Data Structures and Algorithms – COMS21103

2015/2016

Dynamic Programming

Largest Empty Square and Weighted Interval Scheduling

Benjamin Sach

University of BRISTOL

The name

Dynamic Programming is an approach to algorithm design...

why does it sound like an alternative to Agile Software Development?

The name

Dynamic Programming is an approach to algorithm design...

why does it sound like an alternative to Agile Software Development?

Serious answer:

- Richard Bellman invented Dynamic programming around 1950

  a 'program' referred to finding an optimal schedule or programme of activities

The name

Dynamic Programming is an approach to algorithm design. . .

why does it sound like an alternative to Agile Software Development?

Serious answer:

- Richard Bellman invented Dynamic programming around 1950

  a ‘program’ referred to finding an optimal schedule or programme of activities

---

Real answer:

“The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research... His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical... I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

- Richard Bellman

What problems can Dynamic Programming solve?

- Longest Common Subsequence
  (used heavily in Bioinformatics for DNA similarity)
- Edit Distance
  (used heavily in Bioinformatics for sequence alignment)
- Text justification
- Seam Carving
  (Google this later, it's really awesome)
- Solving the Towers of Hanoi
- Predicting cricket scores
- Assembly Line Scheduling
- Matrix Chain Multiplication
- Playing Tetris perfectly
- Dynamic Time Warping
  (used extensively in computer vision)
- Finding optimal Binary Search Trees
  (when you know the likely frequencies of searches)
- The Travelling Salesman Problem *(though still slowly)*
- Knapsack *(though still slowly)*

and **loads** of other problems

Introduction

Dynamic programming is a technique for finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea:

1. Find a recursive formula for the problem
- in terms of answers to subproblems.

(typically this is the hard bit)

2. Write down a naive recursive algorithm

(typically this algorithm will take exponential time)

3. Speed it up by storing the solutions to subproblems

(to avoid recomputing the same thing over and over)

4. Derive an iterative algorithm by solving the subproblems in a good order

(iterative algorithms are often better in practice, easier to analyse and prettier)

in other words. . .

Dynamic programming is *recursion without repetition*

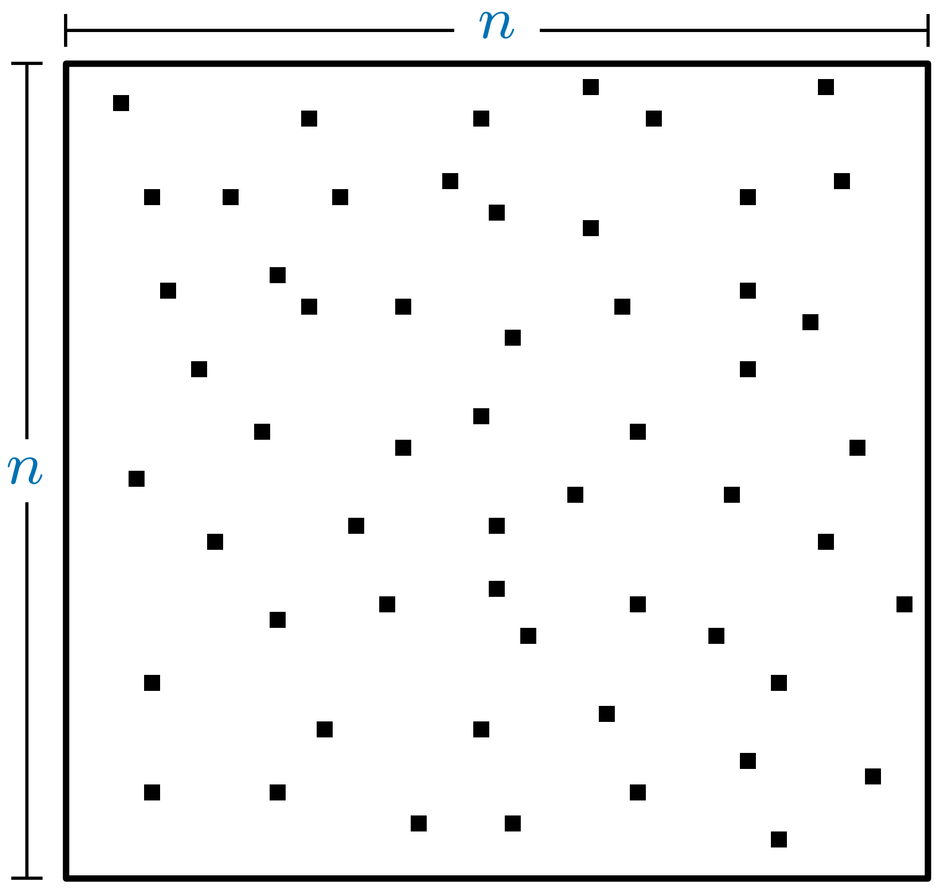
Part one

Largest Empty Square

Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

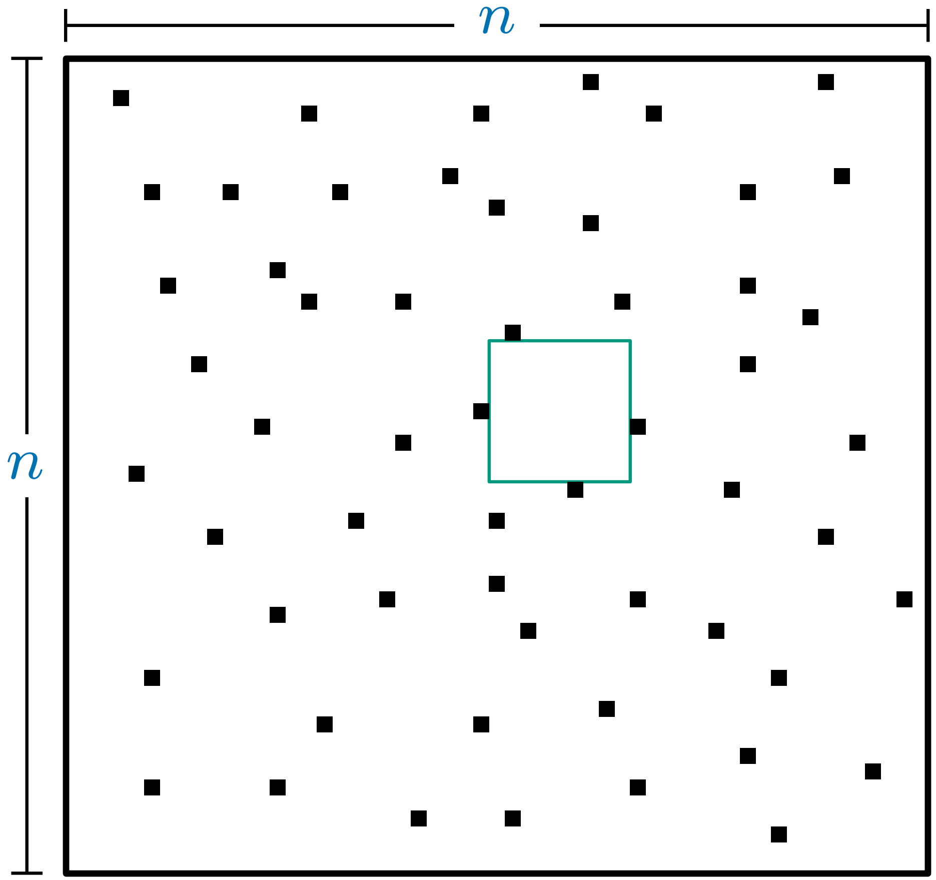
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

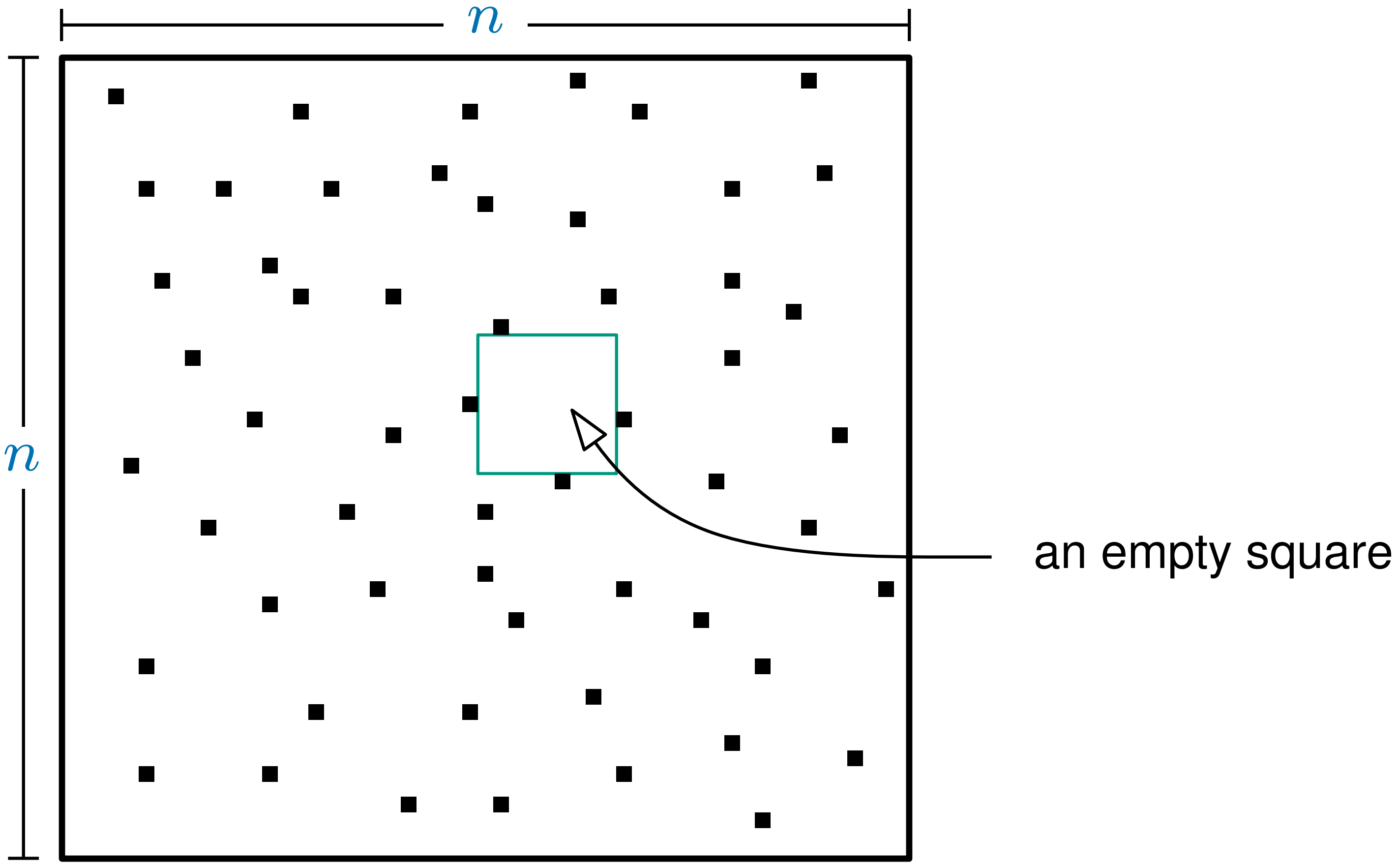
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

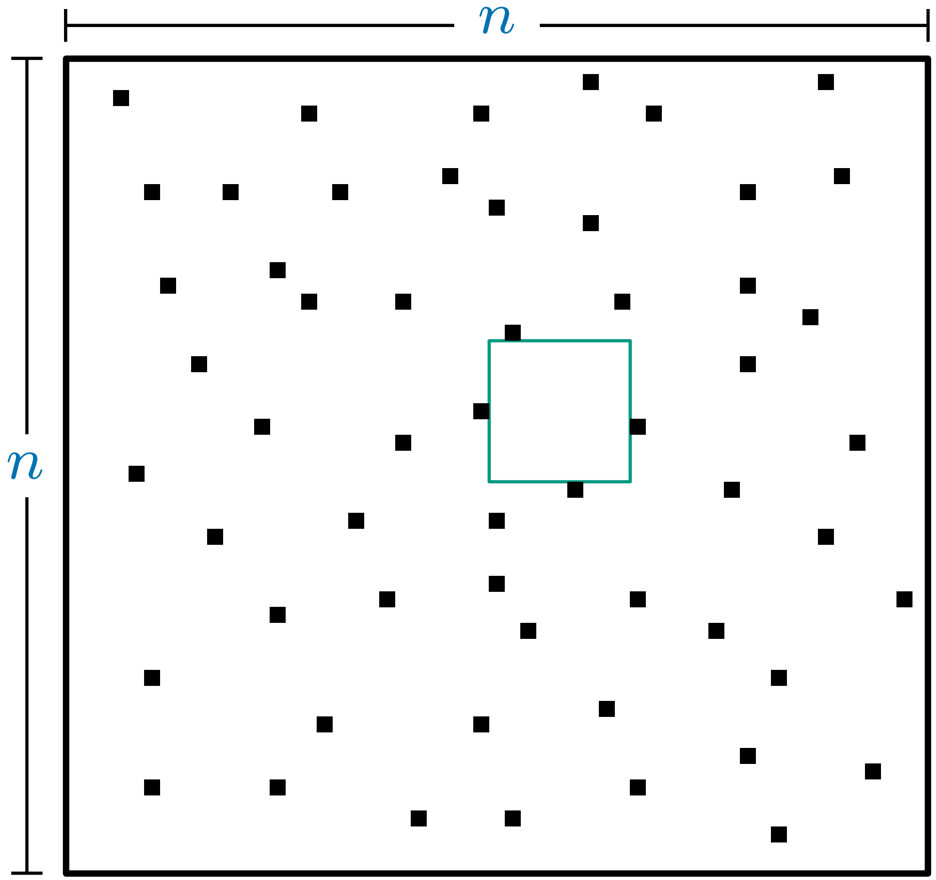
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

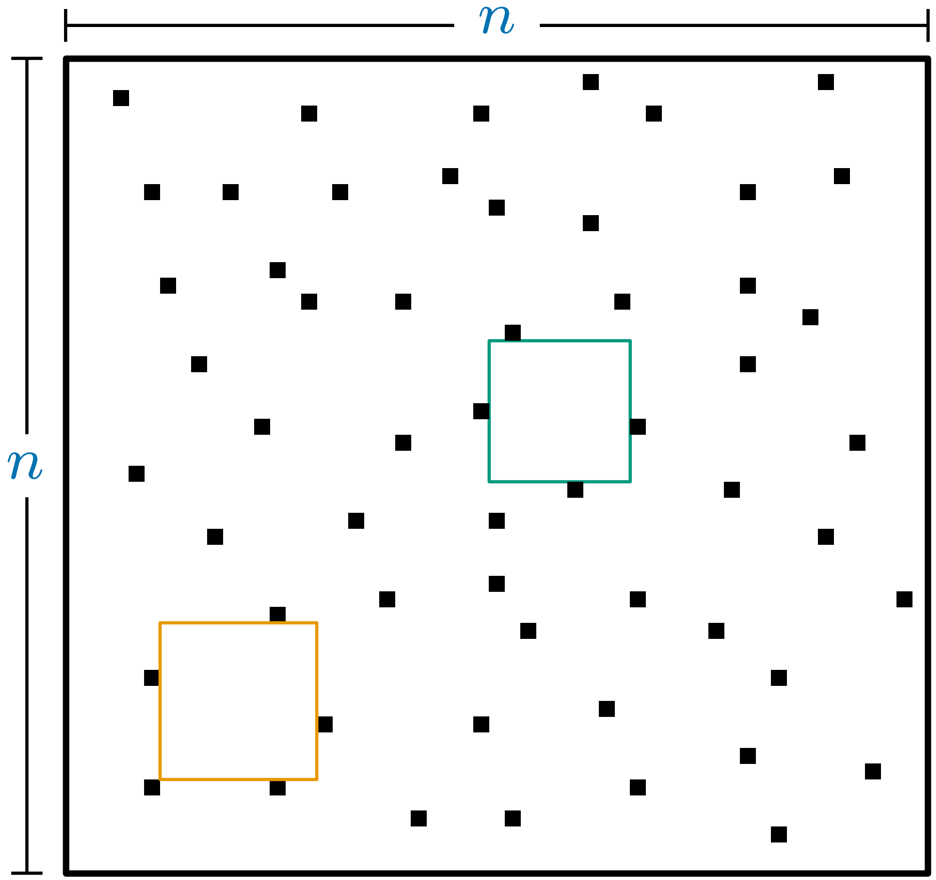
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

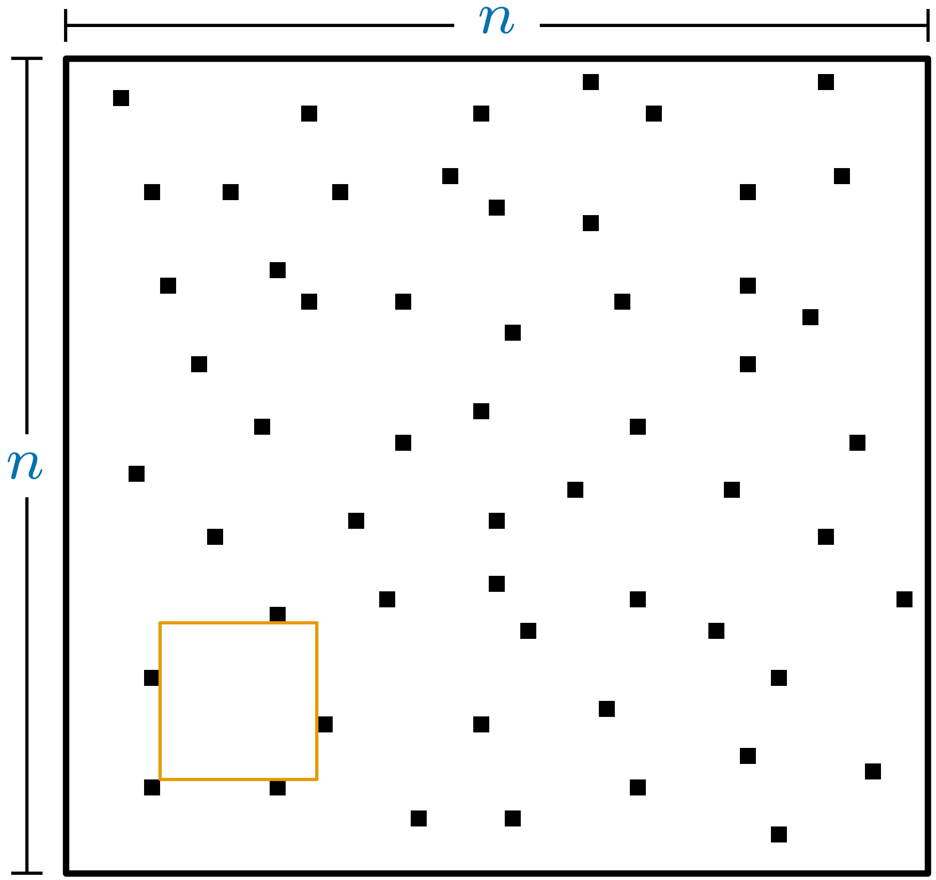
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

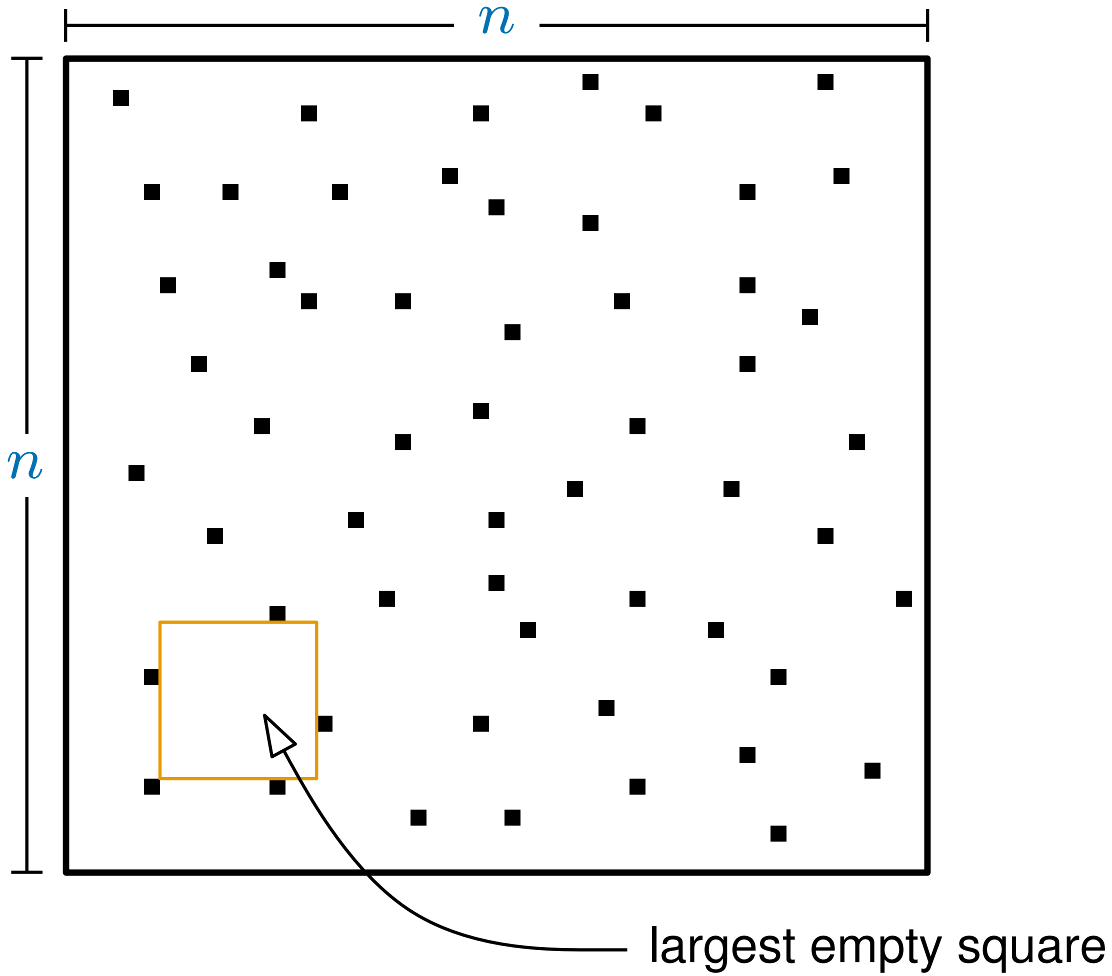
i.e. without any black pixels



Largest Empty Square

Problem Given an $n \times n$ monochrome image, find the largest empty square.

i.e. without any black pixels



1. Find a recursive formula

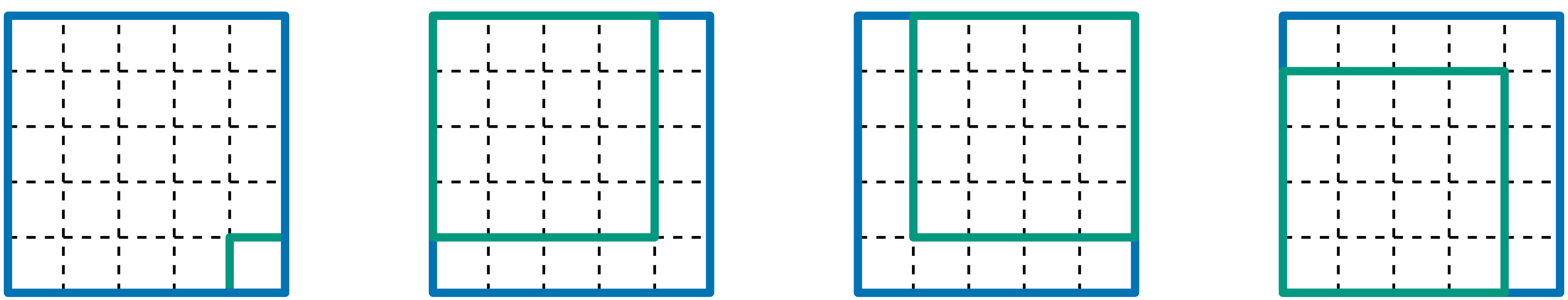
To find a recursive formulation of this problem, consider the following fact:

Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**



1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

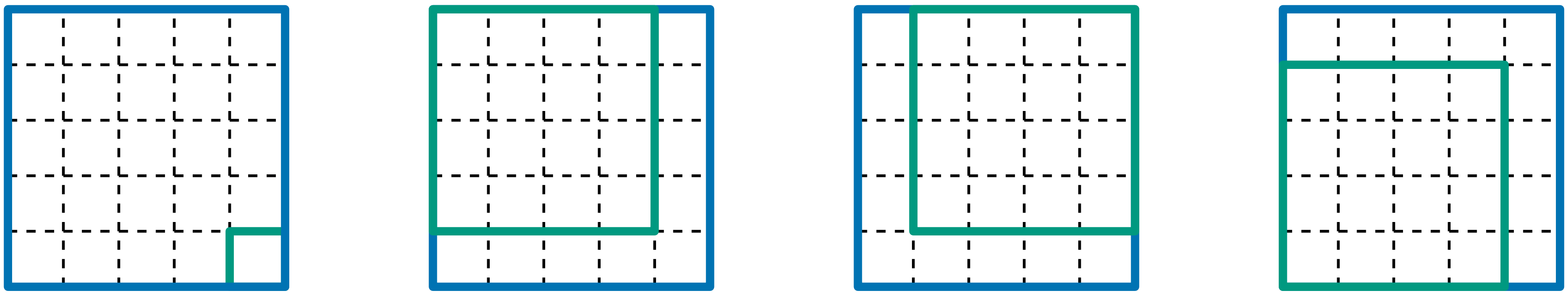
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

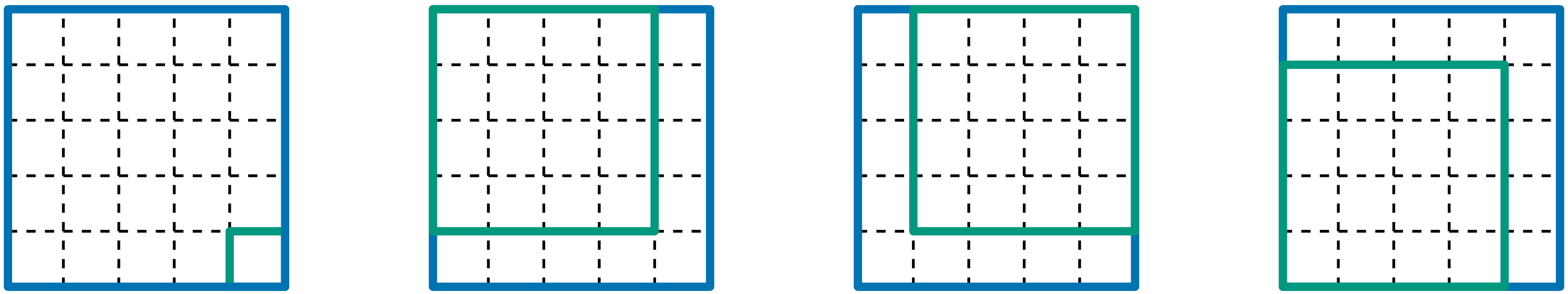
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If S is empty then all four ☐ are empty

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

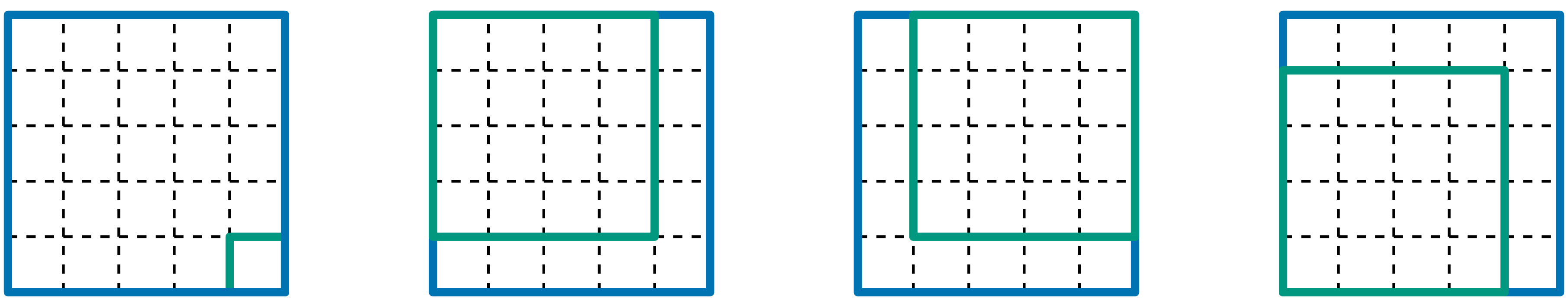
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

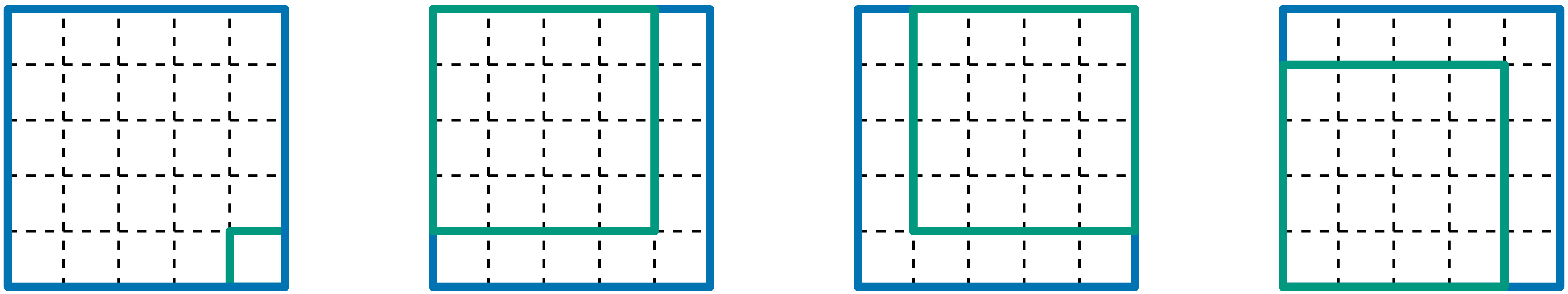
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all  are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

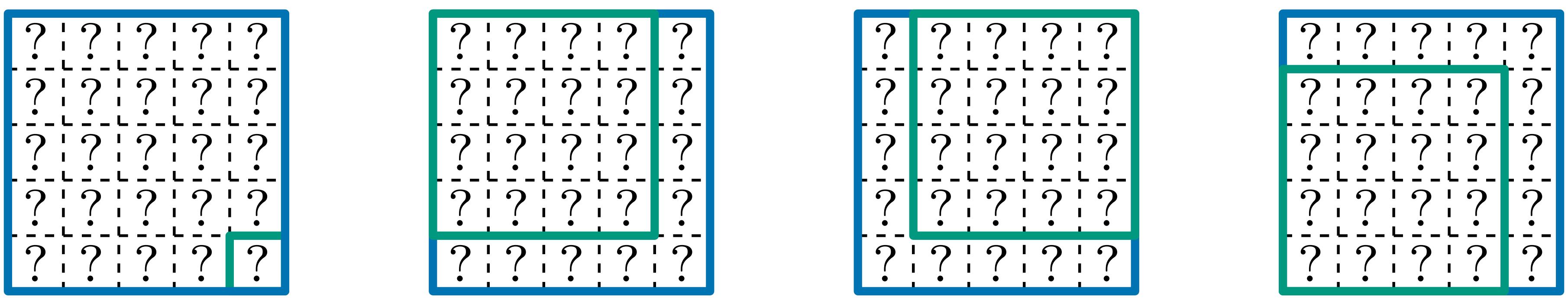
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all $\square$ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

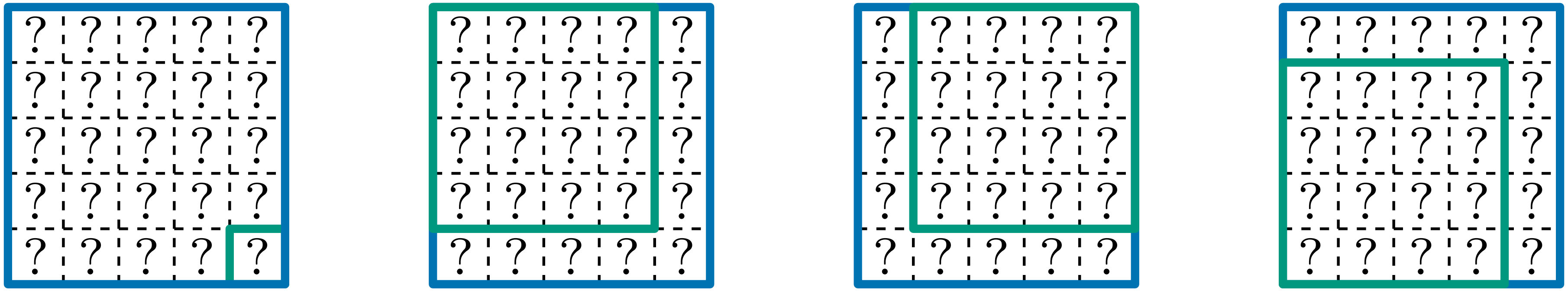
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



empty

If all ☐ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

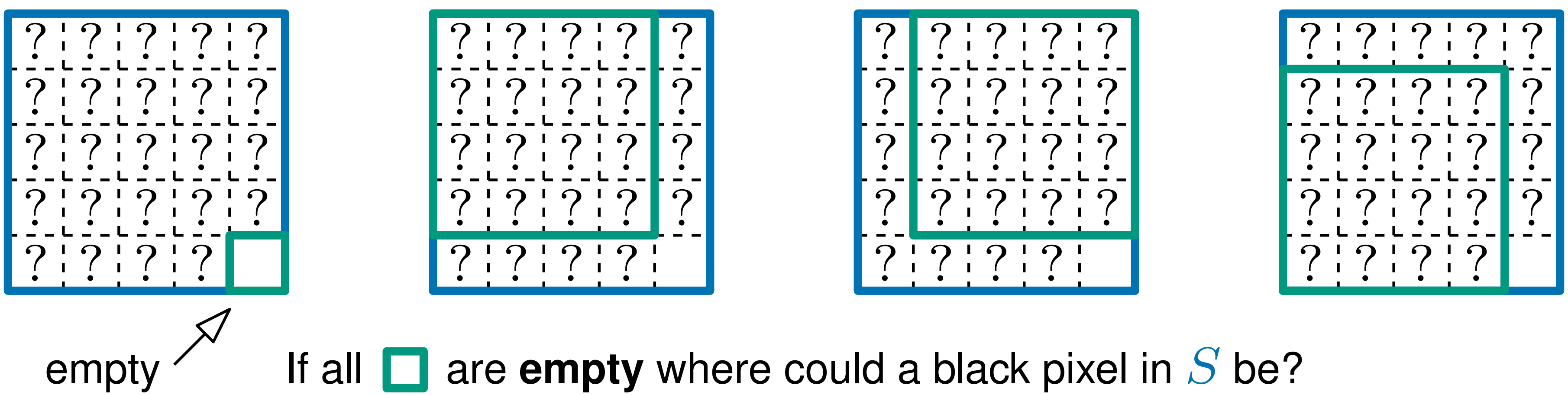
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all □ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

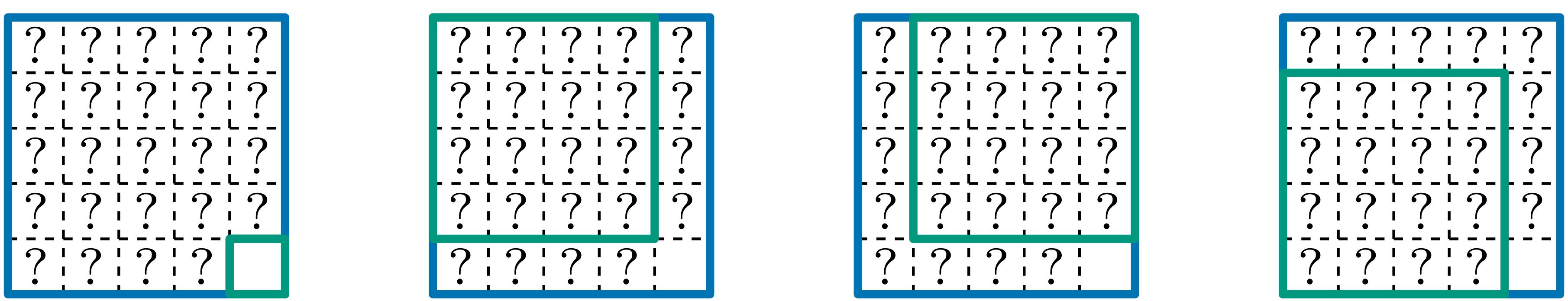
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

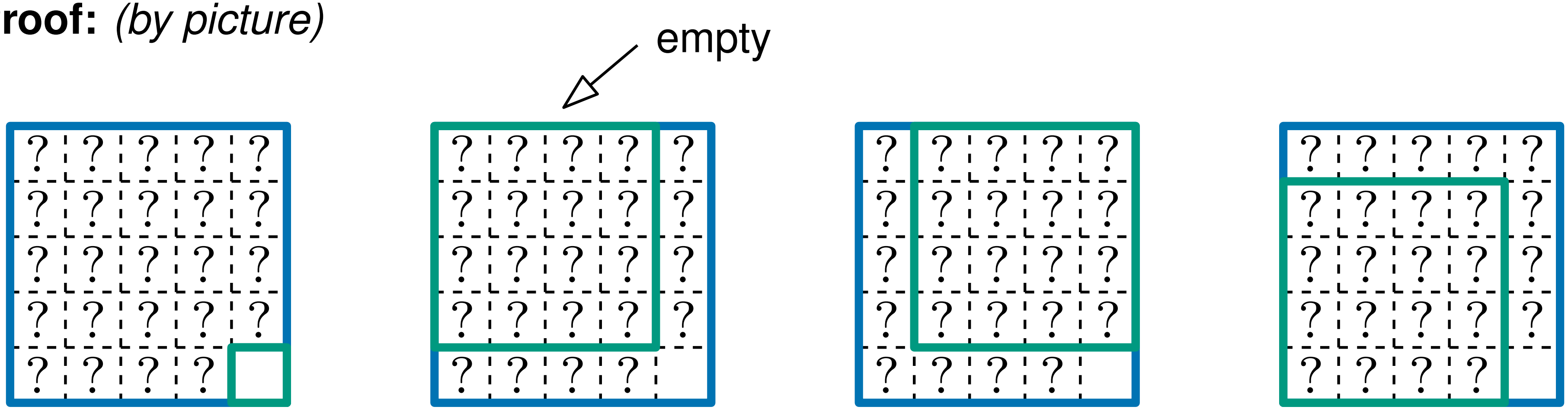
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all ☐ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

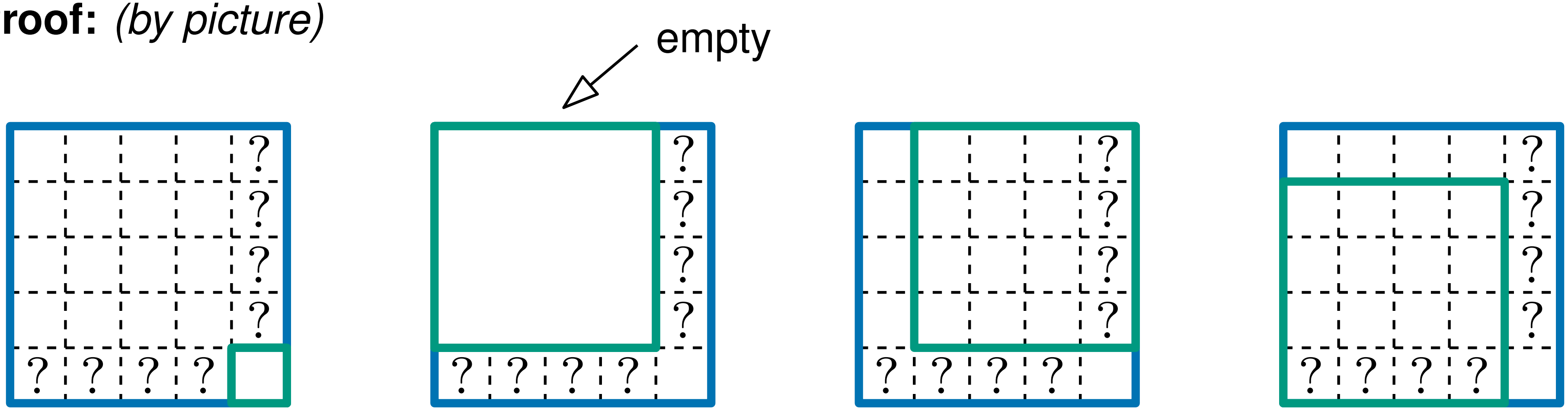
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all ☐ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

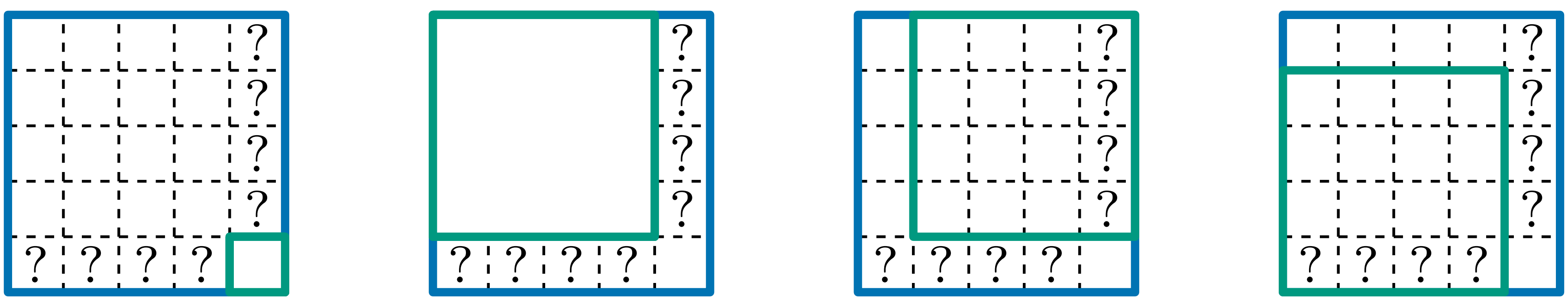
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all $\square$ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

