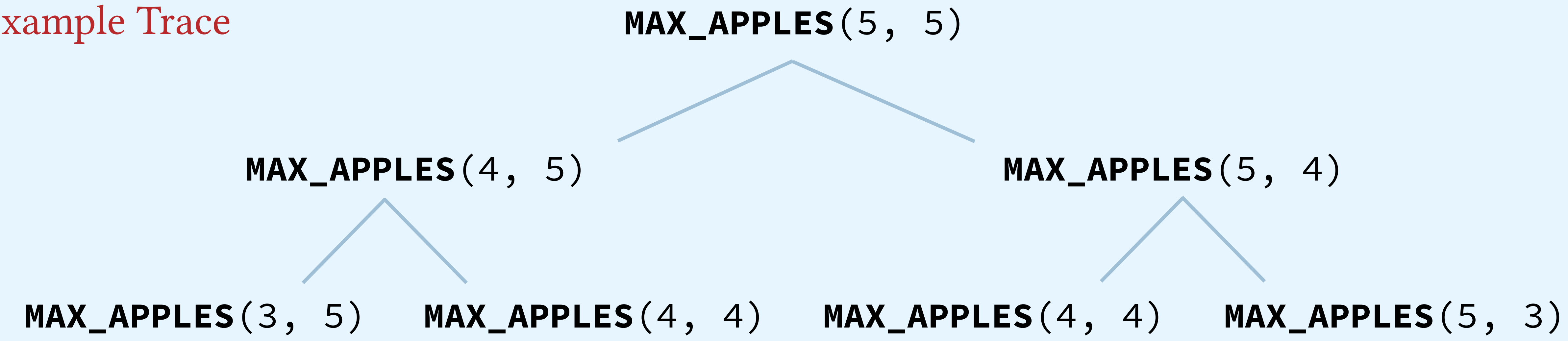
Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example Trace



MAX_APPLES(5, 5)

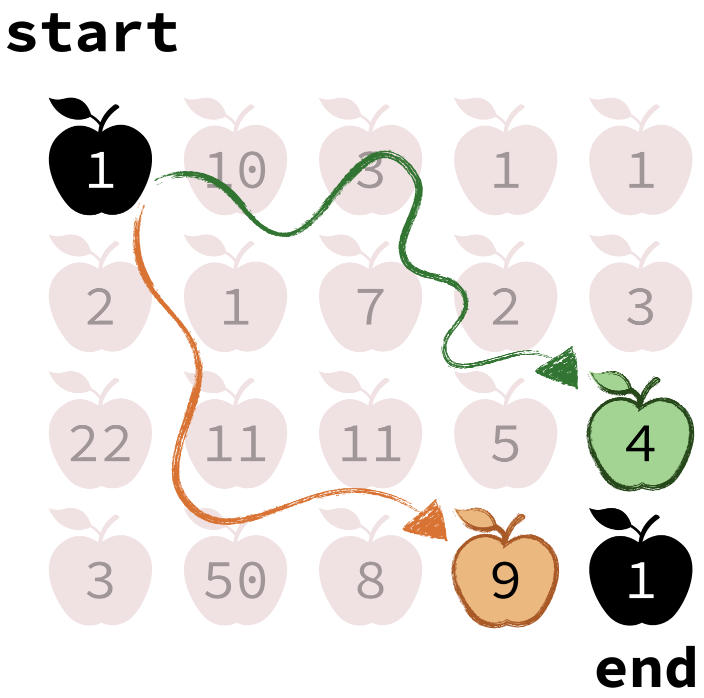
MAX_APPLES(4, 5)   MAX_APPLES(5, 4)

MAX_APPLES(3, 5)   MAX_APPLES(4, 4)   MAX_APPLES(4, 4)   MAX_APPLES(5, 3)

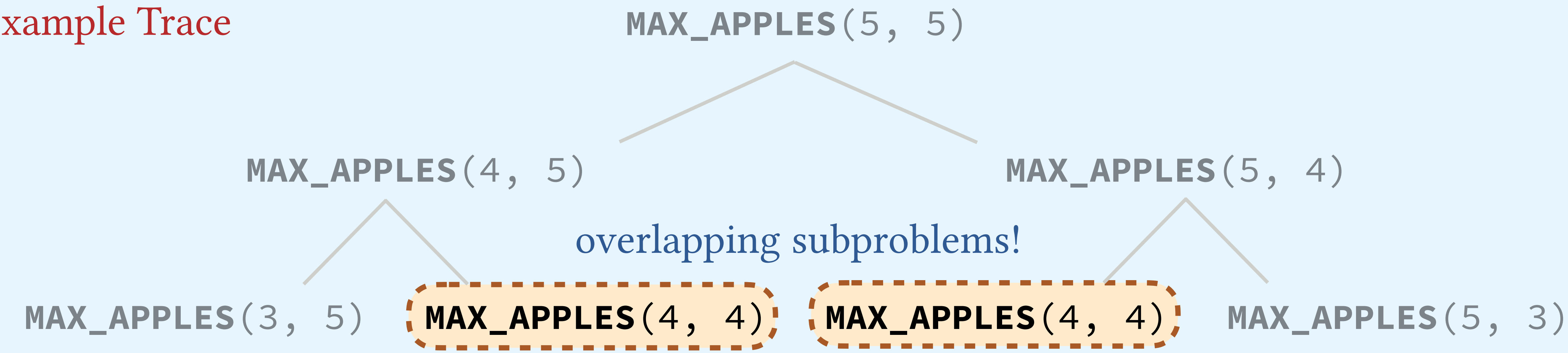
Example: Collecting Apples

Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```



Example Trace



MAX_APPLES(5, 5)

MAX_APPLES(4, 5) MAX_APPLES(5, 4)

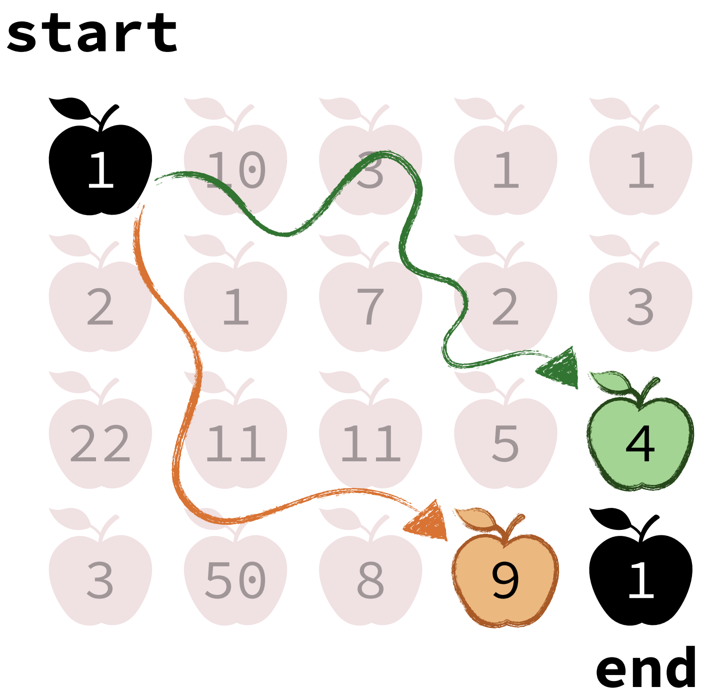
overlapping subproblems!

MAX_APPLES(3, 5) **MAX_APPLES(4, 4)** **MAX_APPLES(4, 4)** MAX_APPLES(5, 3)

Example: Collecting Apples

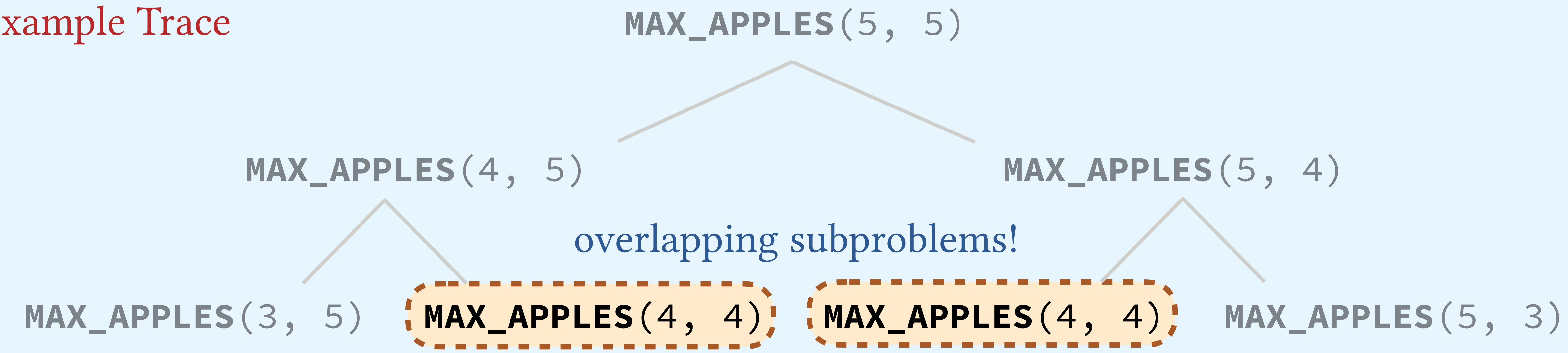
Recursive Solution

```
MAX_APPLES(i, j, apples[])

  if (i == 0 and j == 0): return apples[0][0]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  return apples[i][j] + MAX(max_left, max_up)
```



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example Trace



MAX_APPLES(5, 5)

MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

overlapping subproblems!

MAX_APPLES(3, 5)

MAX_APPLES(4, 4)

MAX_APPLES(4, 4)

MAX_APPLES(5, 3)

Running Time. if $N == M$: $T(N) = O(2^{2N})$ and $\Omega(2^N)$

A binary tree with:
N ≤ # of levels ≤ N+M

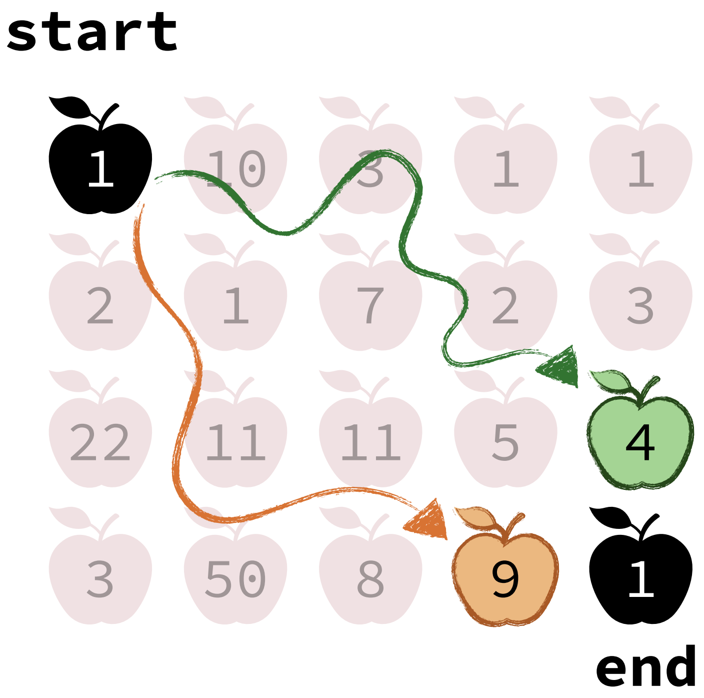
Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])
    create array result[N][M]
```

stores the solution for each subproblem

result[][]



Example: Collecting Apples

Memoized Solution

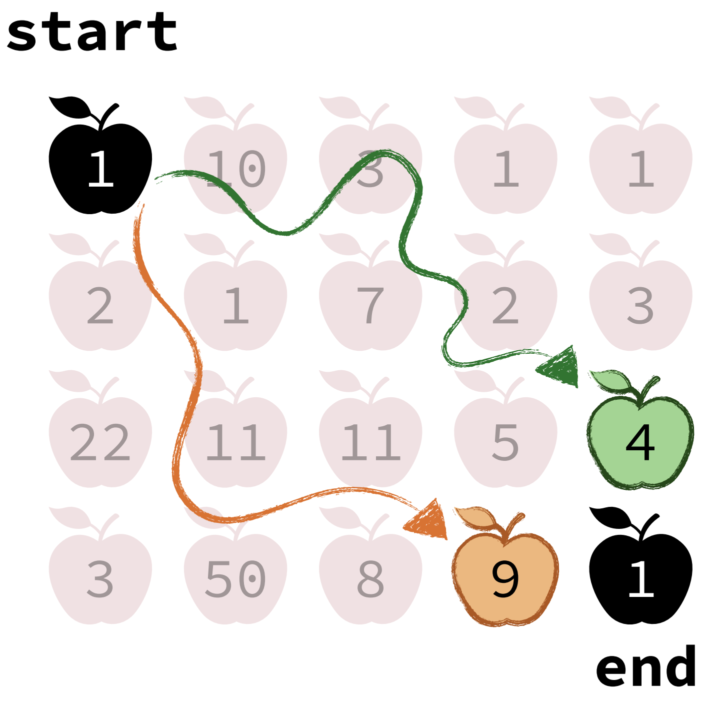
```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]
```

initially, only **MAX_APPLES**(0,0) has a solution

result[][]

1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

stores the solution for each subproblem



Example: Collecting Apples

Memoized Solution

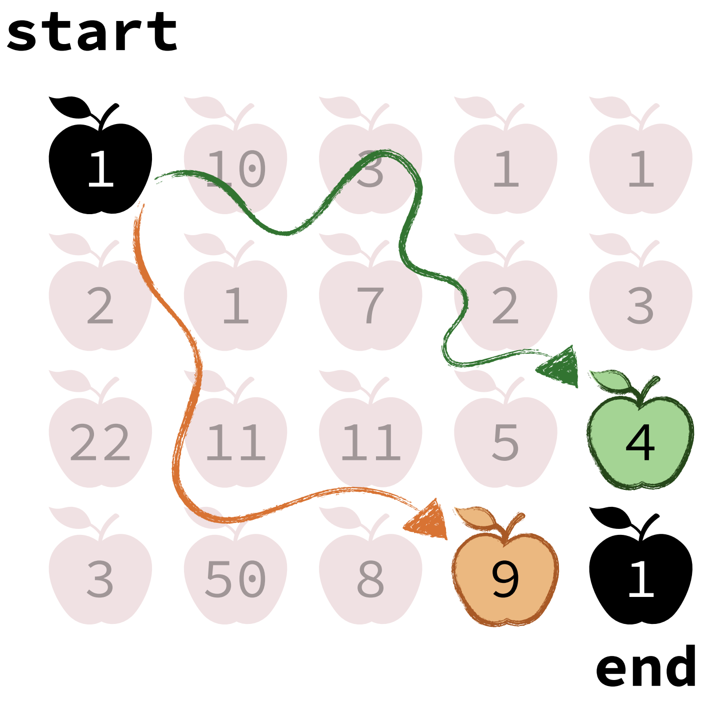
```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
```

fill the table

result[][]

1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

stores the solution for each subproblem



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

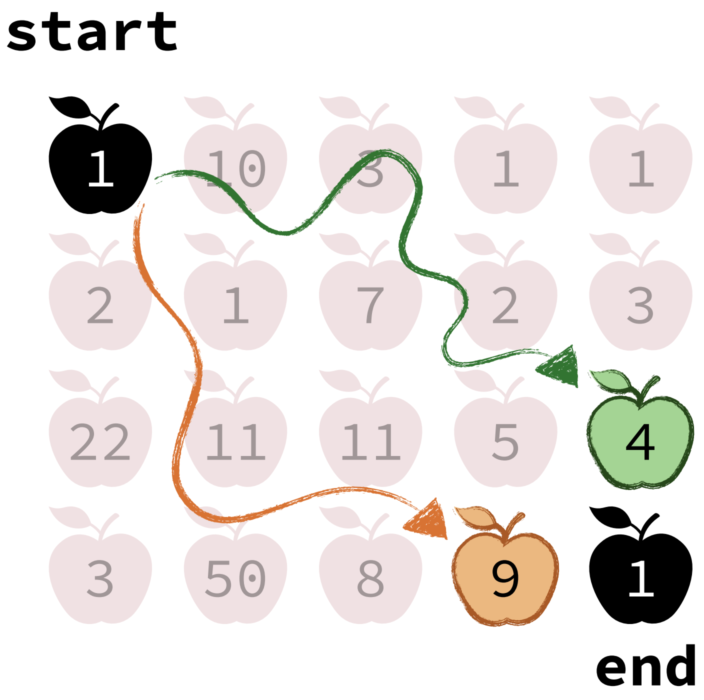
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

result[][]

1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

this is where the final result will be!



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

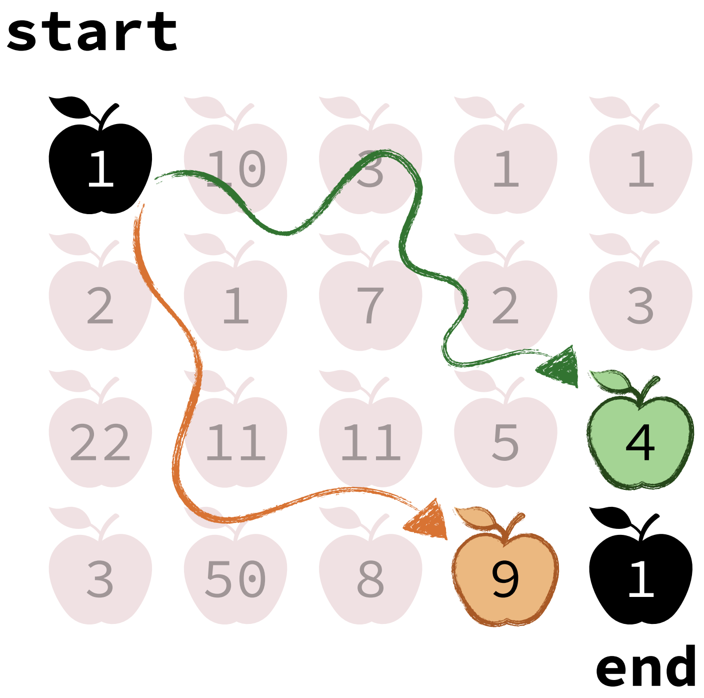
result[][]

1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

this is where the final result will be!

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]
```

base case: if we solved this subproblem before, return the solution!



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

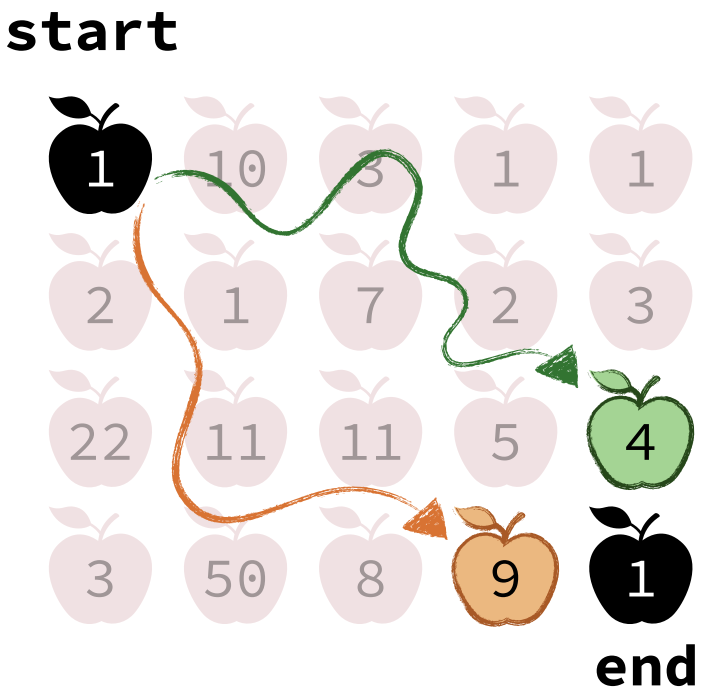
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)
```

recursively solve the needed subproblems and store the result

result[][]

1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

this is where the final result will be!



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Memoized Solution

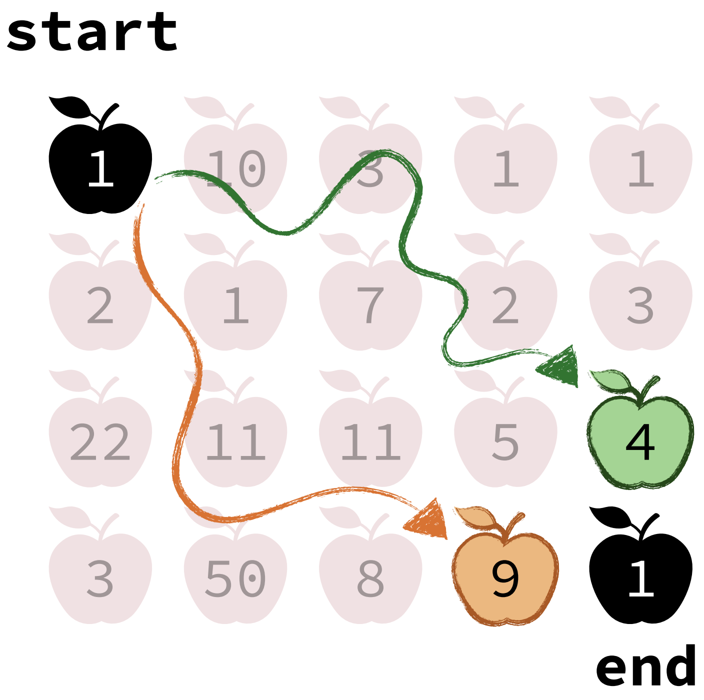
```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

result[][]

1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

this is where the final result will be!



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

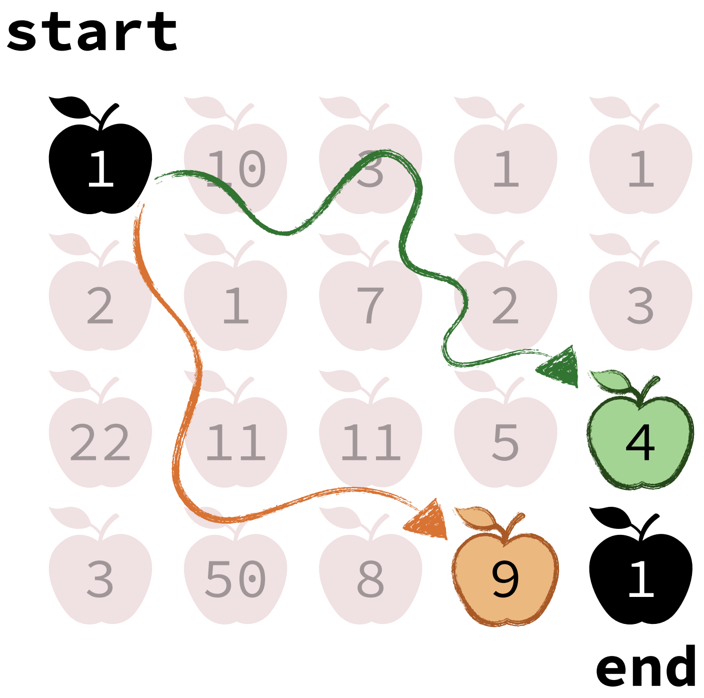
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem was not solved before. We need to solve the *left* and *upper* subproblems