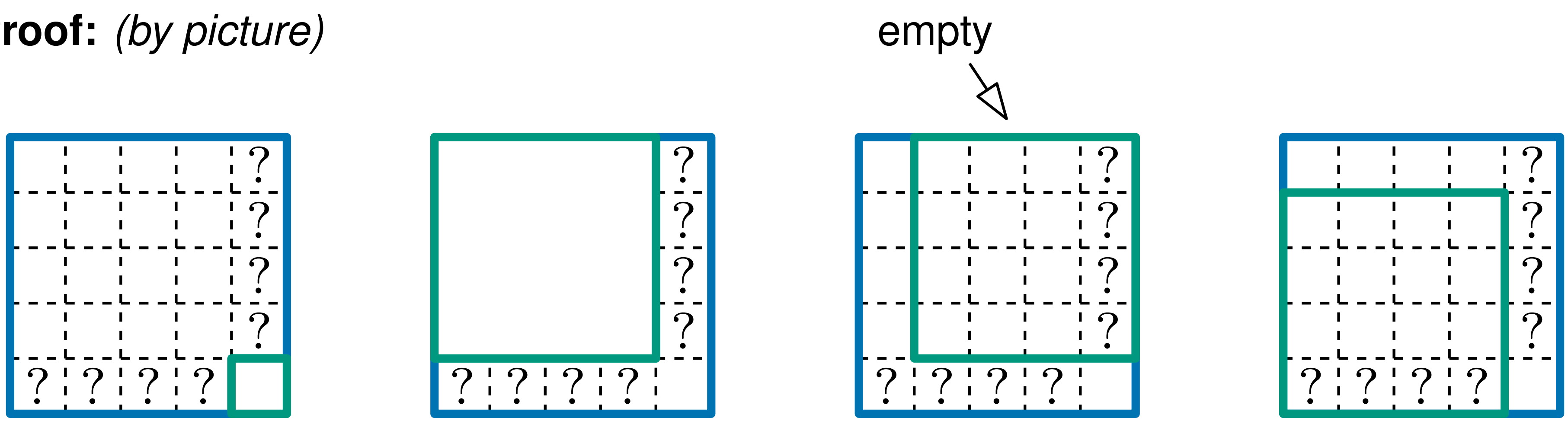
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all ☐ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

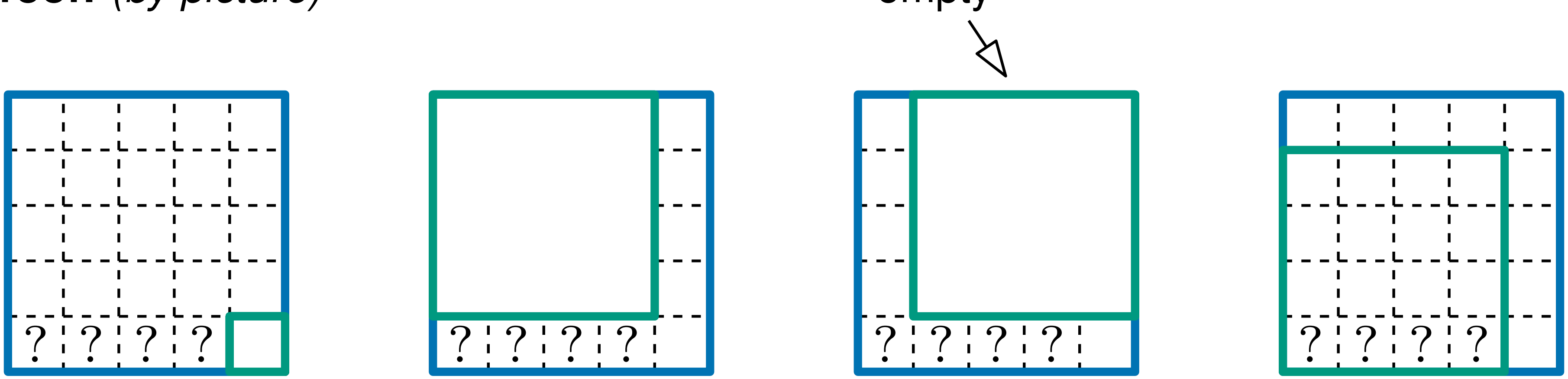
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all $\square$ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

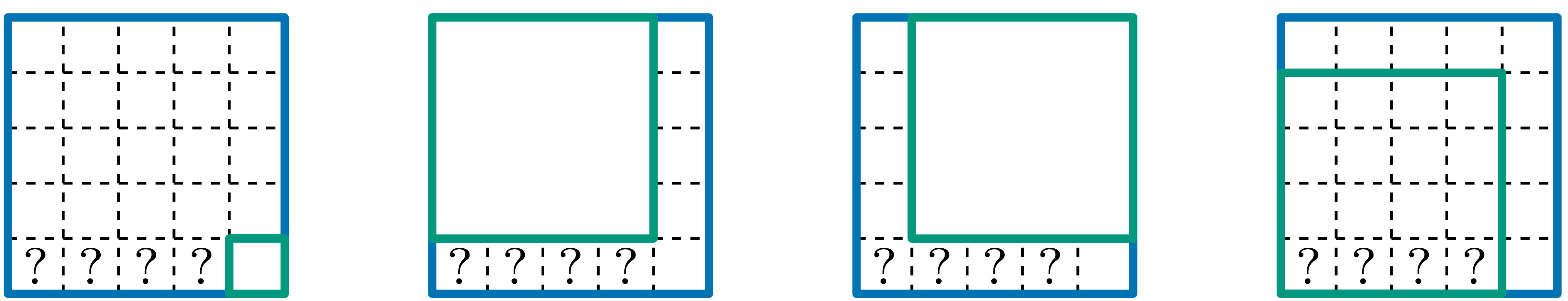
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all ☐ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

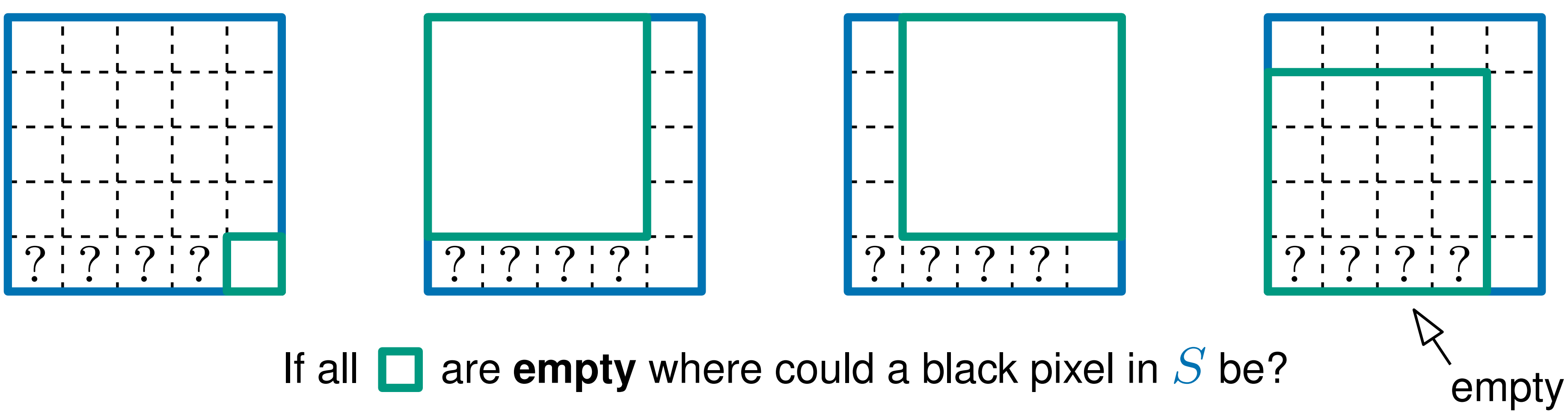
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all □ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

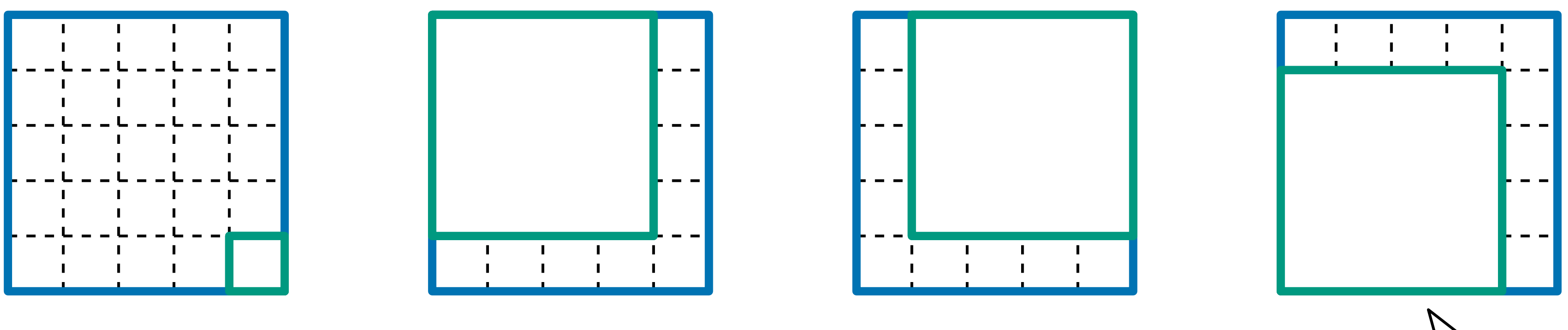
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all  are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

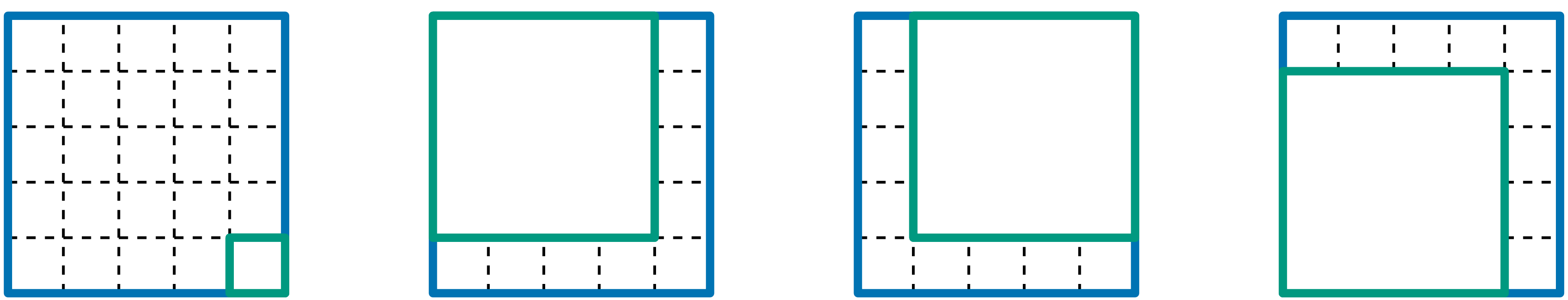
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*



If all $\square$ are **empty** where could a black pixel in S be?

1. Find a recursive formula

To find a recursive formulation of this problem, consider the following fact:

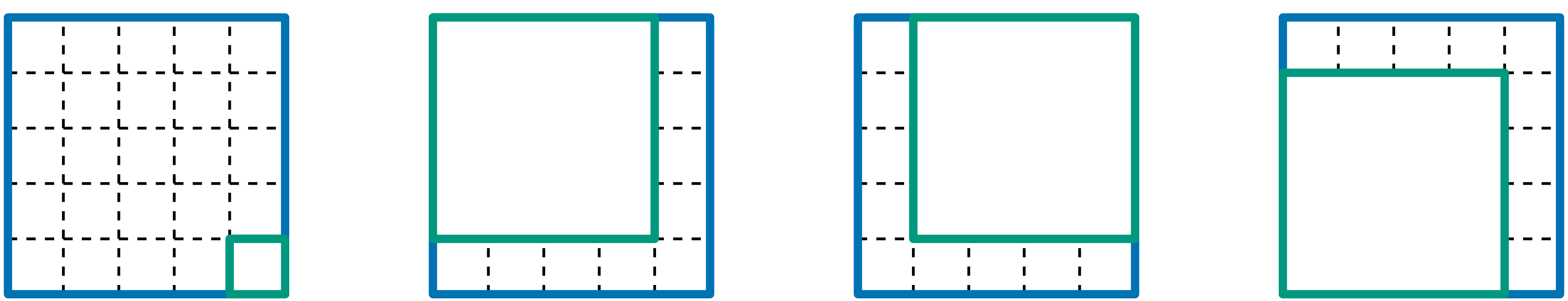
Any $m \times m$ square of pixels, S is **empty** *if and only if*

The bottom right pixel of S is **empty** and

The three $(m-1) \times (m-1)$ squares in the

top left, top right and bottom left of S are **empty**

Proof: *(by picture)*

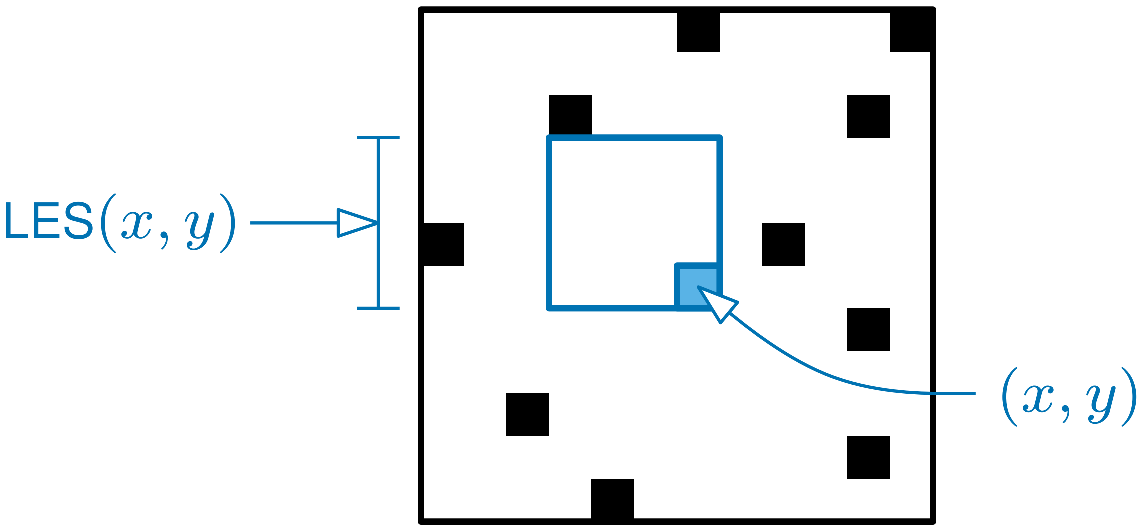


If all $\square$ are **empty** where could a black pixel in S be?

If all $\square$ are **empty** then S is **empty**

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$.

If (x, y) is empty and in the first row or column,

$$\text{LES}(x, y) = 1.$$

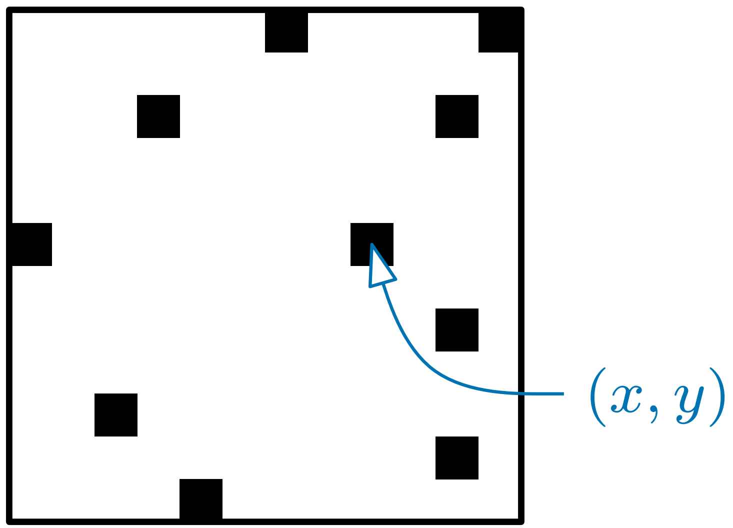
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)

LES$(x, y) = 0$



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$.

If (x, y) is empty and in the first row or column,

$$\text{LES}(x, y) = 1.$$

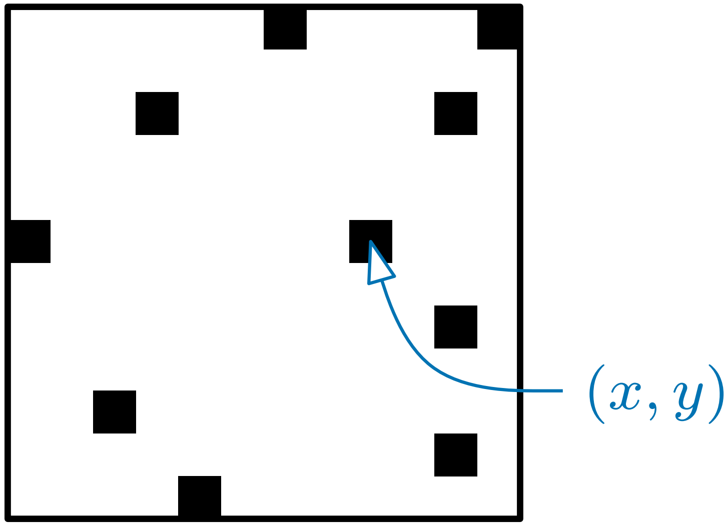
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)

$\text{LES}(x, y) = 0$



Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

$$\text{LES}(x, y) = 1.$$

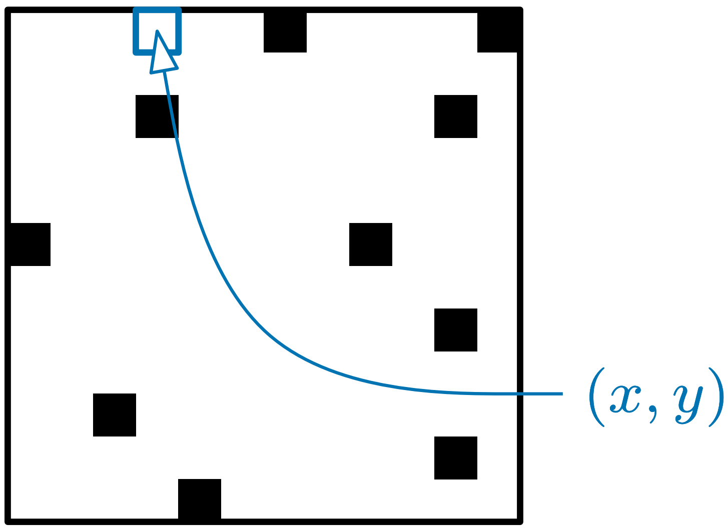
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)

$\text{LES}(x, y) = 1$



Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$.

If (x, y) is empty and in the first row or column,

$$\text{LES}(x, y) = 1.$$

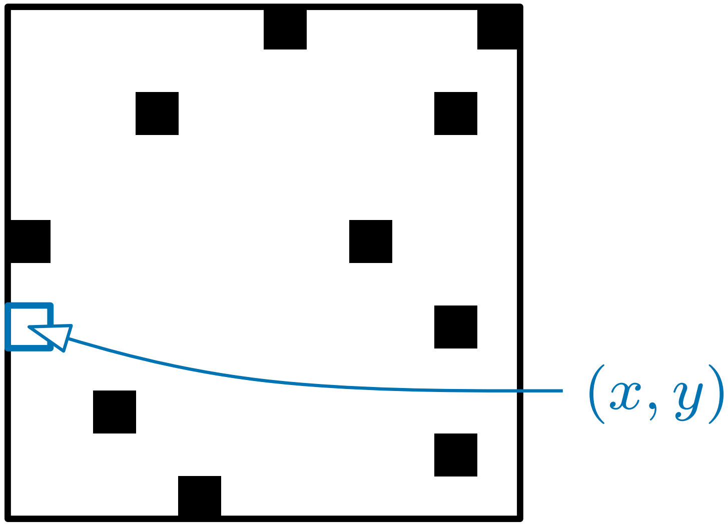
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)

$\text{LES}(x, y) = 1$



Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

$$\text{LES}(x, y) = 1.$$

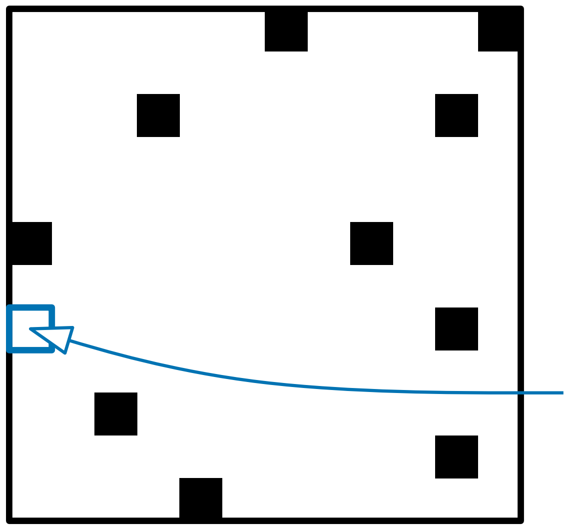
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)

LES$(x, y) = 1$



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

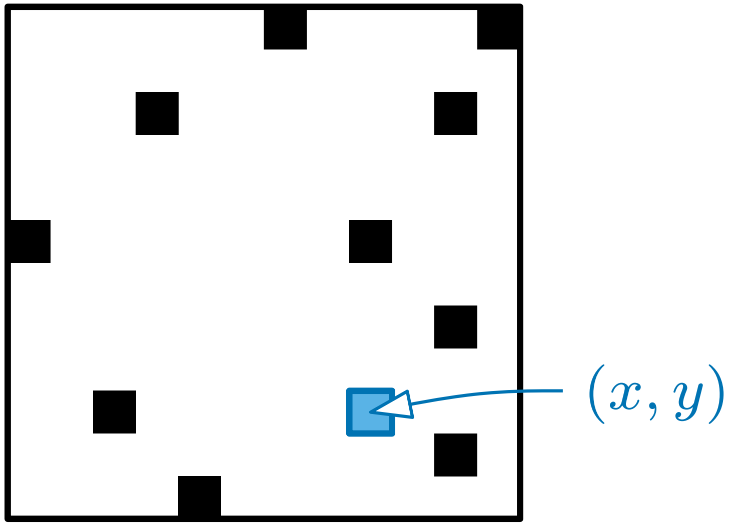
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$.

If (x, y) is empty and in the first row or column,

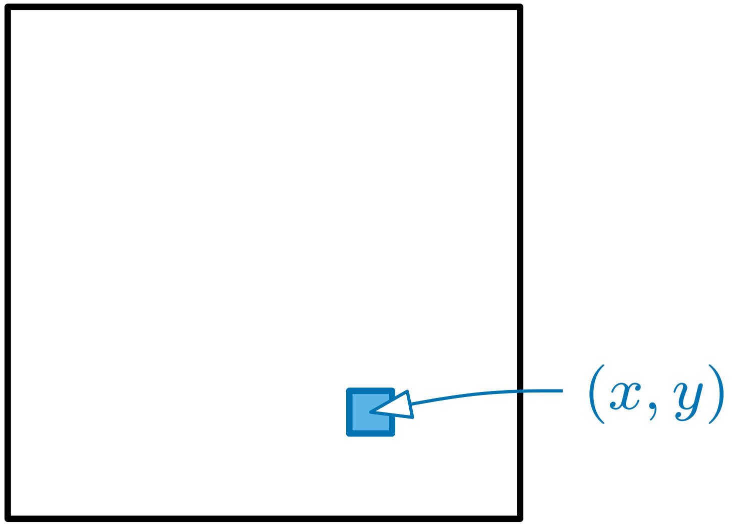
$$\text{LES}(x, y) = 1.$$

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

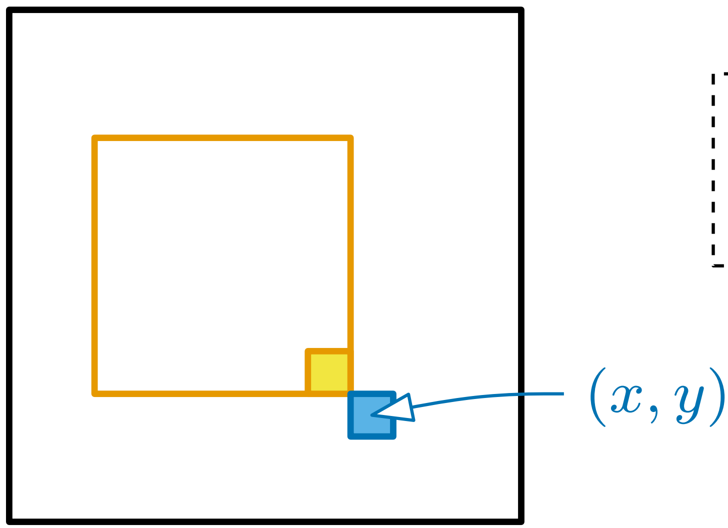
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\mathsf{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



$$\mathsf{LES}(x, y) \leqslant \mathsf{LES}(x-1, y-1) + 1$$

Then:

If the pixel (x, y) is not empty then $\mathsf{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

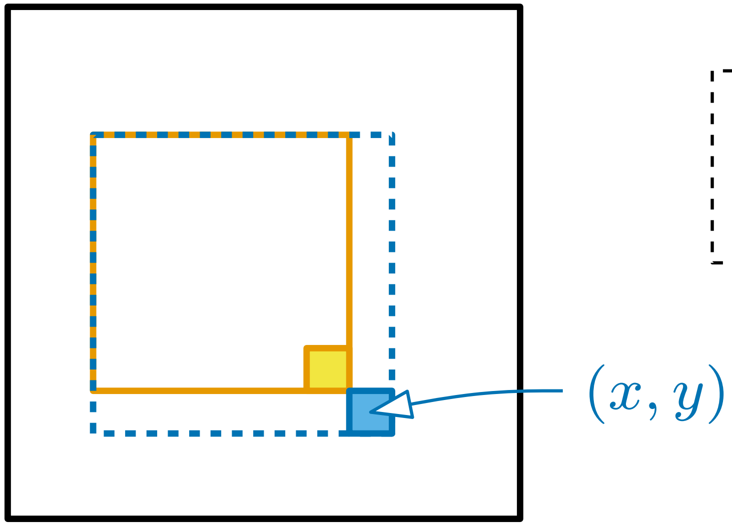
$$\mathsf{LES}(x, y) = 1.$$ ✓

If (x, y) is empty and not in the first row or column,

$$\mathsf{LES}(x, y) = \min(\mathsf{LES}(x-1, y-1), \mathsf{LES}(x-1, y), \mathsf{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



$\text{LES}(x, y) \leqslant \text{LES}(x-1, y-1) + 1$

Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

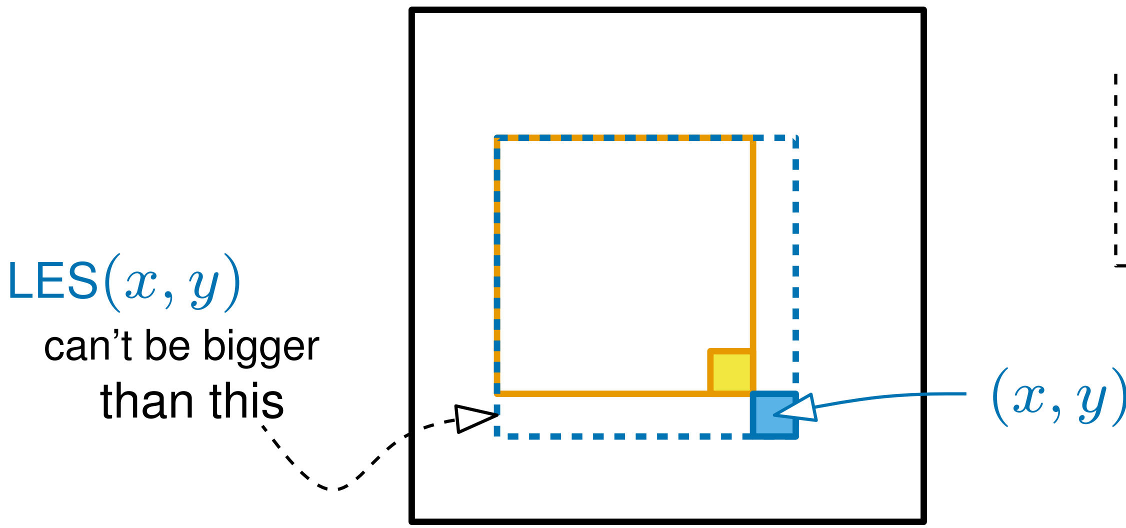
$$\text{LES}(x, y) = 1. \checkmark$$

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



$$\text{LES}(x, y) \leqslant \text{LES}(x-1, y-1) + 1$$

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

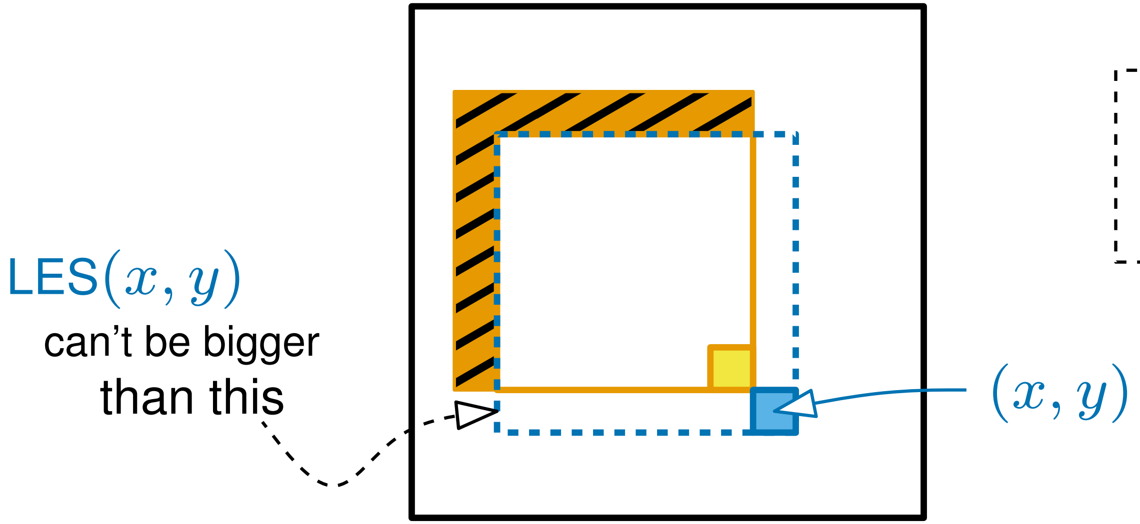
$$\text{LES}(x, y) = 1. ✓$$

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



LES$(x, y) \leqslant$ LES$(x-1, y-1)+1$

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

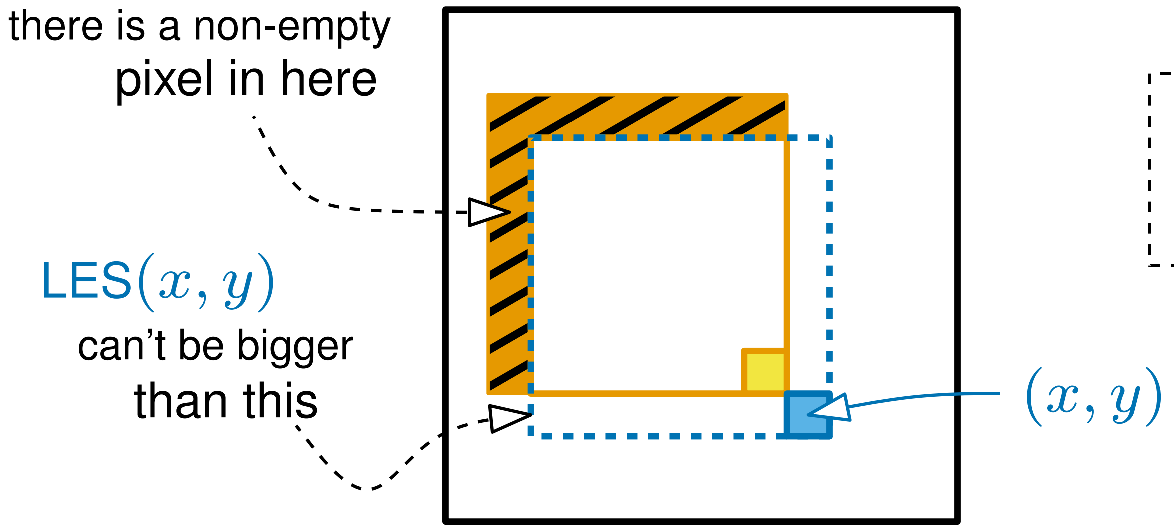
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



LES$(x, y) \leqslant$ LES$(x - 1, y - 1) + 1$

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

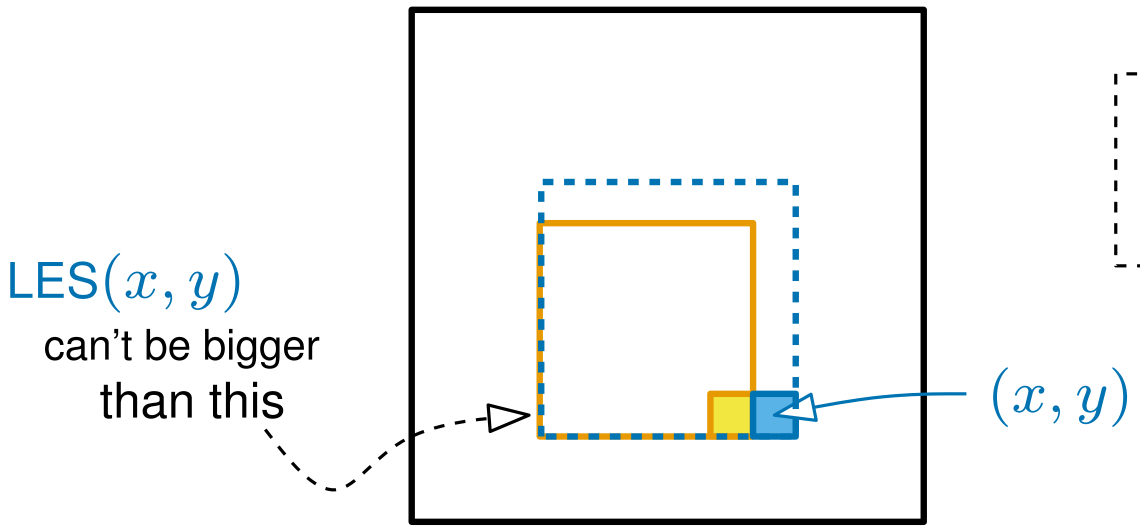
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



LES$(x, y) \leqslant$ LES$(x-1, y)+1$

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$.

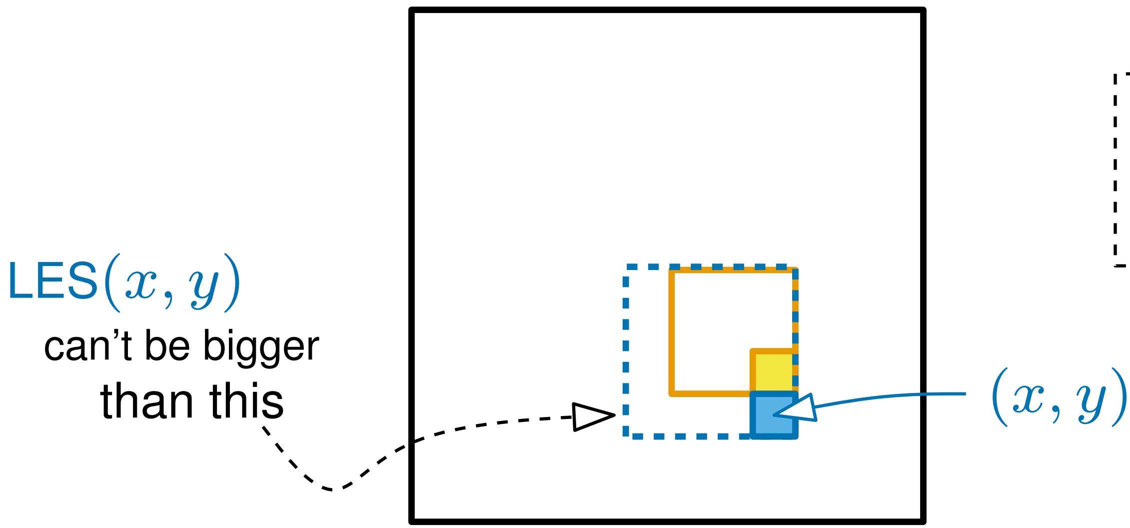
If (x, y) is empty and in the first row or column,
LES$(x, y) = 1$.

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



LES$(x, y) \leqslant$ LES$(x, y-1) + 1$

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

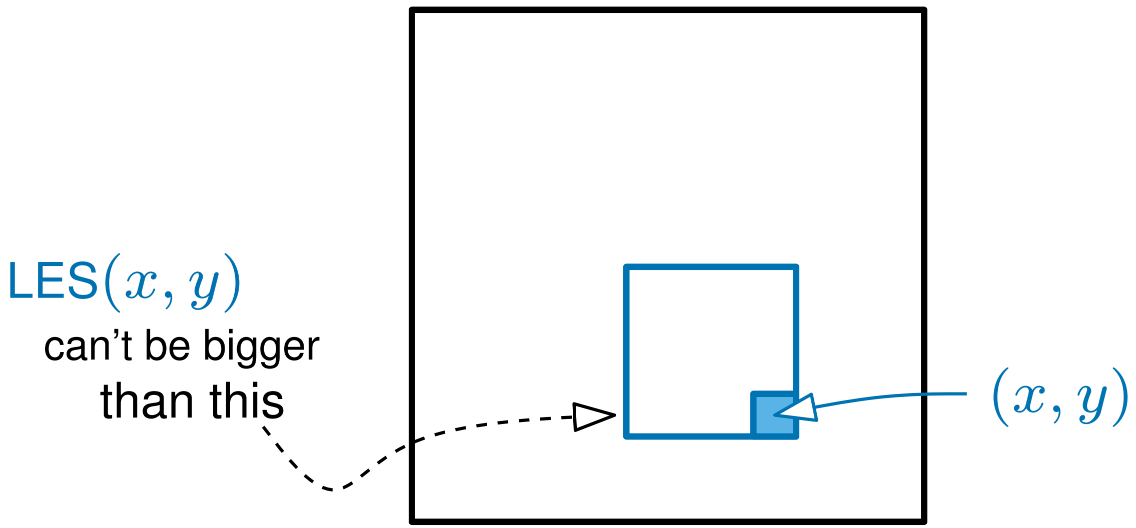
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

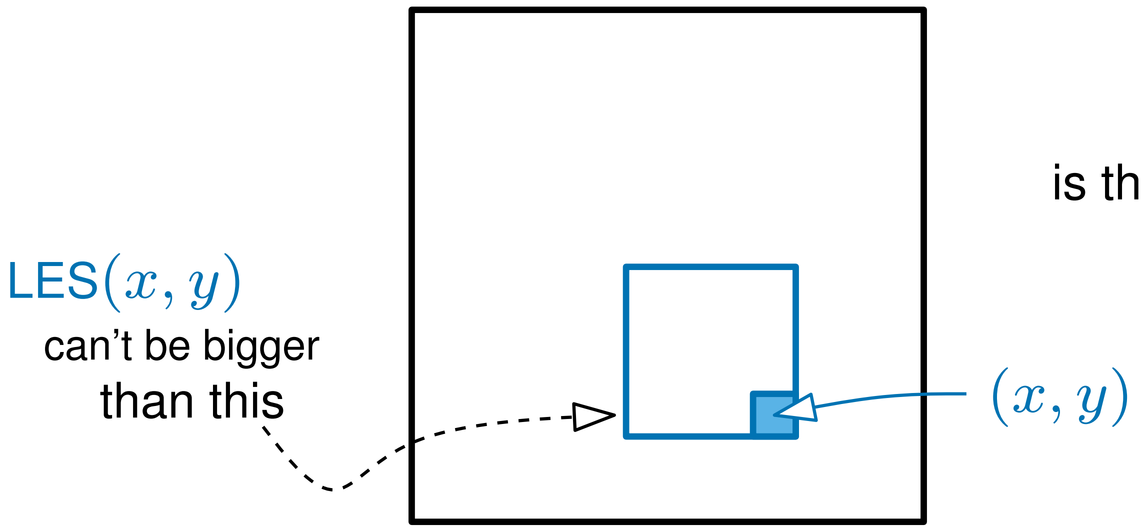
$$\text{LES}(x, y) = 1.$$ ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

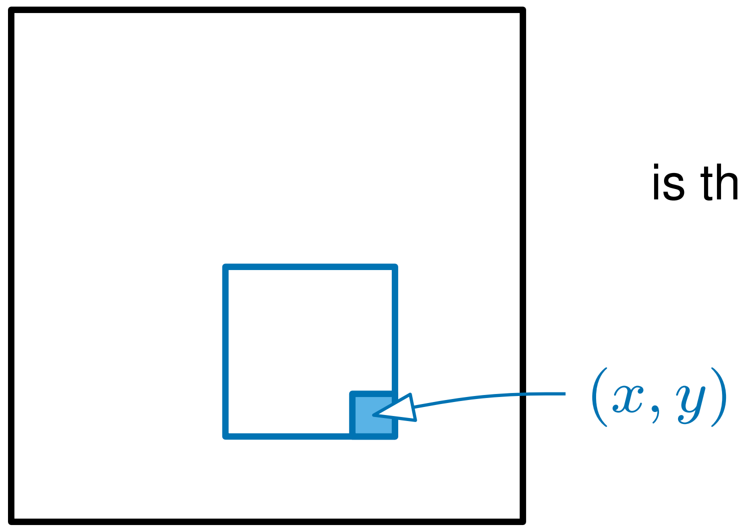
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