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

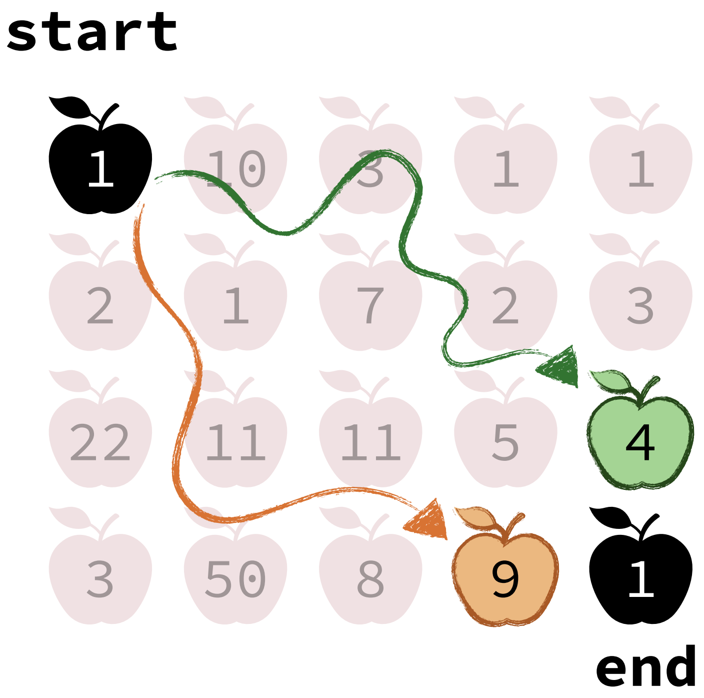
```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

This problem was not solved before. We need to solve the *left* and *upper* subproblems



Example: Collecting Apples

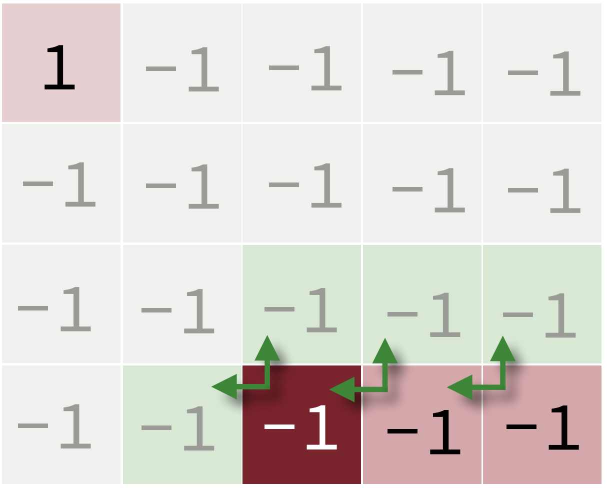
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

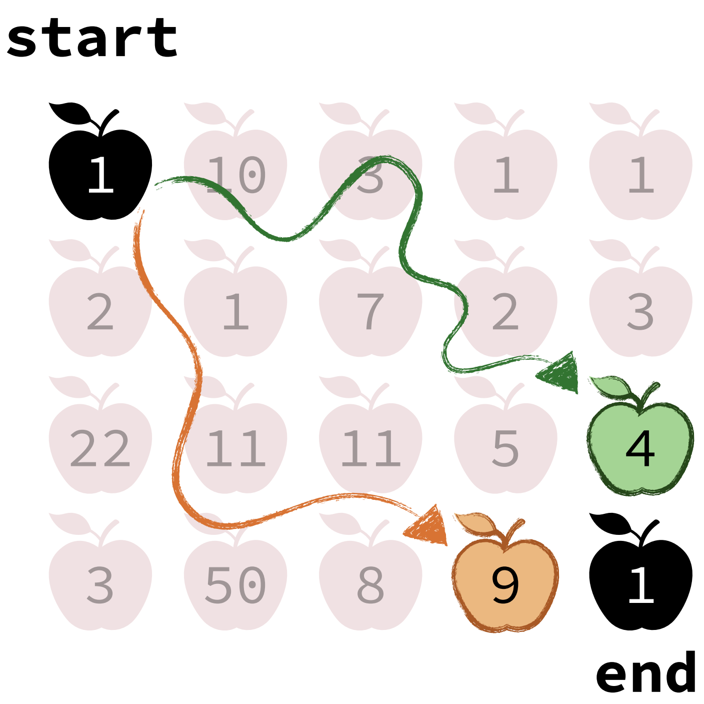
Trace

result[][]



1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

This problem was not solved before. We need to solve the *left* and *upper* subproblems



Example: Collecting Apples

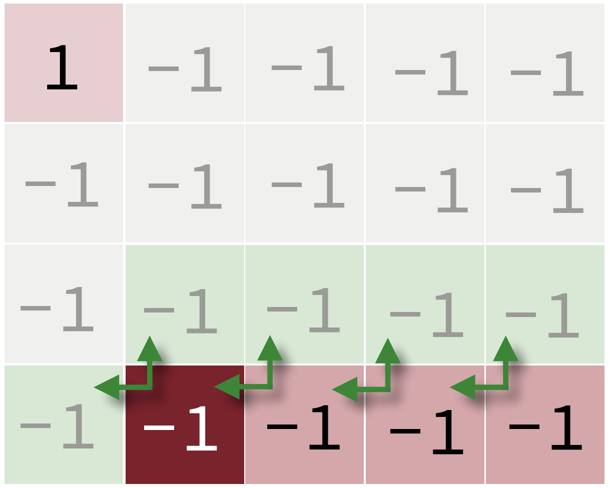
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

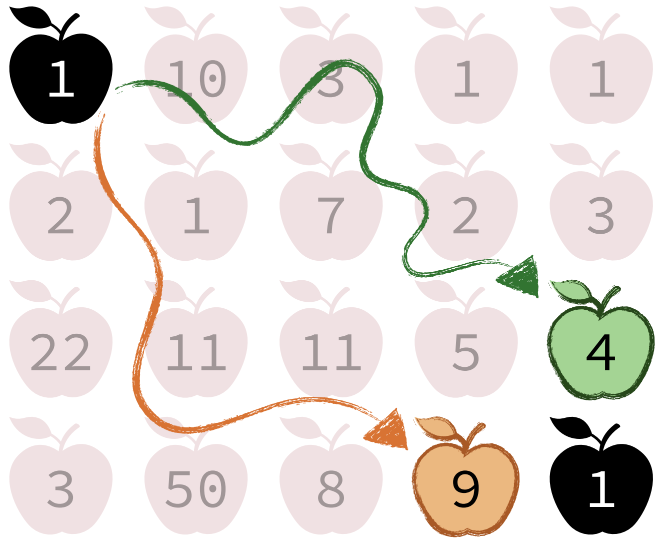
result[][]



1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem was not solved before. We need to solve the *left* and *upper* subproblems

start



end

Example: Collecting Apples

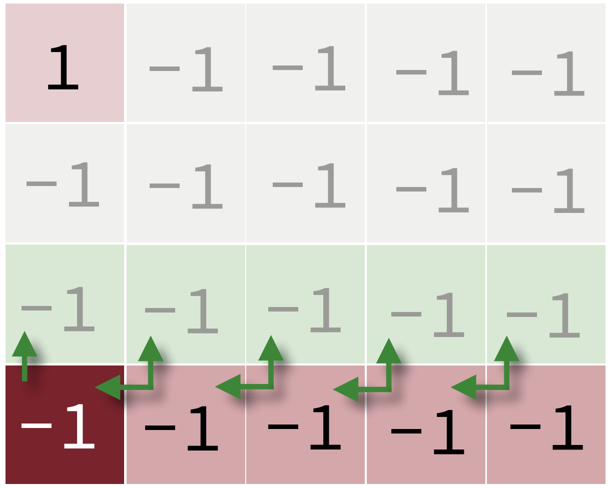
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

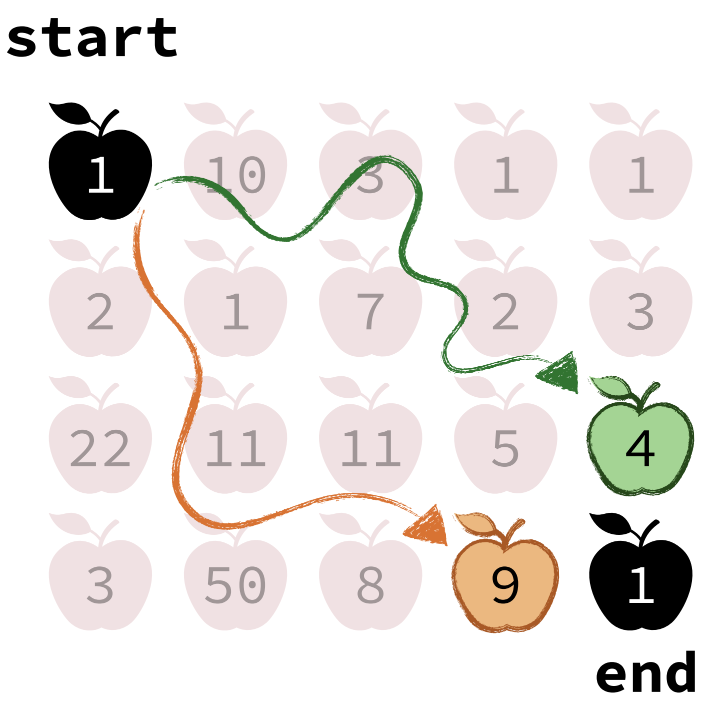
Trace

result[][]



1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

This problem was not solved before. We need to solve the *upper* subproblem



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

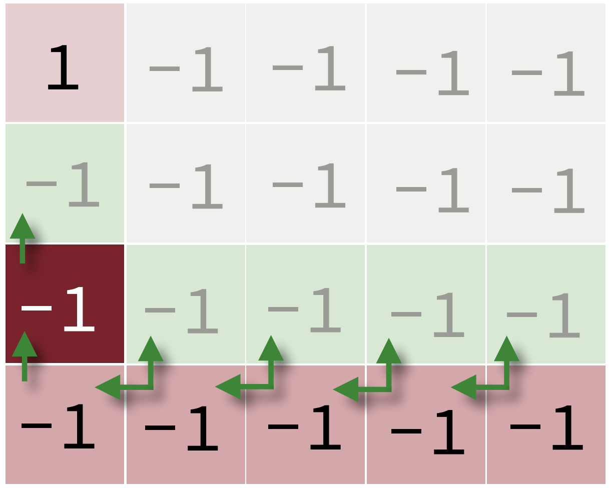
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

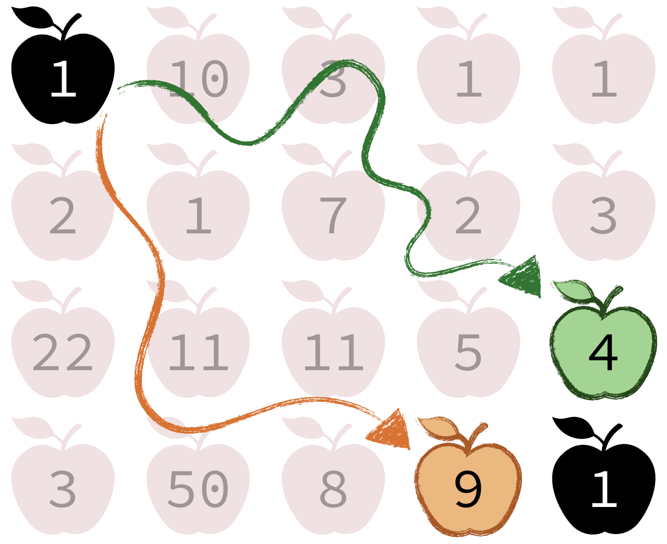
result[][]



1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

This problem was not solved before. We need to solve the *upper* subproblem

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

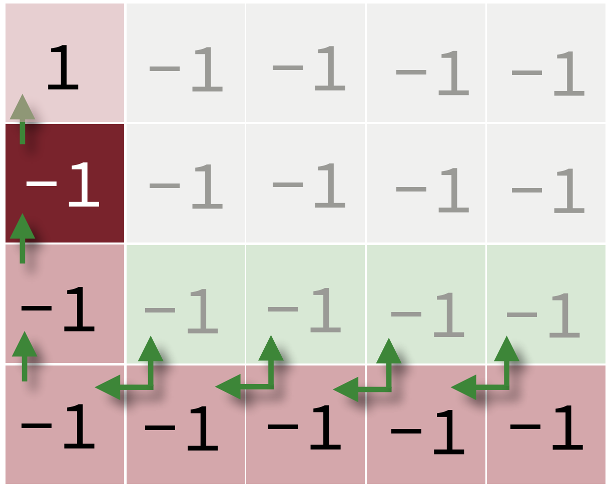
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

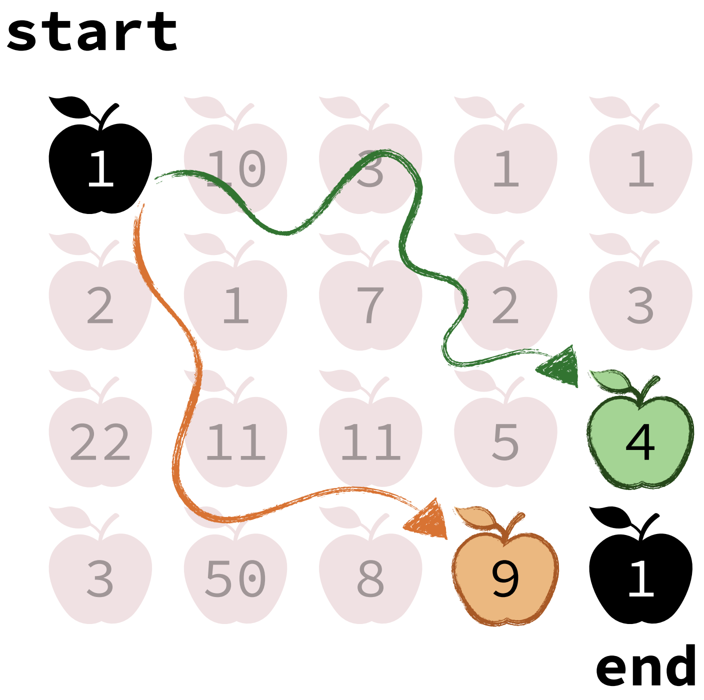
Trace

result[][]



1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem was not solved before. We need to solve the *upper* subproblem



Example: Collecting Apples

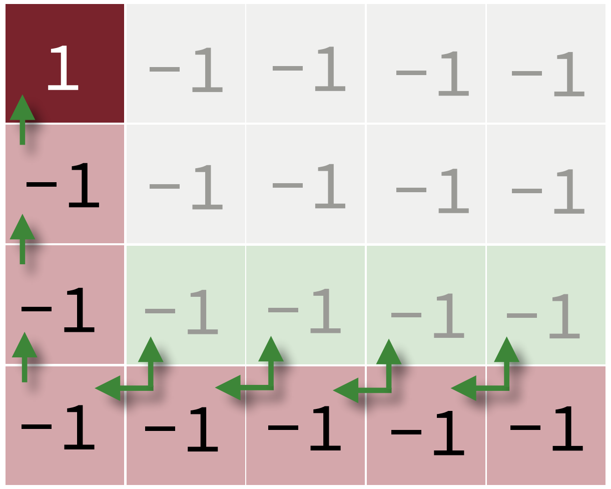
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

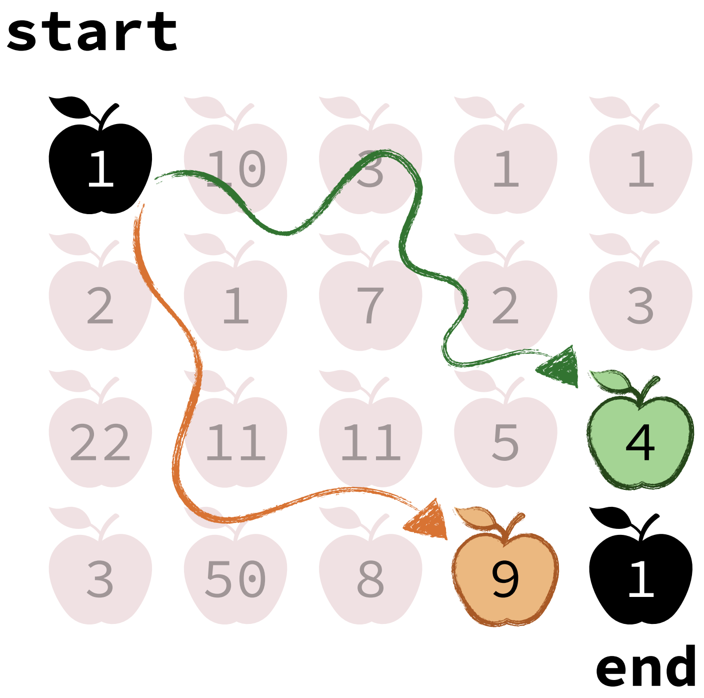
Trace

result[][]



1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

This problem was solved before. It is a *base case!*



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

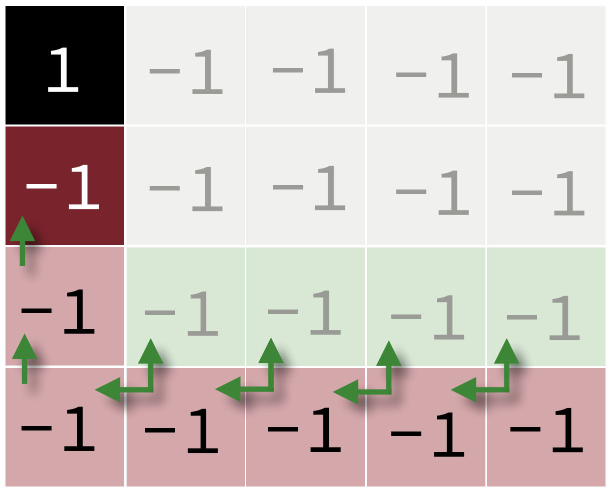
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

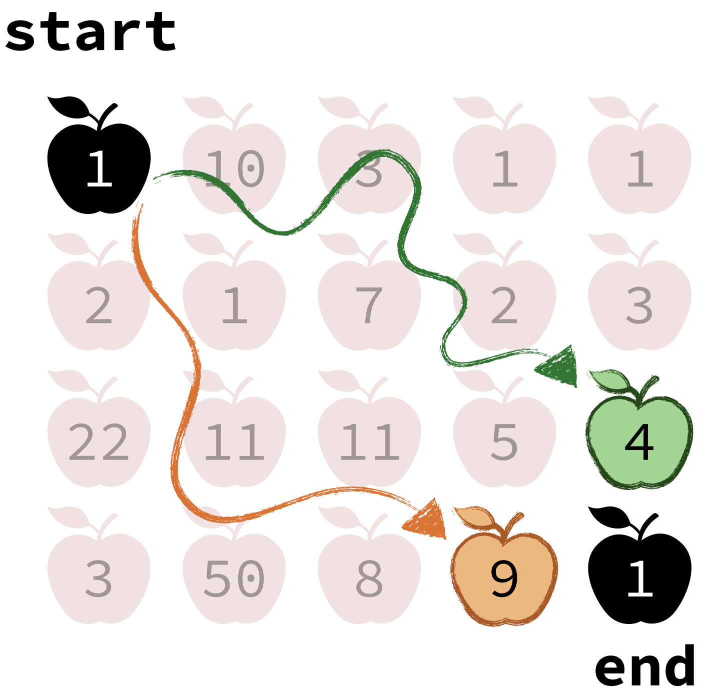
Trace

result[][]



1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem has what it needs (solution of the *upper* subproblem)



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

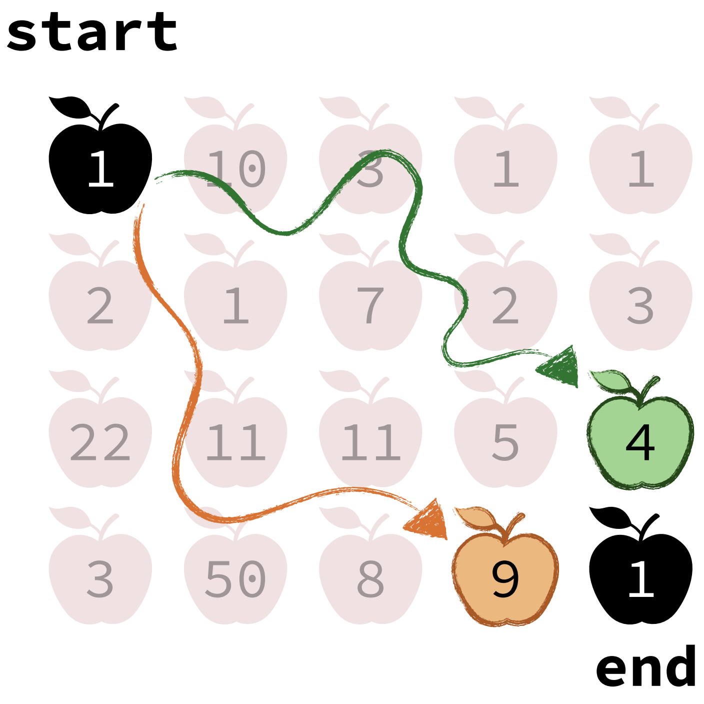
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	-1	-1	-1	-1
3	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem has what it needs (solution of the *upper* subproblem)



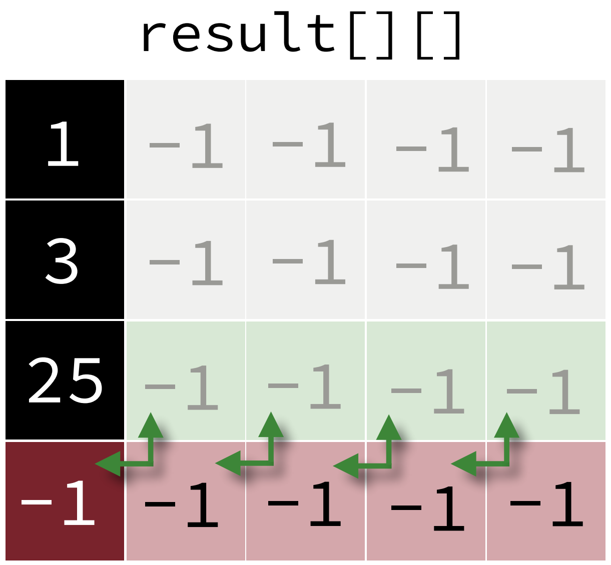
Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

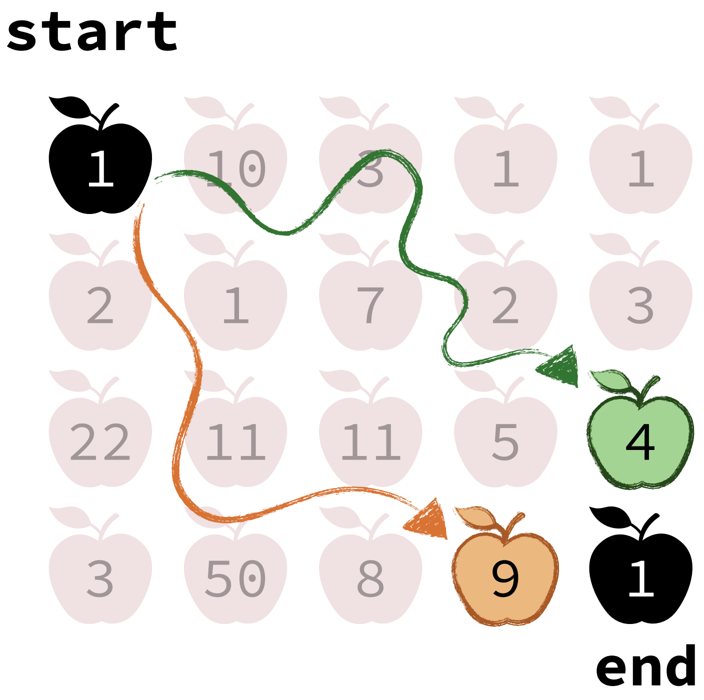
Trace



result[][]

1	-1	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
-1	-1	-1	-1	-1

This problem has what it needs (solution of the *upper* subproblem)



Example: Collecting Apples

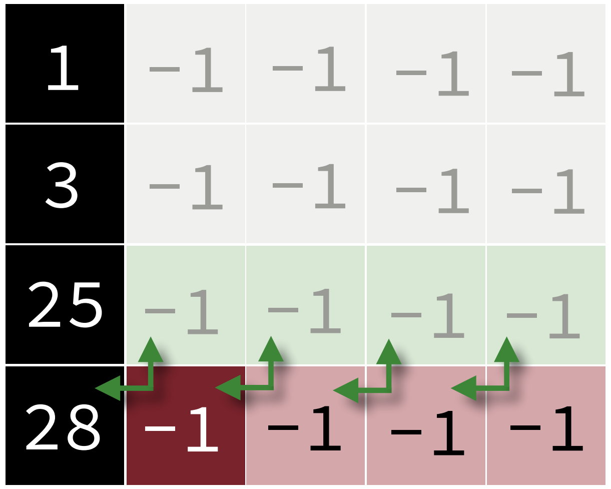
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

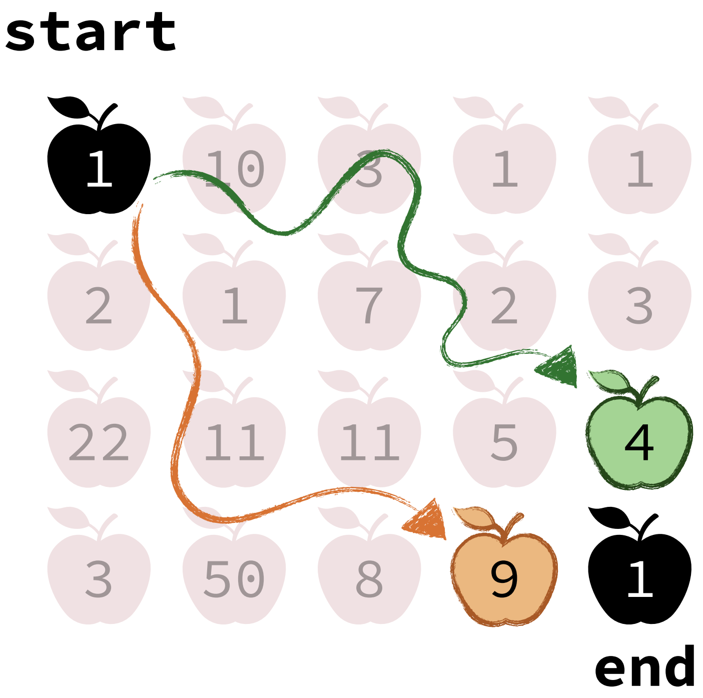
Trace

result[][]



1	-1	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* subproblem but not the *upper* subproblem.



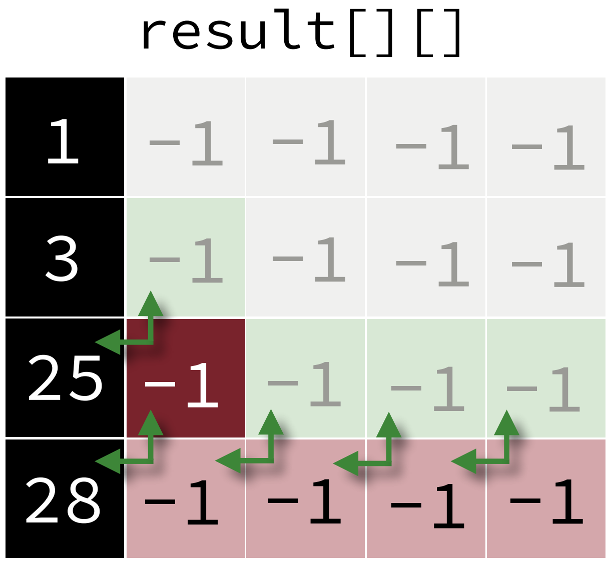
Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

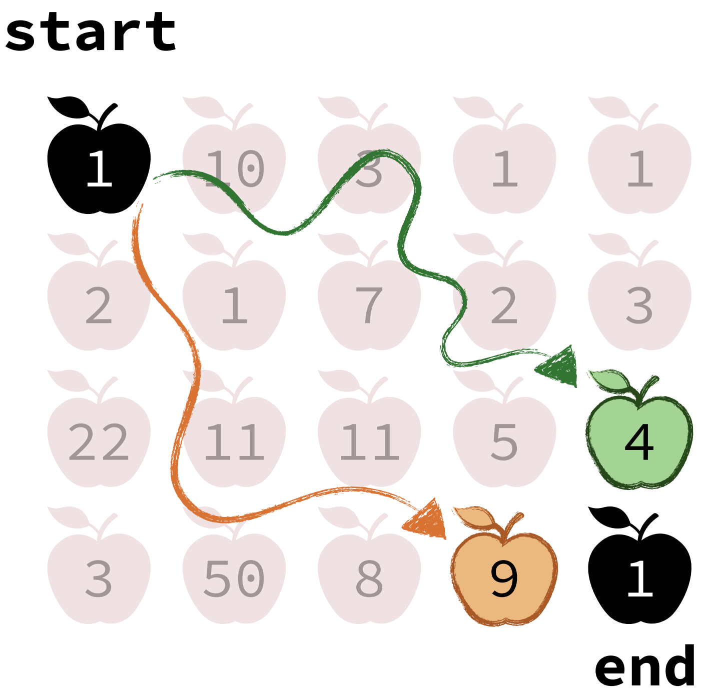
Trace



result[][]

1	-1	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* subproblem but not the *upper* subproblem.



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

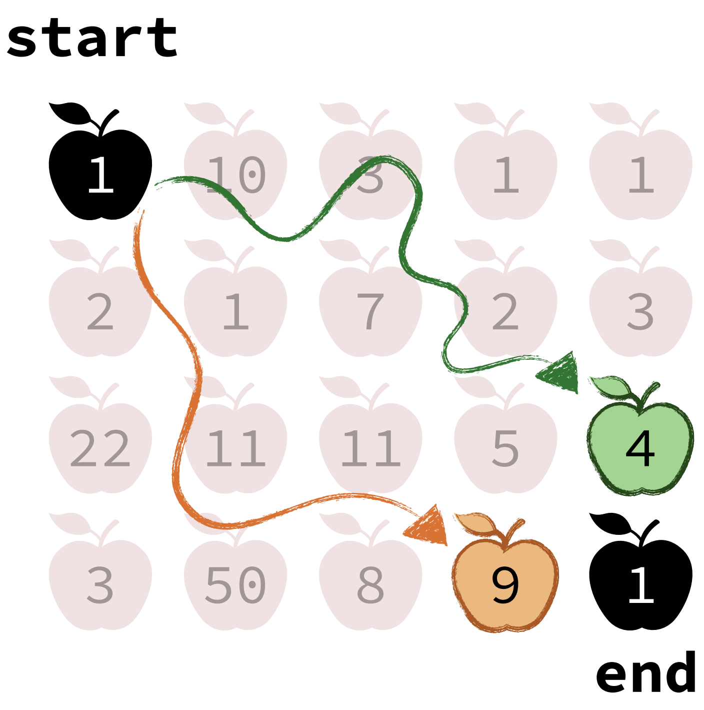
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	-1	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* subproblem but not the *upper* subproblem.



Example: Collecting Apples

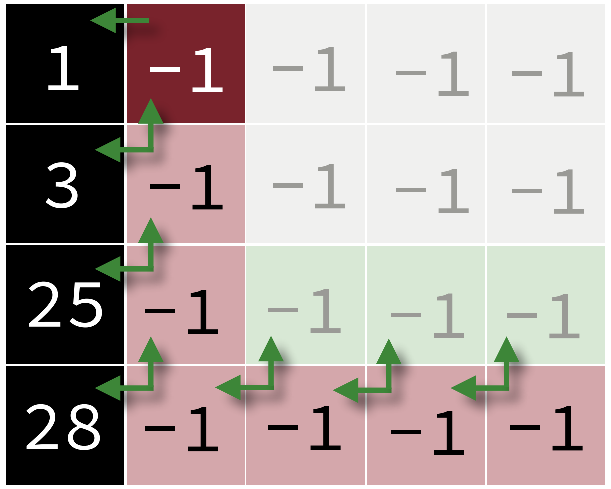
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

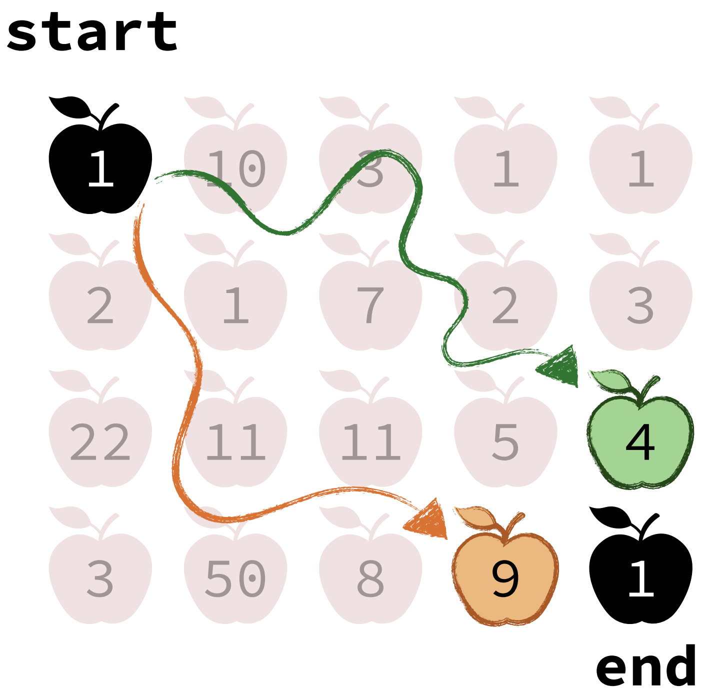
Trace

result[][]



1	-1	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* subproblem.



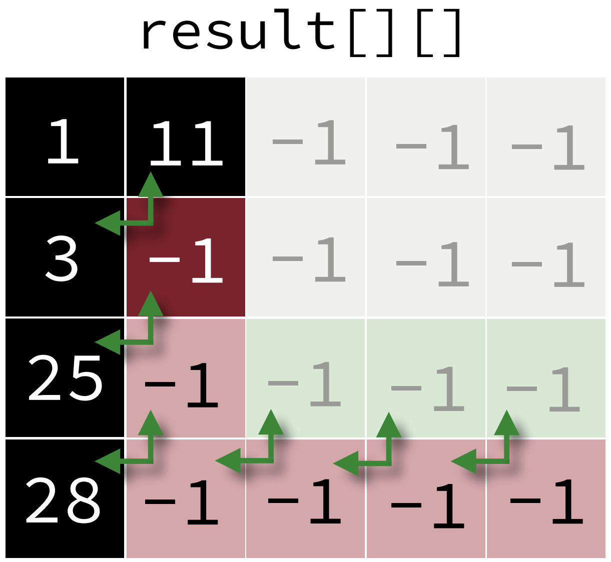
Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

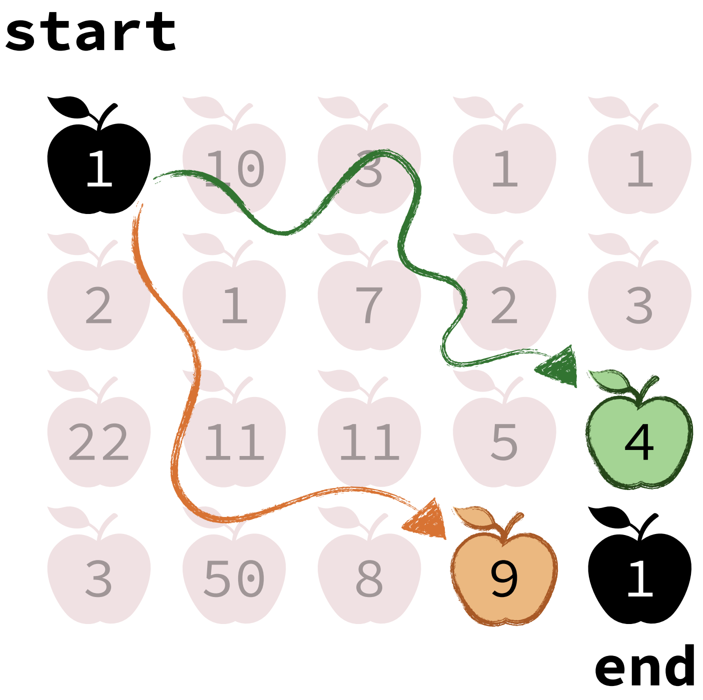
Trace



result[][]

1	11	-1	-1	-1
3	-1	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* and *upper* subproblems.



Example: Collecting Apples

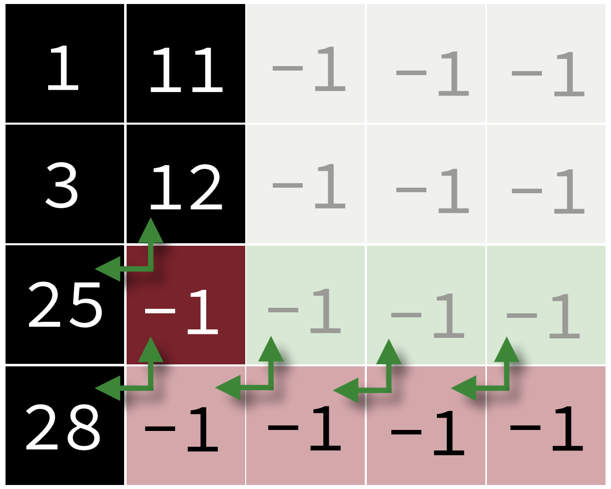
Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

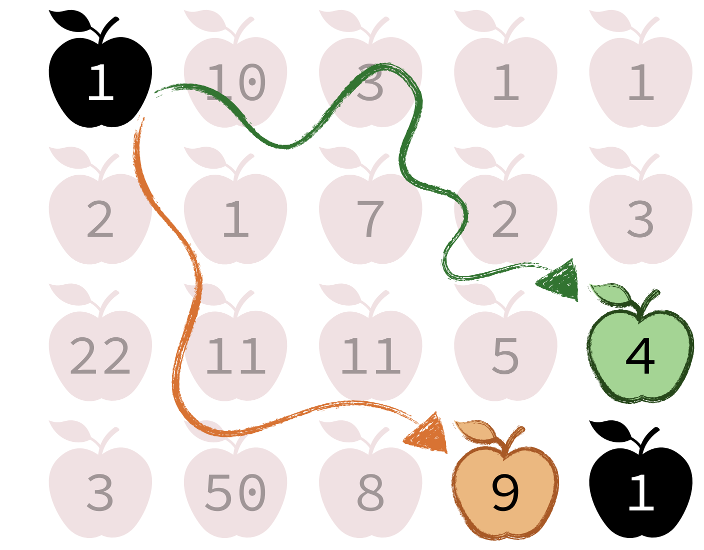
result[][]



1	11	-1	-1	-1
3	12	-1	-1	-1
25	-1	-1	-1	-1
28	-1	-1	-1	-1

This problem has the solution to the *left* and *upper* subproblems.

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

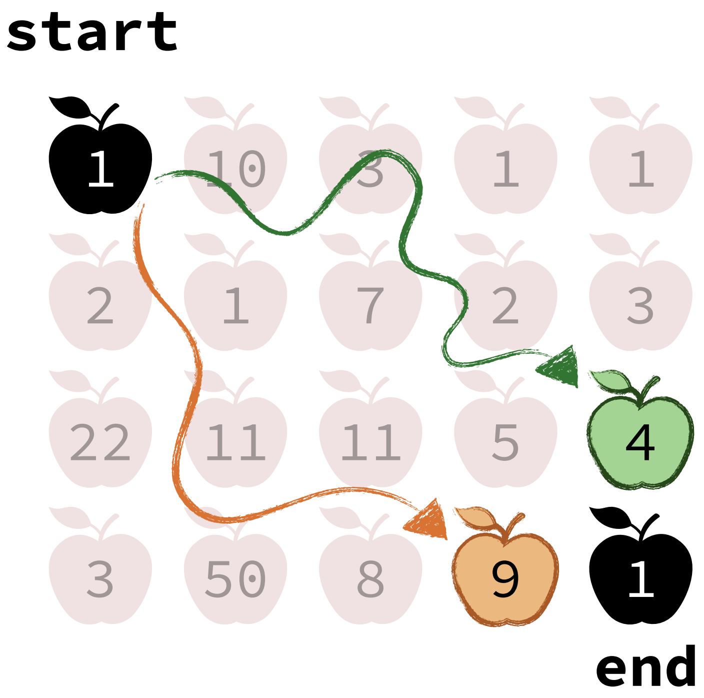
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	11	−1	−1	−1
3	12	−1	−1	−1
25	36	−1	−1	−1
28	−1	−1	−1	−1

This problem has the solution to the *left* and *upper* subproblems.



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

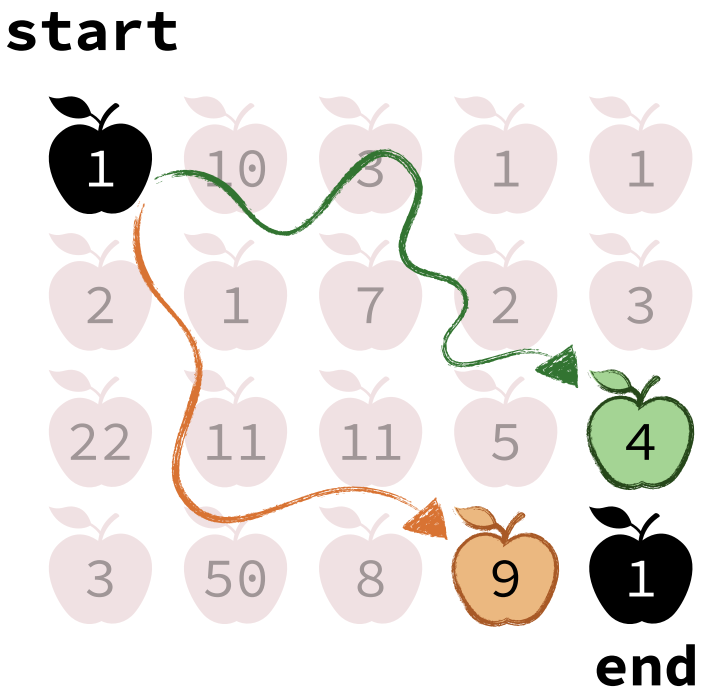
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	11	−1	−1	−1
3	12	−1	−1	−1
25	36	−1	−1	−1
28	86	−1	−1	−1

This problem has the solution to the *left* but still needs the *upper*



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

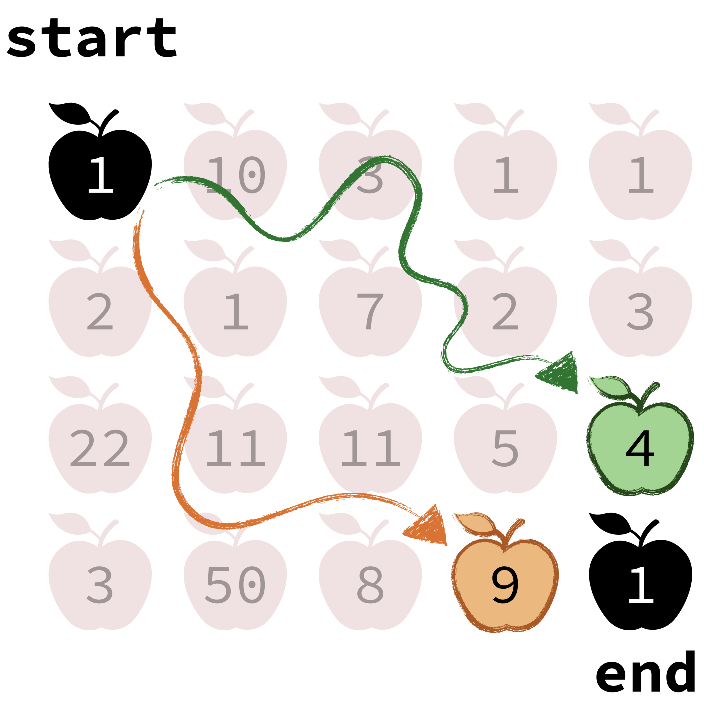
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	-1

Eventually, the main problem has the solution to the *left* and *upper* subproblems



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

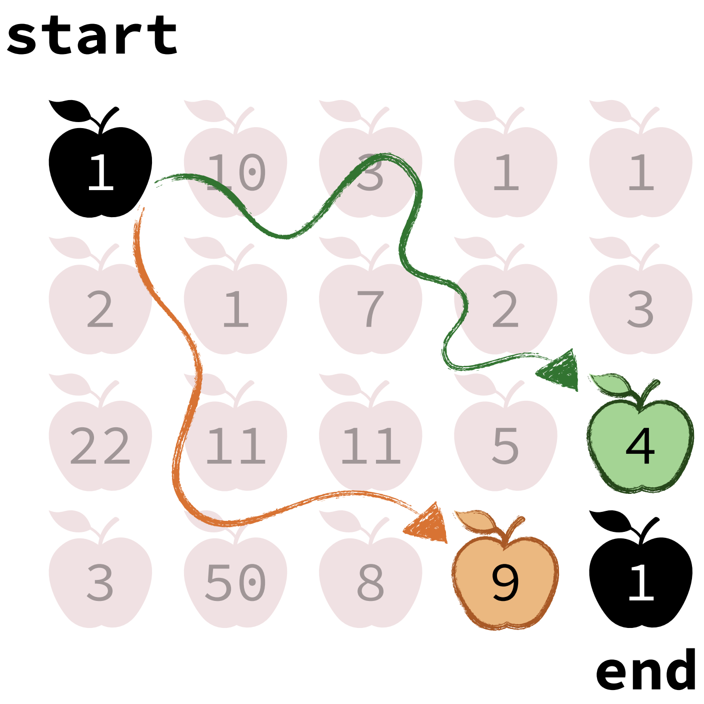
```
MAX_APPLES(i, j, apples[], result[])

 if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	-1

Eventually, the main problem has the solution to the *left* and *upper* subproblems



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

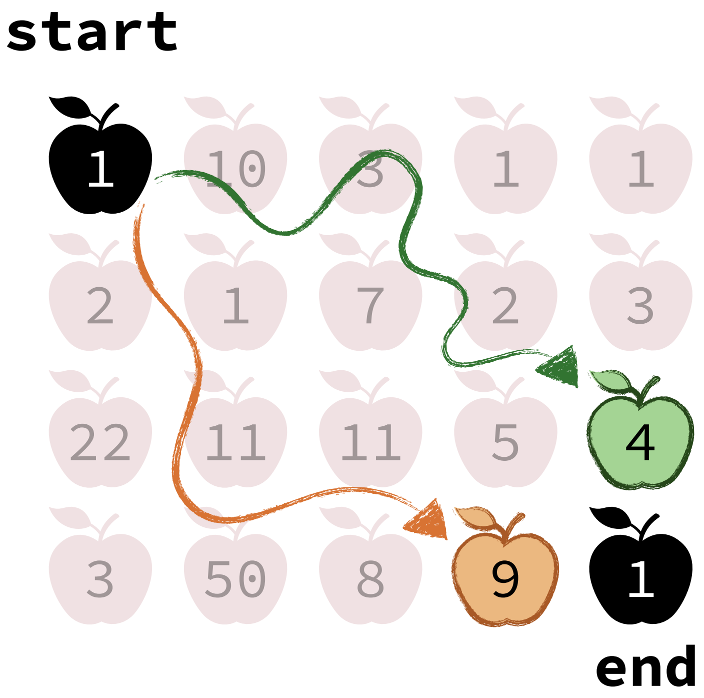
```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

  max_left = 0, max_up = 0
  if (j > 0): max_left = MAX_APPLES(i, j-1)
  if (i > 0): max_up   = MAX_APPLES(i-1, j)

  result[i][j] = apples[i][j] +
                 MAX(max_left, max_up)

  return result[i][j]
```

Trace

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	104

Eventually, the main problem has the solution to the *left* and *upper* subproblems



Example: Collecting Apples

Memoized Solution

```
COLLECT_APPLES(apples[])

  create array result[N][M]
  initialize result[][] to -1
  result[0][0] = apples[0][0]

  MAX_APPLES(N-1, M-1, apples, result)
  return result[N-1][M-1]
```

Running Time.

$\Theta(NM)$

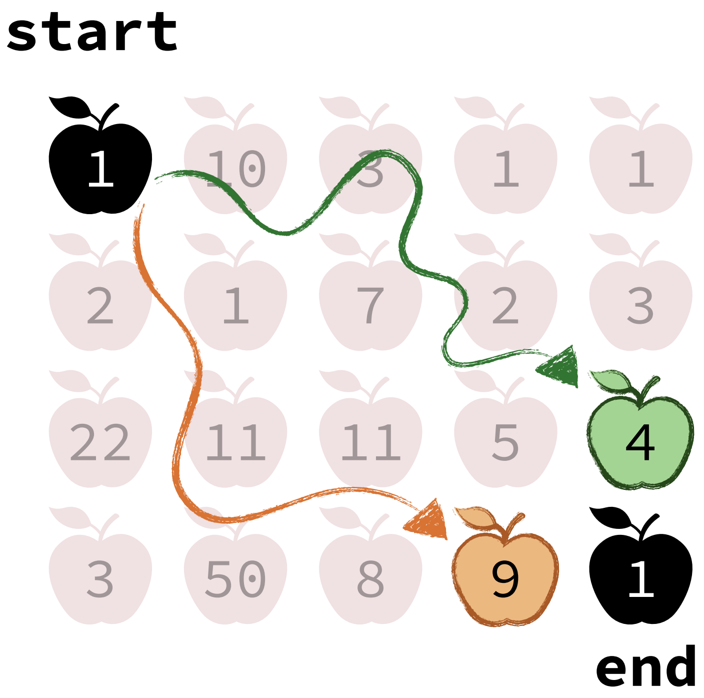
(NM subproblems solved, each once)

🎉 🎉

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	104

```
MAX_APPLES(i, j, apples[], result[])

if (result[i][j] != -1): return result[i][j]

 max_left = 0, max_up = 0
 if (j > 0): max_left = MAX_APPLES(i, j-1)
 if (i > 0): max_up   = MAX_APPLES(i-1, j)

 result[i][j] = apples[i][j] +
                MAX(max_left, max_up)

 return result[i][j]
```



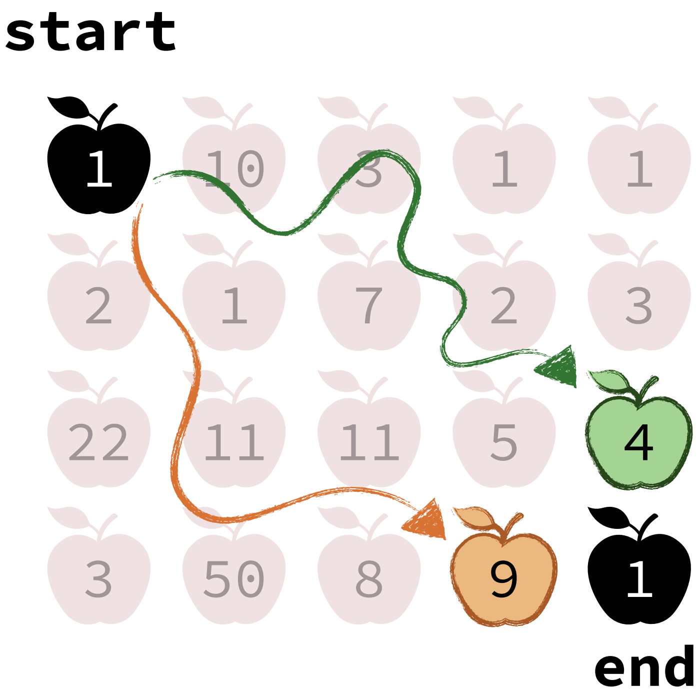
Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]
```

result[][]

stores the solution for each subproblem



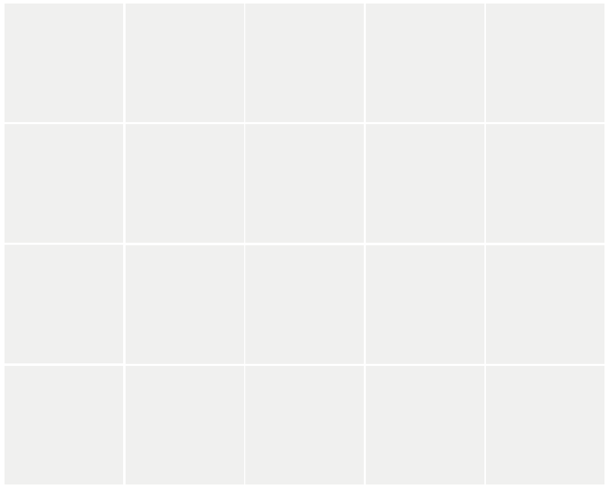
1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

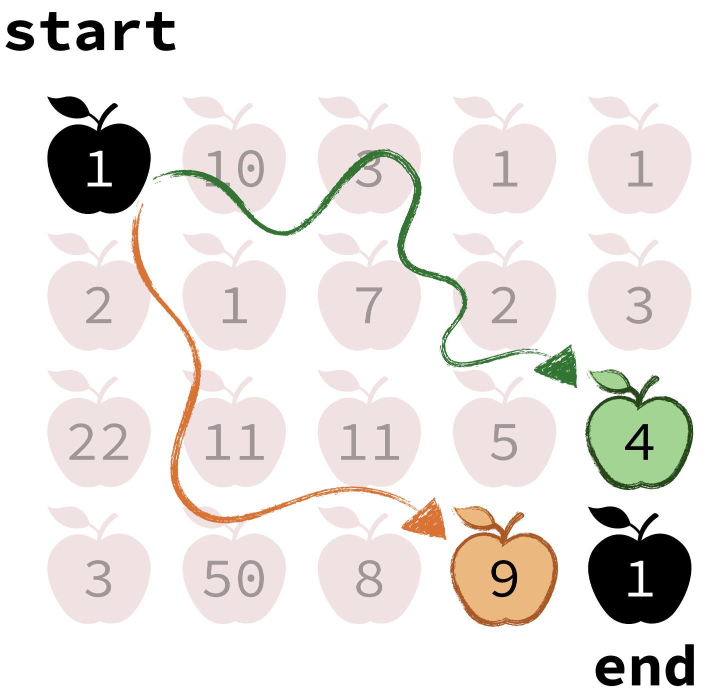
Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]
```

result[][]



stores the solution for each subproblem



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Note:

- The problem at [i][j] needs the problems **above** and **left**.
- Therefore, result[i-1][j] and result[i][j-1] must be filled before the result[i][j].
- This can be done by going row-by-row or column-by-column.