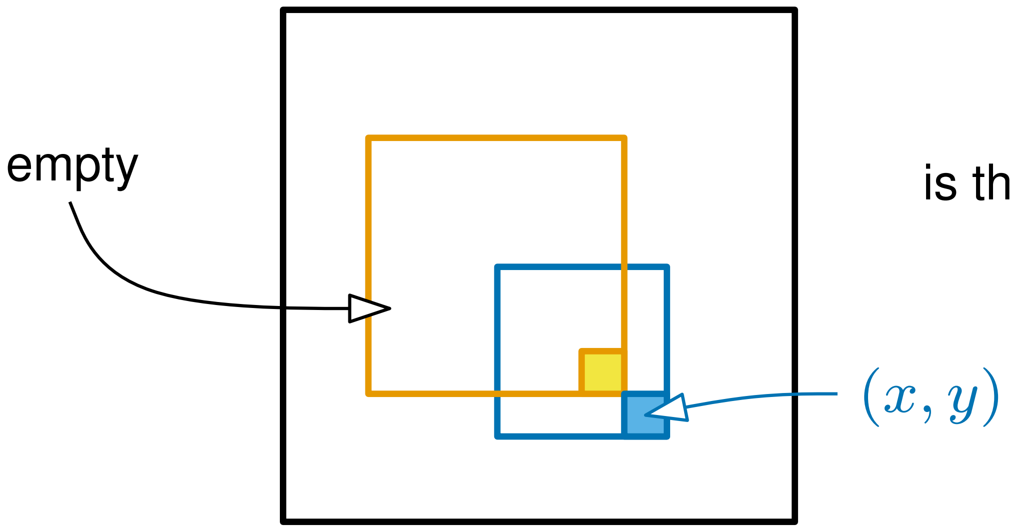
$$\text{LES}(x, y) = 1. \checkmark$$

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

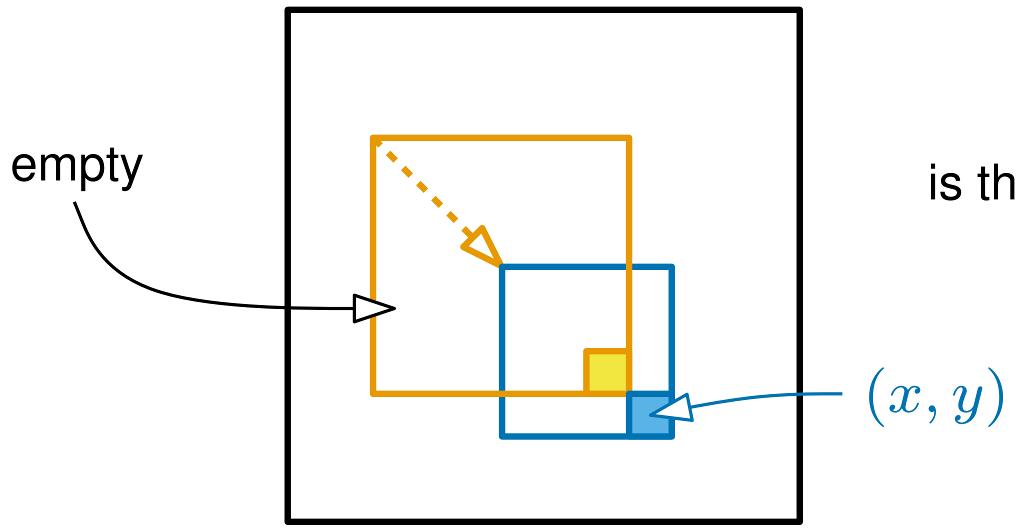
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

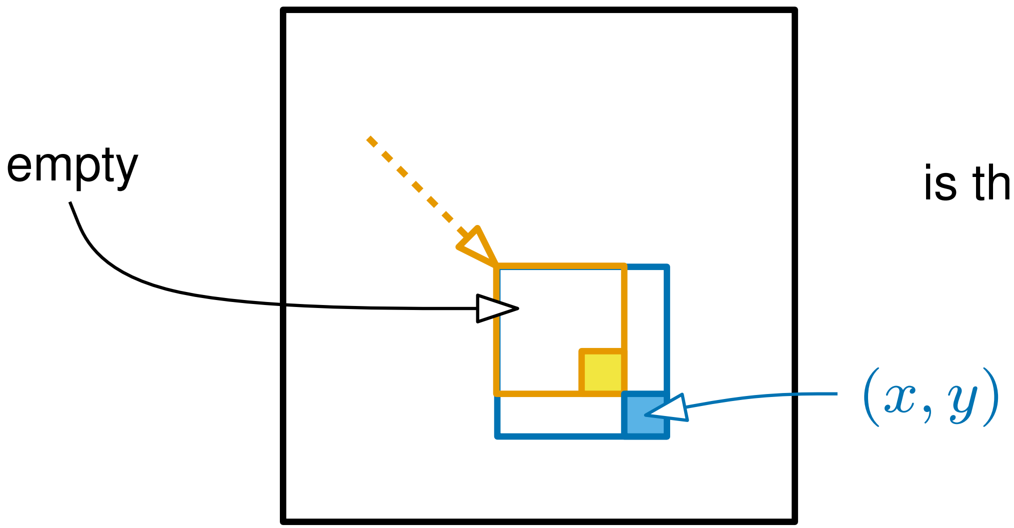
$$\text{LES}(x, y) = 1.$$ ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

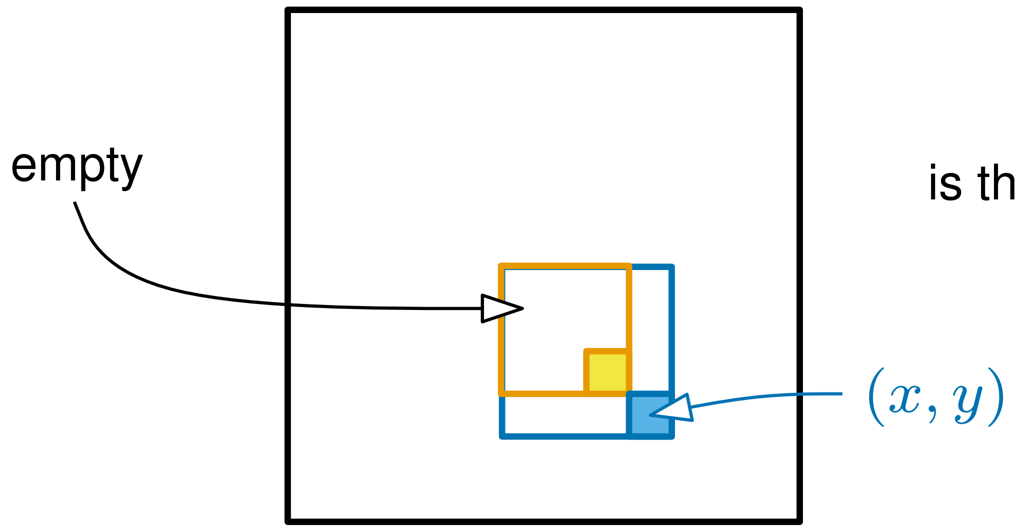
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\text{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then $\text{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

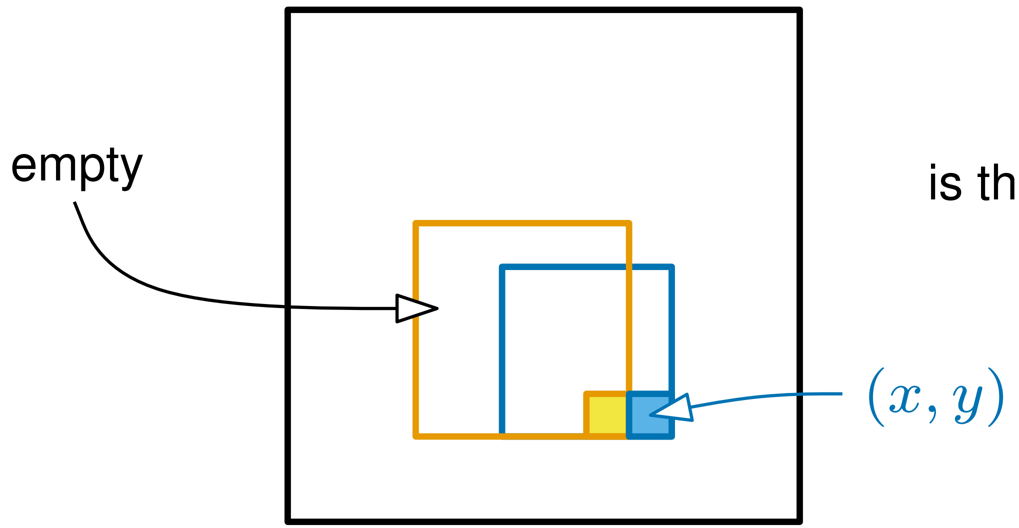
$$\text{LES}(x, y) = 1. ✓$$

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

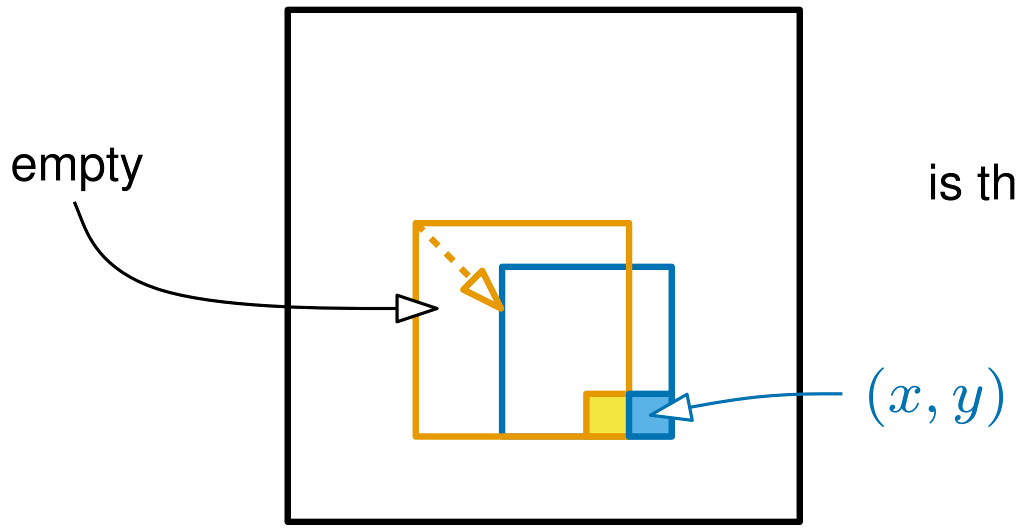
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

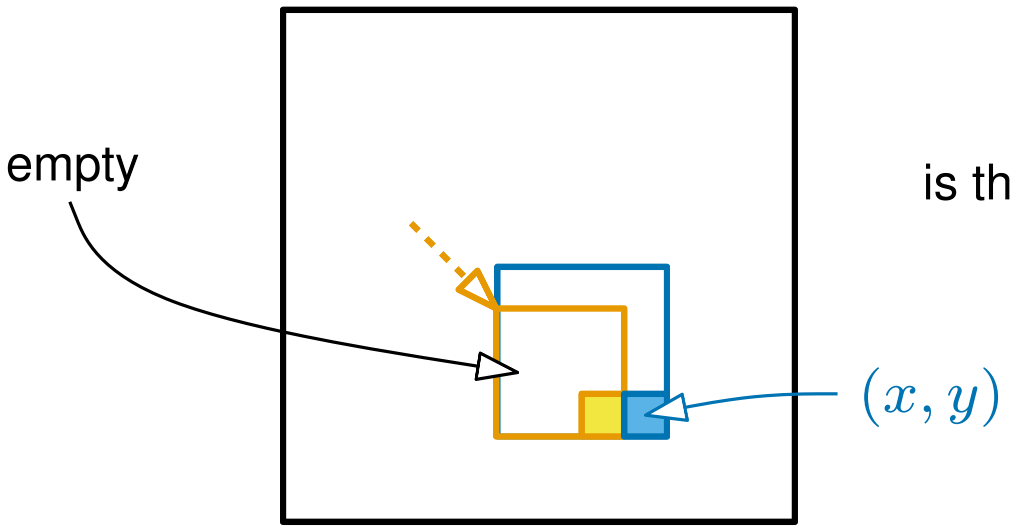
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$.

If (x, y) is empty and in the first row or column,

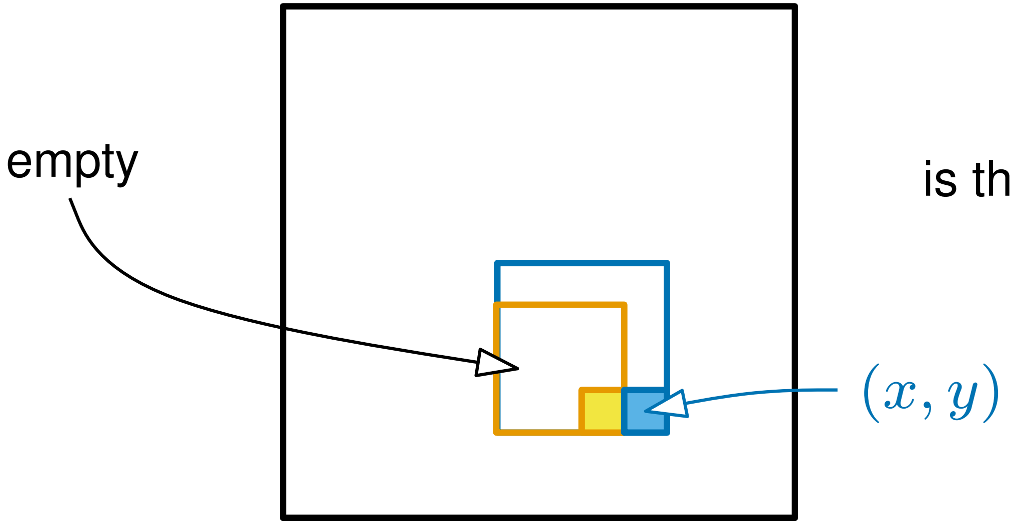
LES$(x, y) = 1$.

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

LES$(x, y) = 1$. ✓

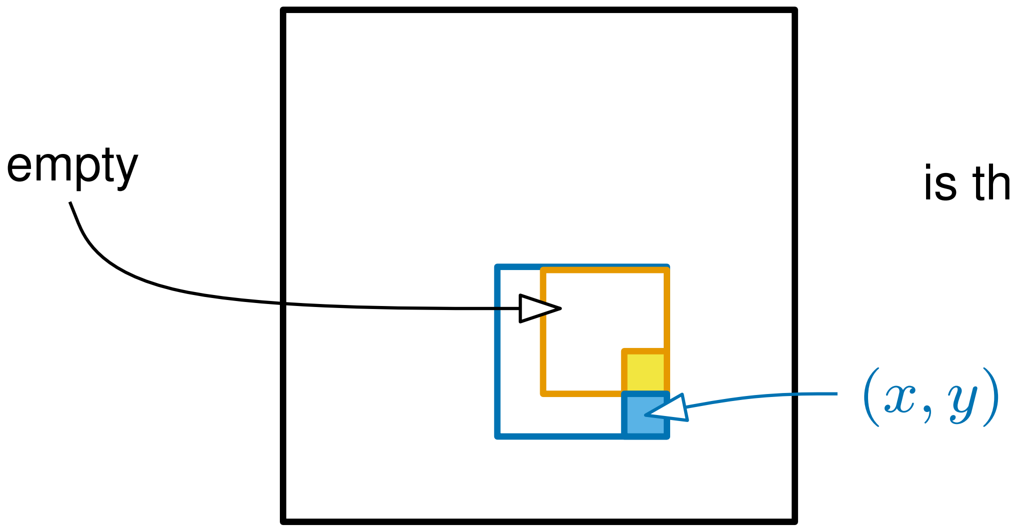
If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square

whose bottom right is at (x, y)



is this square always empty?

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

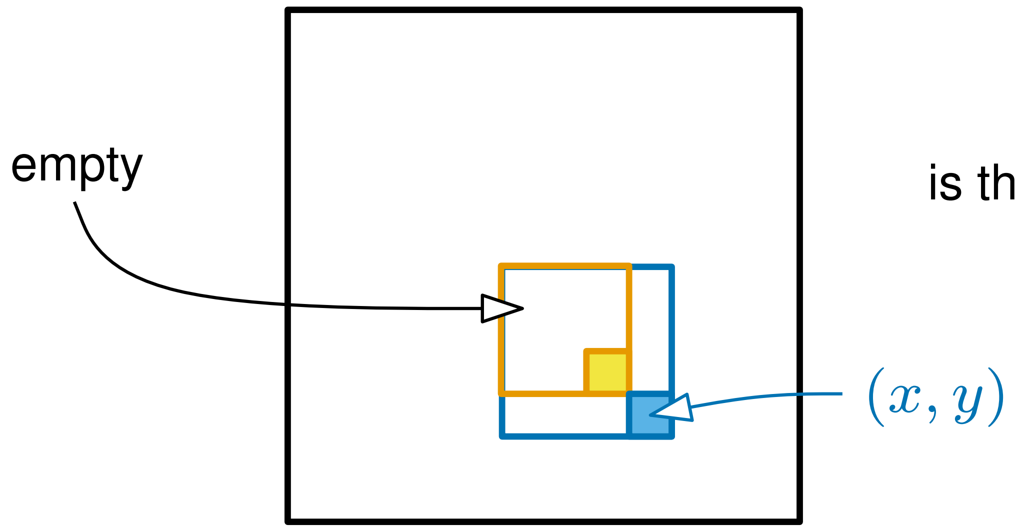
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

LES$(x, y) = \min($LES$(x-1, y-1)$, LES$(x-1, y)$, LES$(x, y-1)) + 1$.

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

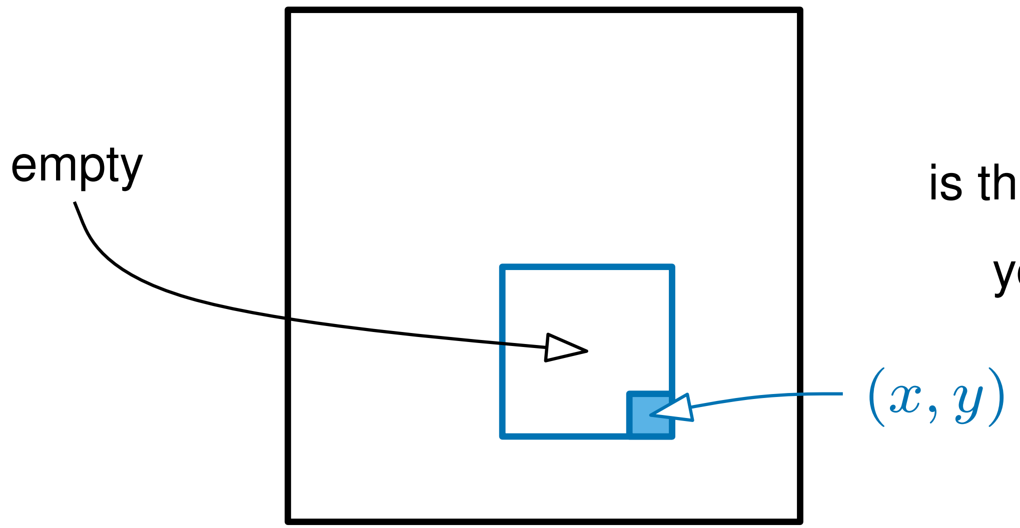
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let $\mathsf{LES}(x, y)$ be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

yes, by the proof on the previous slide

Then:

If the pixel (x, y) is not empty then $\mathsf{LES}(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

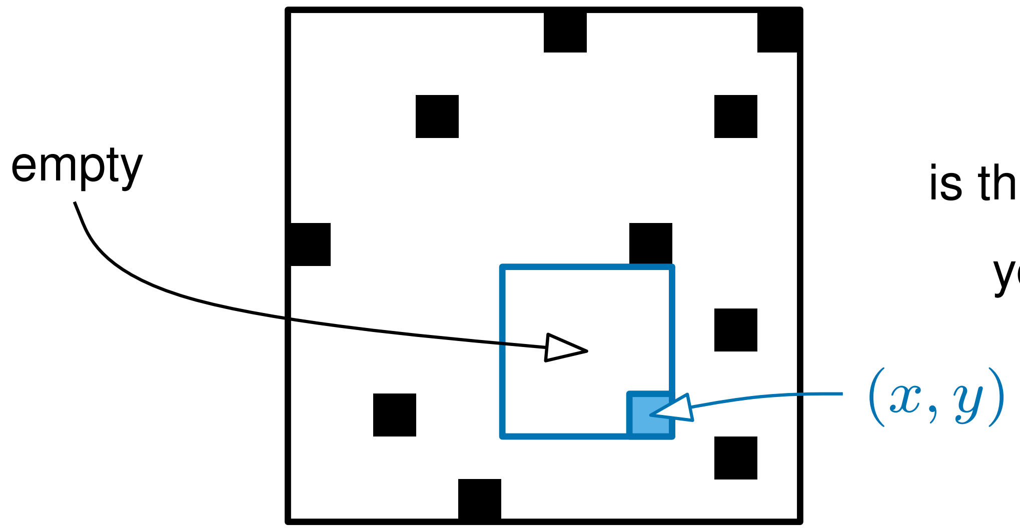
$\mathsf{LES}(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\mathsf{LES}(x, y) = \min(\mathsf{LES}(x-1, y-1), \mathsf{LES}(x-1, y), \mathsf{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

yes, by the proof on the previous slide

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

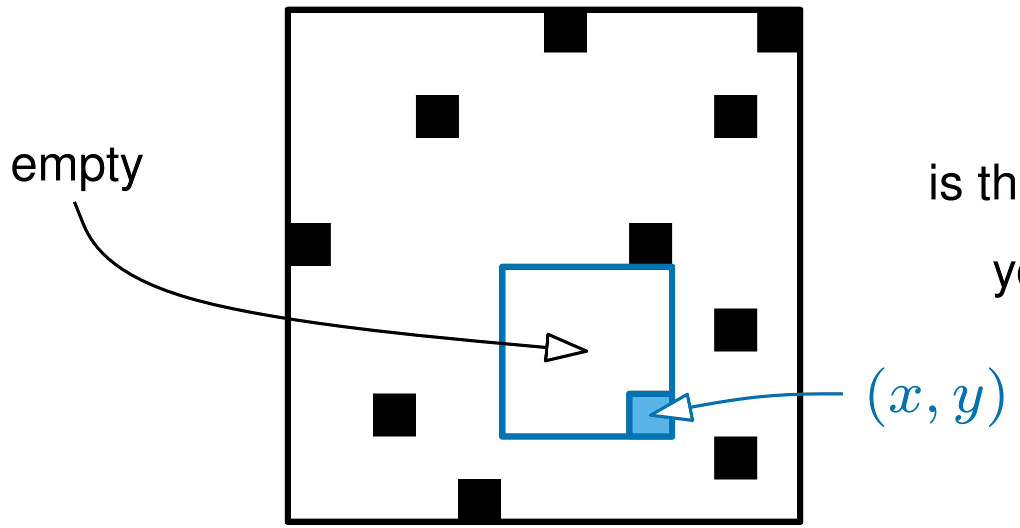
LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$

1. Find a recursive formula

Let LES(x, y) be the size (i.e. side length) of the largest empty square whose bottom right is at (x, y)



is this square always empty?

yes, by the proof on the previous slide

Then:

If the pixel (x, y) is not empty then LES$(x, y) = 0$. ✓

If (x, y) is empty and in the first row or column,

LES$(x, y) = 1$. ✓

If (x, y) is empty and not in the first row or column,

$$\text{LES}(x, y) = \min(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)) + 1.$$ ✓

2. Write down a recursive algorithm

We can use the recursive formula to get a recursive algorithm...

LES(x, y)

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x-1, y-1), LES(x-1, y), LES(x, y-1)) + 1
```

LES(x, y) computes the size of the largest empty square whose bottom right is at (x, y)

Therefore, the maximum of LES(x, y) over all x and y gives the size of the largest empty square in the whole image

2. Write down a recursive algorithm

We can use the recursive formula to get a recursive algorithm...

LES(x, y)

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x-1, y-1), LES(x-1, y), LES(x, y-1)) + 1
```

LES(x, y) computes the size of the largest empty square whose bottom right is at (x, y)

Therefore, the maximum of LES(x, y) over all x and y gives the size of the largest empty square in the whole image

What is the time complexity of this algorithm?

How efficient is the recursive algorithm?

```
LES(x, y)
```

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$...

How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

How efficient is the recursive algorithm?

LES(x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\mathrm{LES}(x-1, y-1), \mathrm{LES}(x-1, y), \mathrm{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

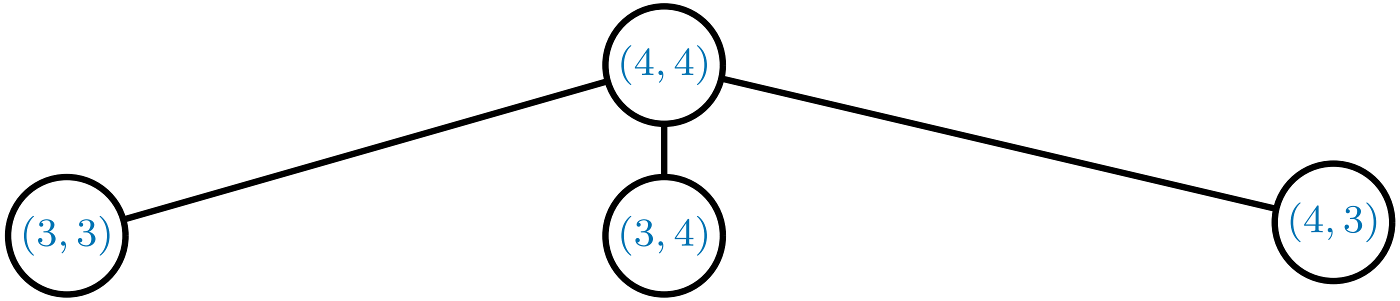
$(4, 4)$

How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

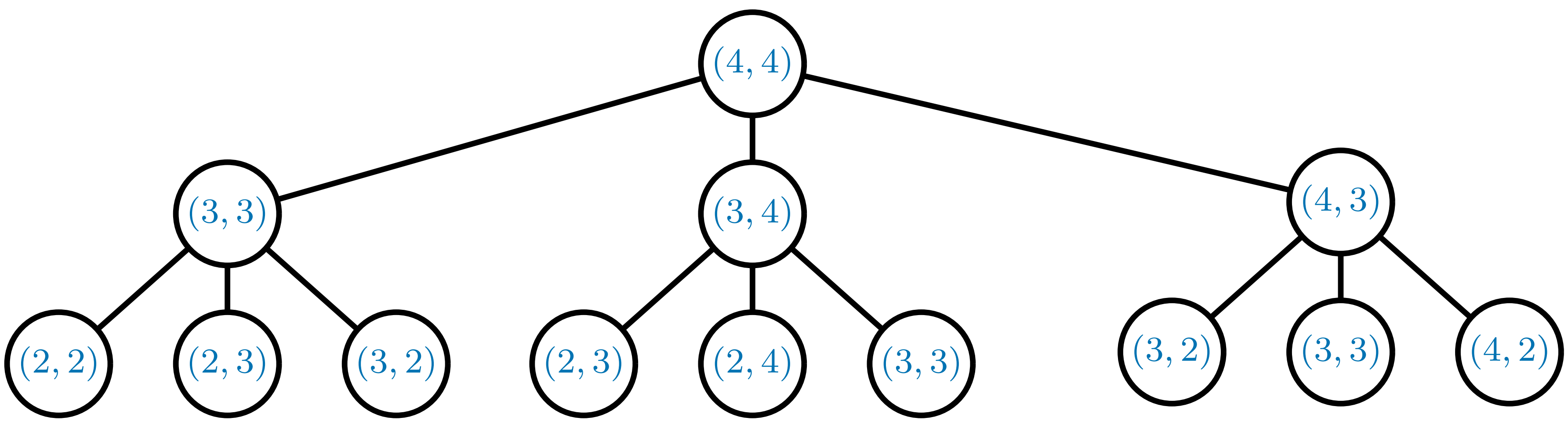


How efficient is the recursive algorithm?

LES (x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

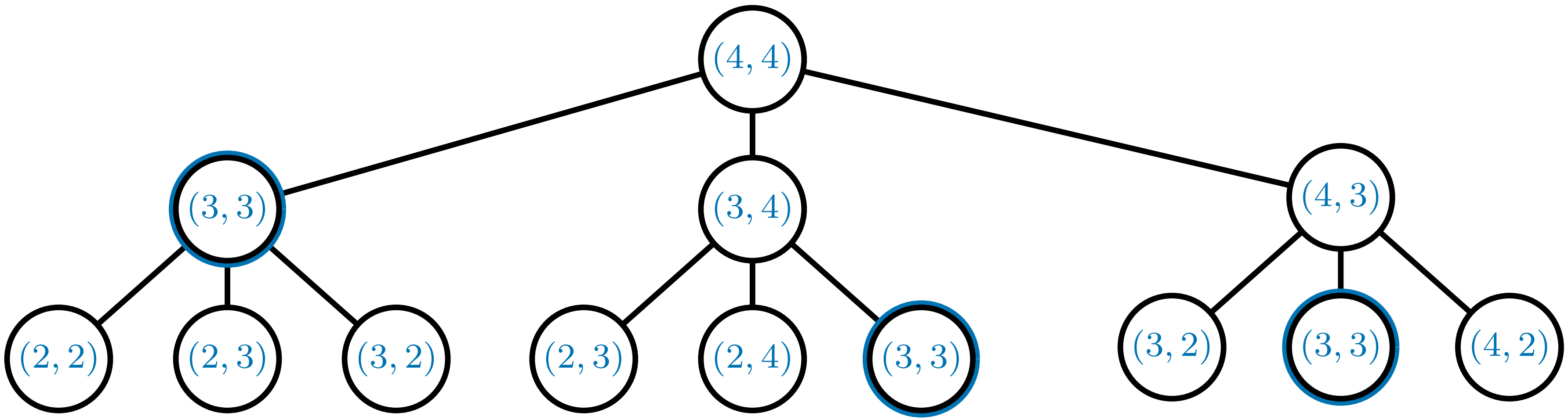


How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x-1, y-1), LES(x-1, y), LES(x, y-1)) + 1
```

Let's compute LES$(4,4)$... *(and consider the recursive calls)*

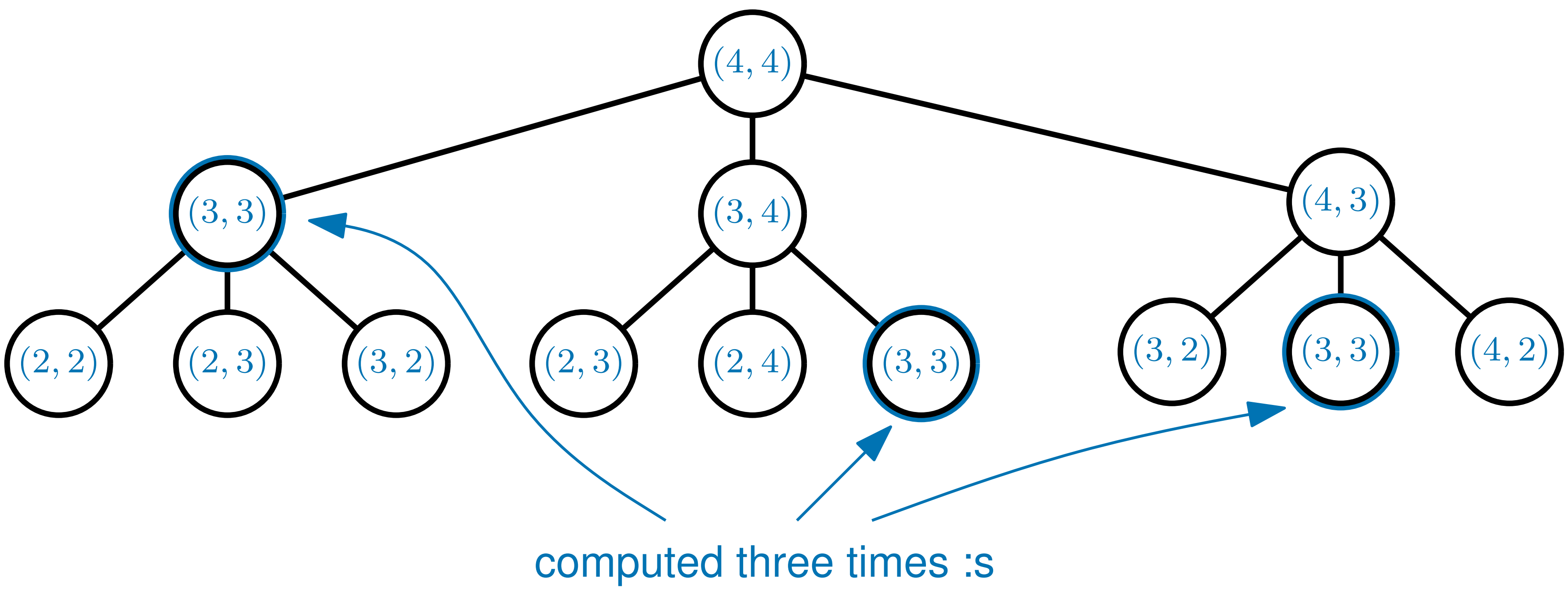


How efficient is the recursive algorithm?

LES (x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

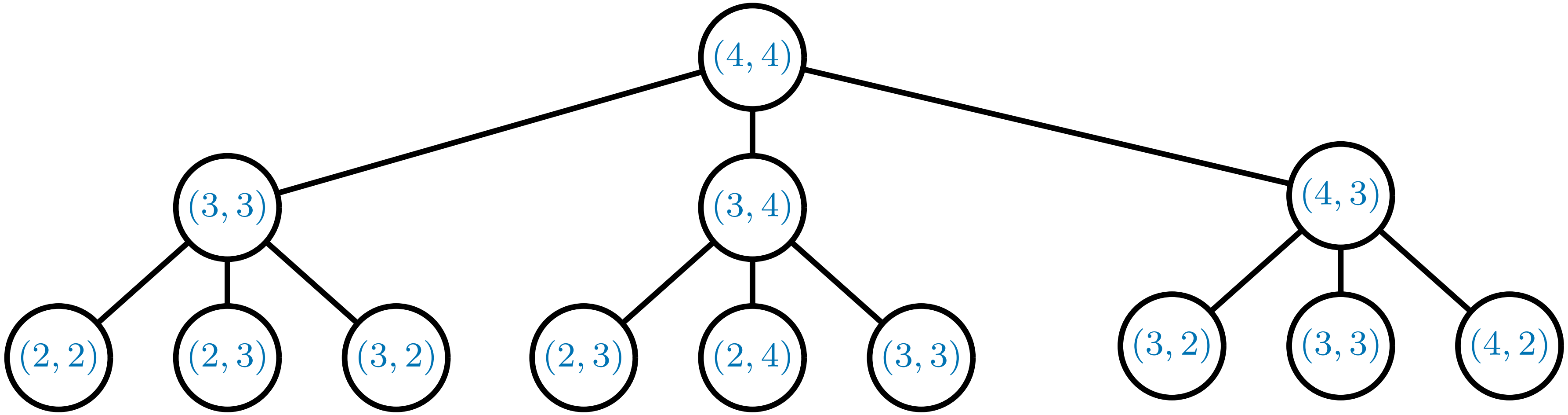


How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4,4)$... *(and consider the recursive calls)*

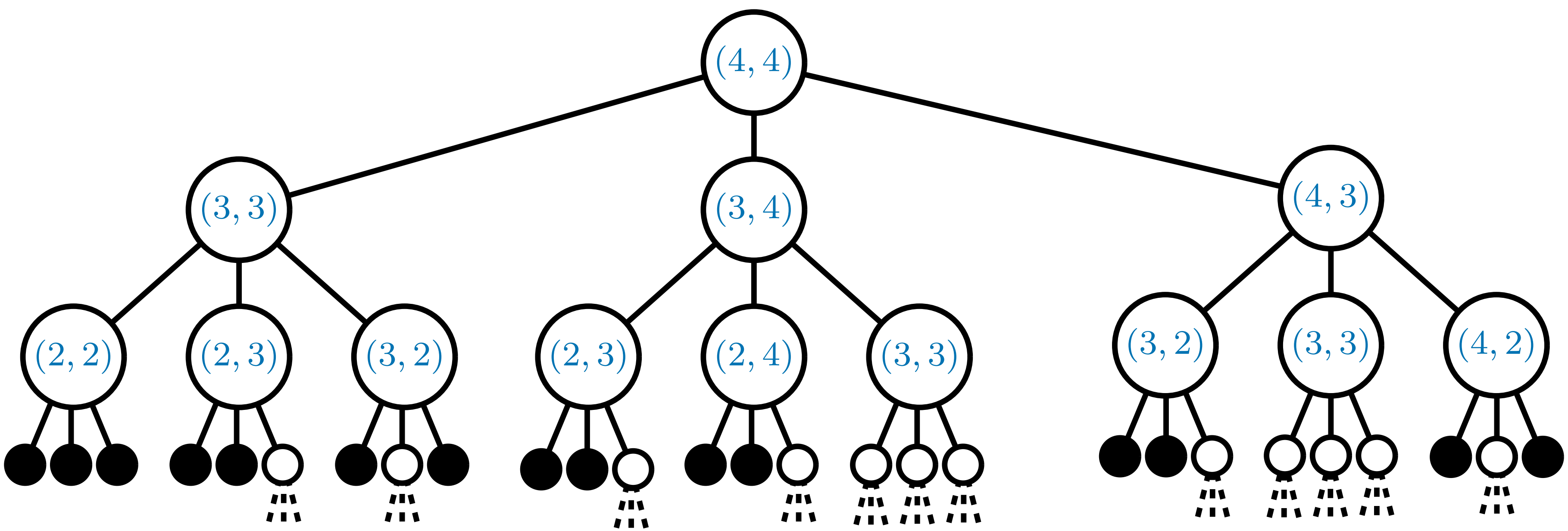


How efficient is the recursive algorithm?

LES (x, y)

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x-1,y-1), LES(x-1,y), LES(x,y-1)) + 1
```

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

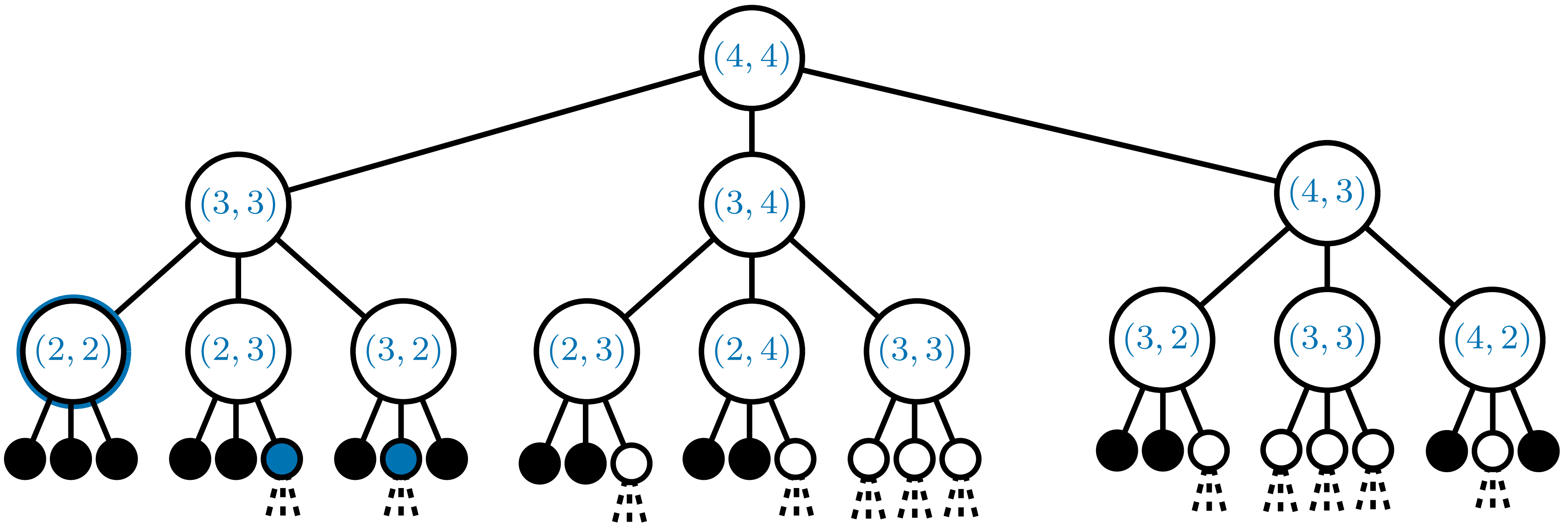


How efficient is the recursive algorithm?

LES (x, y)

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min (LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

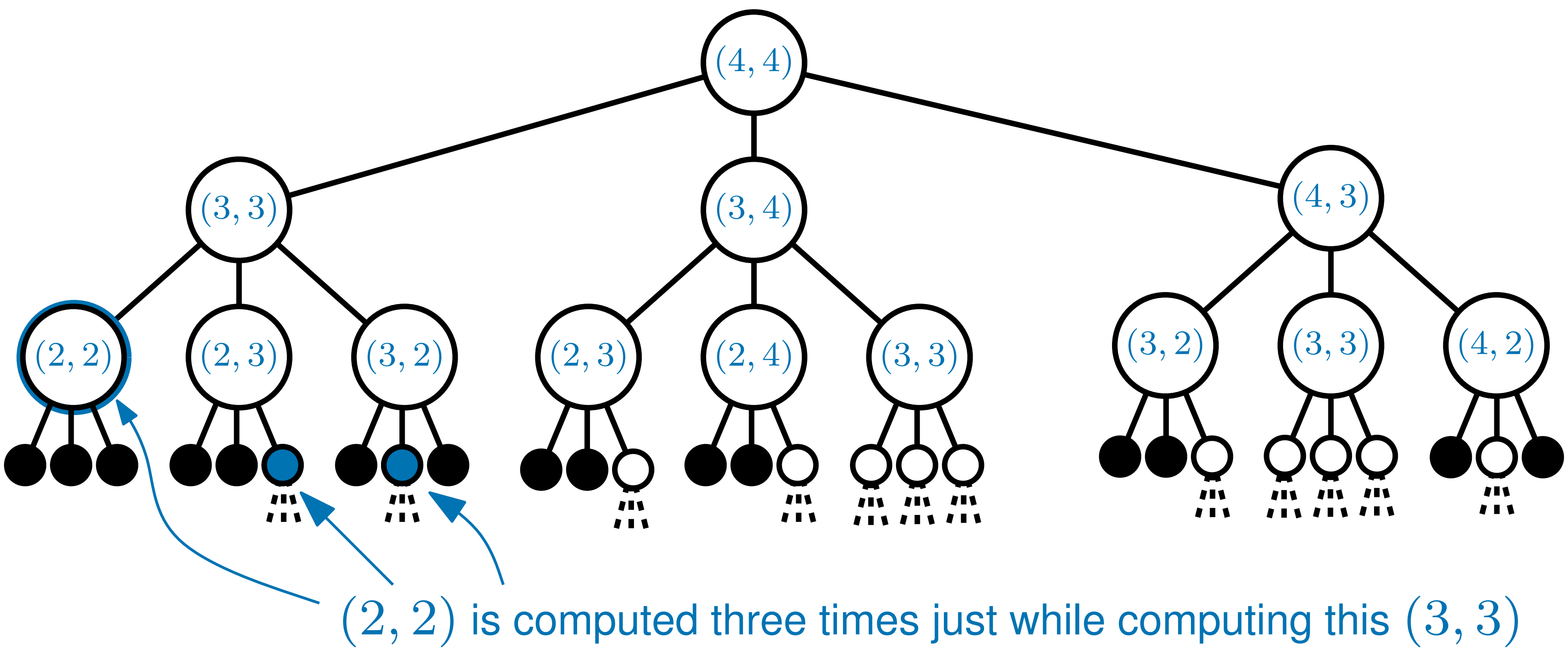


How efficient is the recursive algorithm?

LES(x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\left(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\right) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

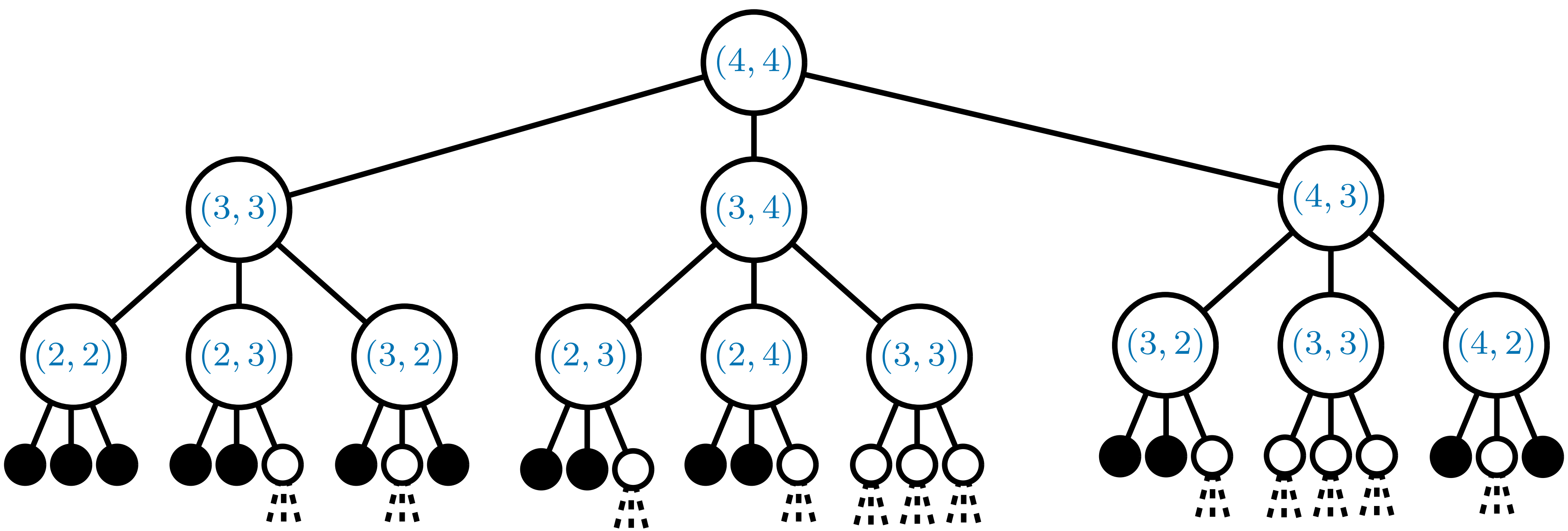


How efficient is the recursive algorithm?

LES (x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*

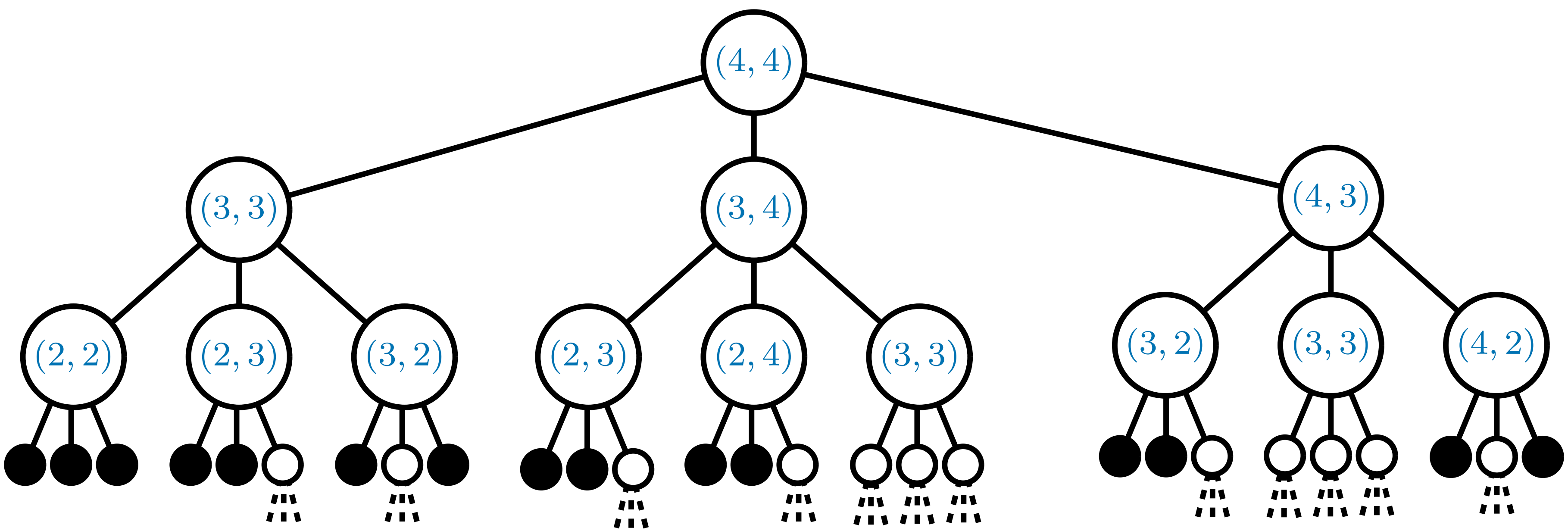


How efficient is the recursive algorithm?

LES(x, y)

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x-1, y-1), LES(x-1, y), LES(x, y-1)) + 1
```

Let's compute LES$(4,4)$... *(and consider the recursive calls)*



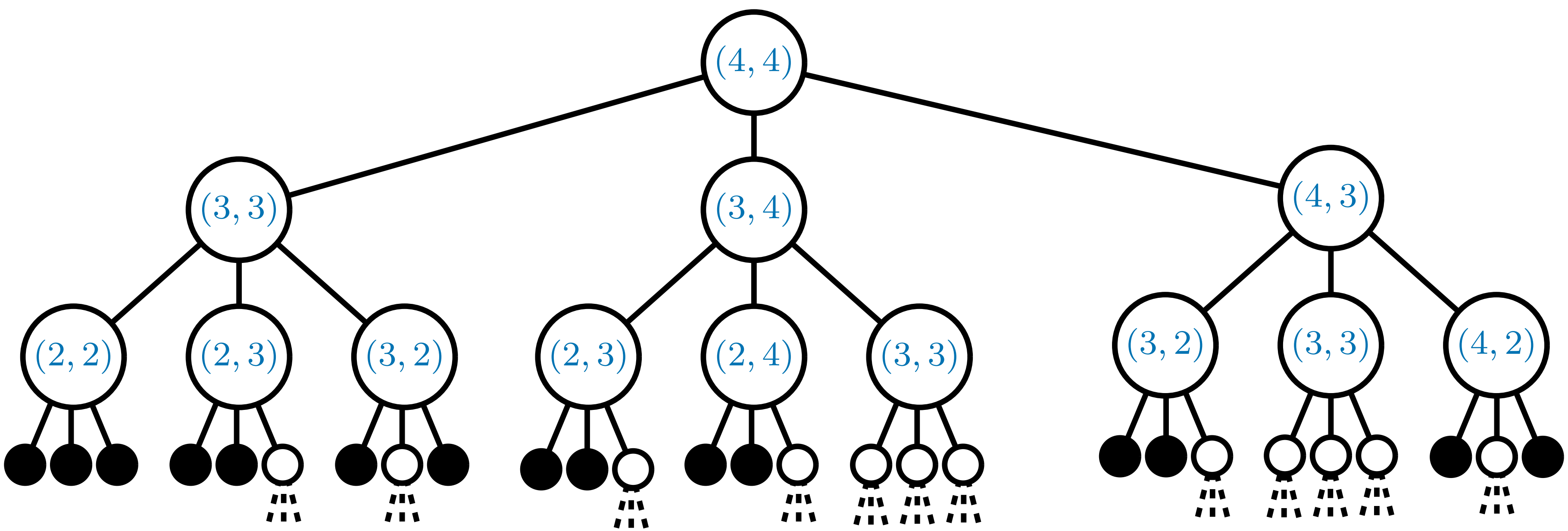
This doesn't look good!

How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x - 1, y - 1), LES(x - 1, y), LES(x, y - 1)) + 1
```

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*



This doesn't look good!

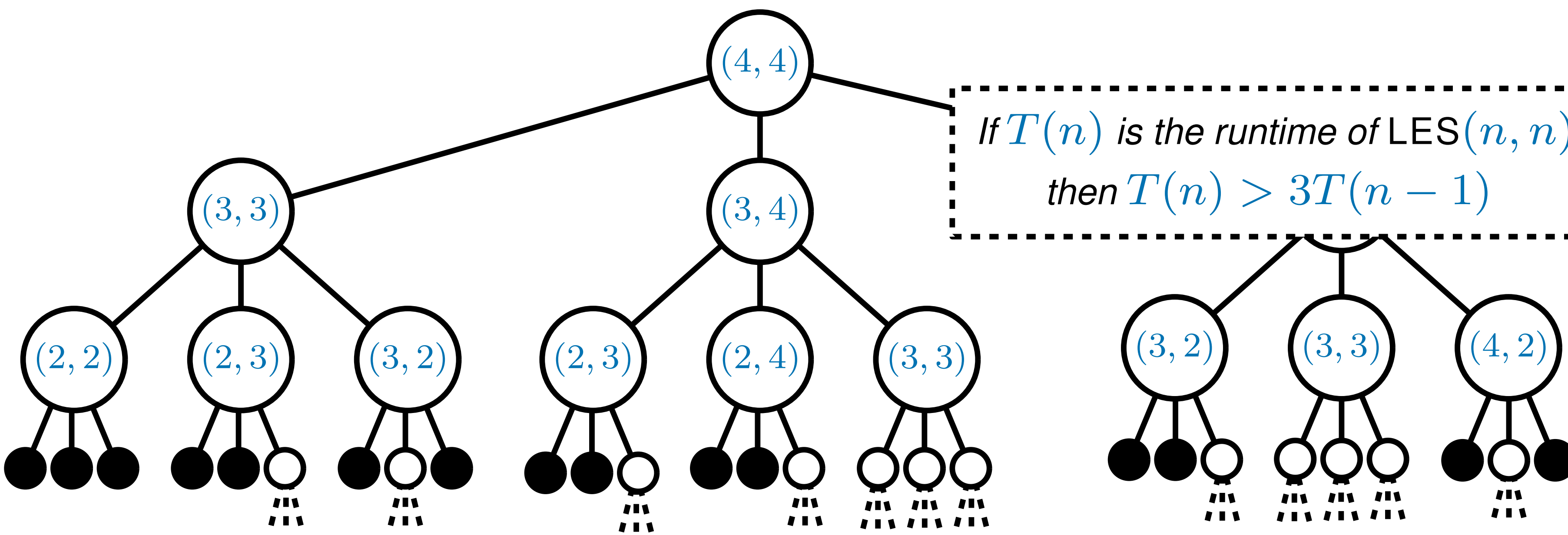
In fact the running time of LES(n, n) is *exponential* in n

How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4,4)$... *(and consider the recursive calls)*



If $T(n)$ *is the runtime of* LES(n,n) *then* $T(n) > 3T(n-1)$

This doesn't look good!

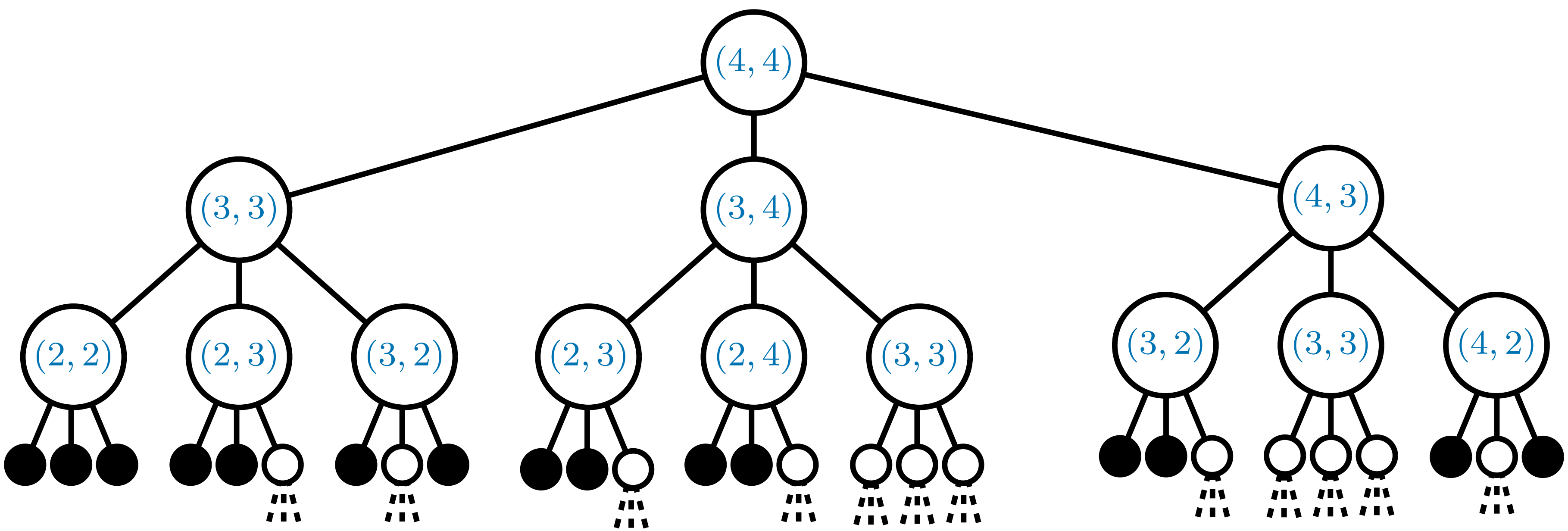
In fact the running time of LES(n,n) is *exponential* in n

How efficient is the recursive algorithm?

```
LES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
Return min(LES(x − 1, y − 1), LES(x − 1, y), LES(x, y − 1)) + 1
```

Let's compute LES$(4,4)$... *(and consider the recursive calls)*

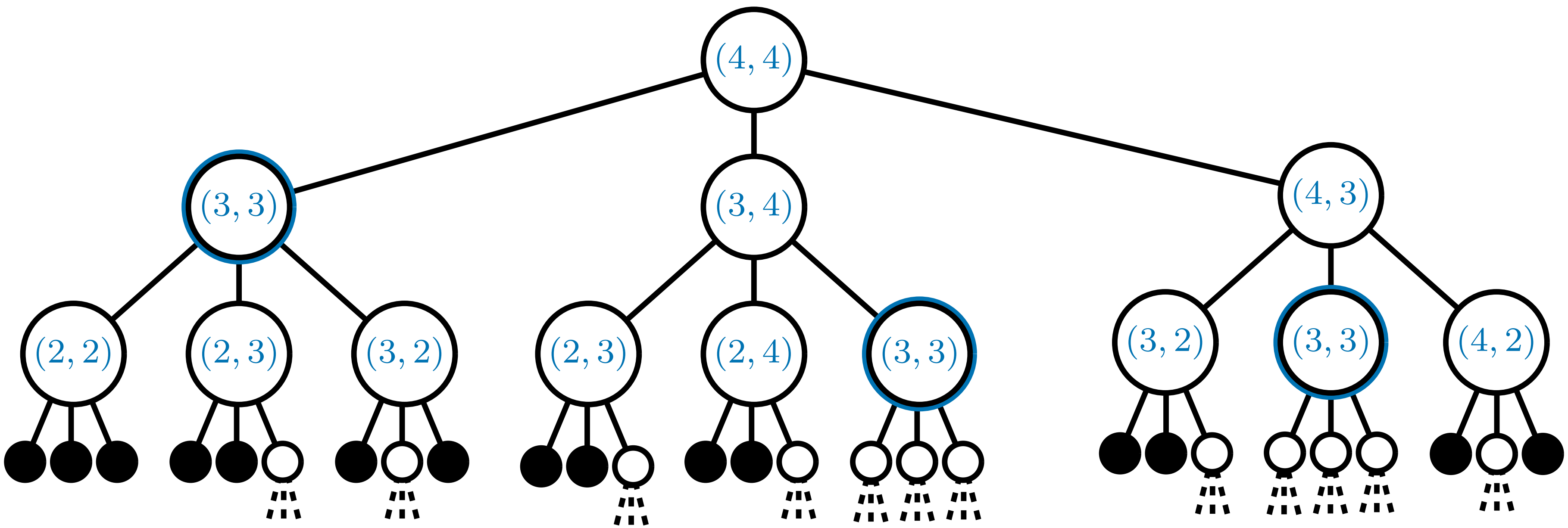


How efficient is the recursive algorithm?

LES(x, y)

If pixel (x, y) is not empty
  Return 0
If $(x = 1)$ or $(y = 1)$
  Return 1
Return $\min\big(\text{LES}(x-1, y-1), \text{LES}(x-1, y), \text{LES}(x, y-1)\big) + 1$

Let's compute LES$(4, 4)$... *(and consider the recursive calls)*



What should we do about all this repeated computation?

3. Store the solutions to subproblems

$\textsc{MemLES}(x, y)$

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
If LES[x, y] undefined
  LES[x, y] = min (MemLES(x − 1, y − 1), MemLES(x − 1, y), MemLES(x, y − 1)) + 1
Return LES[x, y]
```

In the $\textsc{MemLES}$ version of the algorithm
we store solutions to previously computed subproblems
in an $(n \times n)$ 2D array called LES

3. Store the solutions to subproblems

MEMLES(x, y)

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
If LES[x,y] undefined
  LES[x,y] = min(MEMLES(x−1, y−1), MEMLES(x−1, y), MEMLES(x, y−1)) + 1
Return LES[x,y]
```

In the MEMLES version of the algorithm
we store solutions to previously computed subproblems
in an $(n \times n)$ 2D array called LES

This is called memoization *(not memorization)*

3. Store the solutions to subproblems

$\textsc{MemLES}(x, y)$

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
If LES[x,y] undefined
  LES[x,y] = min(MemLES(x-1,y-1), MemLES(x-1,y), MemLES(x,y-1)) + 1
Return LES[x,y]
```

In the $\textsc{MemLES}$ version of the algorithm
we store solutions to previously computed subproblems
in an $(n \times n)$ 2D array called LES

This is called memoization *(not memorization)*

Crucially, now each entry $\text{LES}[x, y]$ is only computed *once*

3. Store the solutions to subproblems

```
MEMLES(x, y)
```

```
If pixel (x, y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
If LES[x, y] undefined
  LES[x, y] = min (MEMLES(x − 1, y − 1), MEMLES(x − 1, y), MEMLES(x, y − 1)) + 1
Return LES[x, y]
```

In the MEMLES version of the algorithm

we store solutions to previously computed subproblems

in an $(n \times n)$ 2D array called LES

This is called memoization *(not memorization)*

Crucially, now each entry $\text{LES}[x, y]$ is only computed *once*

The time complexity of computing $\text{MEMLES}(n, n)$ is now $O(n^2)$

3. Store the solutions to subproblems