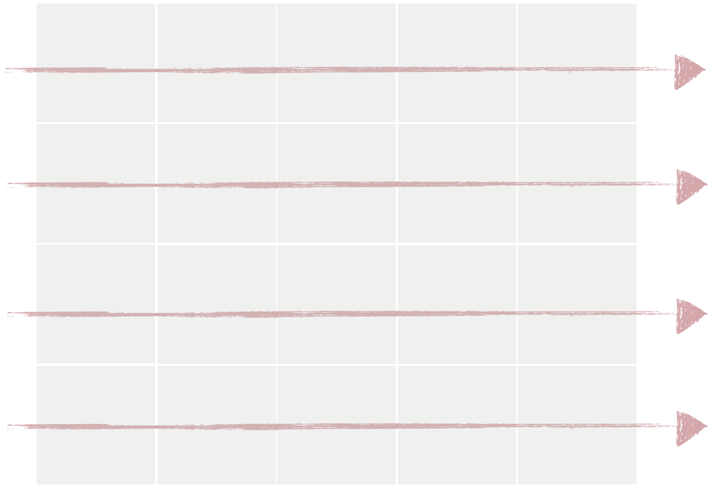
Example: Collecting Apples

Bottom-up Solution.

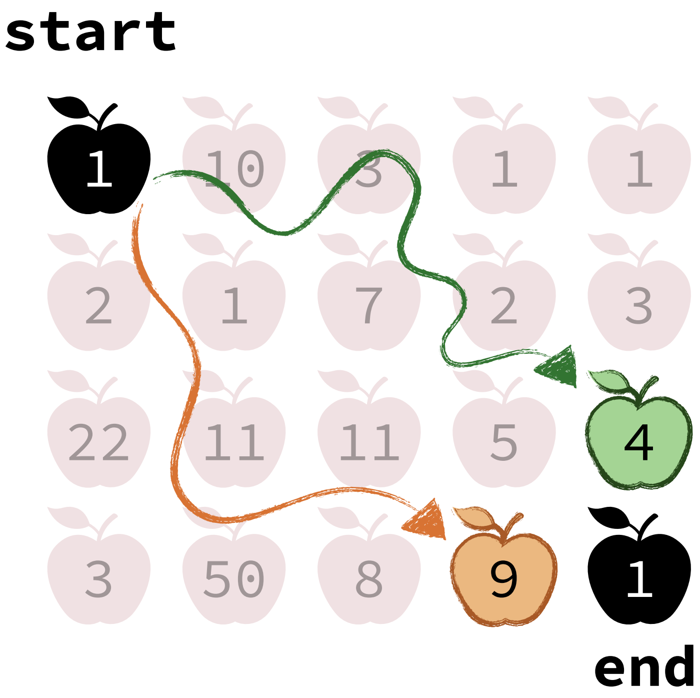
```
MAX_APPLES(i, j, apples[])

create array result[N][M]

for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
```

solve all subproblems from smallest to largest (row by row)

`result[][]`



stores the solution for each subproblem



Note:

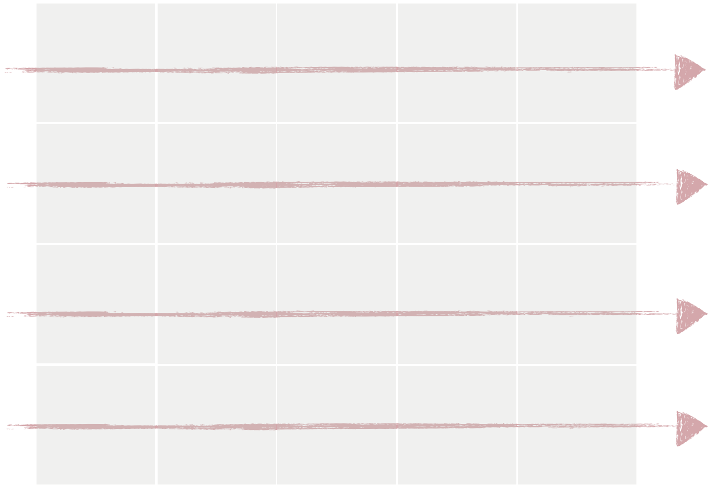
- The problem at `[i][j]` needs the problems **above** and **left**.
- Therefore, `result[i-1][j]` and `result[i][j-1]` must be filled before the `result[i][j]`.
- This can be done by going row-by-row or column-by-column.

Example: Collecting Apples

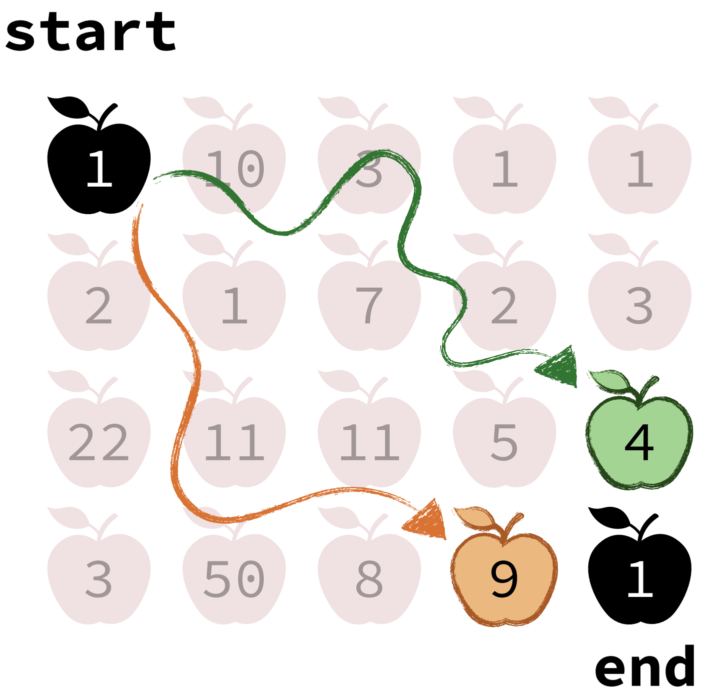
Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

create array result[N][M]

for (i = 0 ⟶ N-1):
  for (j = 0 ⟶ M-1):
    left = 0, up = 0
    if (j > 0): left = result[i][j-1]
    if (i > 0): up   = result[i-1][j]

    result[i][j] = MAX(left, up) + apples[i][j]
```

result[][]



stores the solution for each subproblem



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Note:

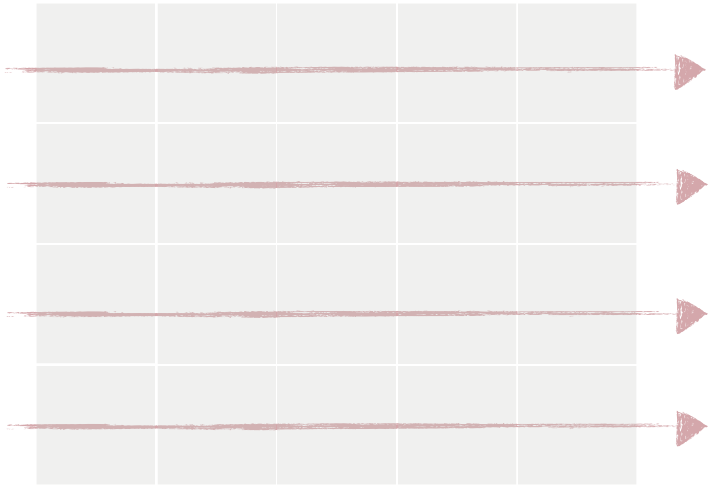
- The problem at [i][j] needs the problems **above** and **left**.
- Therefore, result[i-1][j] and result[i][j-1] must be filled before the result[i][j].
- This can be done by going row-by-row or column-by-column.

Example: Collecting Apples

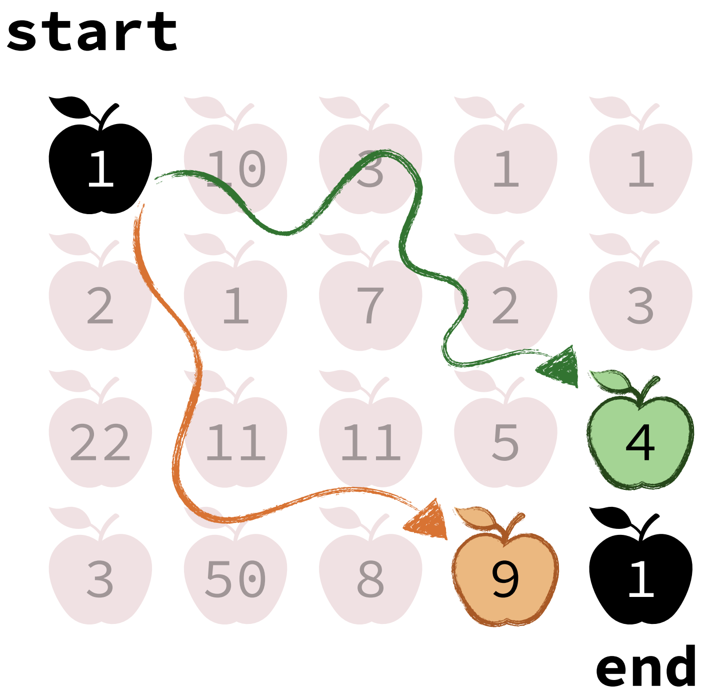
Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
       left = 0, up = 0
       if (j > 0): left = result[i][j-1]
       if (i > 0): up   = result[i-1][j]

       result[i][j] = MAX(left, up) + apples[i][j]

   return result[N-1][M-1]
```

result[][]



stores the solution for each subproblem



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Note:

- The problem at `[i][j]` needs the problems **above** and **left**.
- Therefore, `result[i-1][j]` and `result[i][j-1]` must be filled before the `result[i][j]`.
- This can be done by going row-by-row or column-by-column.

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
        left = 0, up = 0
        if (j > 0): left = result[i][j-1]
        if (i > 0): up   = result[i-1][j]

        result[i][j] = MAX(left, up) + apples[i][j]

   return result[N-1][M-1]
```

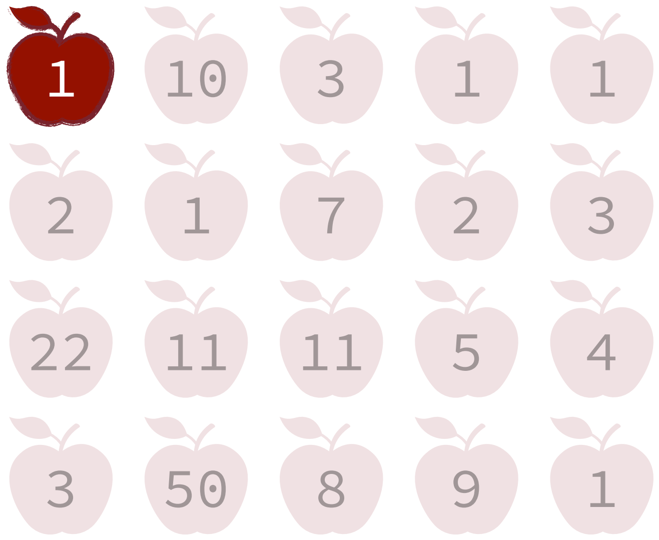
Trace

result[][]

1				

result[0][0] = apples[0][0]

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
       left = 0, up = 0
       if (j > 0): left = result[i][j-1]
       if (i > 0): up   = result[i-1][j]

       result[i][j] = MAX(left, up) + apples[i][j]

   return result[N-1][M-1]
```

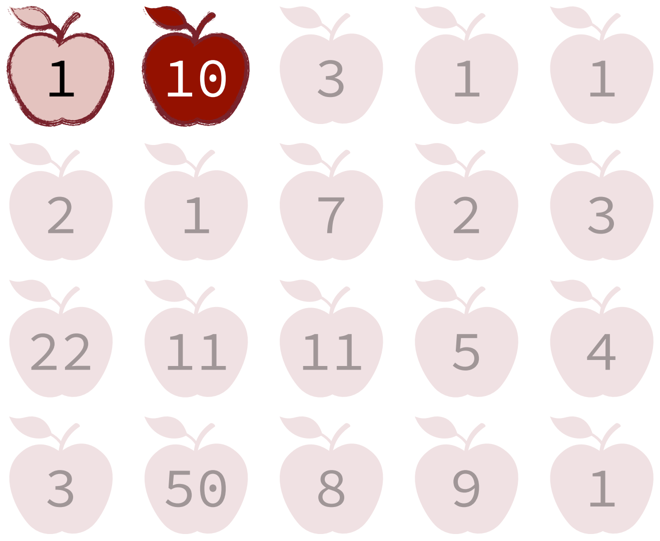
Trace

result[][]