```
MEMLES(x, y)
```

```
If pixel (x,y) is not empty
  Return 0
If (x = 1) or (y = 1)
  Return 1
If LES[x,y] undefined
  LES[x,y] = min (MEMLES(x − 1, y − 1), MEMLES(x − 1, y), MEMLES(x, y − 1)) + 1
Return LES[x,y]
```

In the MEMLES version of the algorithm
we store solutions to previously computed subproblems
in an $(n \times n)$ 2D array called LES

This is called memoization *(not memorization)*

Crucially, now each entry $\mathrm{LES}[x, y]$ is only computed *once*

The time complexity of computing $\mathrm{MEMLES}(n, n)$ is now $O(n^2)$

(in fact, computing $\max_{x,y} \mathrm{MEMLES}(x, y)$ takes $O(n^2)$ time too)

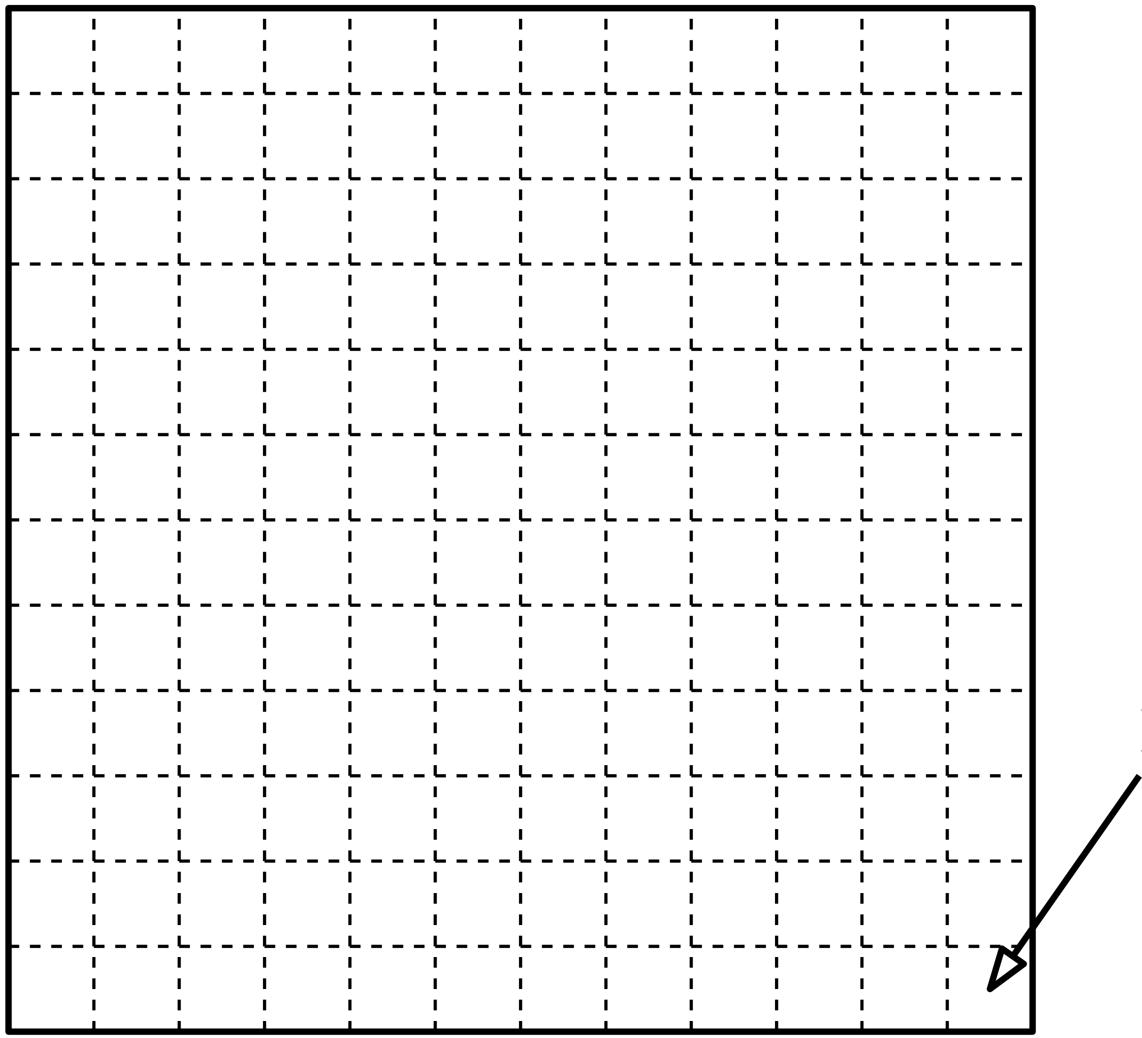
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathrm{LES}:$



$\mathrm{LES}[n,n]$

What information do we need to compute $\mathrm{LES}[n,n]$?

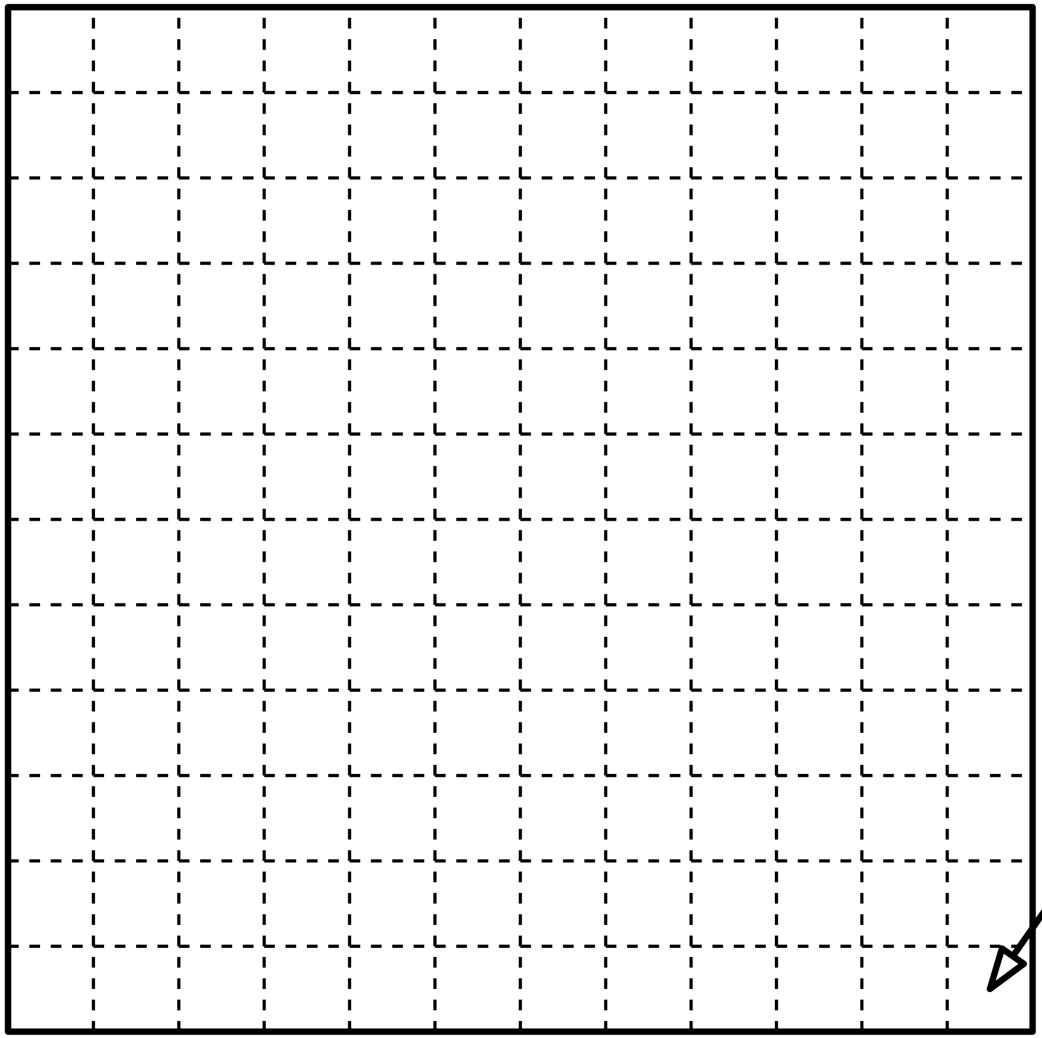
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



to compute

$\mathrm{LES}[n,n]$

What information do we need to compute $\mathrm{LES}[n,n]$?

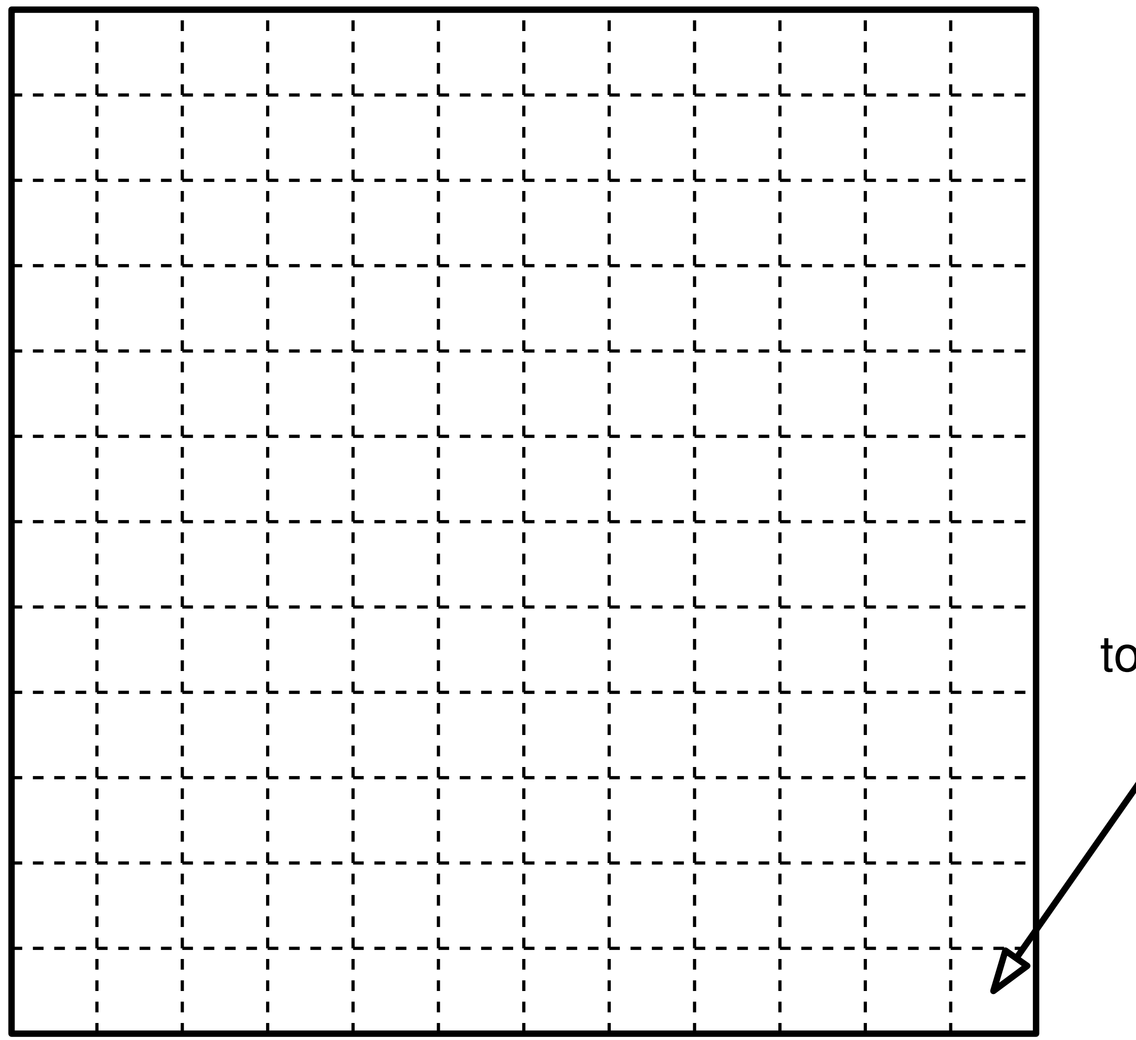
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



to compute

$\mathrm{LES}[n,n]$

we need

What information do we need to compute $\mathrm{LES}[n,n]$?

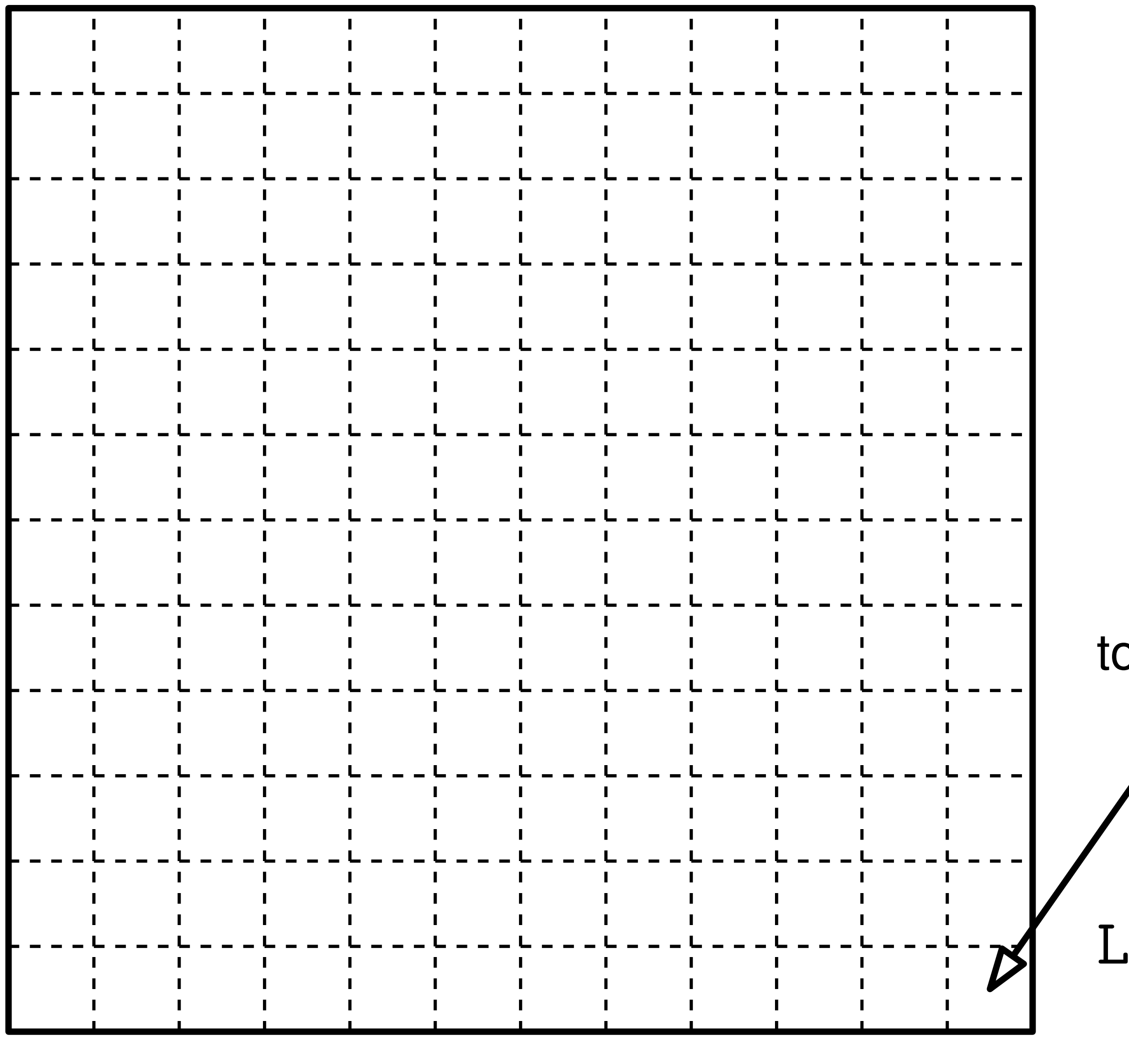
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MemLES}(x-1,y-1), \text{MemLES}(x-1,y), \text{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\text{LES}:$



to compute

$\text{LES}[n,n]$

we need

$\text{LES}[n-1,n-1]$

$\text{LES}[n-1,n]$

and $\text{LES}[n,n-1]$

What information do we need to compute $\text{LES}[n,n]$?

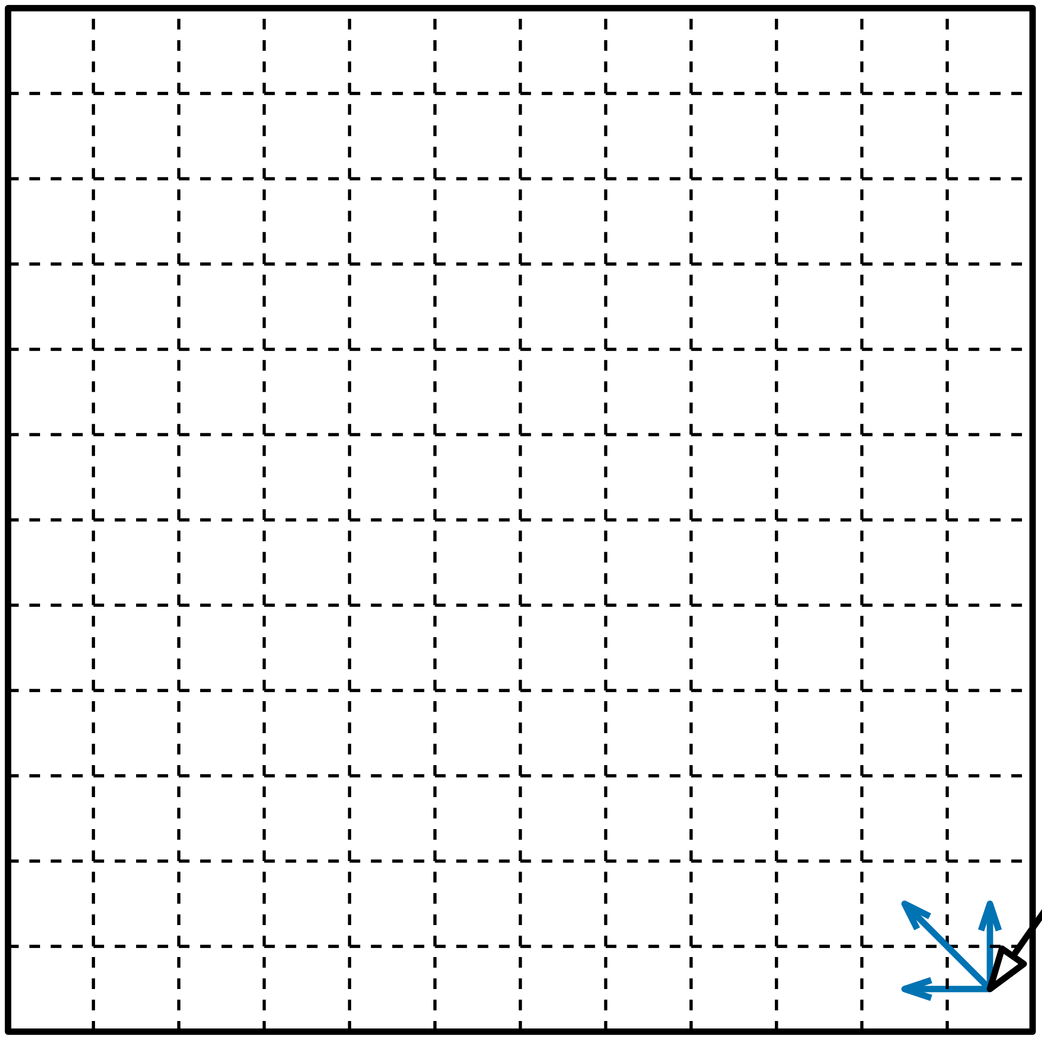
The dependency graph

$$\mathtt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathtt{LES}$:



to compute

$\mathtt{LES}[n,n]$

we need

$\mathtt{LES}[n-1,n-1]$

$\mathtt{LES}[n-1,n]$

and $\mathtt{LES}[n,n-1]$

What information do we need to compute $\mathtt{LES}[n,n]$?

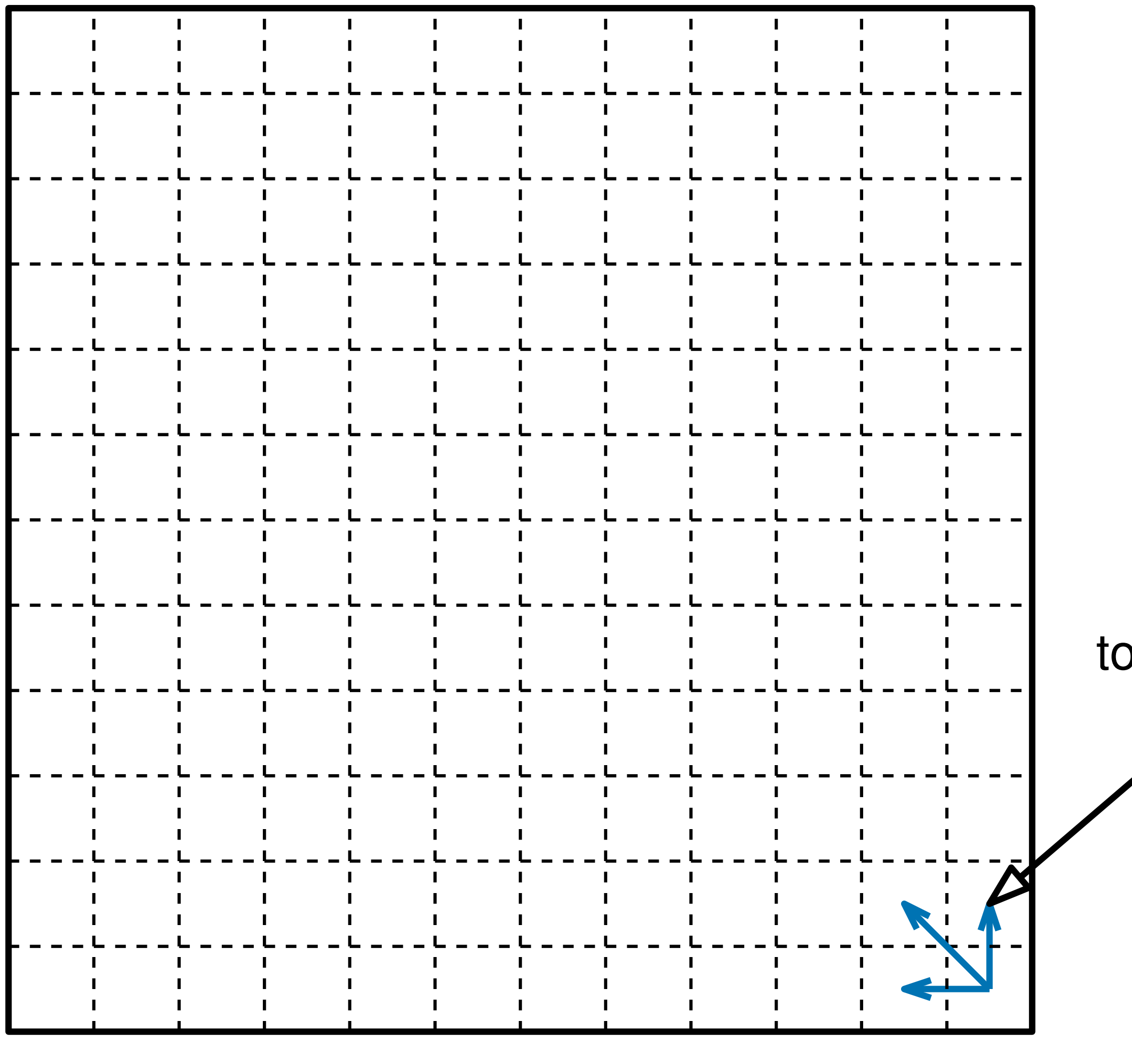
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathrm{LES}:$



to compute

$\mathrm{LES}\,[n, n-1]$

What information do we need to compute $\mathrm{LES}\,[n, n]$?

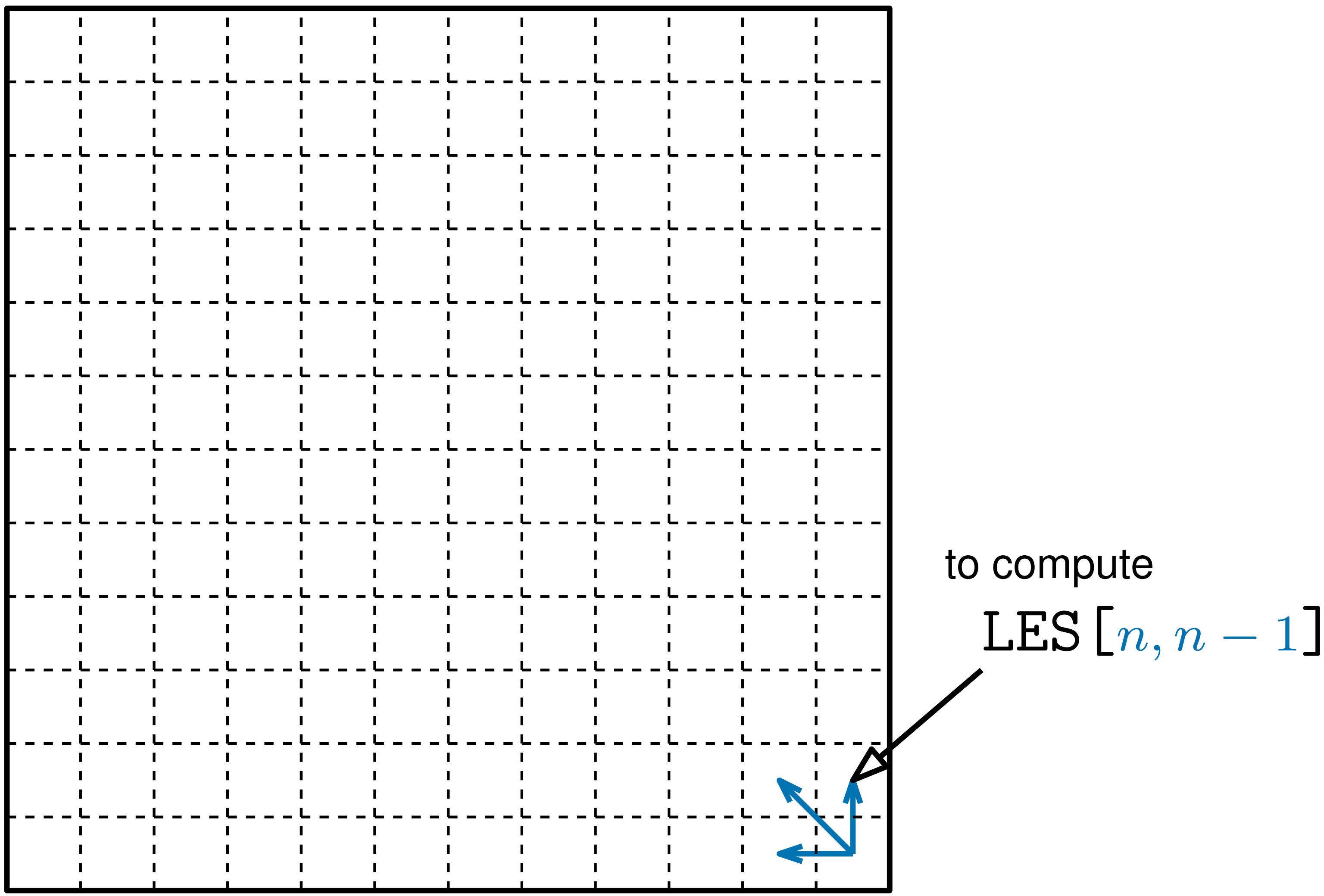
The dependency graph

$$\mathtt{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathtt{LES}$:



What information do we need to compute $\mathtt{LES}\,[n,n]$?

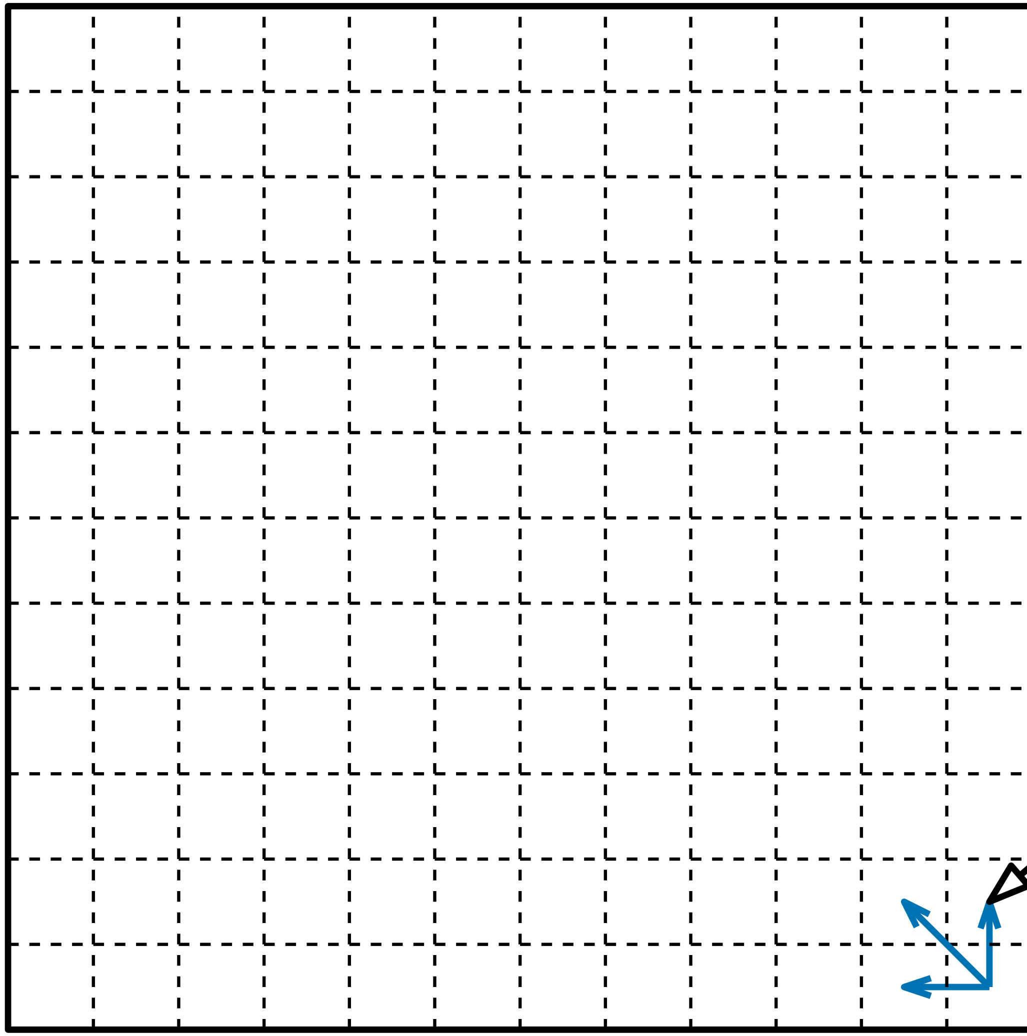
The dependency graph

$$\mathtt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathtt{LES}$:



to compute

$\mathtt{LES}[n, n-1]$

$\mathtt{LES}[n-1, n-2]$

$\mathtt{LES}[n-1, n-1]$

and $\mathtt{LES}[n, n-2]$

What information do we need to compute $\mathtt{LES}[n, n]$?

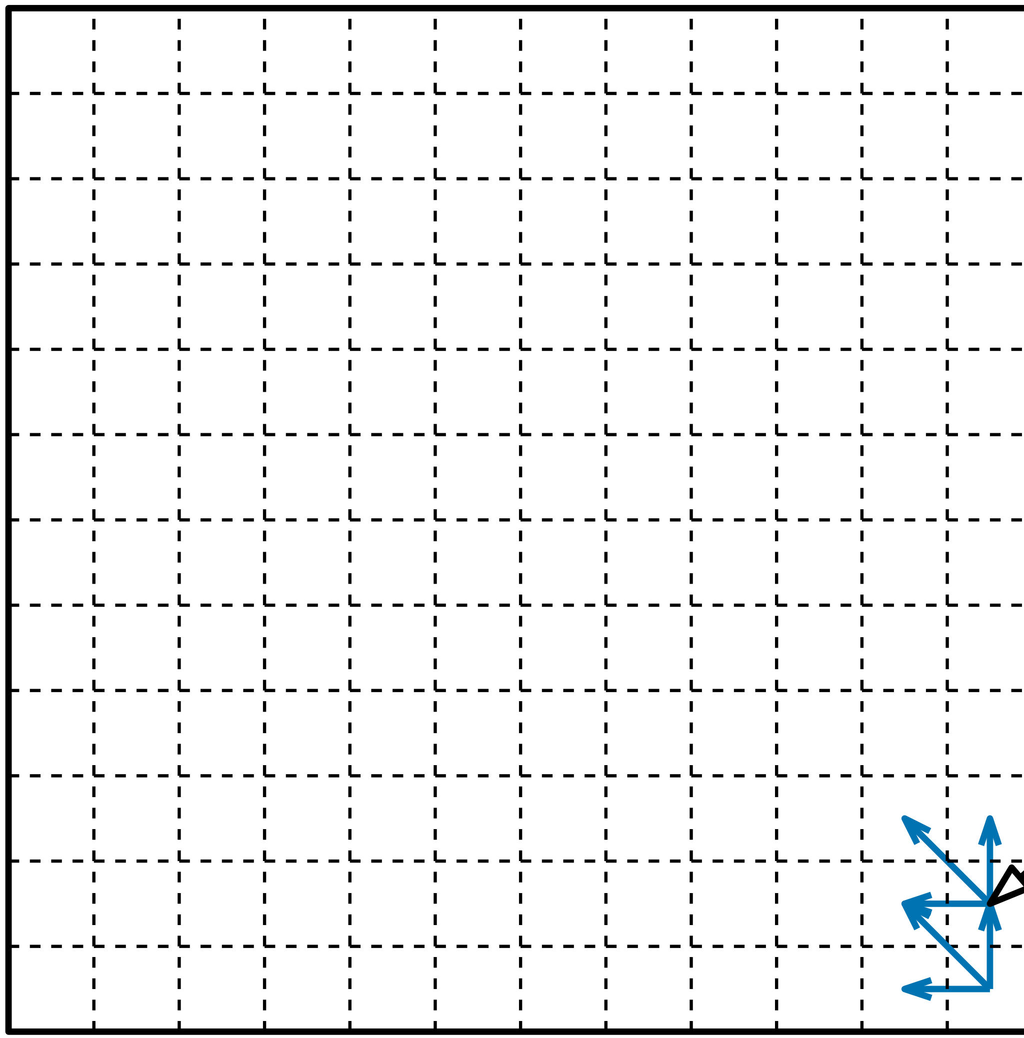