1	11			

result = 10 + 1

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

create array result[N][M]

for (i = 0 ⟶ N-1):
  for (j = 0 ⟶ M-1):
     left = 0, up = 0
     if (j > 0): left = result[i][j-1]
     if (i > 0): up   = result[i-1][j]

     result[i][j] = MAX(left, up) + apples[i][j]

 return result[N-1][M-1]
```

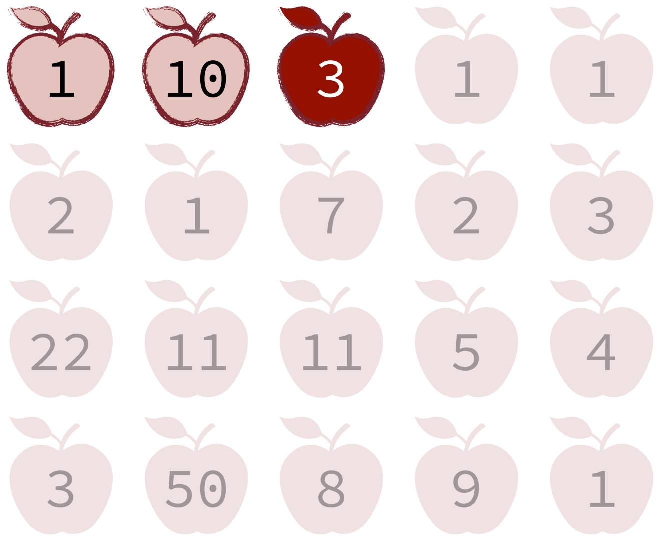
Trace

result[][]

1	11	14		

result = 11 + 3

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

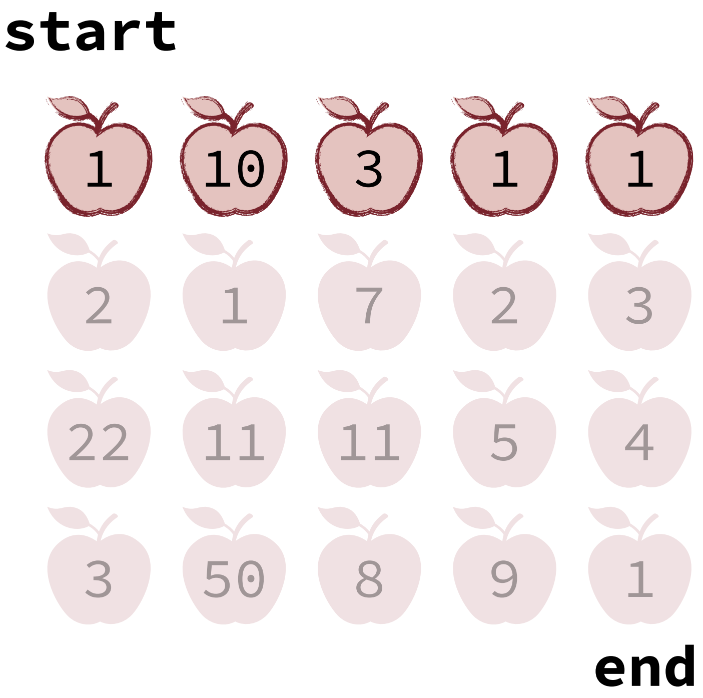
```
MAX_APPLES(i, j, apples[])

create array result[N][M]

for (i = 0 ⟶ N-1):
  for (j = 0 ⟶ M-1):
     left = 0, up = 0
     if (j > 0): left = result[i][j-1]
     if (i > 0): up   = result[i-1][j]

     result[i][j] = MAX(left, up) + apples[i][j]

 return result[N-1][M-1]
```

Trace

result[][]

1	11	14	15	16

result[i][j] = apples[i][j] + result[i][j-1] because there are no subproblems above



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

create array result[N][M]

for (i = 0 ⟶ N-1):
  for (j = 0 ⟶ M-1):
     left = 0, up = 0
     if (j > 0): left = result[i][j-1]
     if (i > 0): up   = result[i-1][j]

     result[i][j] = MAX(left, up) + apples[i][j]

 return result[N-1][M-1]
```

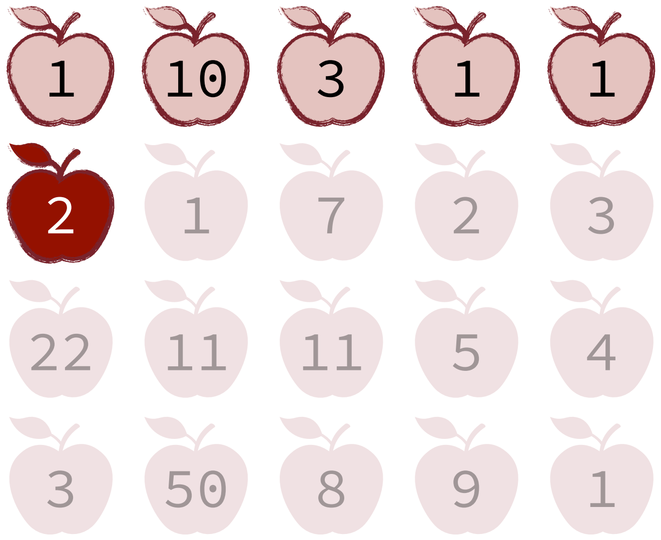
Trace

result[][]

1	11	14	15	16
3				

result = 2 + 1

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
      left = 0, up = 0
      if (j > 0): left = result[i][j-1]
      if (i > 0): up   = result[i-1][j]

      result[i][j] = MAX(left, up) + apples[i][j]

   return result[N-1][M-1]
```

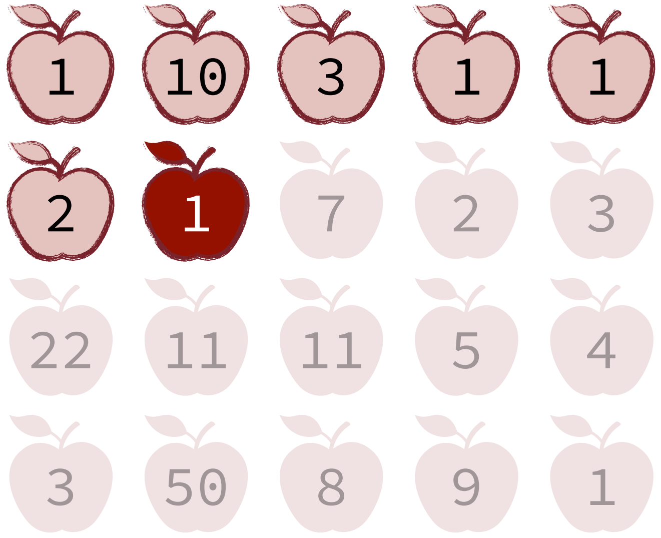
Trace

result[][]

1	11	14	15	16
3	12			

`result = MAX(3, 11) + 1`

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
      left = 0, up = 0
      if (j > 0): left = result[i][j-1]
      if (i > 0): up   = result[i-1][j]

      result[i][j] = MAX(left, up) + apples[i][j]

  return result[N-1][M-1]
```

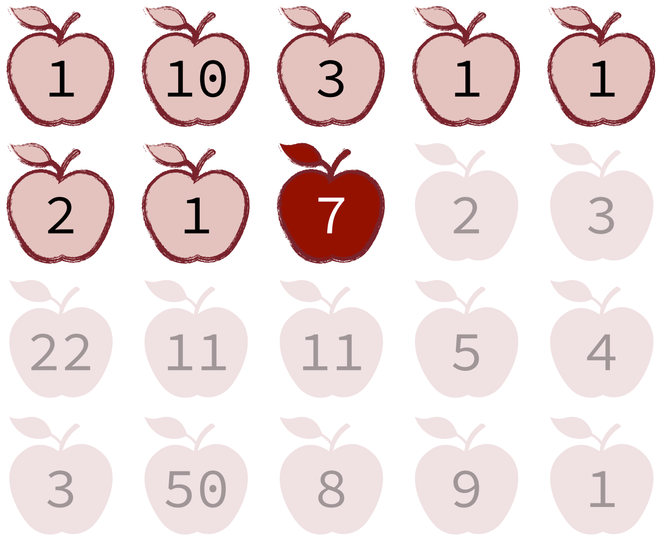
Trace

result[][]

1	11	14	15	16
3	12	21		

```
result = MAX(12, 14) + 7
```

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

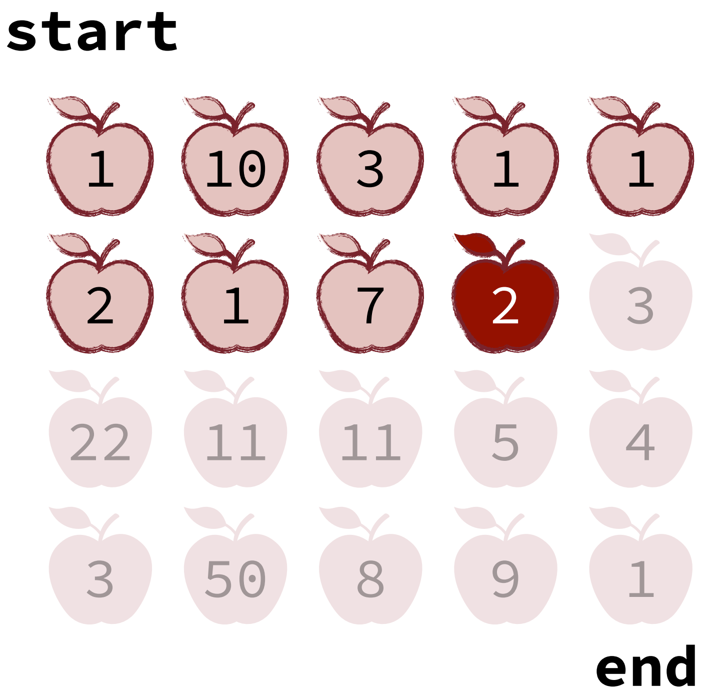
```
MAX_APPLES(i, j, apples[])

 create array result[N][M]

 for (i = 0 ⟶ N-1):
   for (j = 0 ⟶ M-1):
      left = 0, up = 0
      if (j > 0): left = result[i][j-1]
      if (i > 0): up   = result[i-1][j]

      result[i][j] = MAX(left, up) + apples[i][j]

  return result[N-1][M-1]
```

Trace

result[][]

1	11	14	15	16
3	12	21	23	

result = MAX(21, 15) + 2



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

 create array result[N][M]

 for (i = 0 ⟶ N-1):
   for (j = 0 ⟶ M-1):
      left = 0, up = 0
      if (j > 0): left = result[i][j-1]
      if (i > 0): up   = result[i-1][j]

      result[i][j] = MAX(left, up) + apples[i][j]

  return result[N-1][M-1]
```

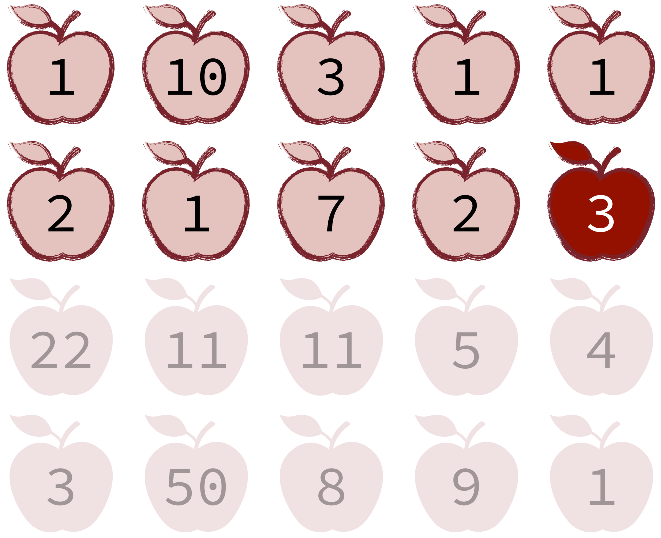
Trace

result[][]

1	11	14	15	16
3	12	21	23	26

result = MAX(23, 16) + 3

start



1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

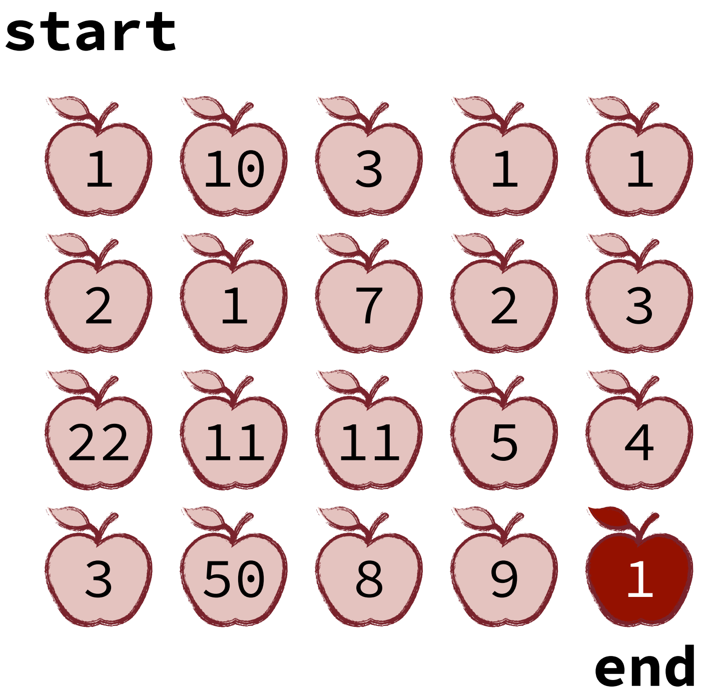
```
MAX_APPLES(i, j, apples[])

  create array result[N][M]

  for (i = 0 ⟶ N-1):
    for (j = 0 ⟶ M-1):
       left = 0, up = 0
       if (j > 0): left = result[i][j-1]
       if (i > 0): up   = result[i-1][j]

       result[i][j] = MAX(left, up) + apples[i][j]

   return result[N-1][M-1]
```

Trace

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	104

result = MAX(103, 56) + 1



start

1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

end

Example: Collecting Apples

Bottom-up Solution.

```
MAX_APPLES(i, j, apples[])

 create array result[N][M]

 for (i = 0 ⟶ N-1):
   for (j = 0 ⟶ M-1):
      left = 0, up = 0
      if (j > 0): left = result[i][j-1]
      if (i > 0): up   = result[i-1][j]

      result[i][j] = MAX(left, up) + apples[i][j]