The dependency graph

$$\mathtt{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\mathtt{LES}$:



to compute

$\mathtt{LES}\,[n, n-1]$

$\mathtt{LES}\,[n-1, n-2]$

$\mathtt{LES}\,[n-1, n-1]$

and $\mathtt{LES}\,[n, n-2]$

What information do we need to compute $\mathtt{LES}\,[n,n]$?

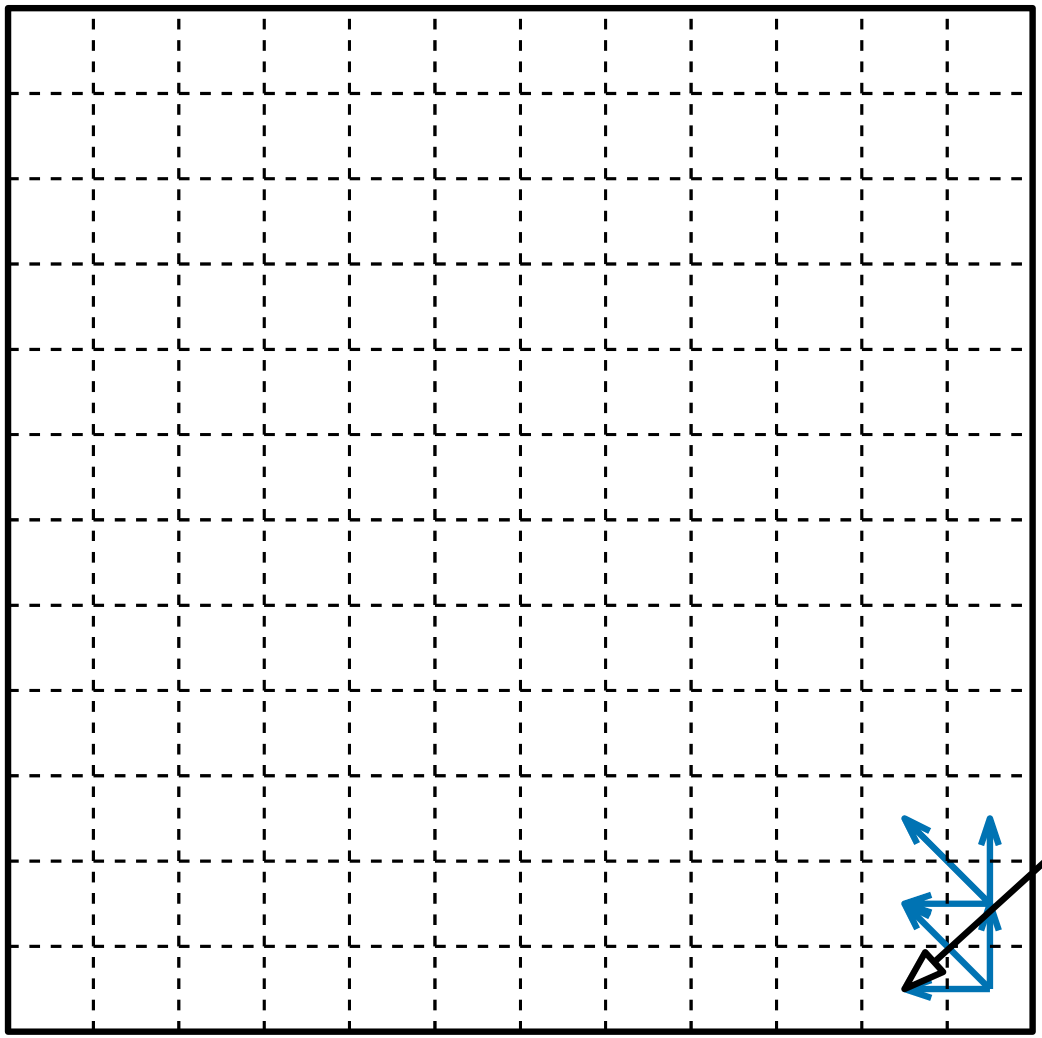
The dependency graph

$$\texttt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\texttt{LES}$:



to compute

$\texttt{LES}[n-1,n]$

What information do we need to compute $\texttt{LES}[n,n]$?

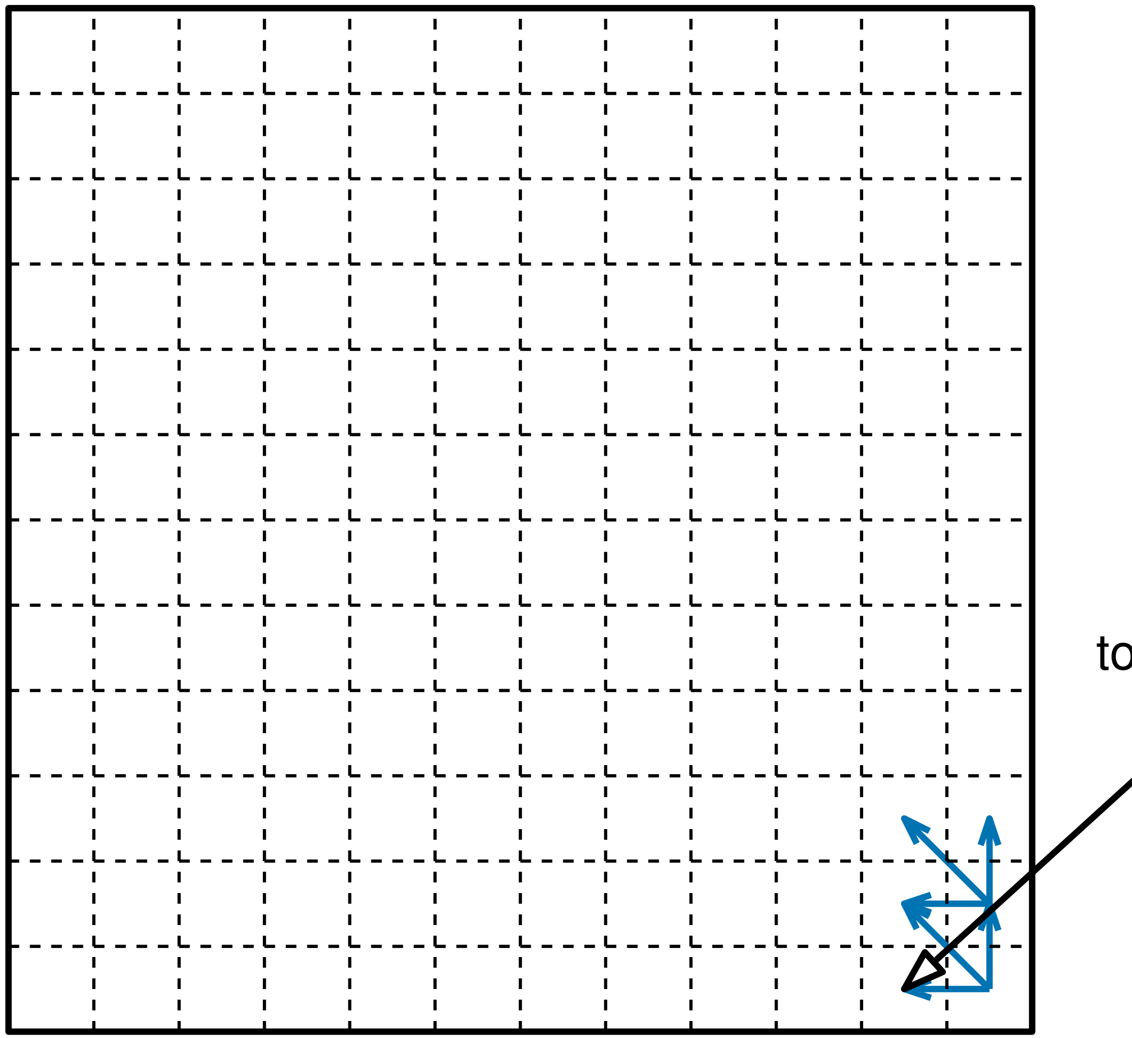
The dependency graph

$$\texttt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\texttt{LES}$:



to compute

$\texttt{LES}[n-1,n]$

we need

What information do we need to compute $\texttt{LES}[n,n]$?

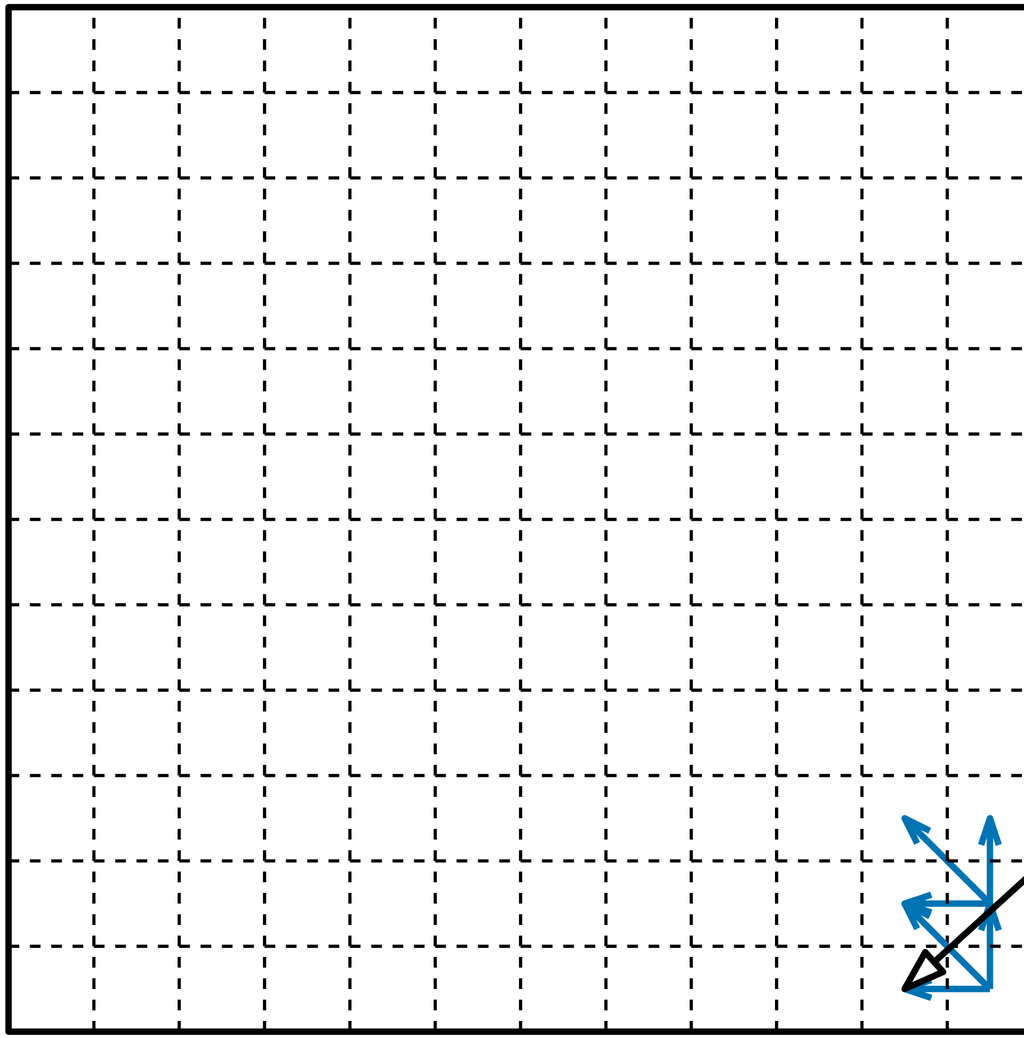
The dependency graph

$$\texttt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\texttt{LES}$:



to compute

$\texttt{LES}[n-1, n]$

we need

$\texttt{LES}[n-2, n-1]$

$\texttt{LES}[n-2, n]$

and $\texttt{LES}[n-1, n-1]$

What information do we need to compute $\texttt{LES}[n,n]$?

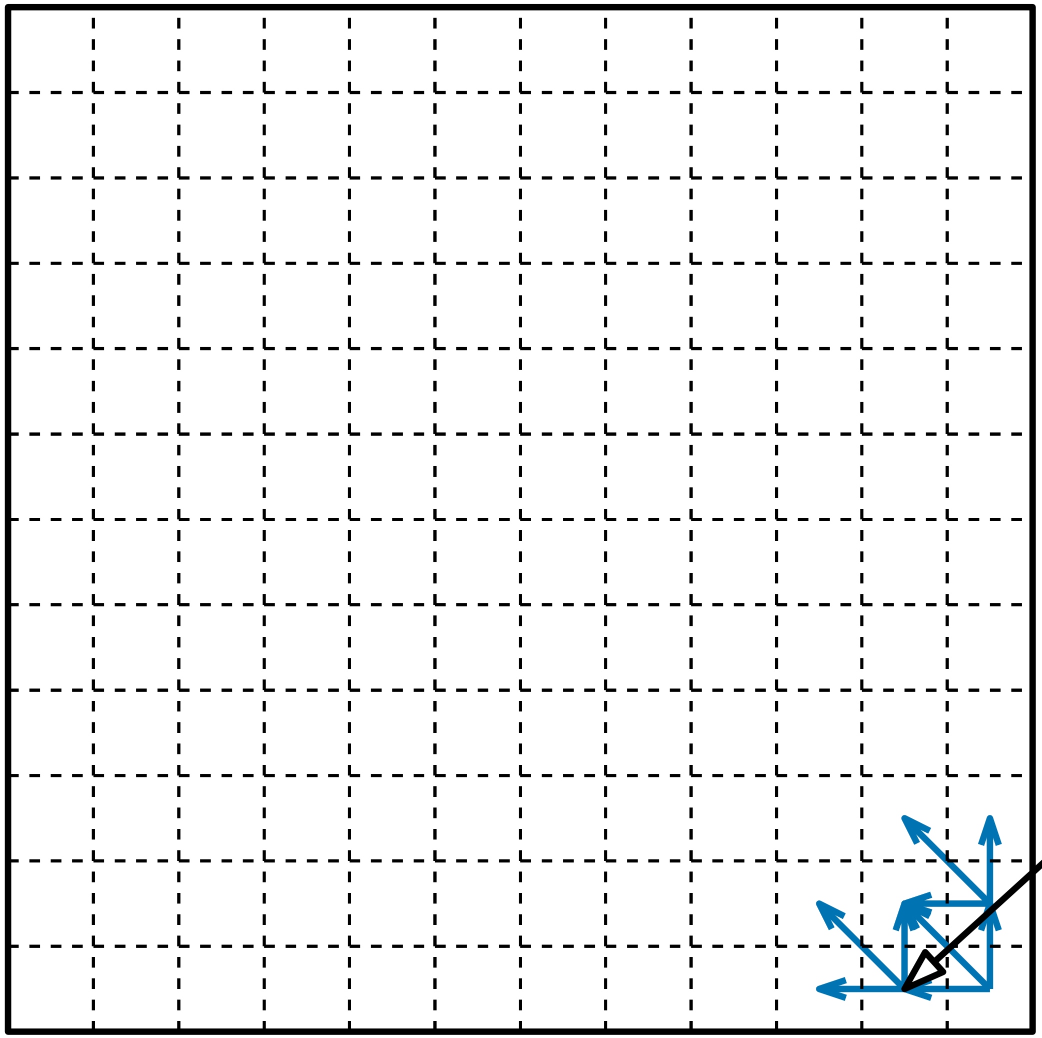
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



to compute

$\mathrm{LES}\,[n-1,n]$

we need

$\mathrm{LES}\,[n-2,n-1]$

$\mathrm{LES}\,[n-2,n]$

and $\mathrm{LES}\,[n-1,n-1]$

What information do we need to compute $\mathrm{LES}\,[n,n]$?

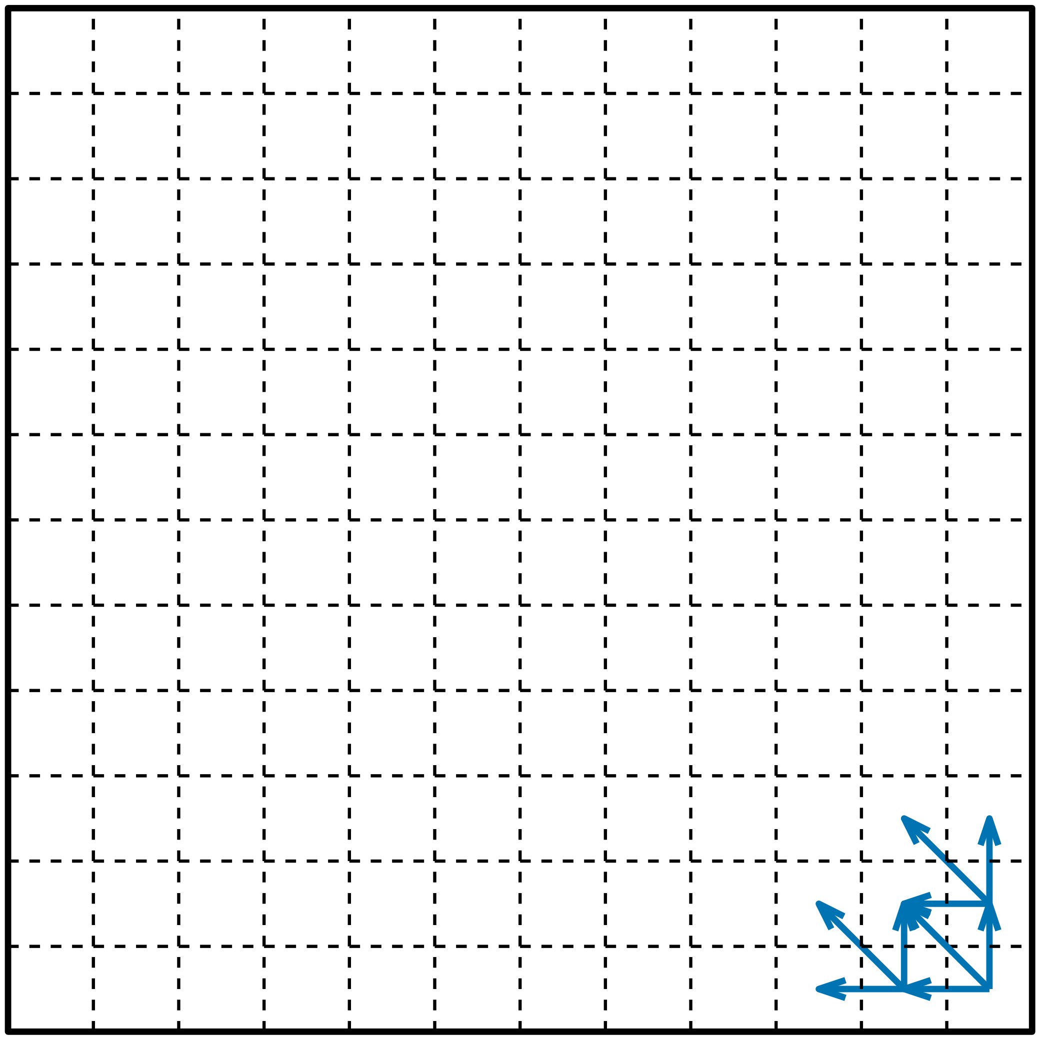
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MEMLES}(x-1,y-1), \text{MEMLES}(x-1,y), \text{MEMLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\text{LES}[n,n]$?

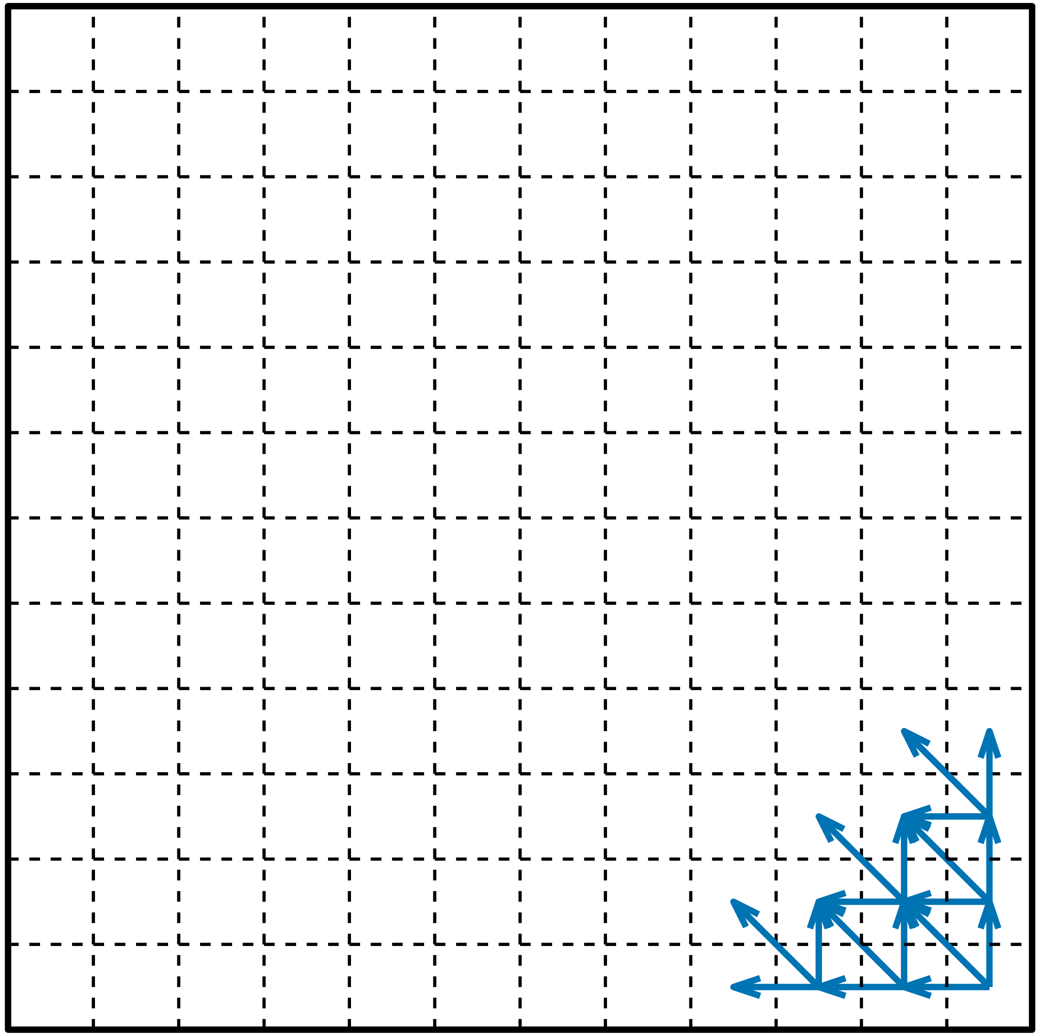
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MemLES}(x-1,y-1), \text{MemLES}(x-1,y), \text{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\text{LES}:$



What information do we need to compute $\text{LES}[n,n]$?

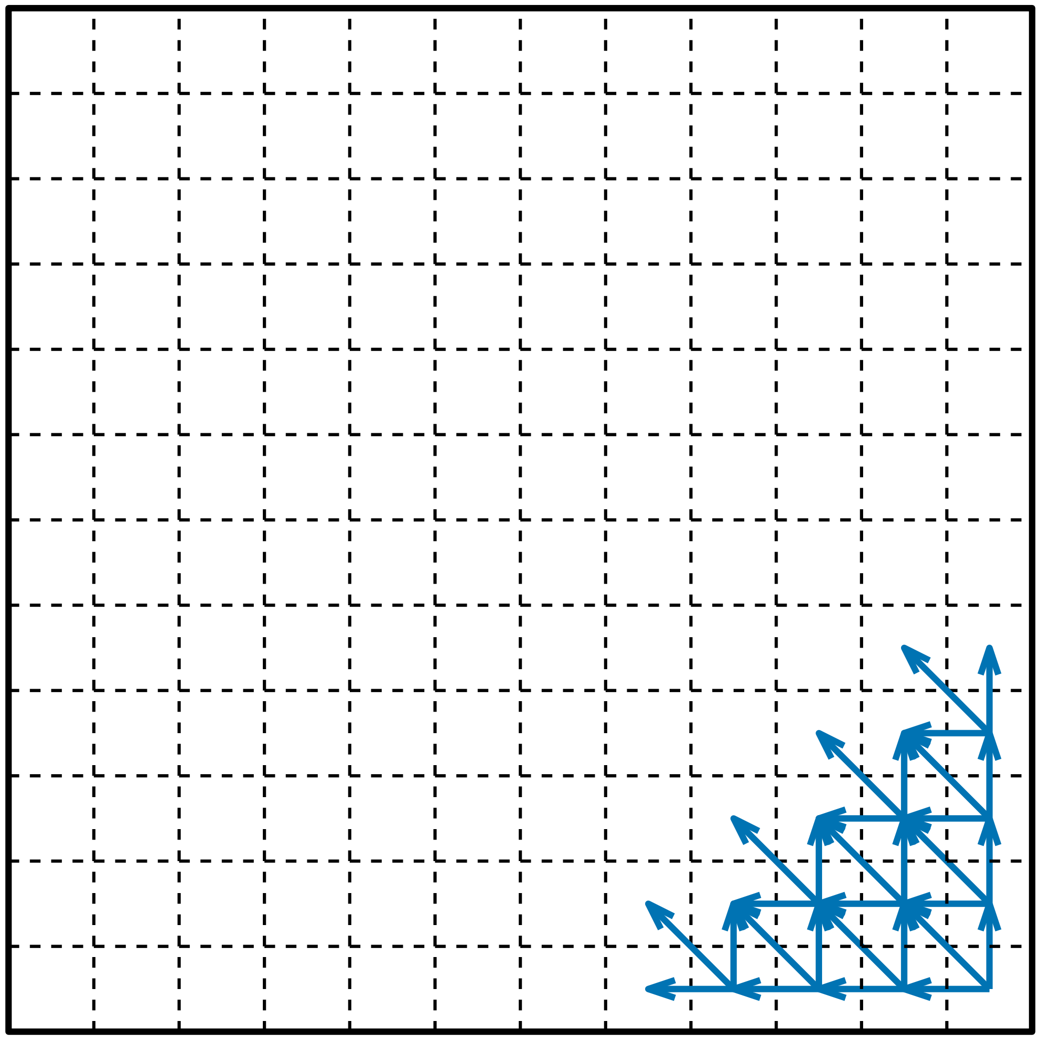
The dependency graph

$$\mathtt{LES}[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathtt{LES}$:



What information do we need to compute $\mathtt{LES}[n, n]$?

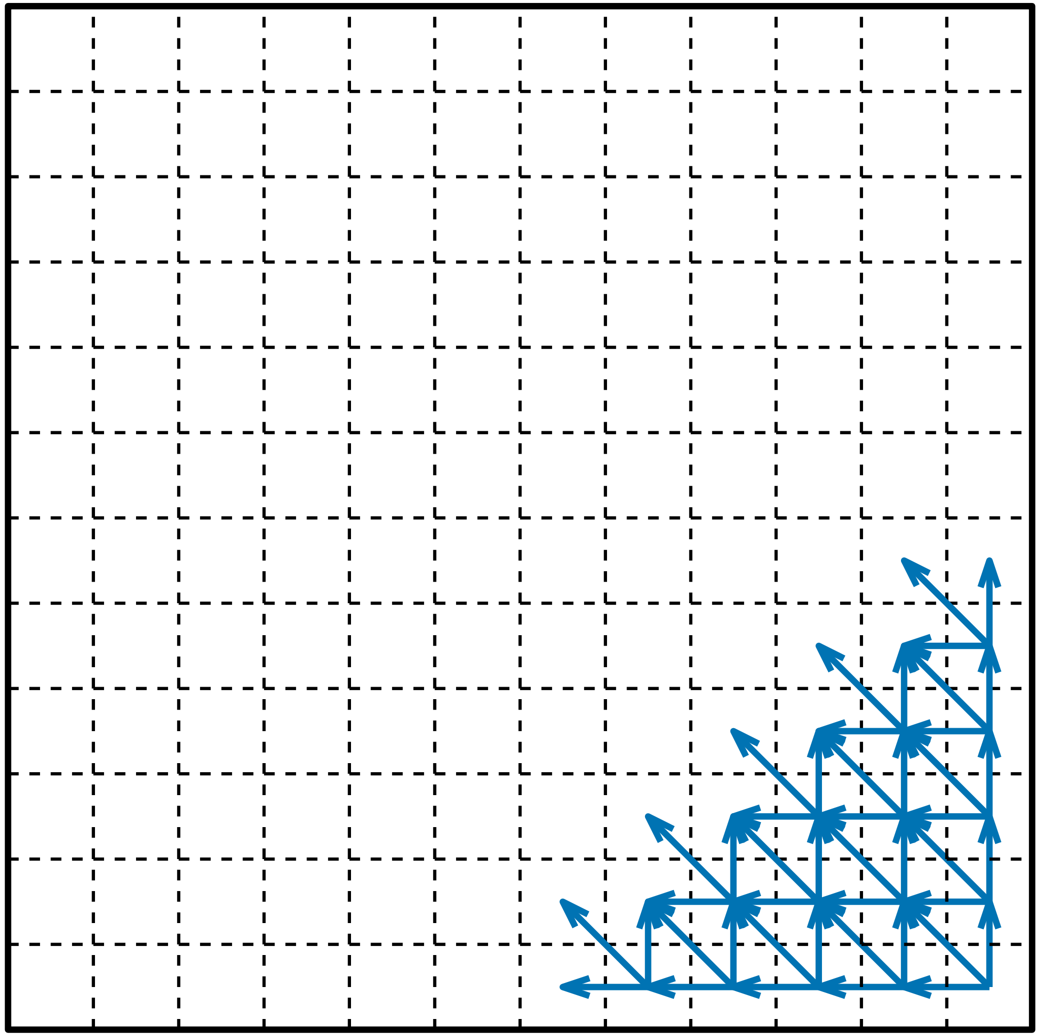
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}\,[n,n]$?

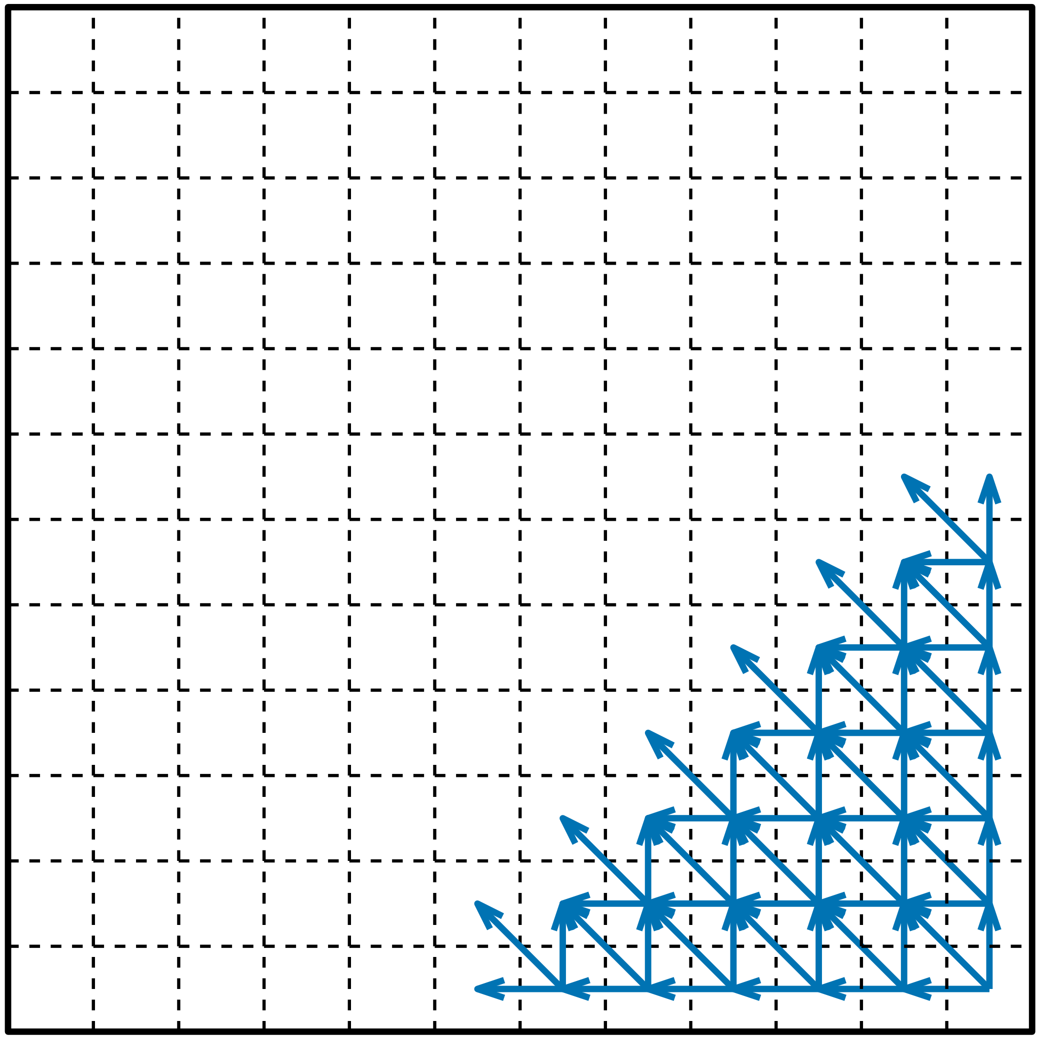
The dependency graph

$$\mathrm{LES}\,[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathrm{LES}:$



What information do we need to compute $\mathrm{LES}\,[n, n]$?

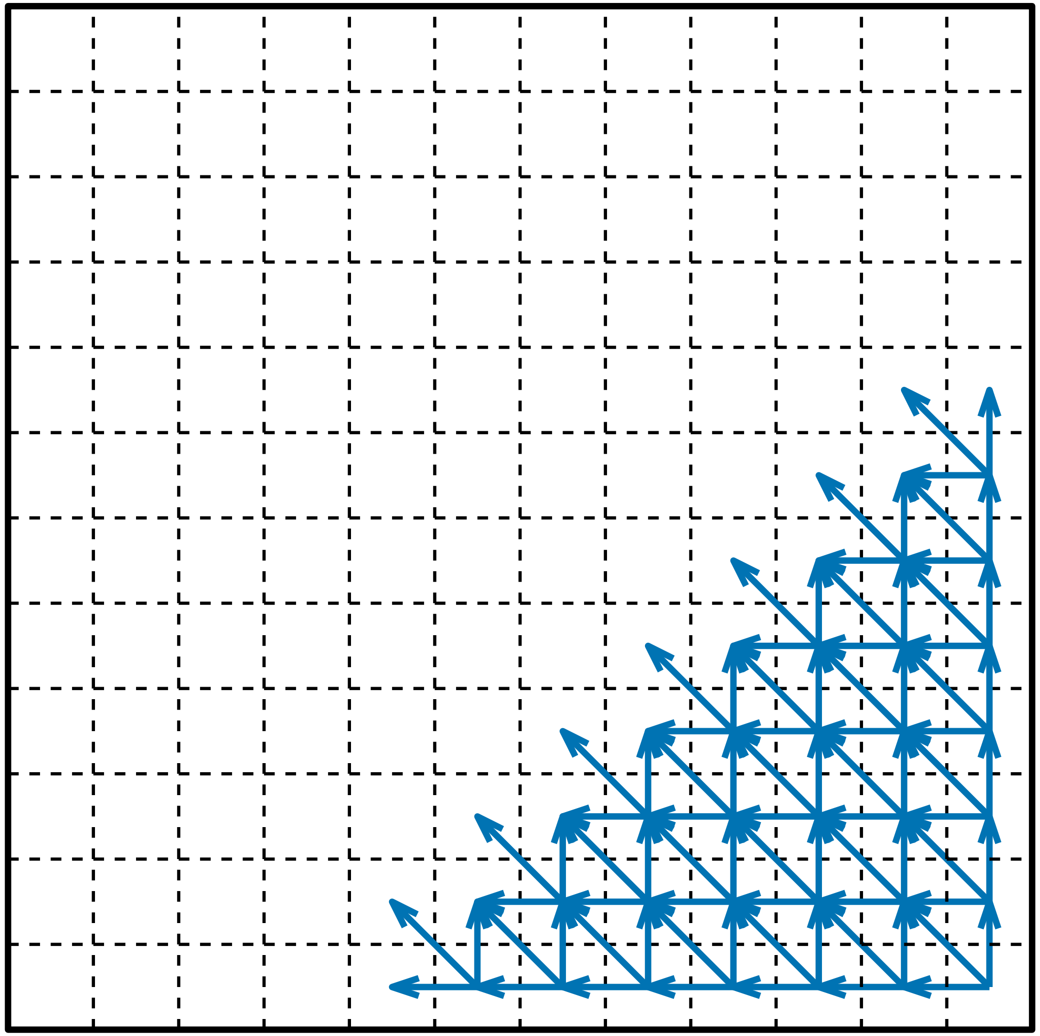
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MEMLES}(x-1,y-1), \text{MEMLES}(x-1,y), \text{MEMLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\text{LES}:$



What information do we need to compute $\text{LES}[n,n]$?

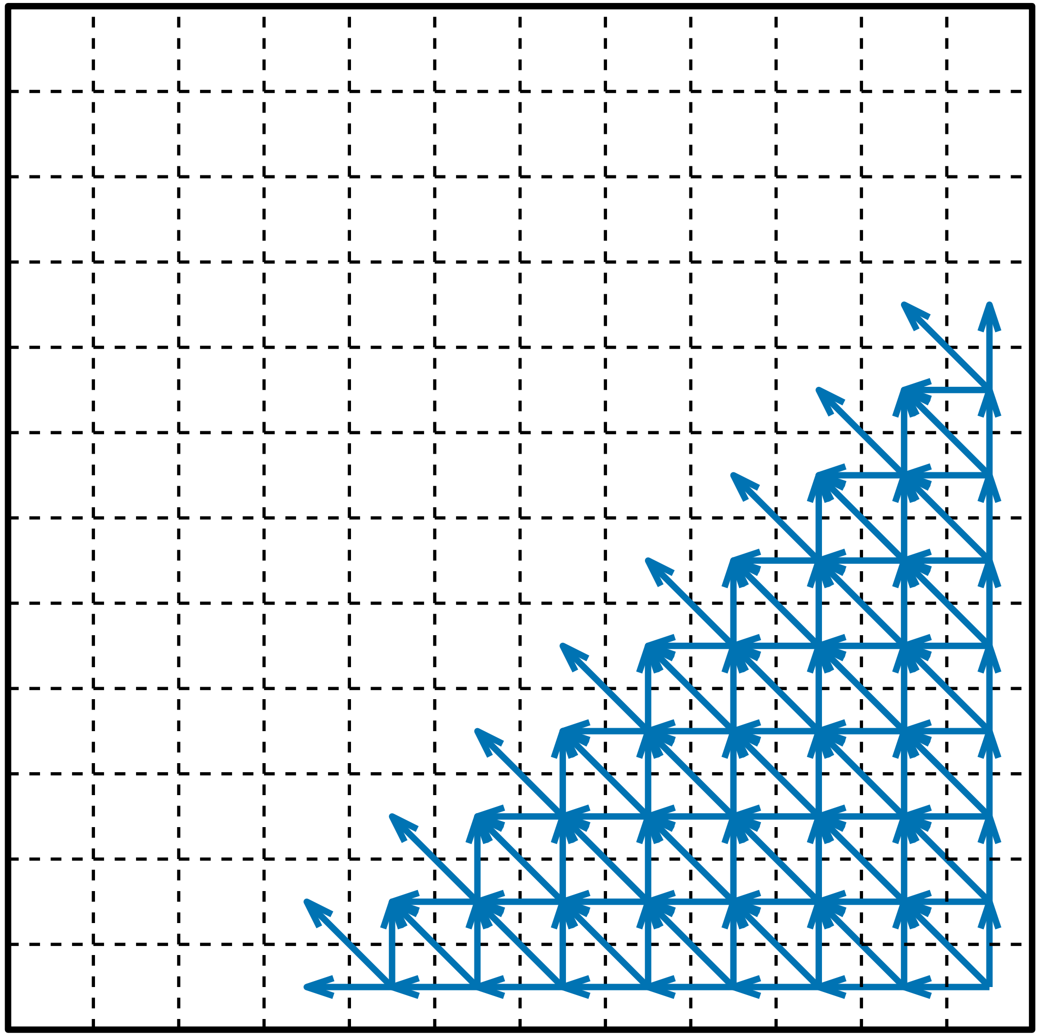
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}\,[n,n]$?

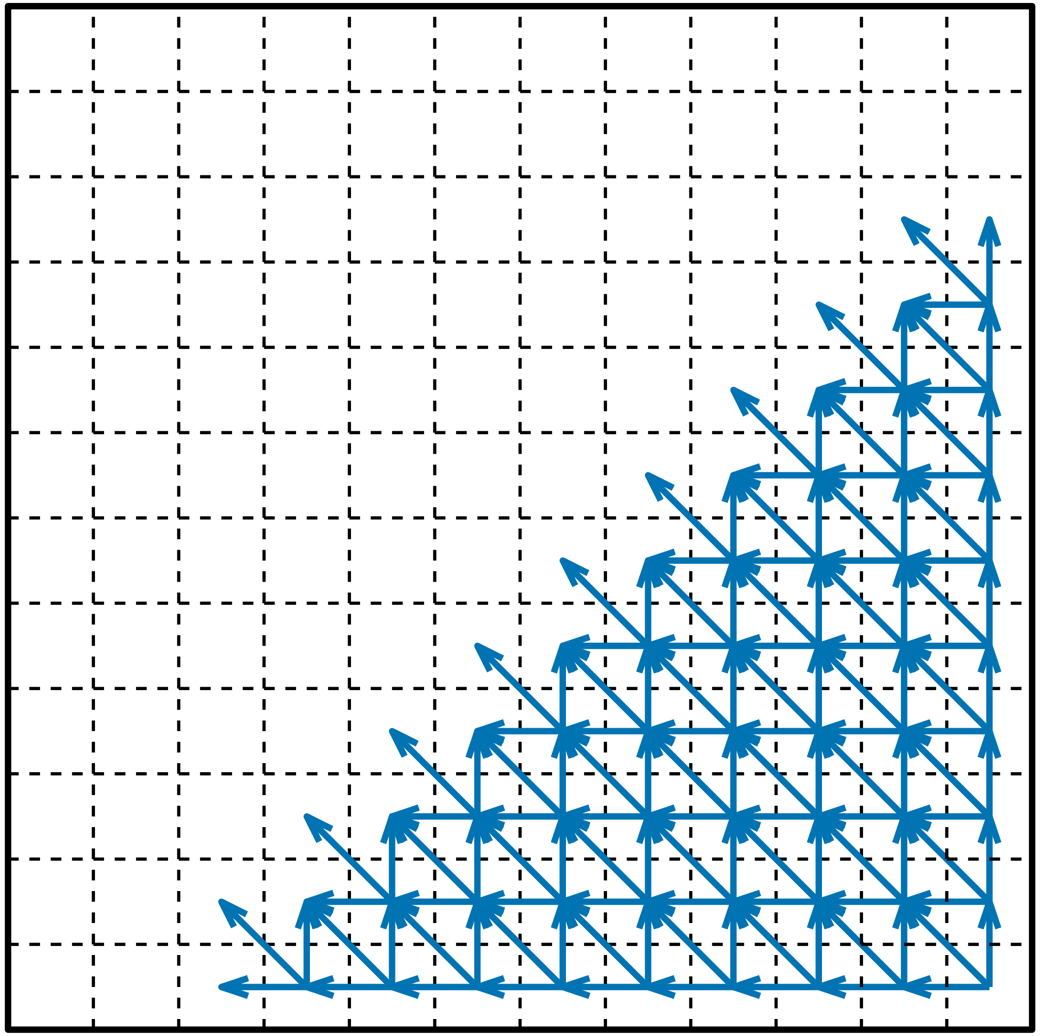
The dependency graph

$$\mathrm{LES}\,[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}\,[n, n]$?

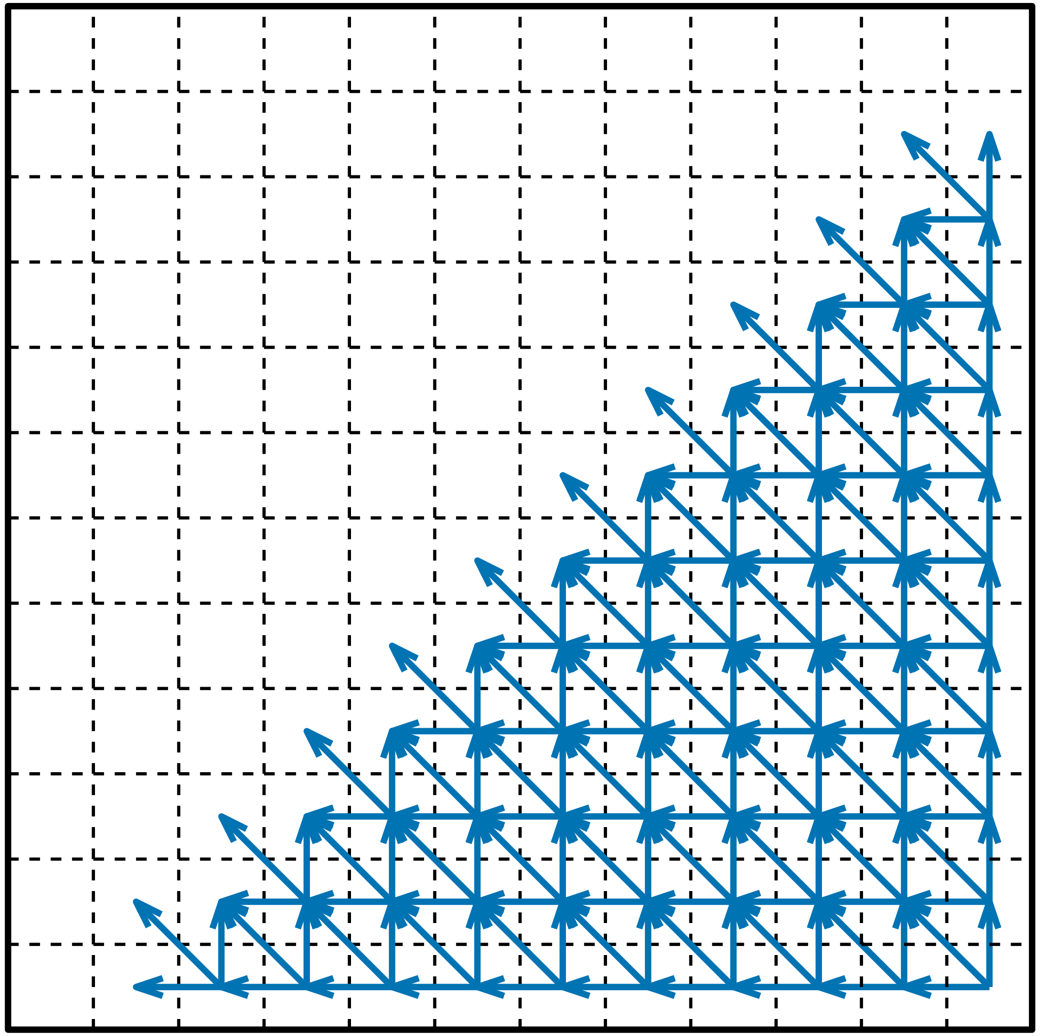
The dependency graph

$$\mathrm{LES}\,[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



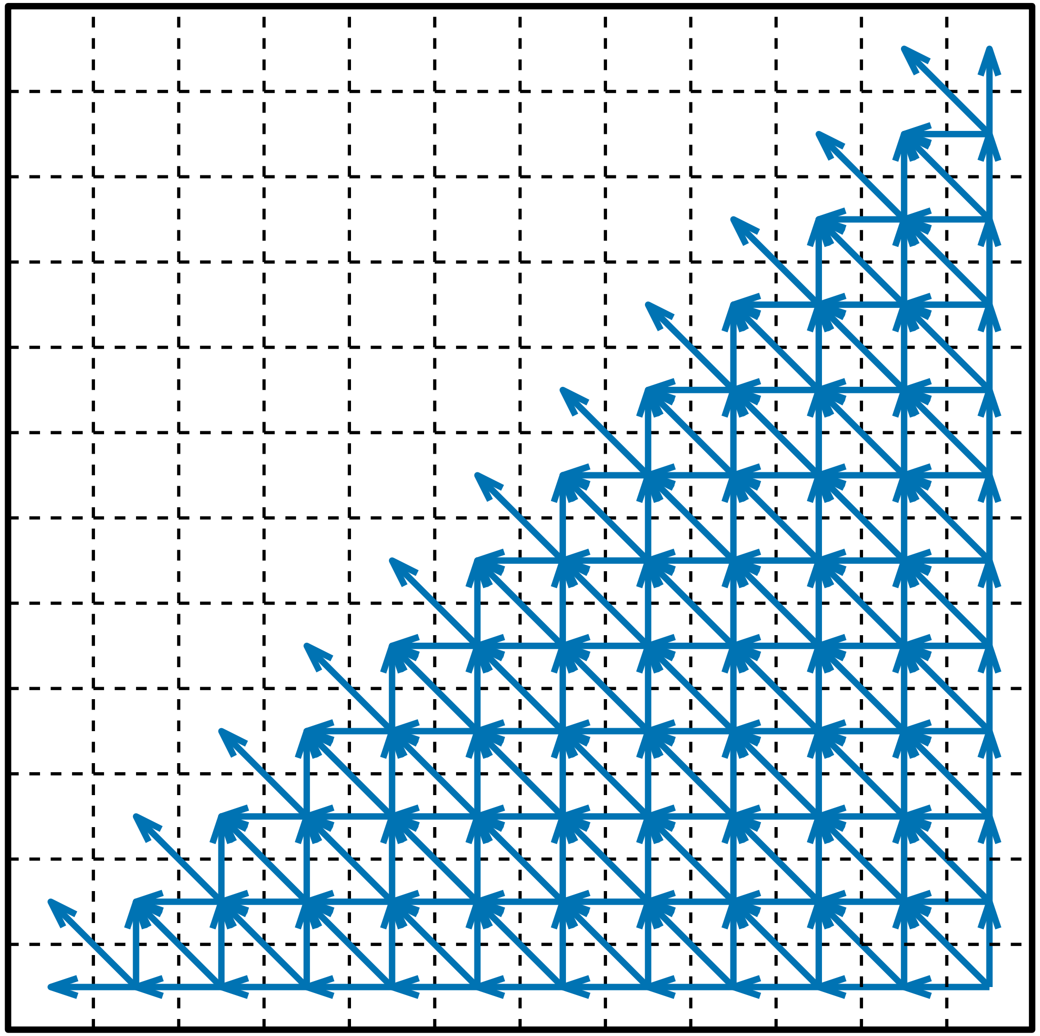
What information do we need to compute $\mathrm{LES}\,[n, n]$?

The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array LES:



What information do we need to compute $\mathrm{LES}[n, n]$?

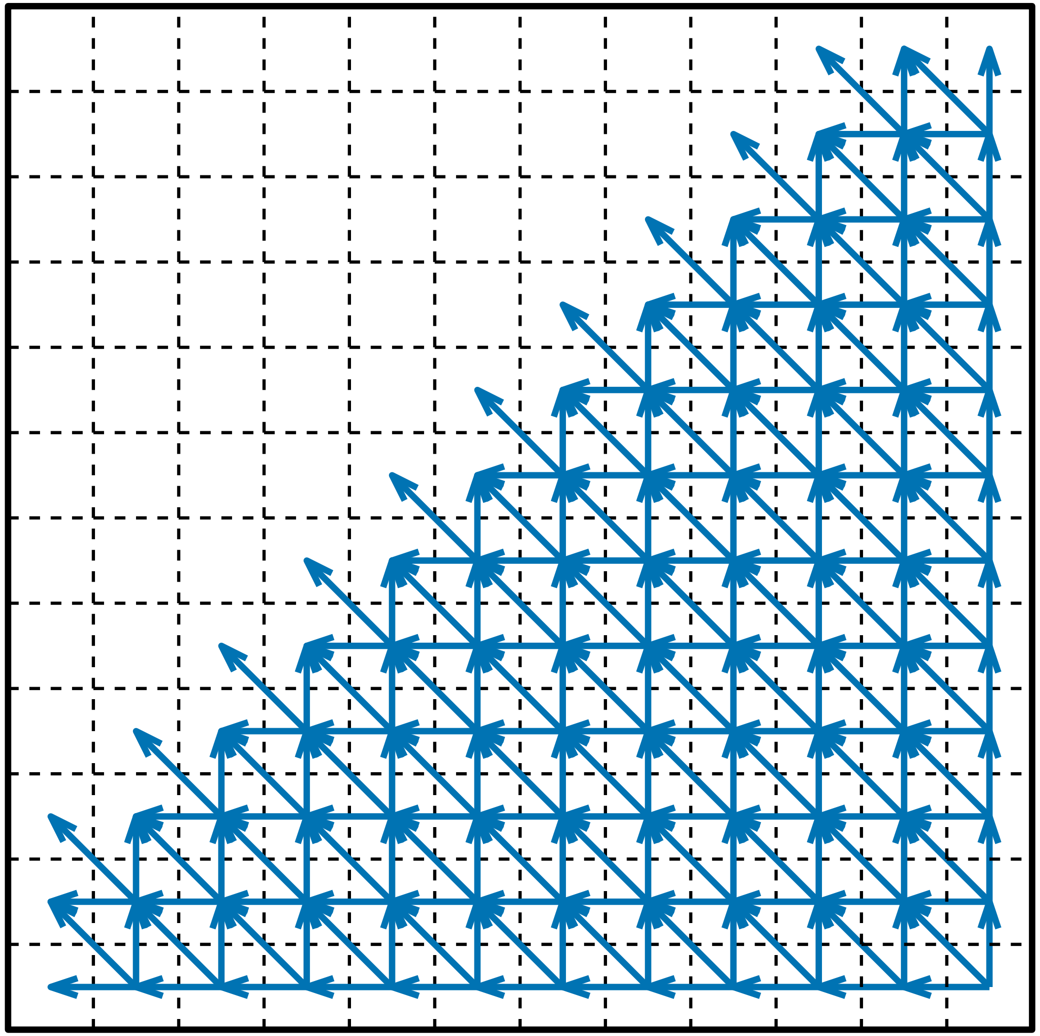
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\mathrm{LES}:$



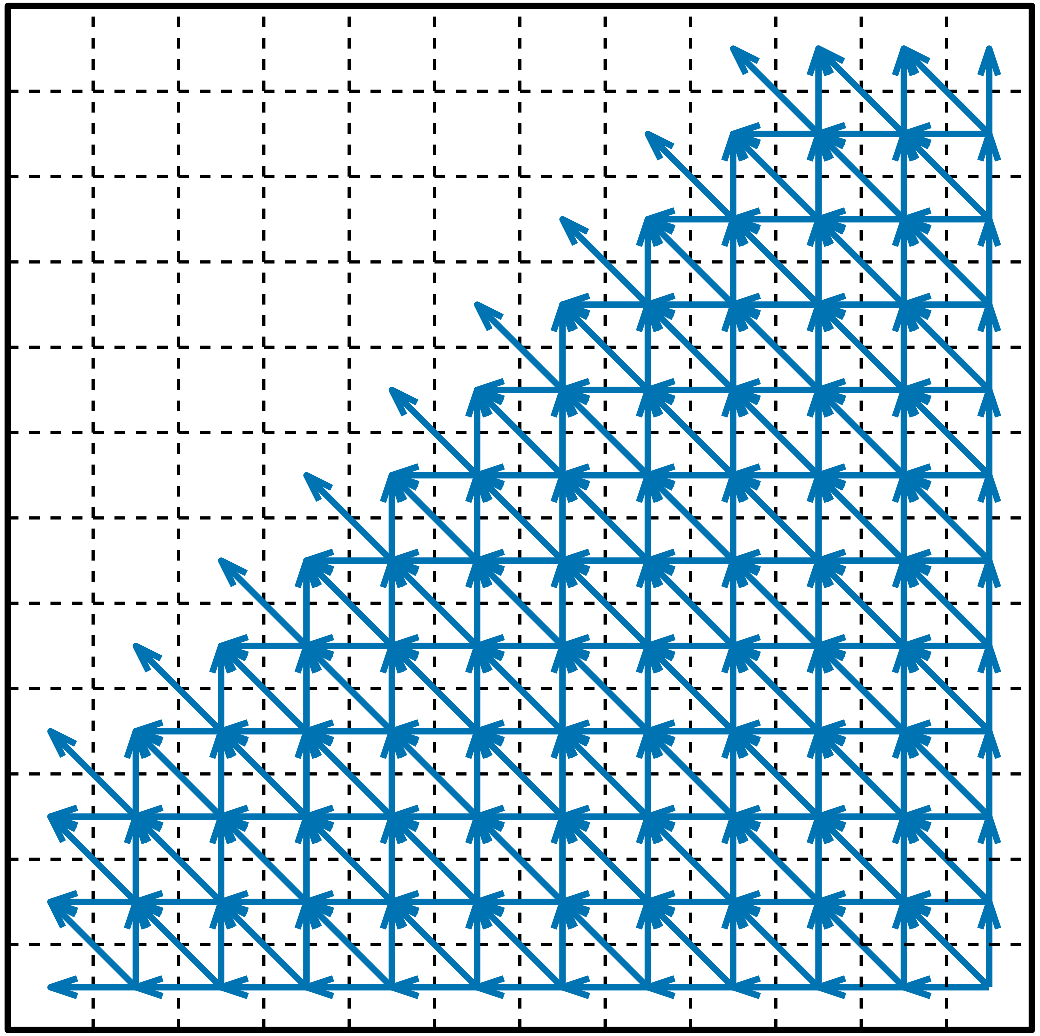
What information do we need to compute $\mathrm{LES}[n,n]$?

The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array LES:



What information do we need to compute $\mathrm{LES}[n,n]$?

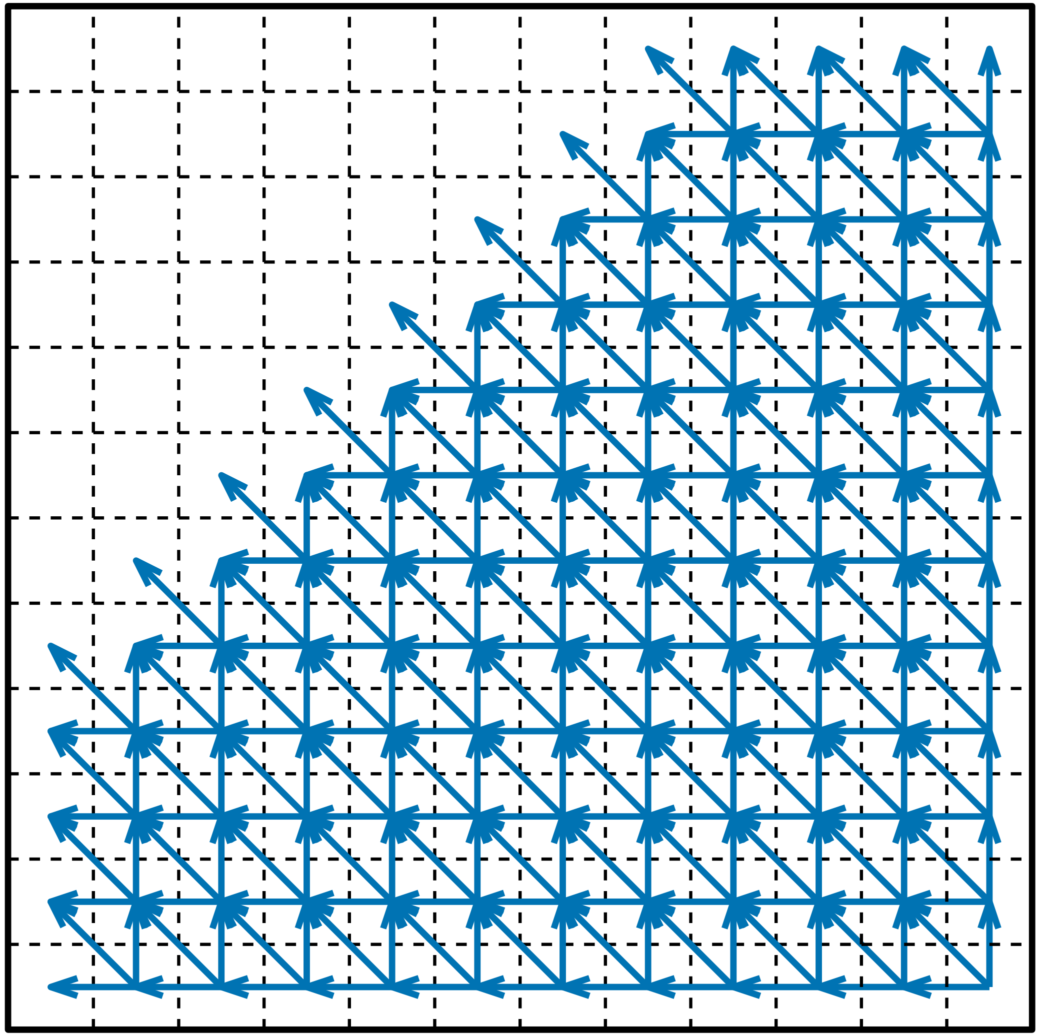
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}[n,n]$?

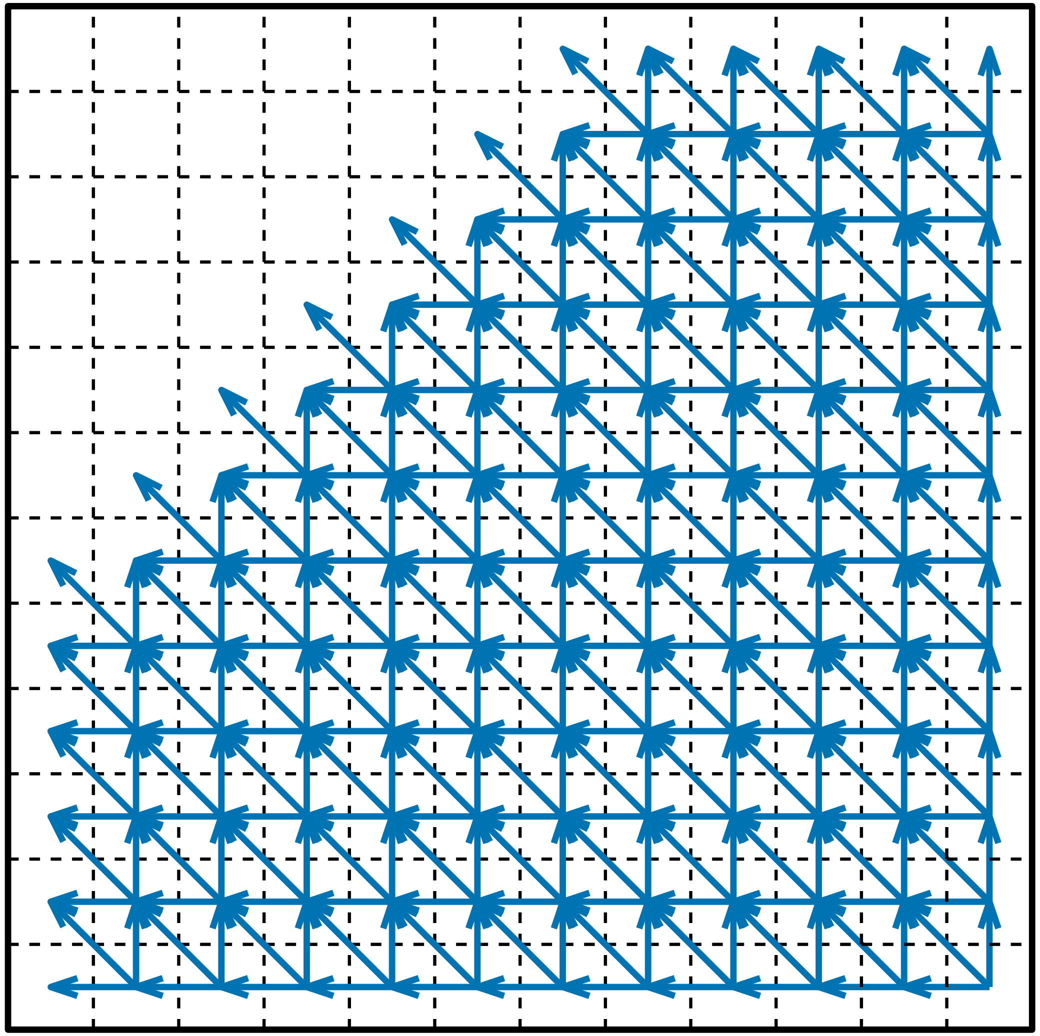
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MemLES}(x-1,y-1), \text{MemLES}(x-1,y), \text{MemLES}(x,y-1)\right)+1$$

(for $x,y>1$ and (x,y) non empty)

The 2D array

LES:



What information do we need to compute $\text{LES}[n,n]$?

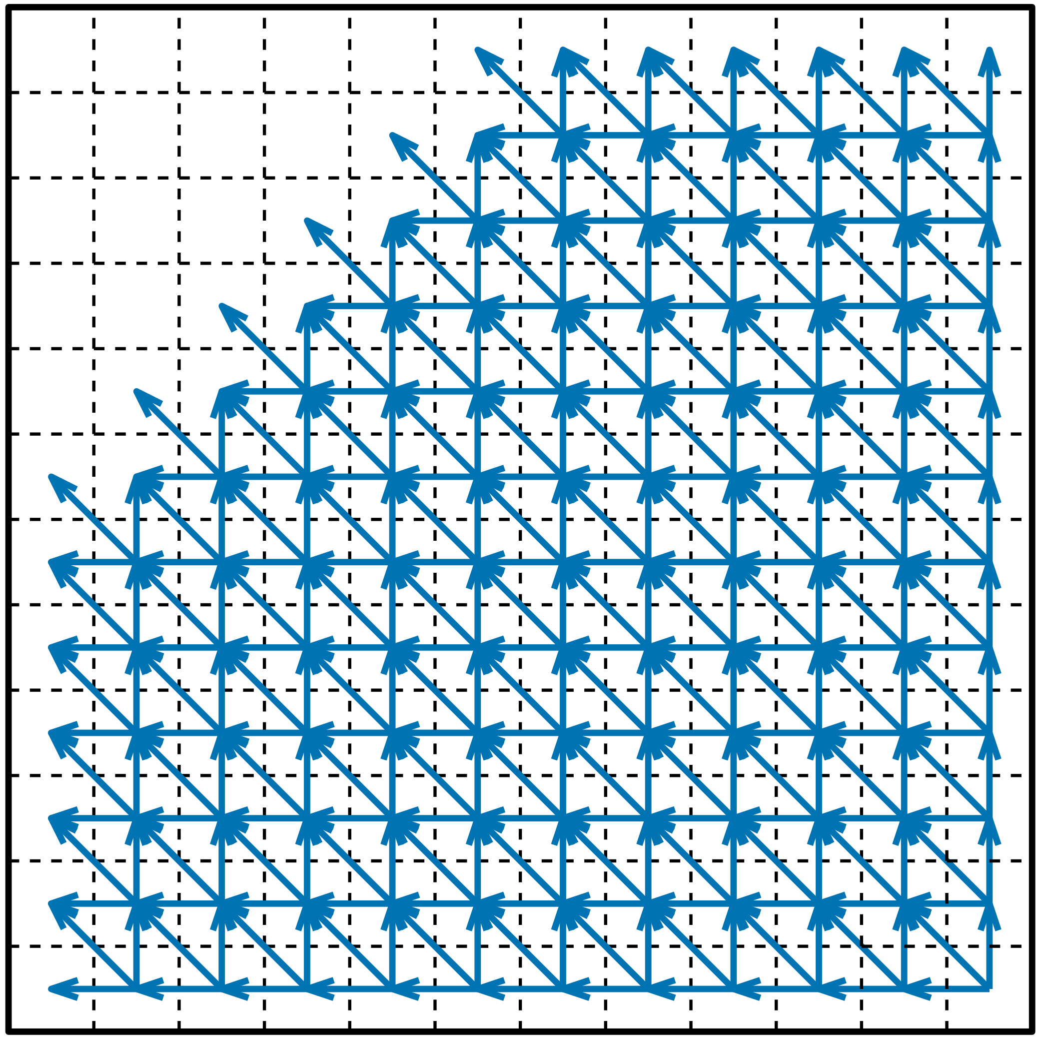
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}\,[n,n]$?

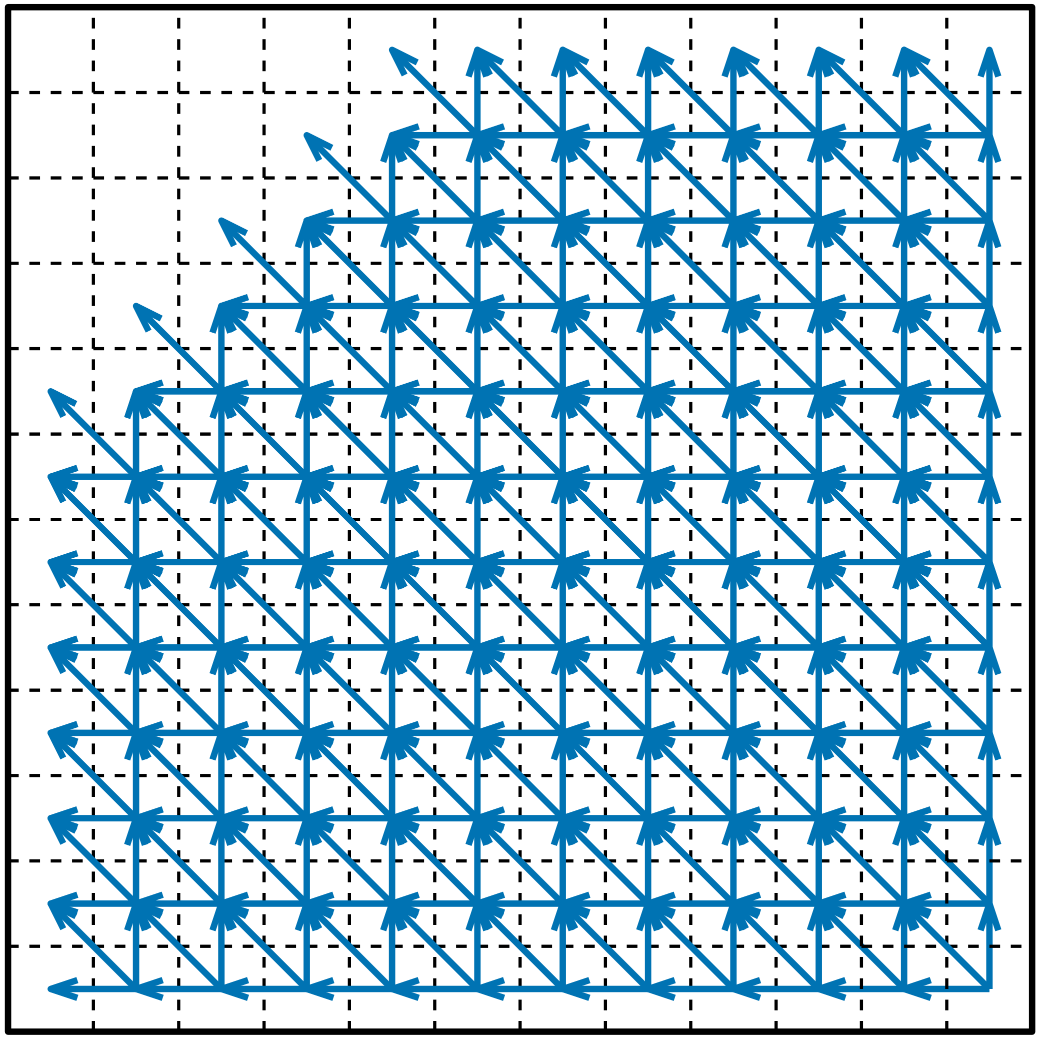
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}\,[n,n]$?

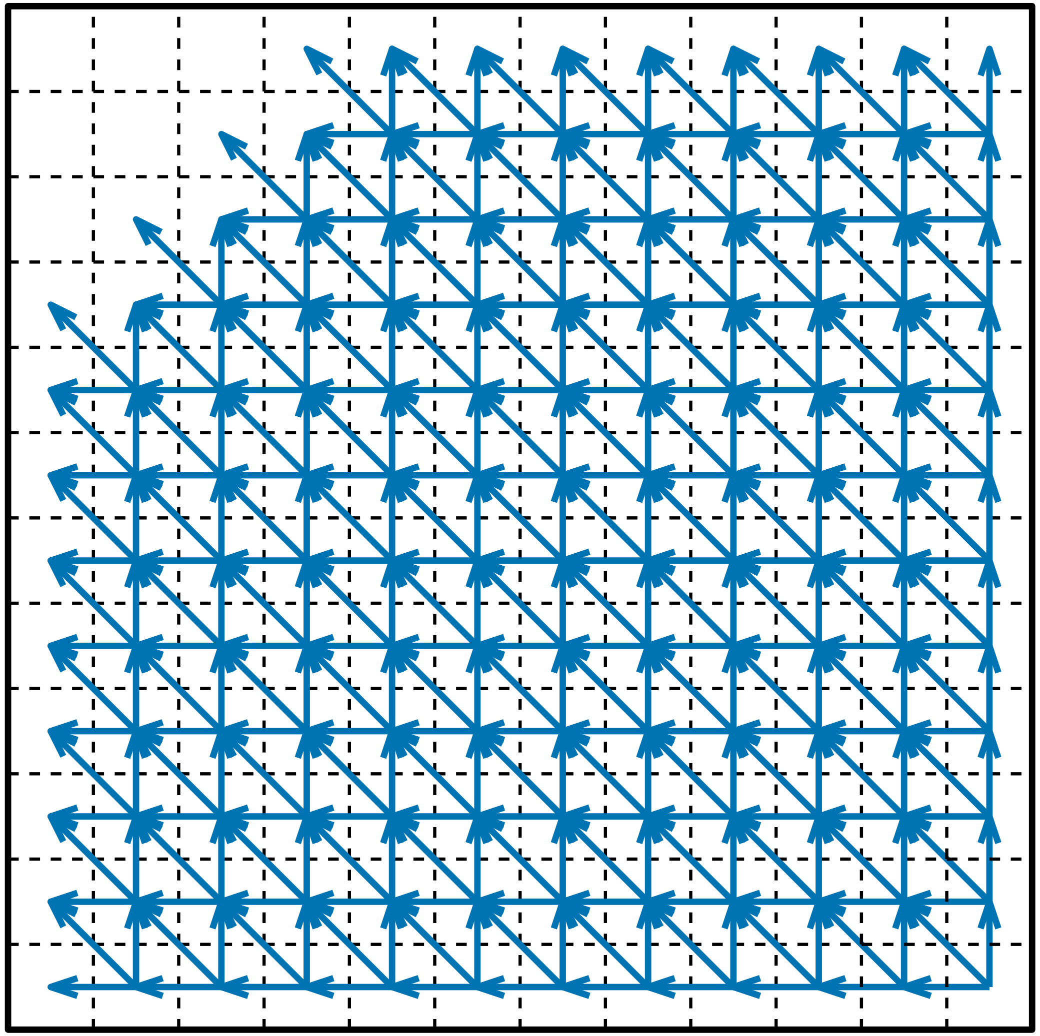
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES:



What information do we need to compute $\mathrm{LES}[n,n]$?

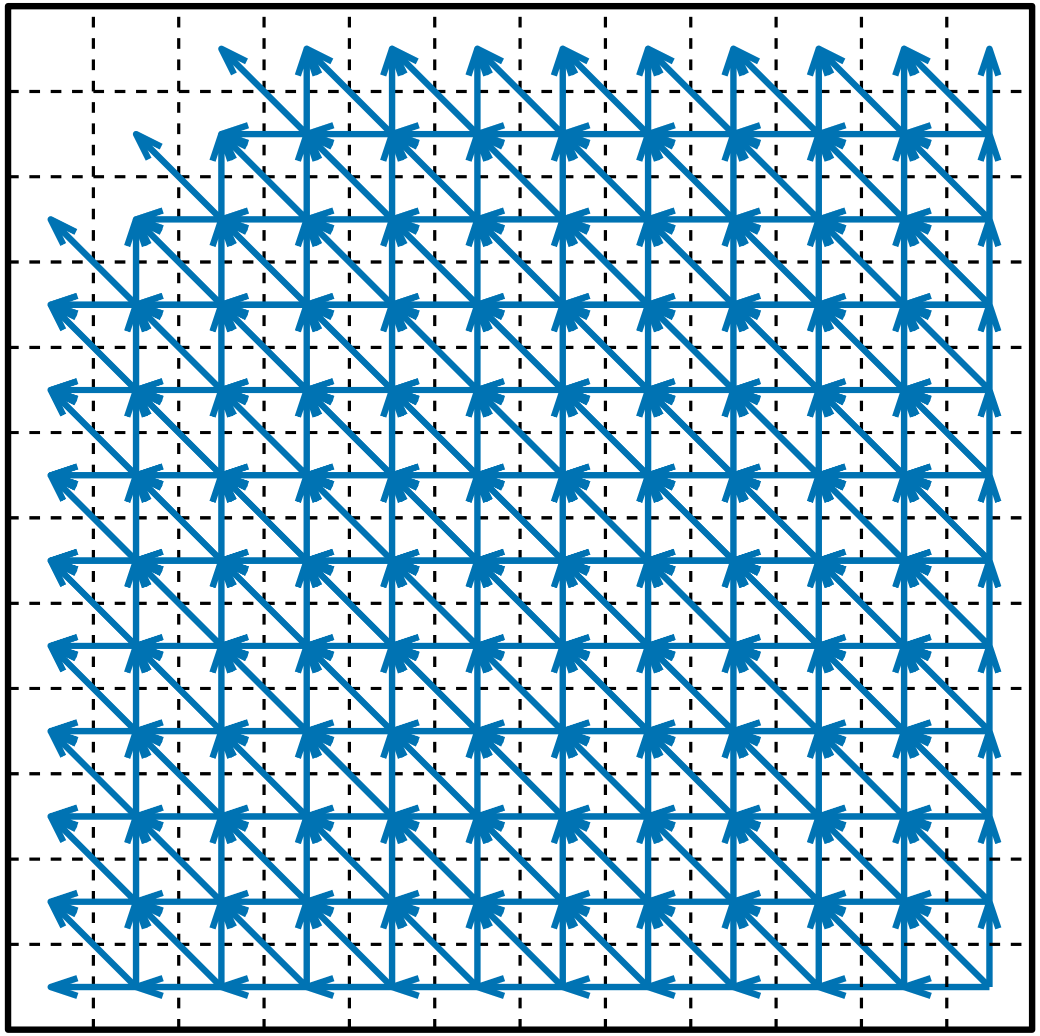
The dependency graph

$$\mathtt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x,y>1$ and (x,y) non empty)

The 2D array

$\mathtt{LES}$:



What information do we need to compute $\mathtt{LES}[n,n]$?

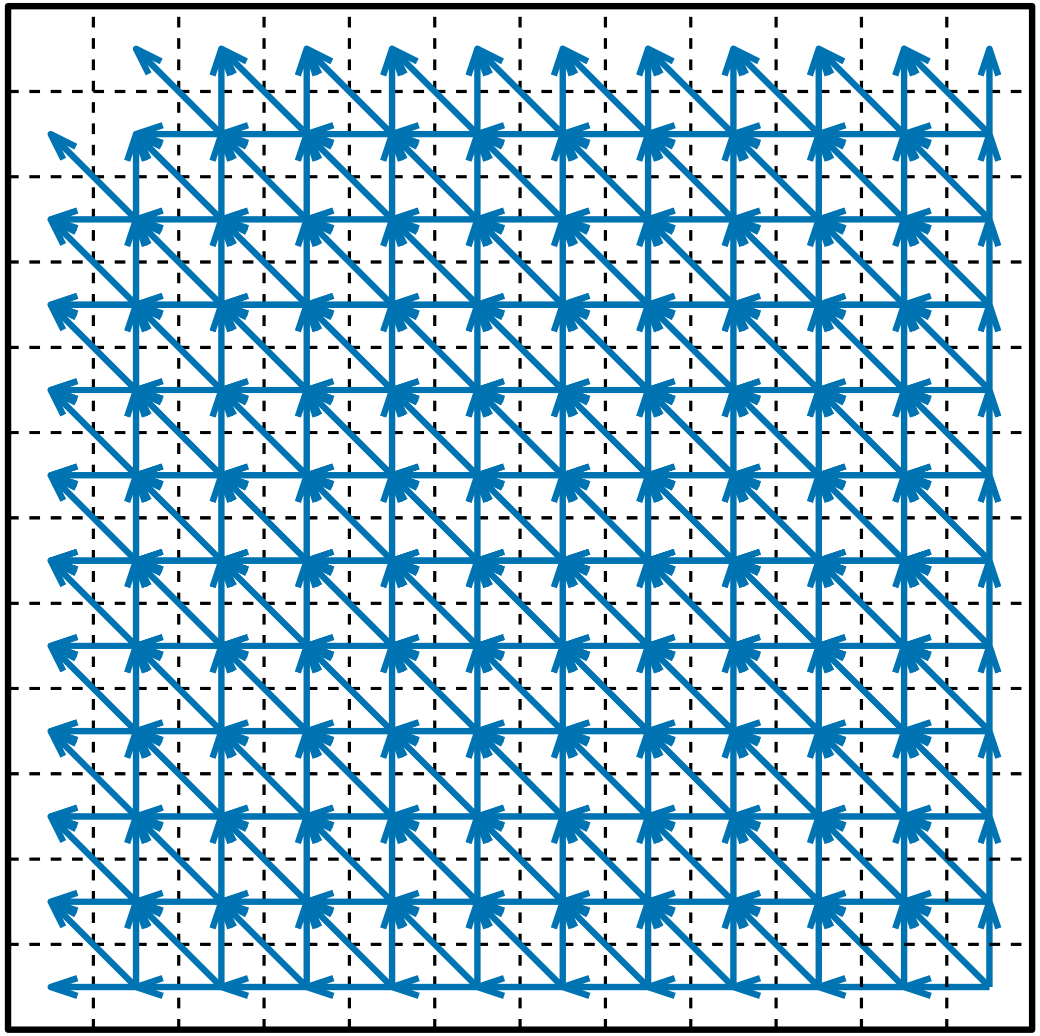
The dependency graph

$$\mathtt{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\mathtt{LES}$:



What information do we need to compute $\mathtt{LES}[n,n]$?

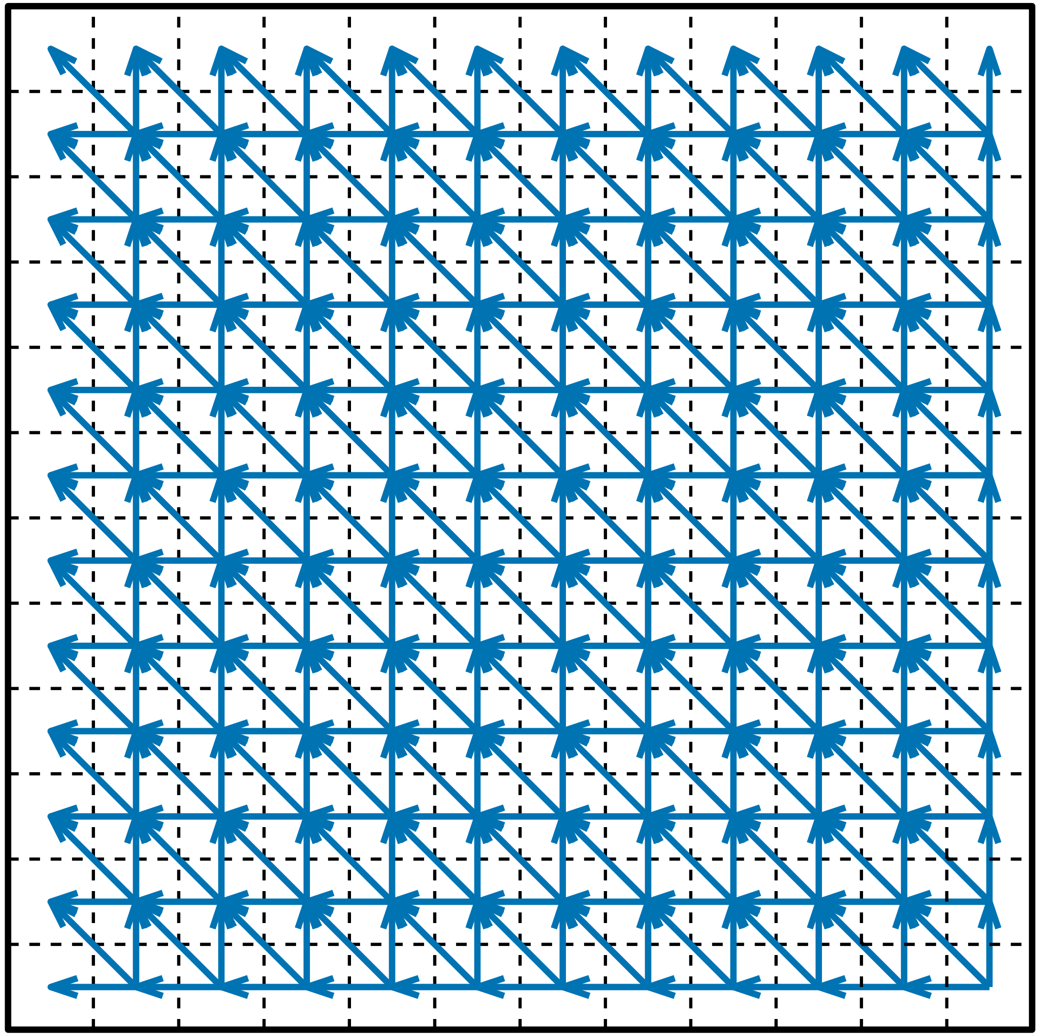
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



What information do we need to compute $\mathrm{LES}[n, n]$?

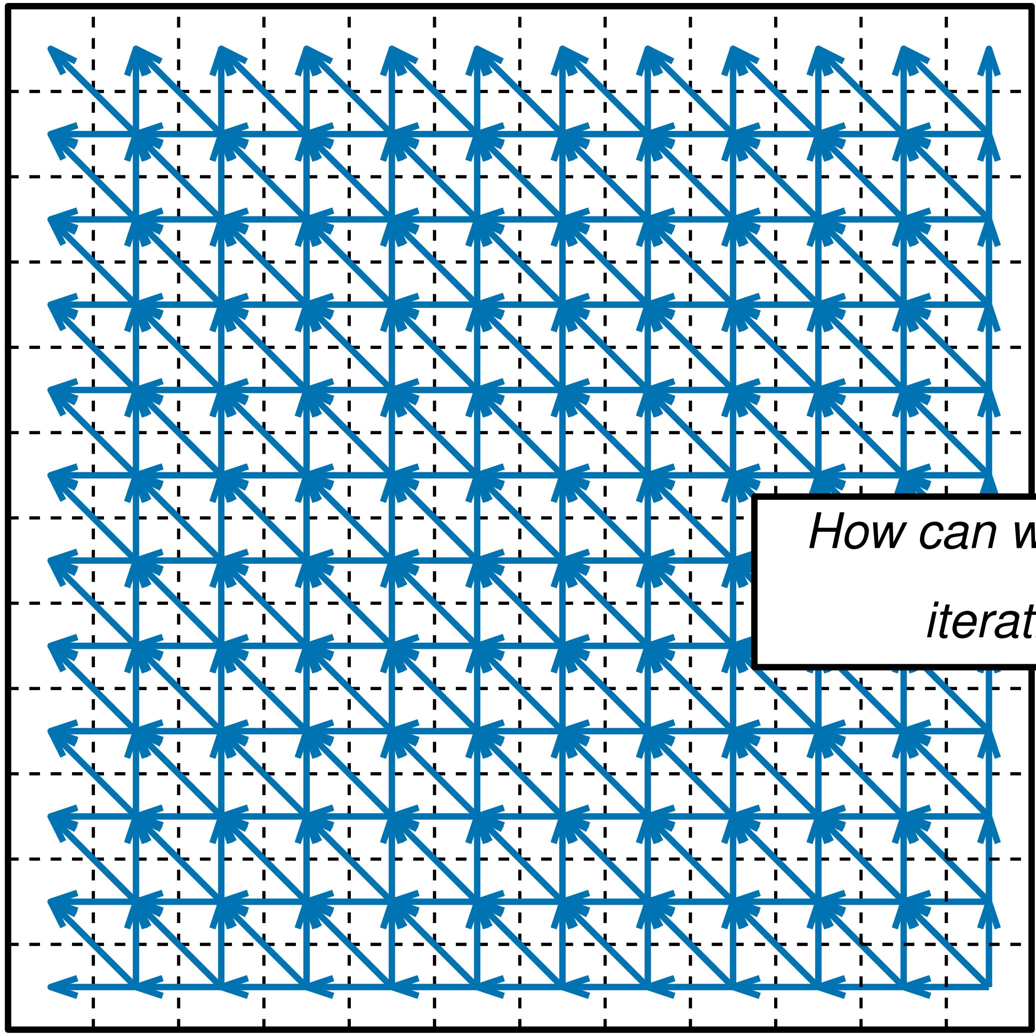
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

What information do we need to compute $\mathrm{LES}[n,n]$?

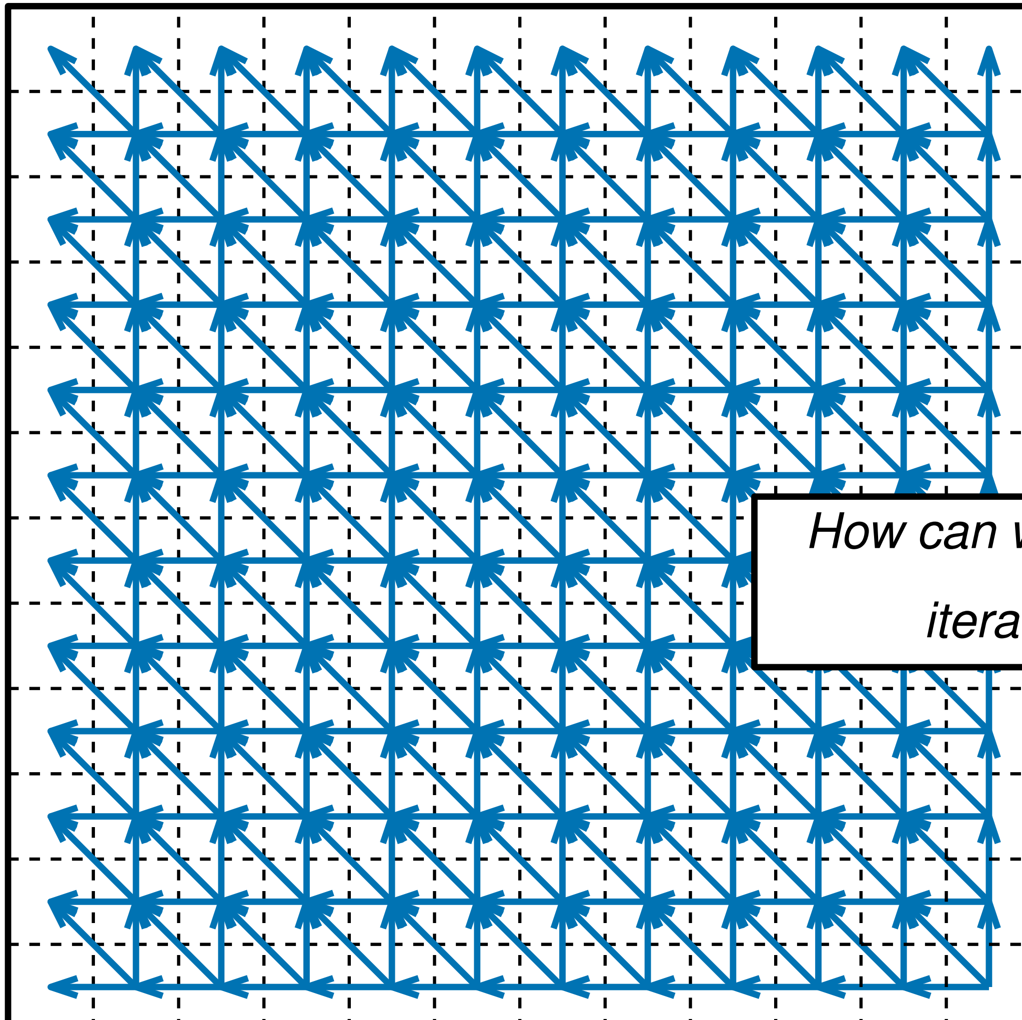
The dependency graph

$$\mathrm{LES}[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n, n]$?

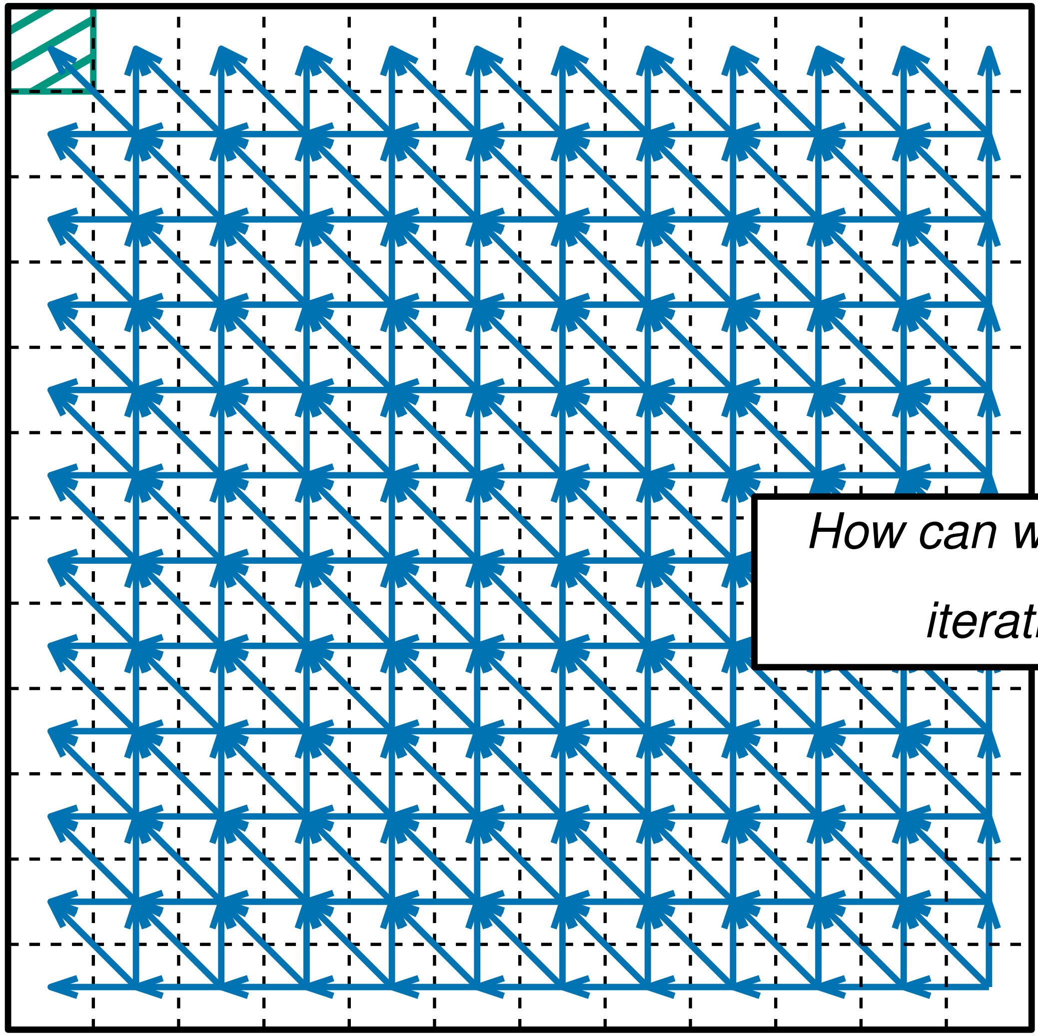