  return result[N-1][M-1]
```

Running Time. $\Theta(NM)$

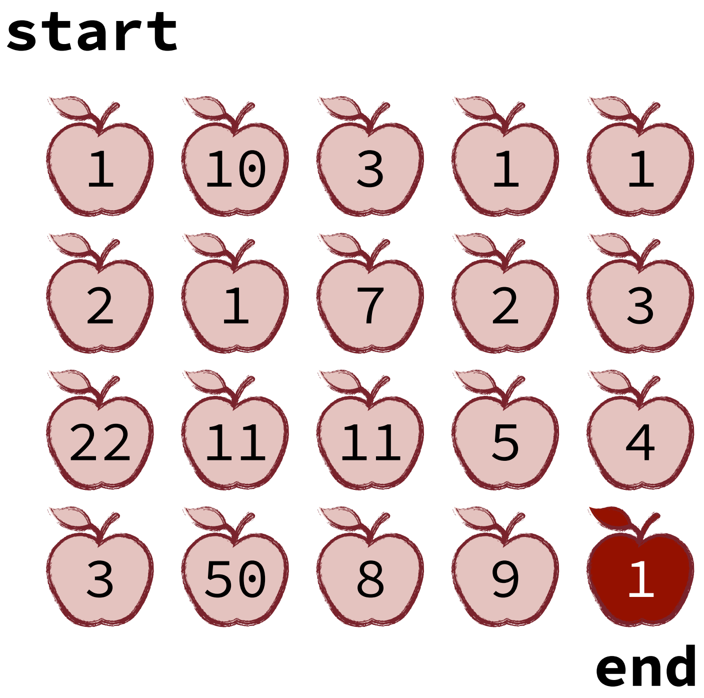
NM cells in the table filled in O(1) each

Trace

result[][]

1	11	14	15	16
3	12	21	23	26
25	36	47	52	56
28	86	94	103	104

result = MAX(103, 56) + 1

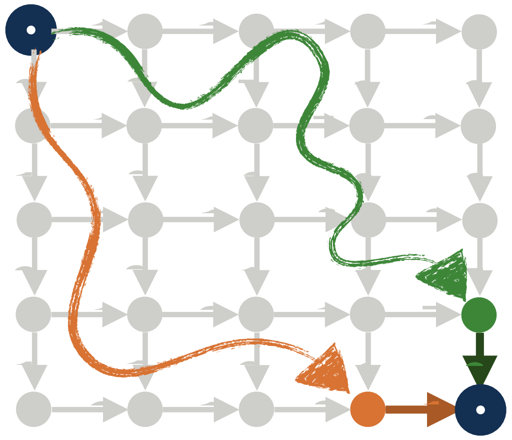


1	10	3	1	1
2	1	7	2	3
22	11	11	5	4
3	50	8	9	1

Collecting Apples: Recap

1. Found the optimal Substructure.

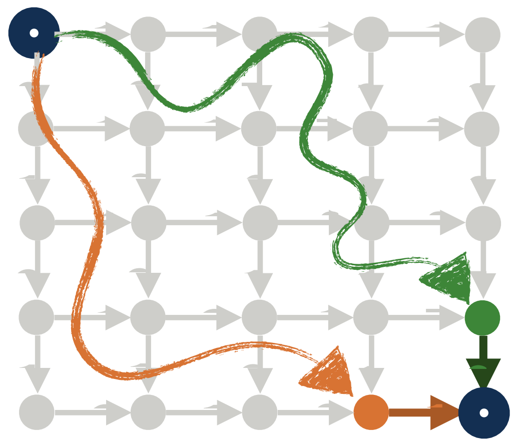
```
max_apples(i, j) = apples[i][j] + MAX(max_apples(i-1, j),
                                      max_apples(i, j-1))
```



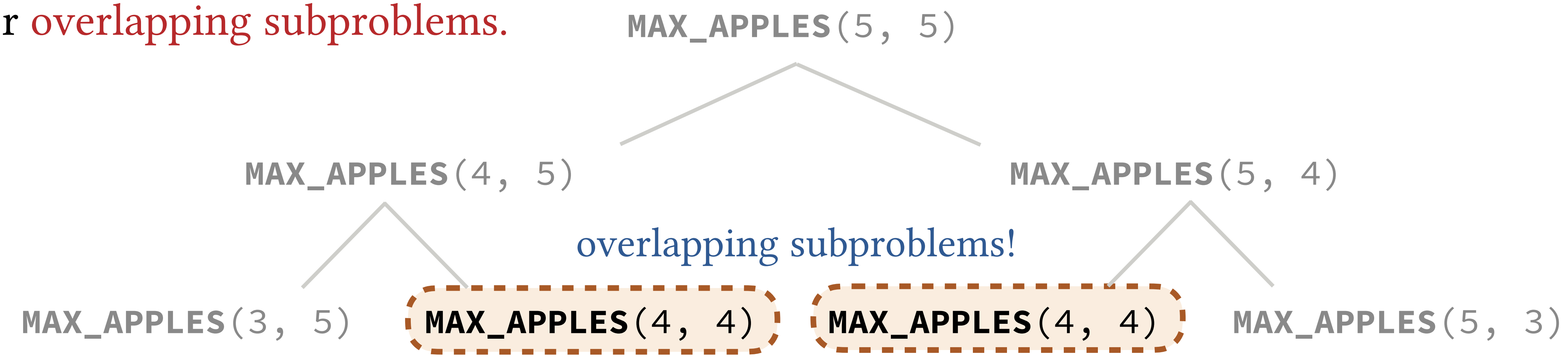
Collecting Apples: Recap

1. Found the optimal Substructure.

```
max_apples(i, j) = apples[i][j] + MAX(max_apples(i-1, j),
                                      max_apples(i, j-1))
```



2. Checked for overlapping subproblems.



MAX_APPLES(5, 5)

MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

overlapping subproblems!

MAX_APPLES(3, 5)

MAX_APPLES(4, 4)

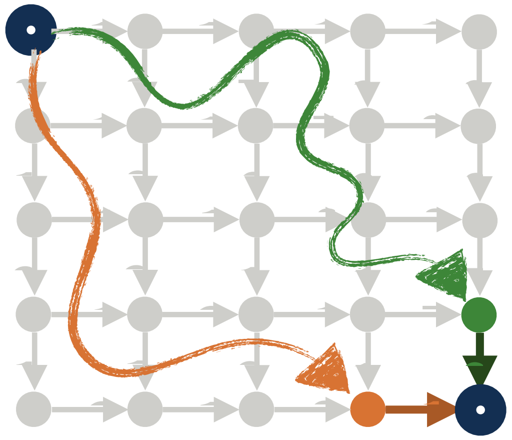
MAX_APPLES(4, 4)

MAX_APPLES(5, 3)

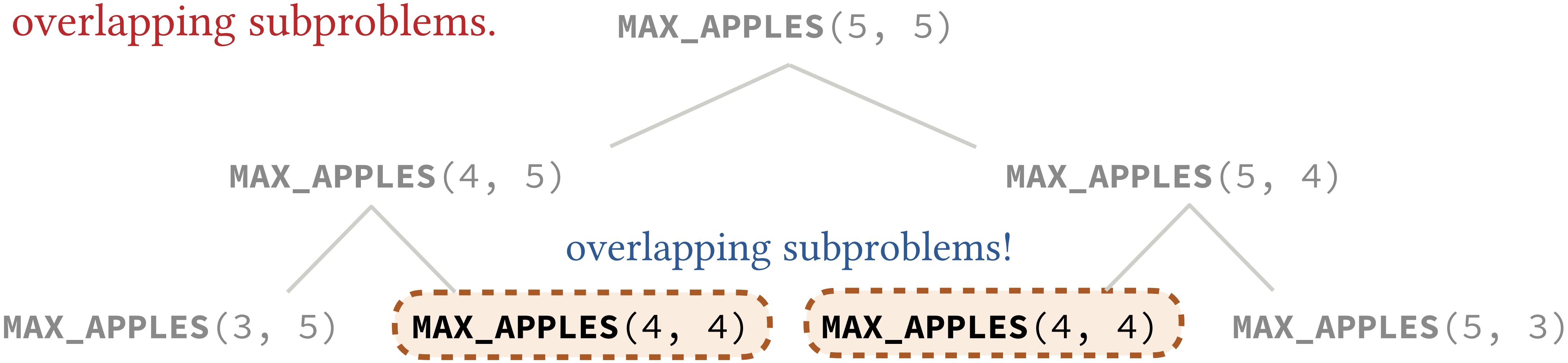
Collecting Apples: Recap

1. Found the optimal Substructure.

```
max_apples(i, j) = apples[i][j] + MAX(max_apples(i-1, j),
                                      max_apples(i, j-1))
```



2. Checked for overlapping subproblems.



MAX_APPLES(5, 5)

MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

overlapping subproblems!

MAX_APPLES(3, 5)

MAX_APPLES(4, 4)

MAX_APPLES(4, 4)

MAX_APPLES(5, 3)

3. Created a table for storing the solutions to subproblems.
Used memoization or bottom-up dynamic programming.

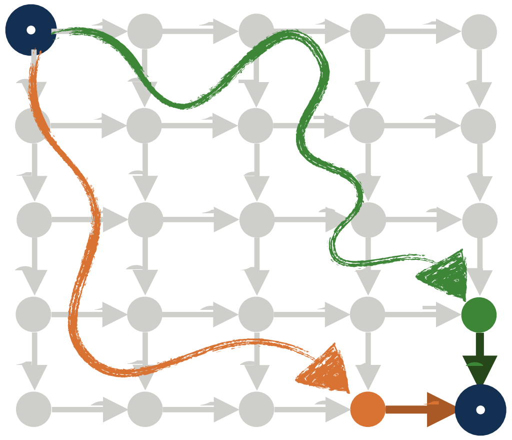
result[][]

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

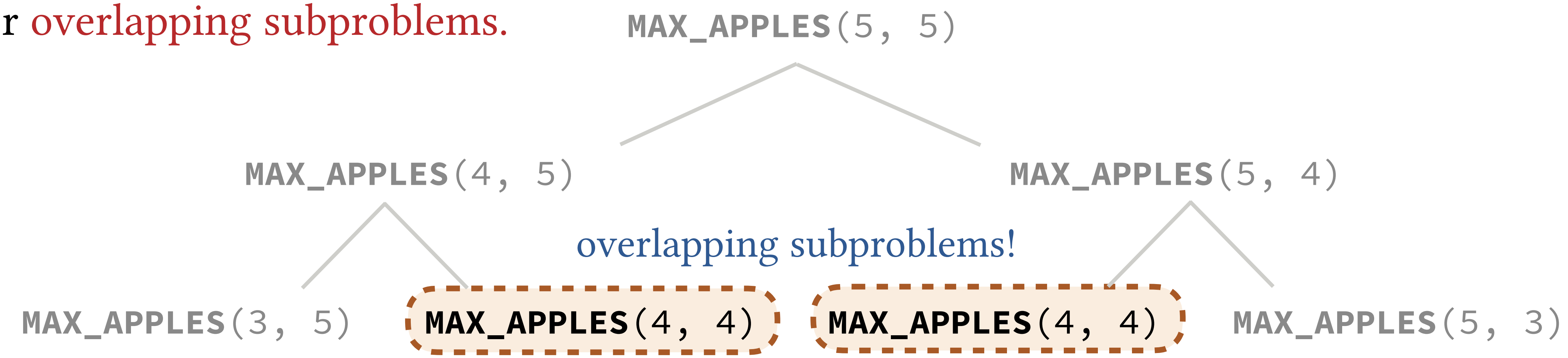
Collecting Apples: Recap

1. Found the optimal Substructure.

```
max_apples(i, j) = apples[i][j] + MAX(max_apples(i-1, j),
                                      max_apples(i, j-1))
```



2. Checked for overlapping subproblems.



MAX_APPLES(5, 5)

MAX_APPLES(4, 5)

MAX_APPLES(5, 4)

overlapping subproblems!

MAX_APPLES(3, 5)

MAX_APPLES(4, 4)

MAX_APPLES(4, 4)

MAX_APPLES(5, 3)

3. Created a table for storing the solutions to subproblems.
Used memoization or bottom-up dynamic programming.

🎉 **Effect.** Reduced the running time from exponential to linear in the number of cells.

result[][]

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

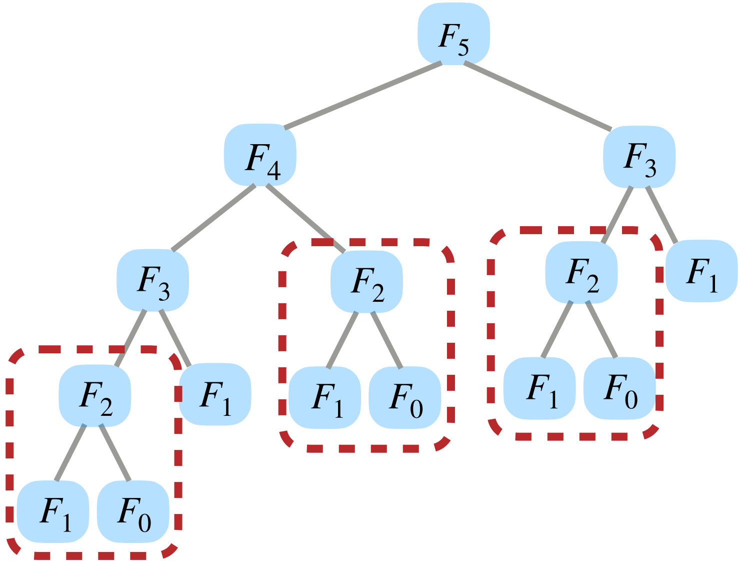
Same Steps for Fibonacci

1. Found the optimal Substructure.

   This was already given by the definition of the problem.

```
fib(n) = fib(n-1) + fib(n-2)
```

2. Checked for overlapping subproblems.

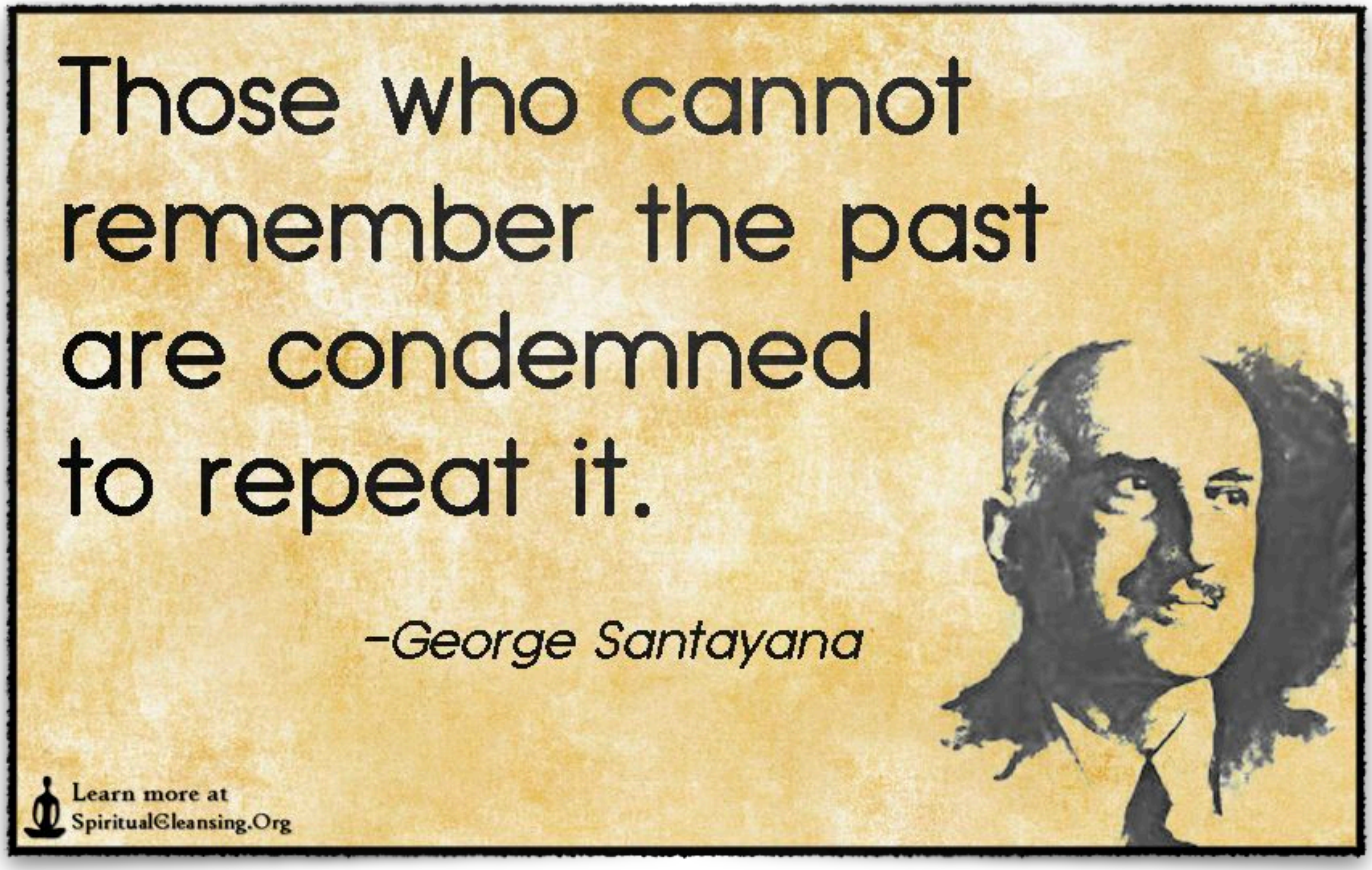


3. Created a table for storing the solutions to subproblems.

   Used memoization or bottom-up dynamic programming.

0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	13

🎉 **Effect.** Reduced the running time from exponential to linear in *n*.



Those who cannot remember the past are condemned to repeat it.

–George Santayana