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MemLES}(x-1,y-1), \text{MemLES}(x-1,y), \text{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}[n, n]$?

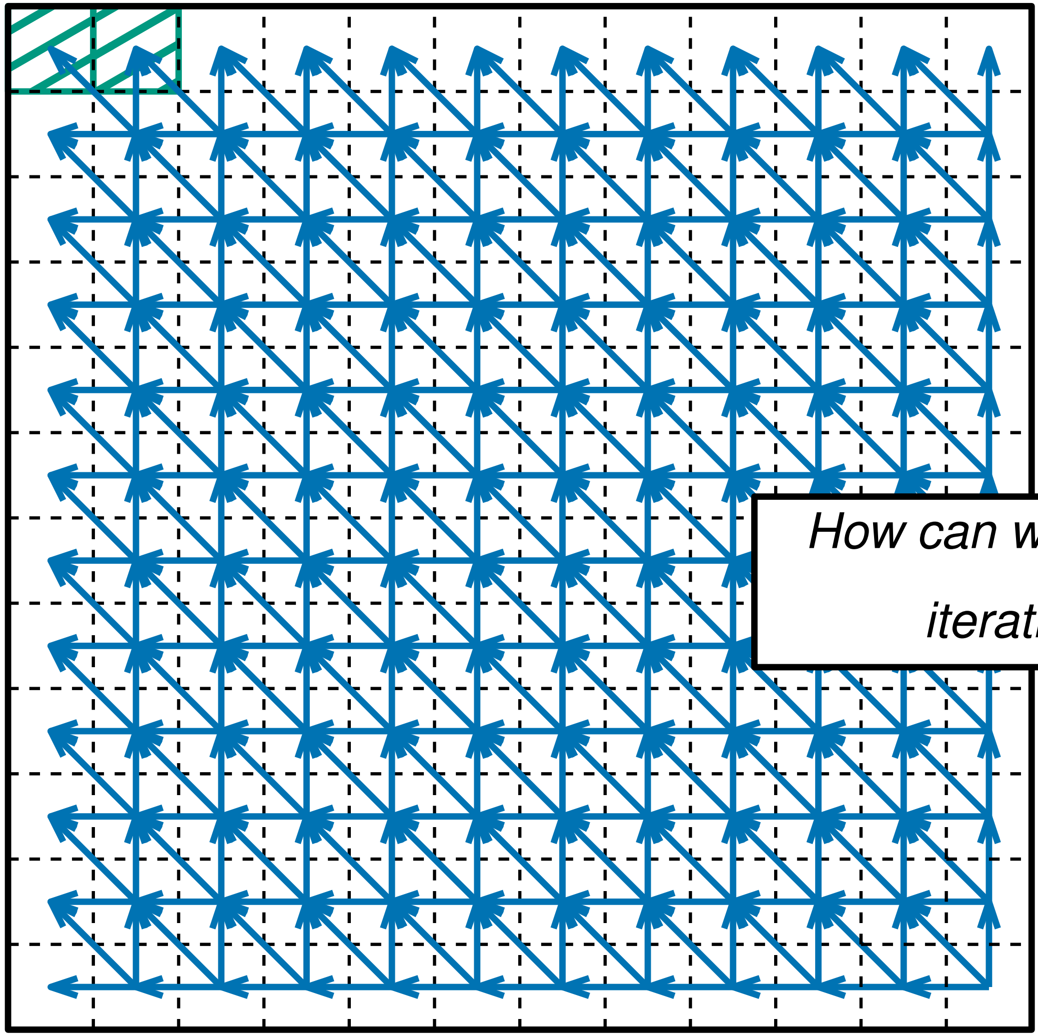
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

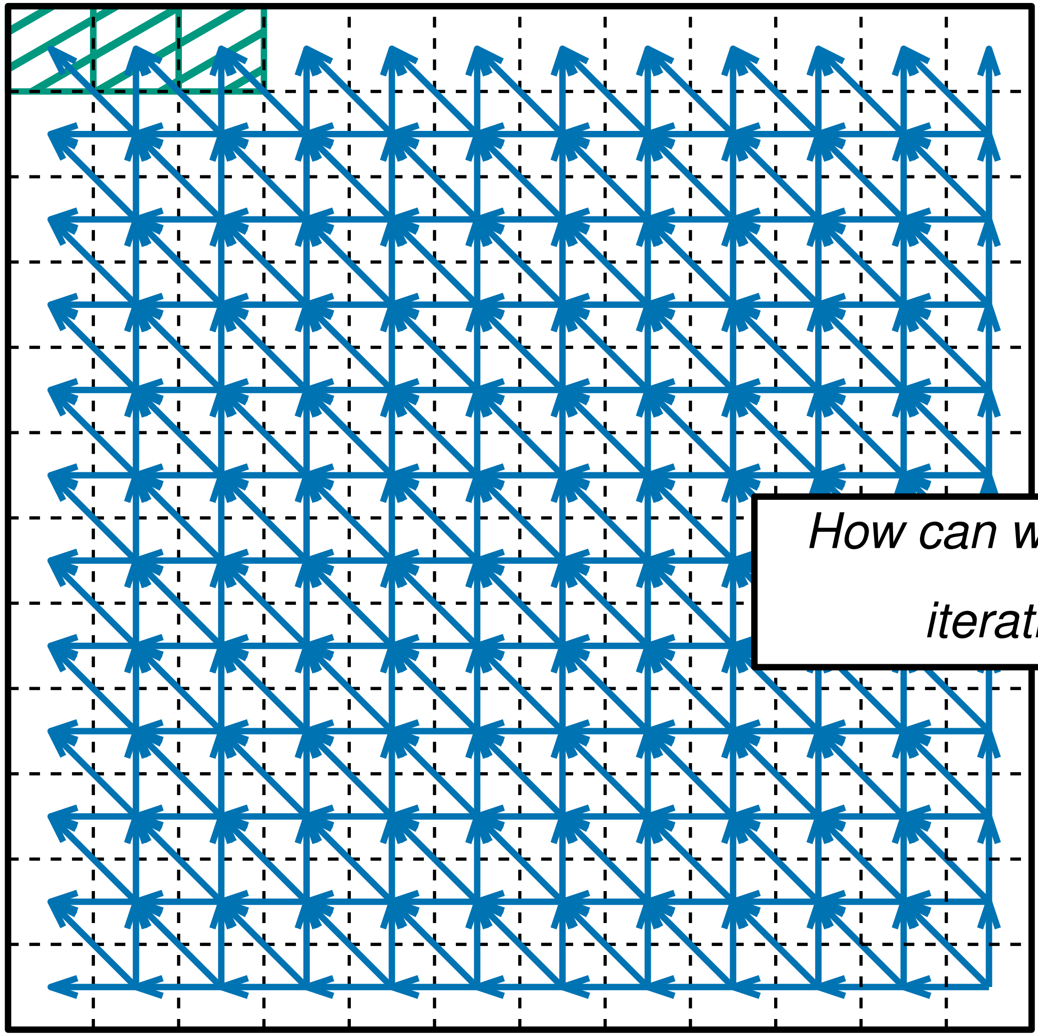
The dependency graph

$$\mathrm{LES}[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n, n]$?

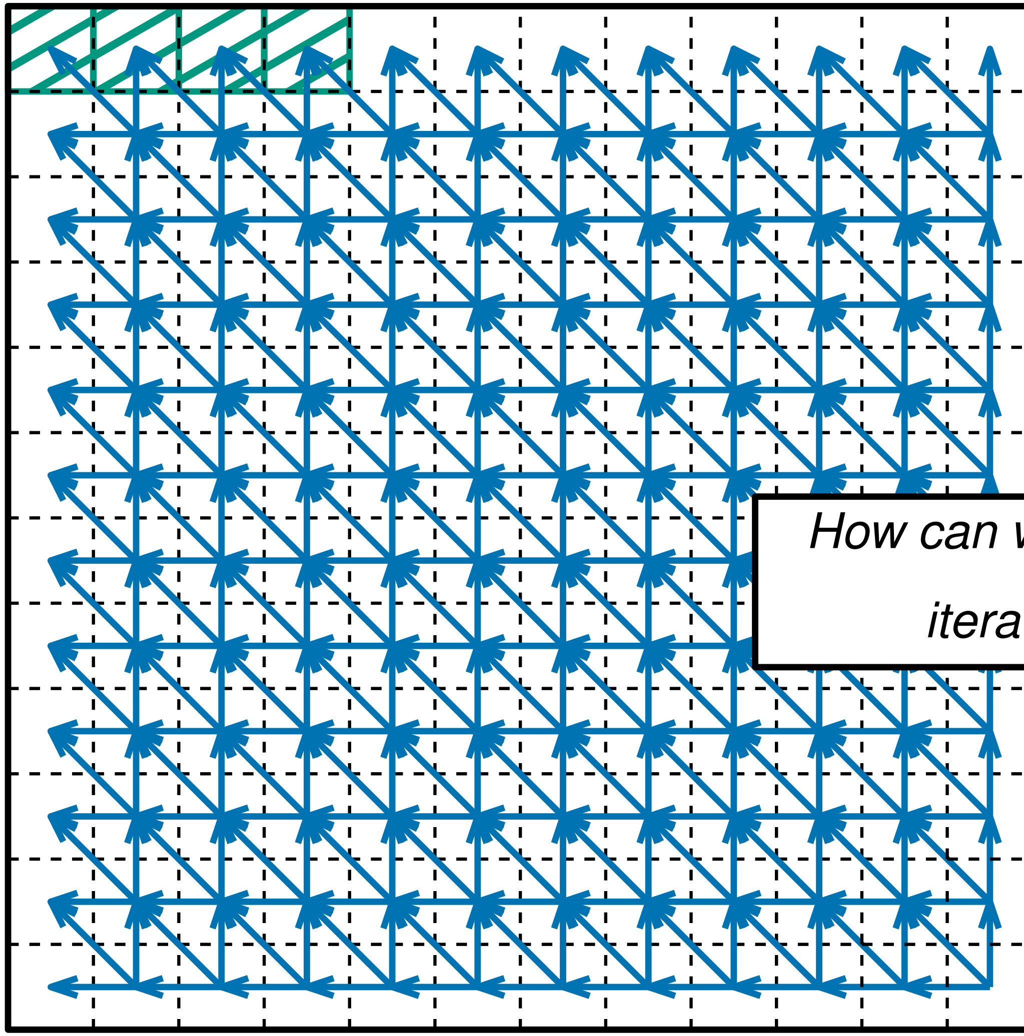
The dependency graph

$$\text{LES}[x, y] = \min\left(\text{MEMLES}(x-1, y-1), \text{MEMLES}(x-1, y), \text{MEMLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}[n, n]$?

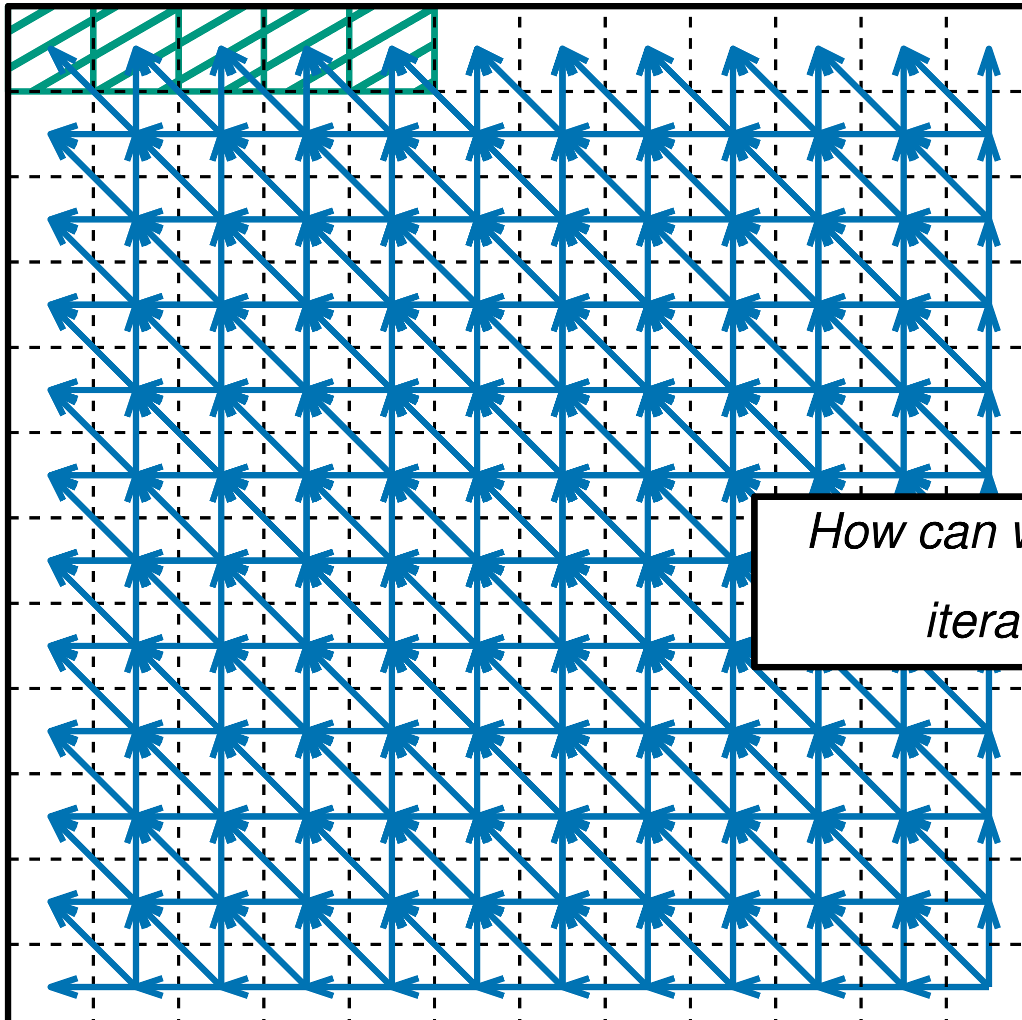
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n, n]$?

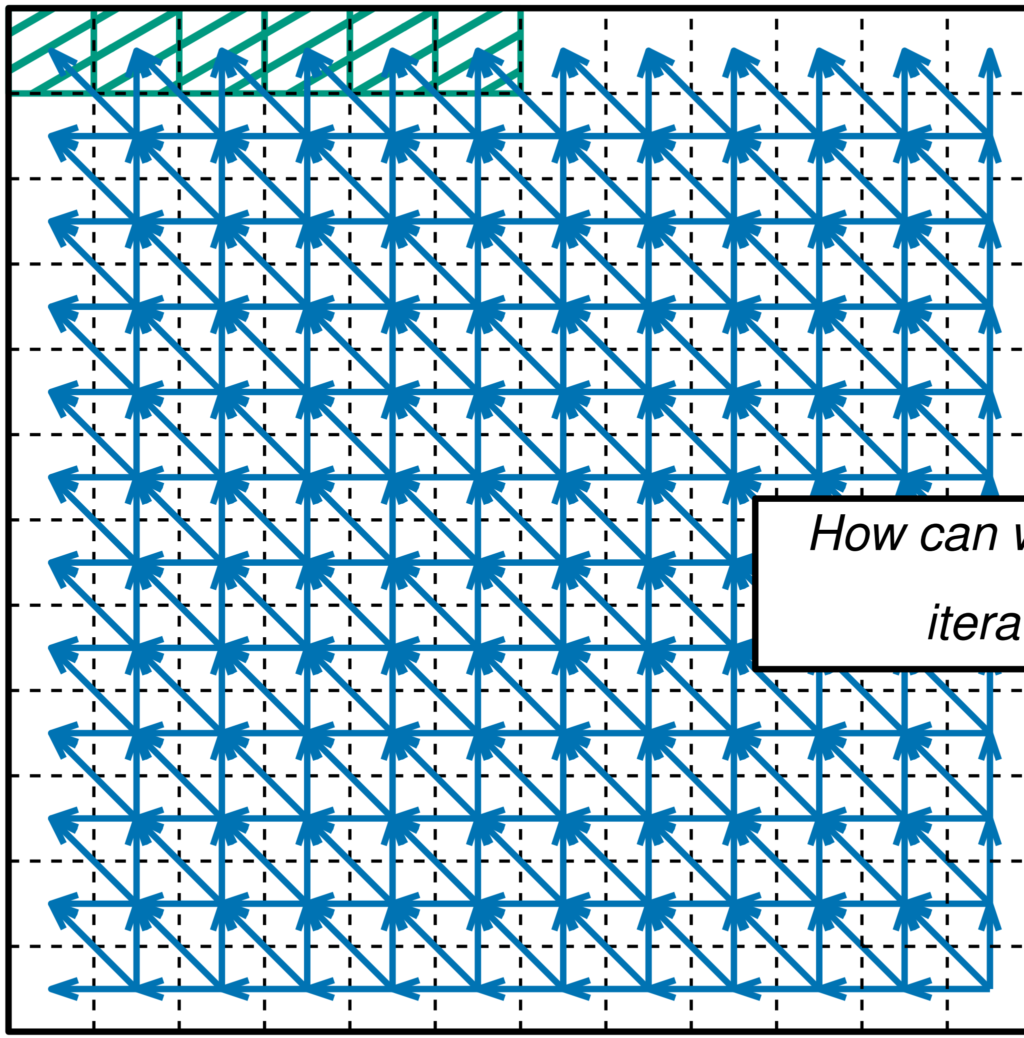
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

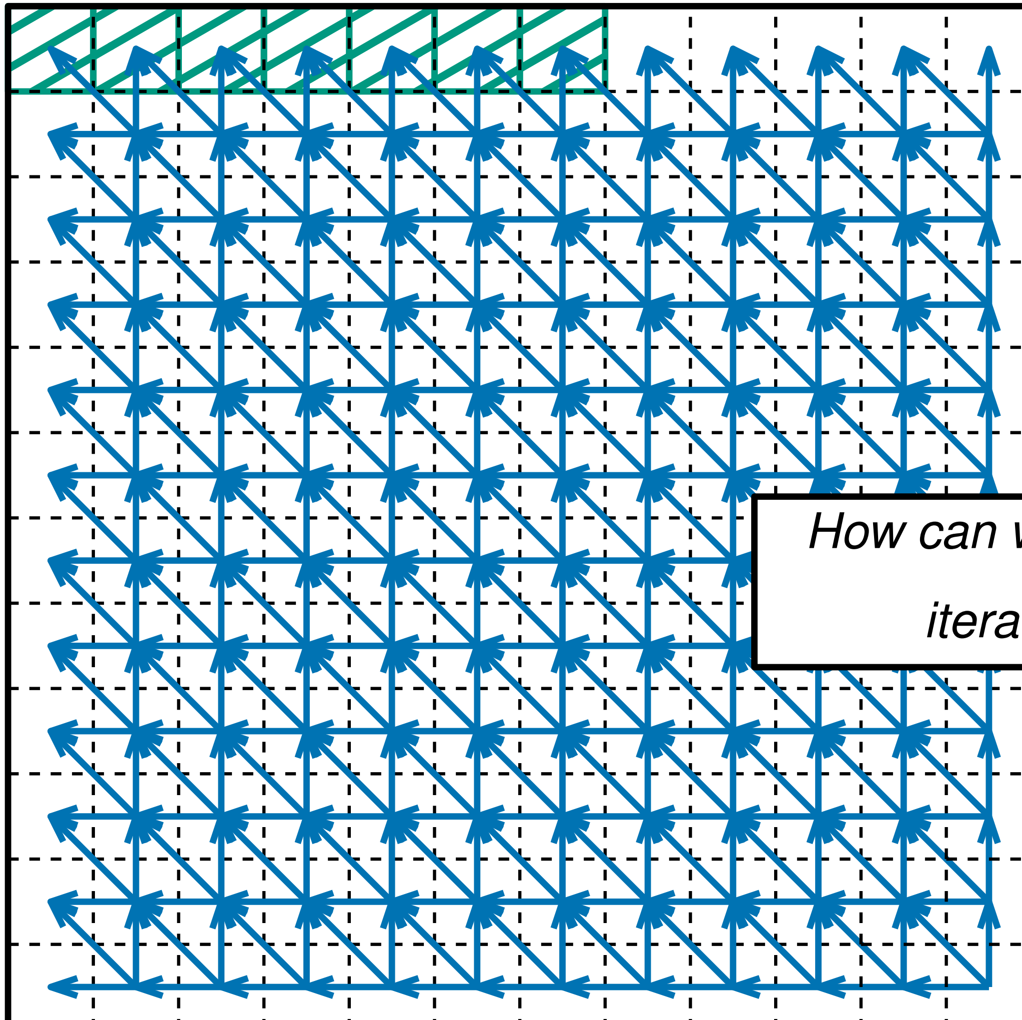
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

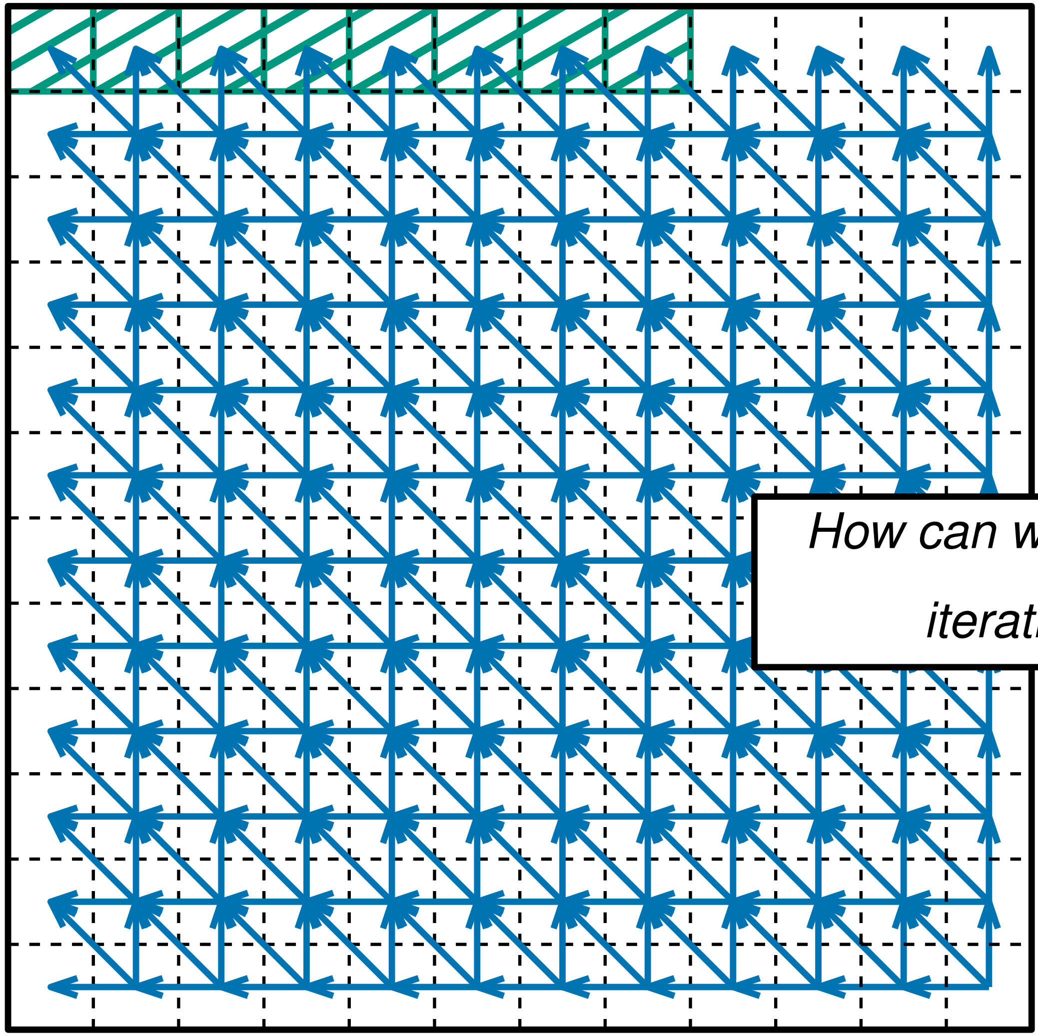
The dependency graph

$$\text{LES}\,[x,y] = \min\left(\text{MEMLES}(x-1,y-1), \text{MEMLES}(x-1,y), \text{MEMLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

$\text{LES}:$



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}\,[n, n]$?

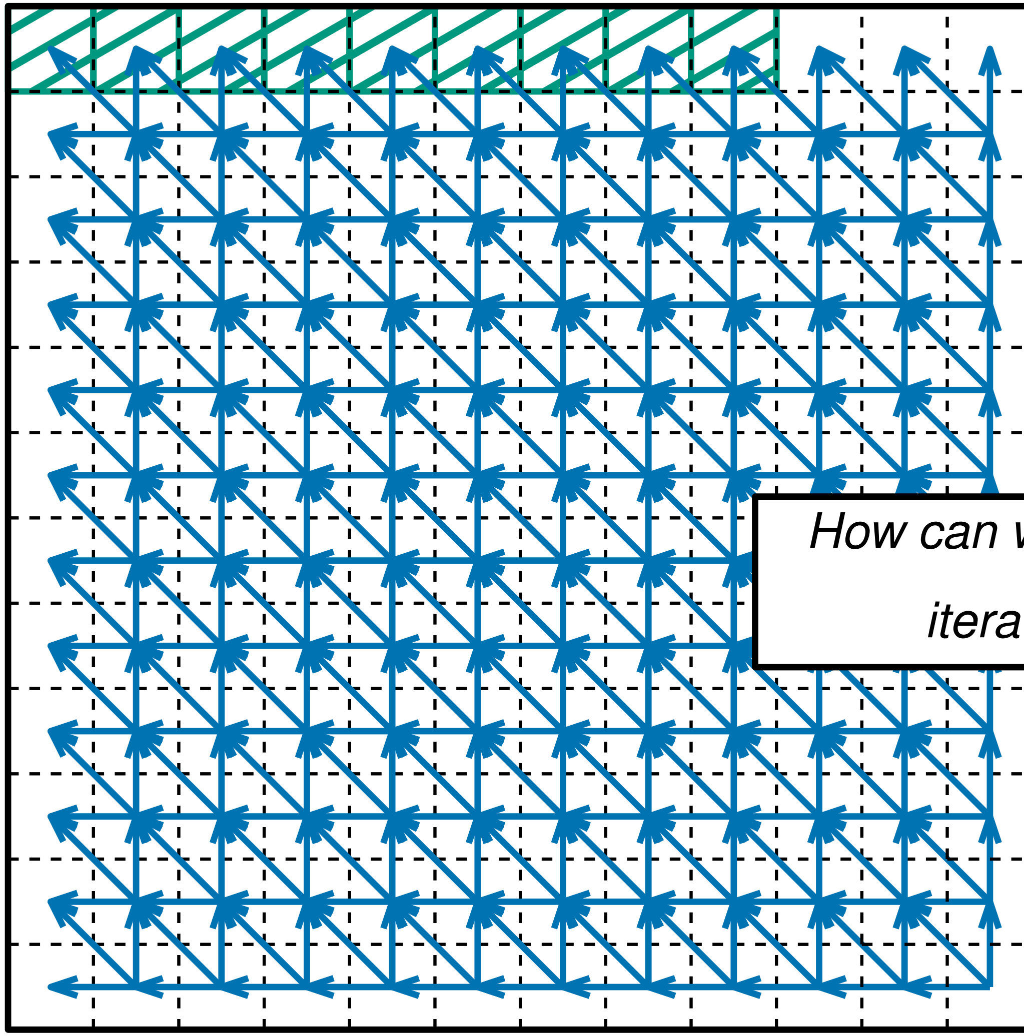
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

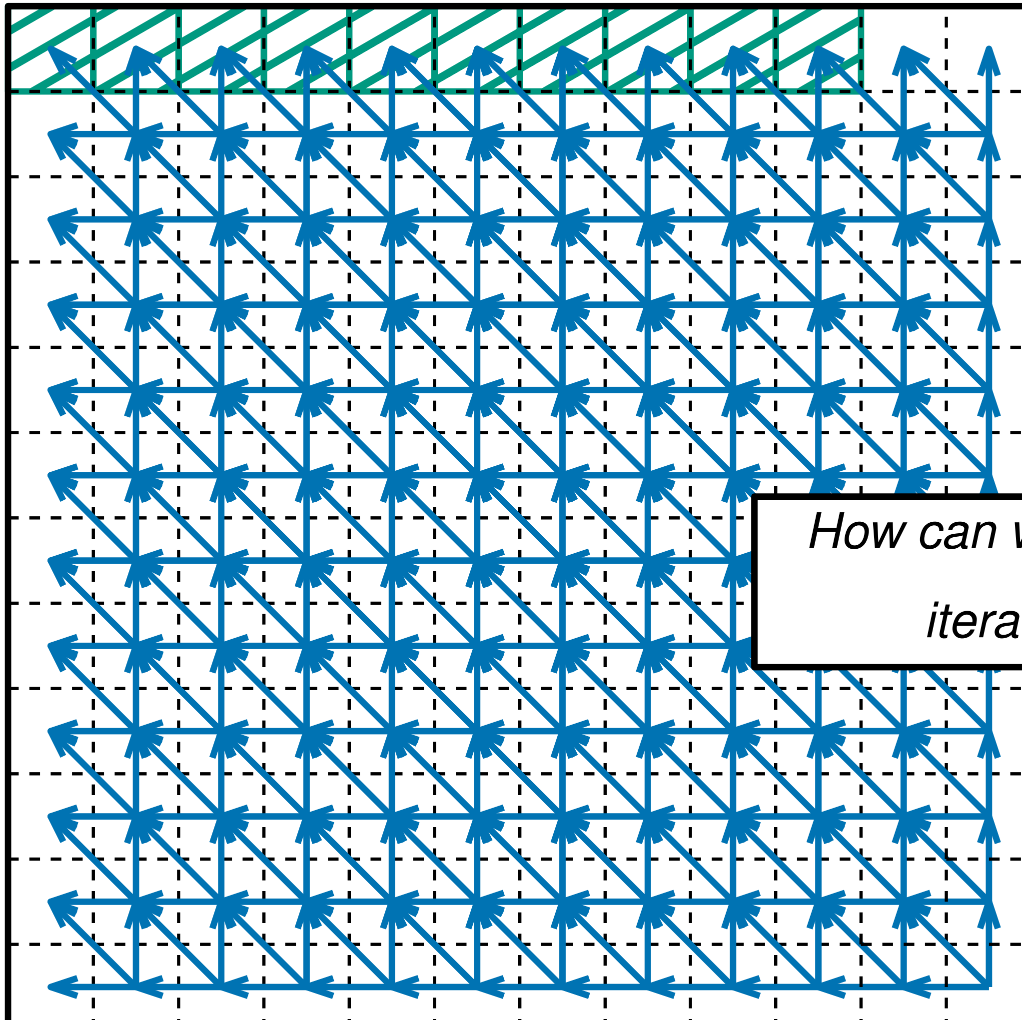
The dependency graph

$$\text{LES}\left[x,y\right] = \min\left(\text{MemLES}(x-1,y-1), \text{MemLES}(x-1,y), \text{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}\left[n,n\right]$?

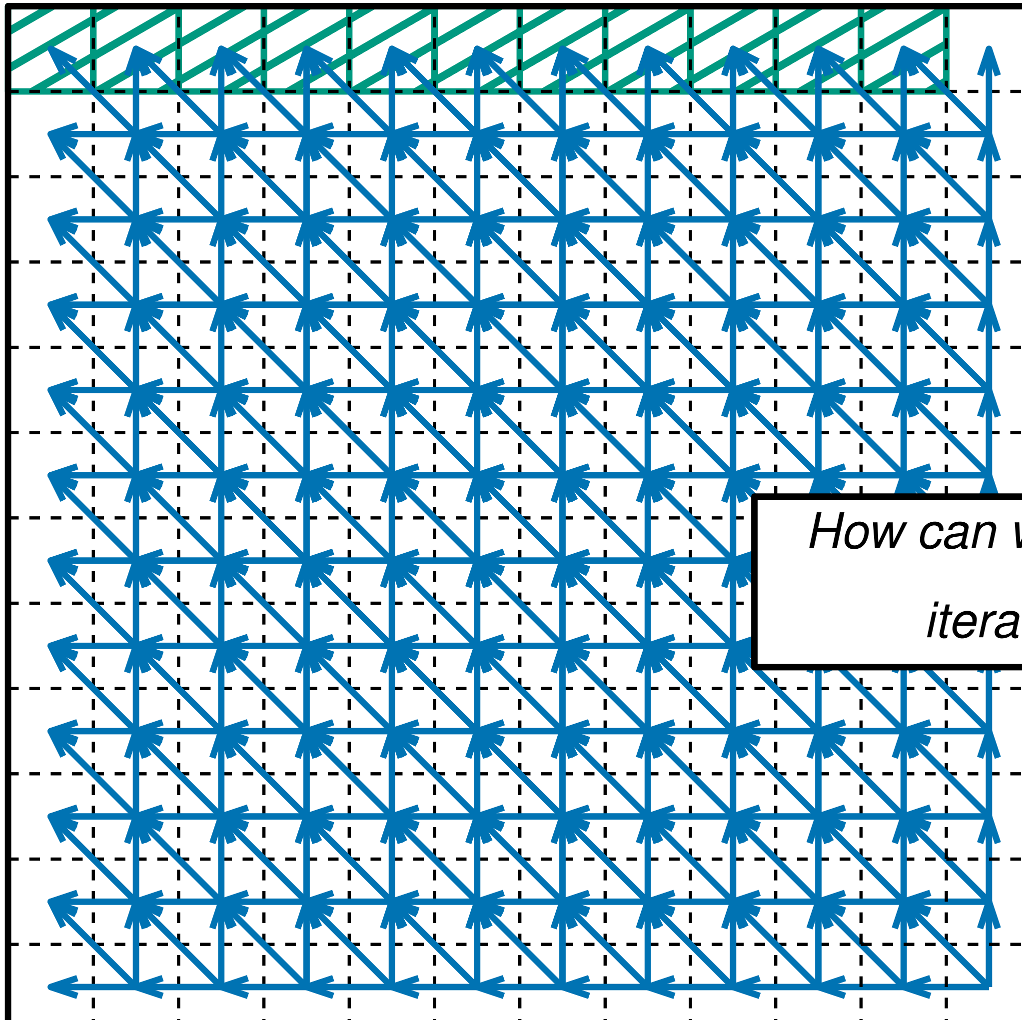
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n, n]$?

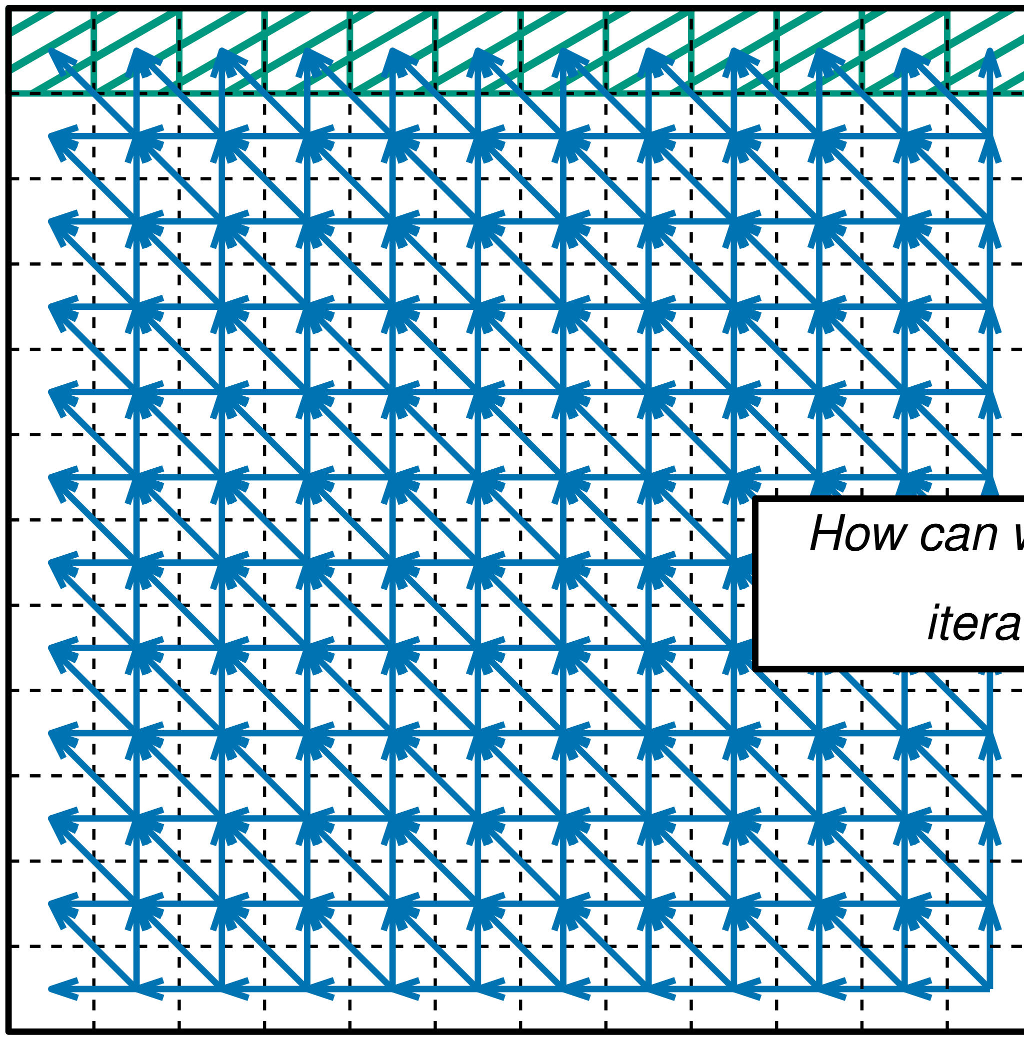
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

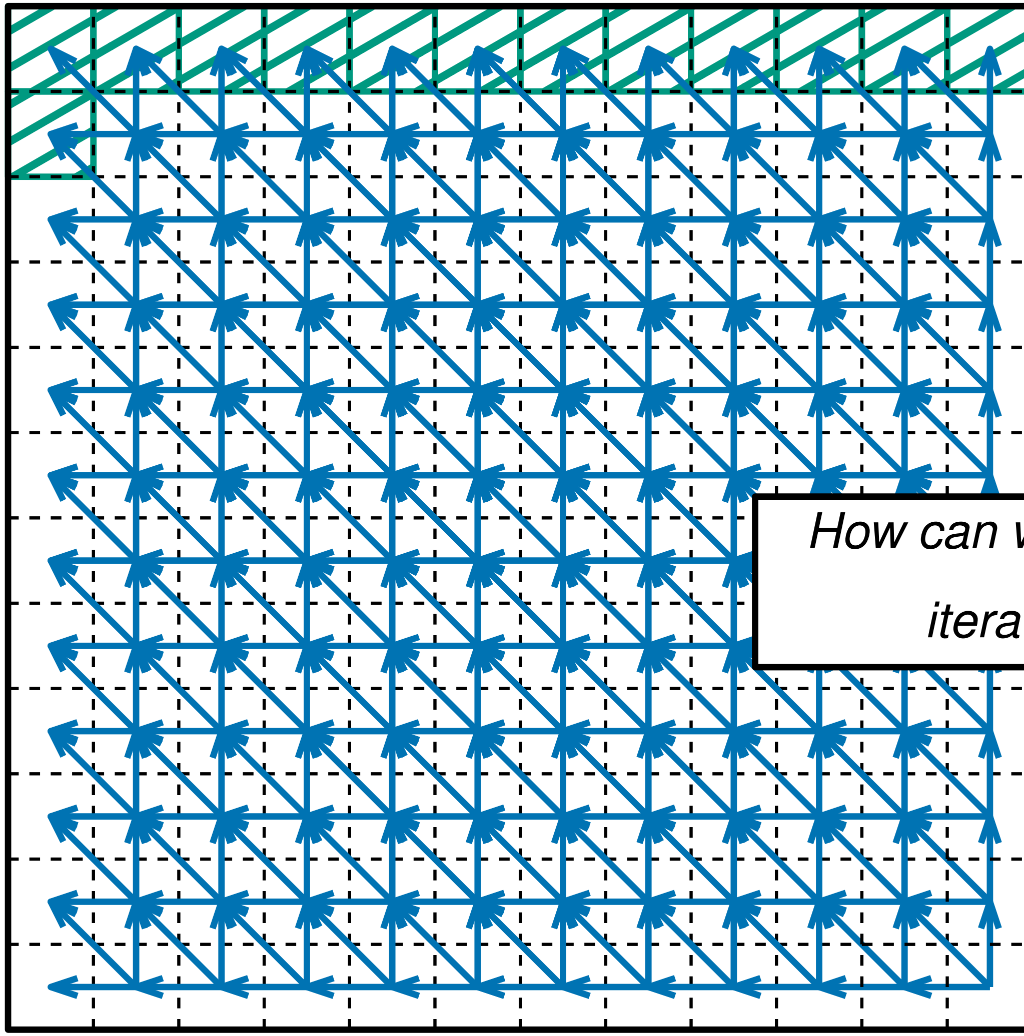
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

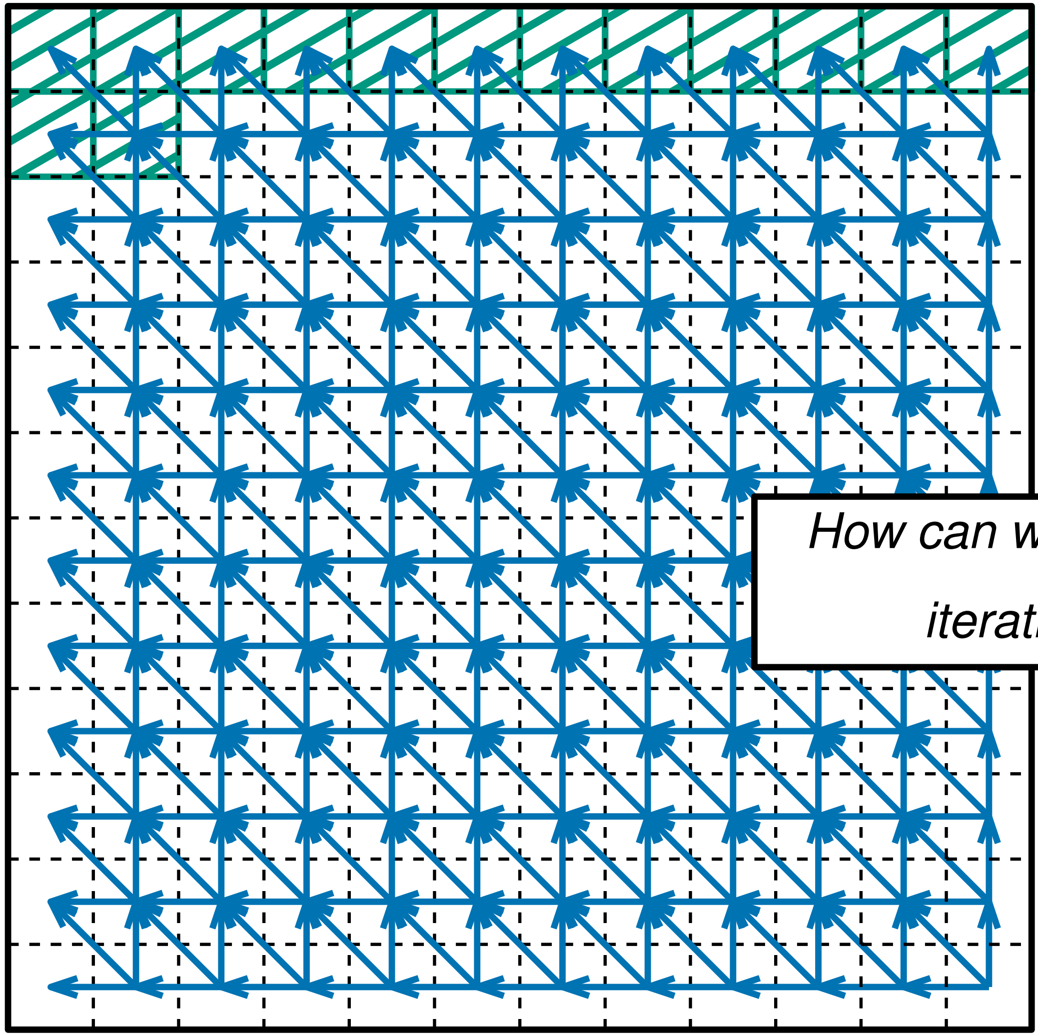
The dependency graph

$$\text{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}[n, n]$?

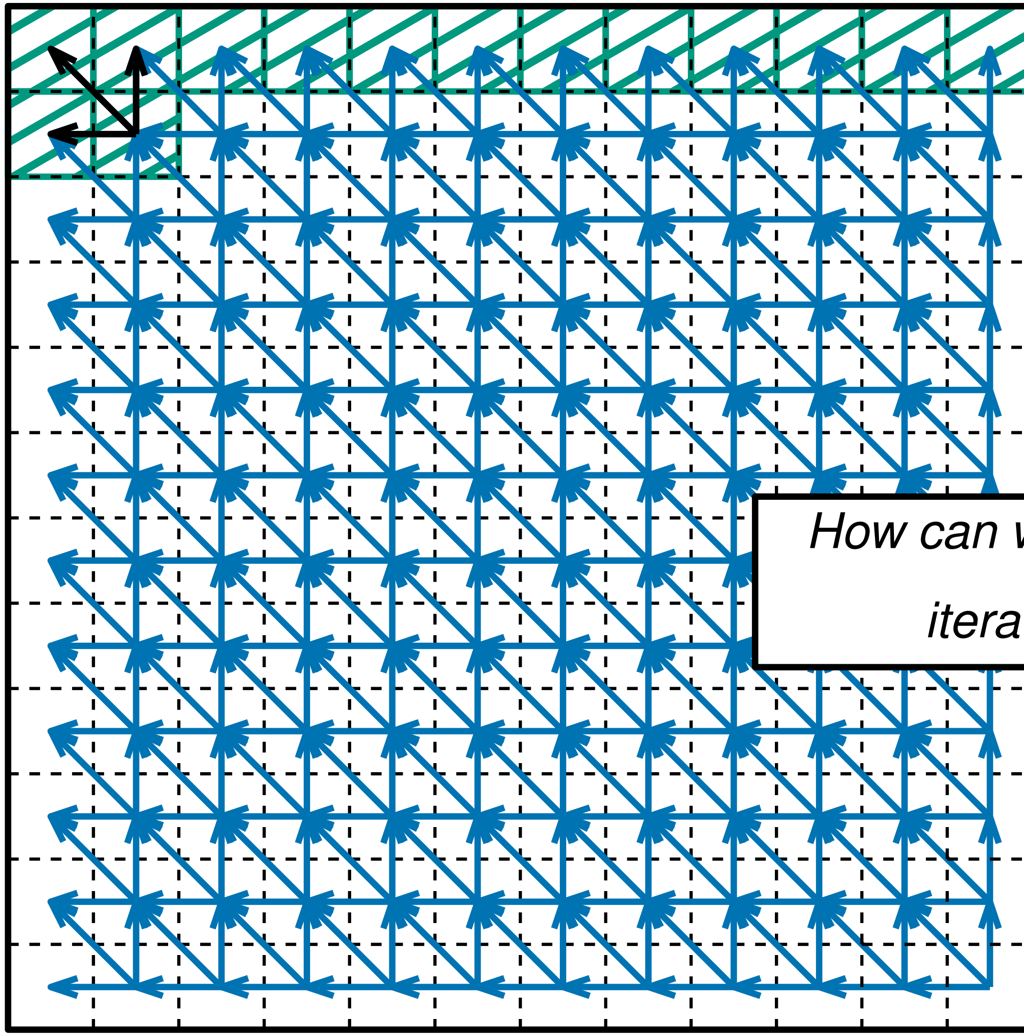
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

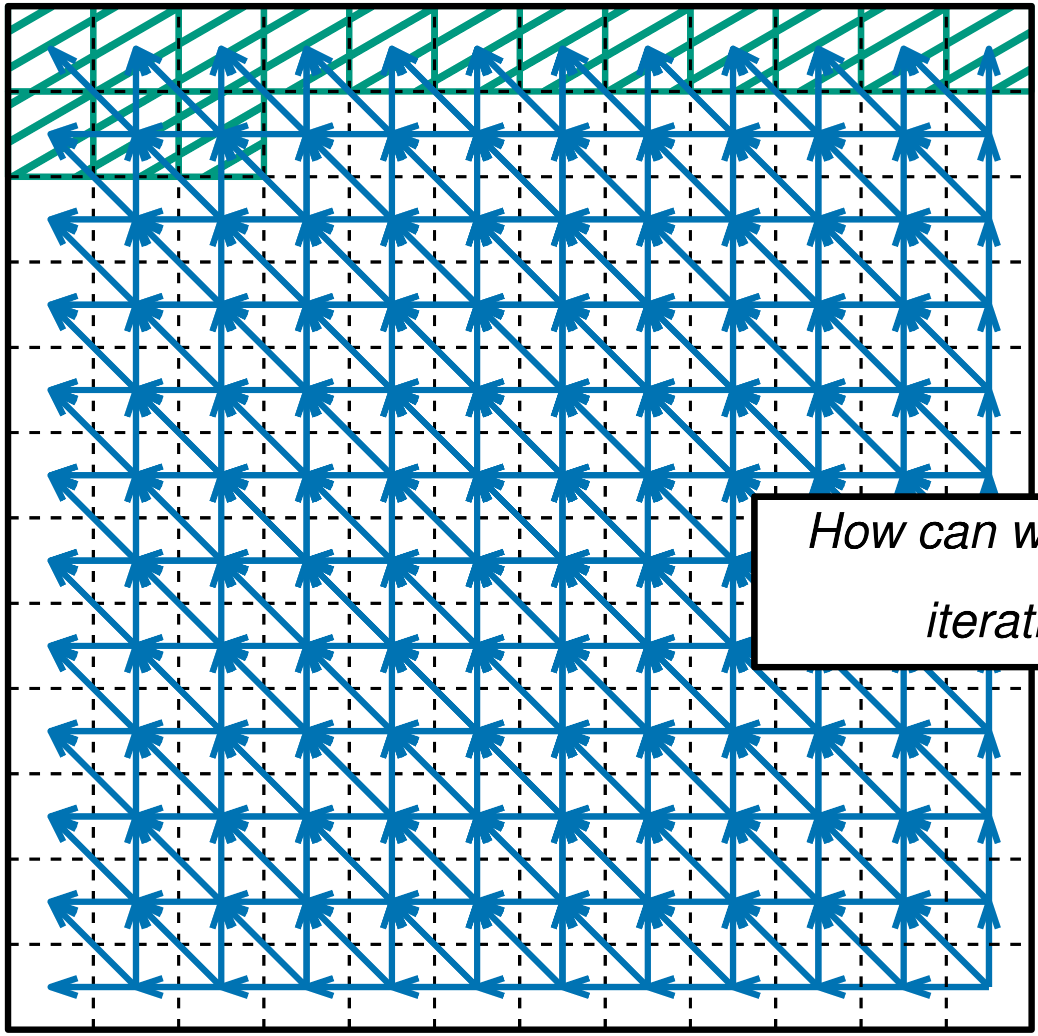
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MEMLES}(x-1,y-1), \text{MEMLES}(x-1,y), \text{MEMLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\text{LES}:$



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}[n,n]$?

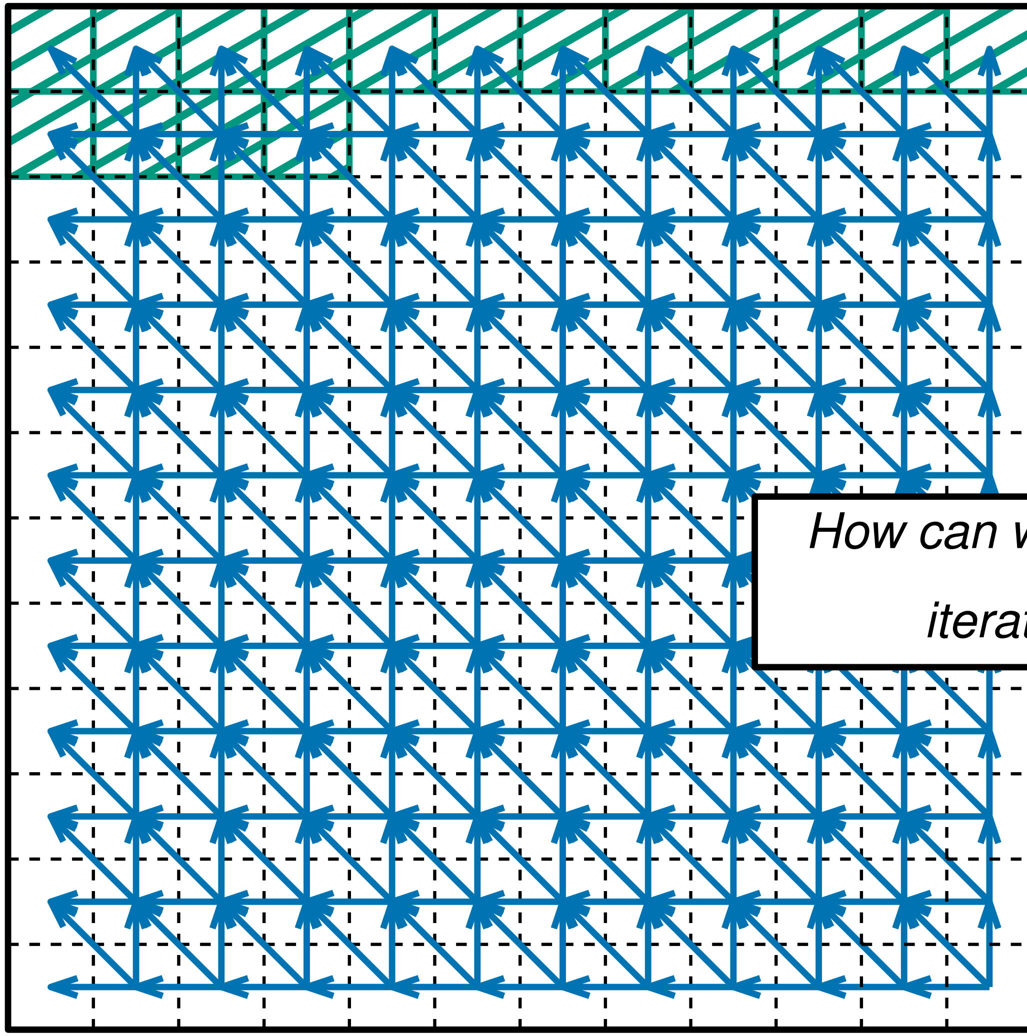
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

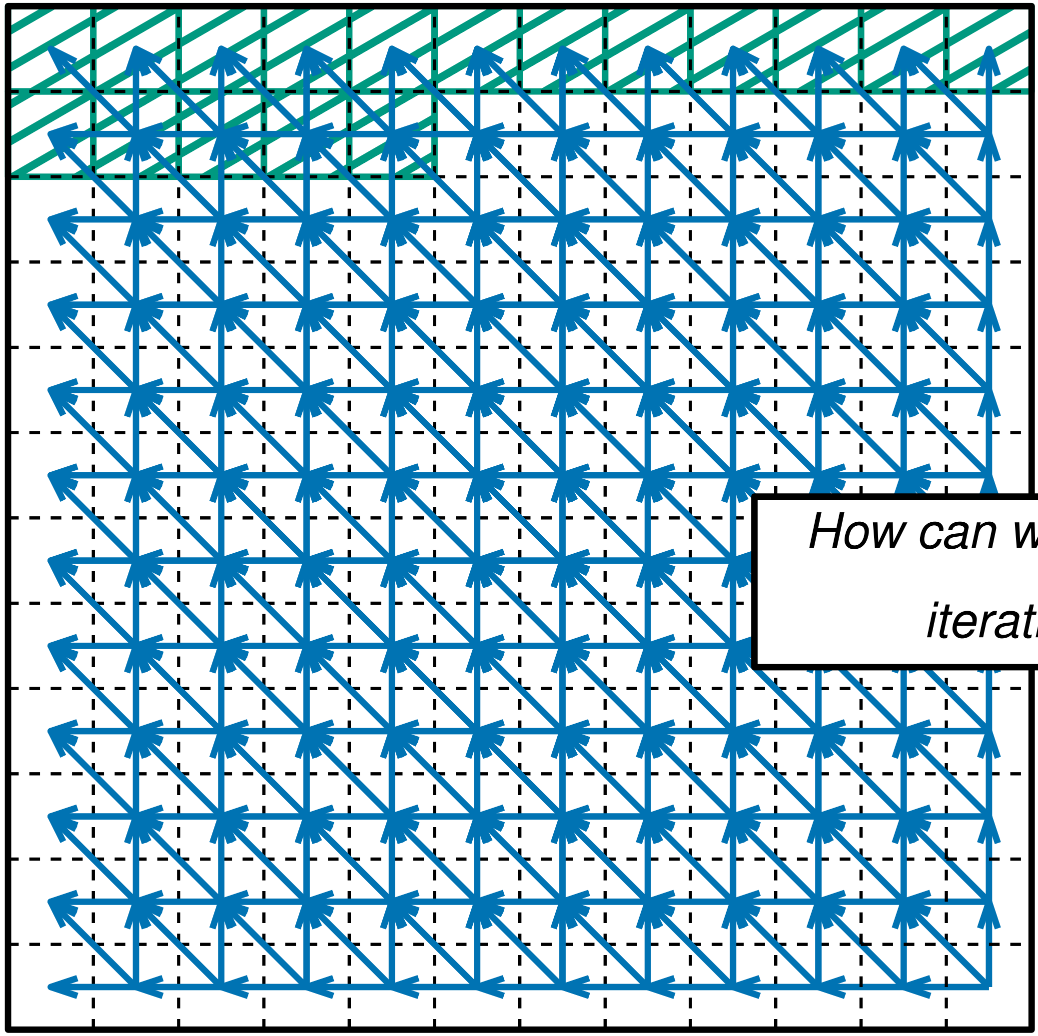
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

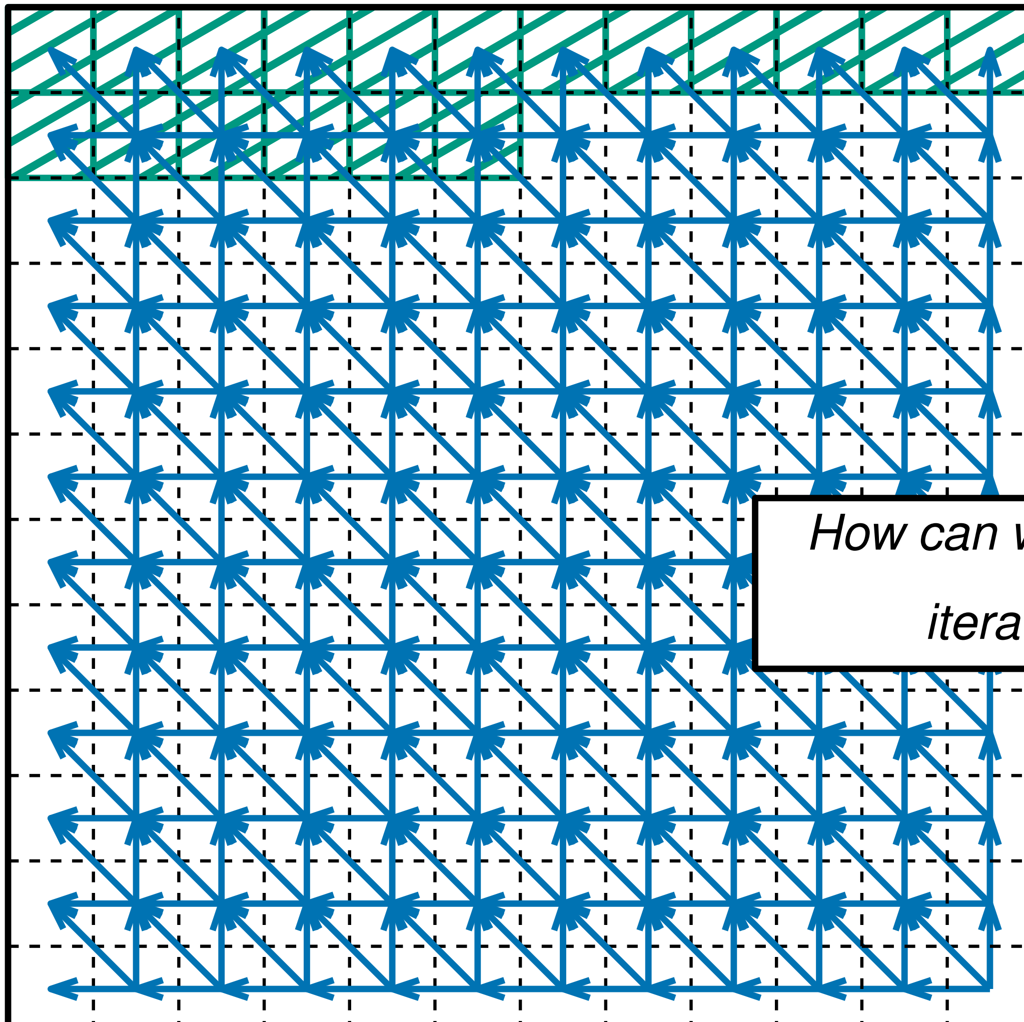
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

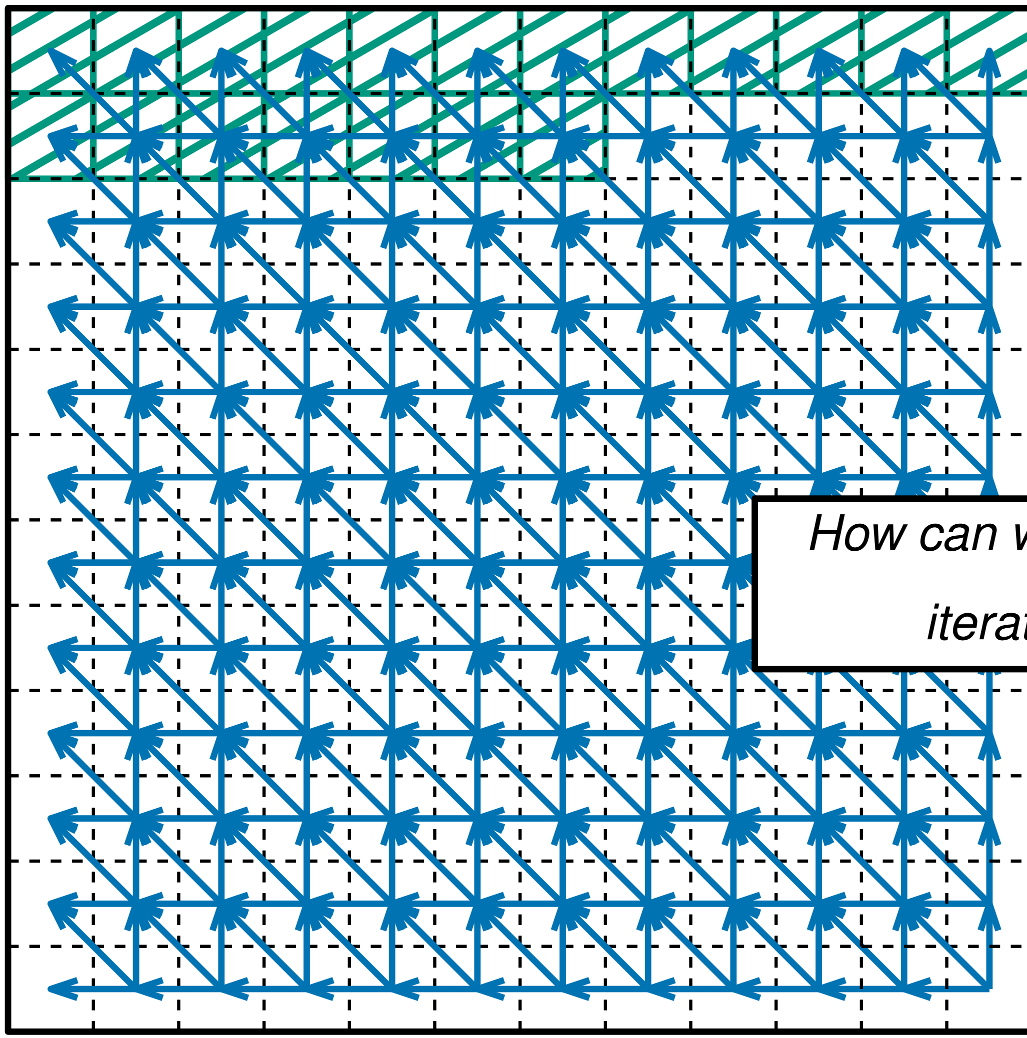
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

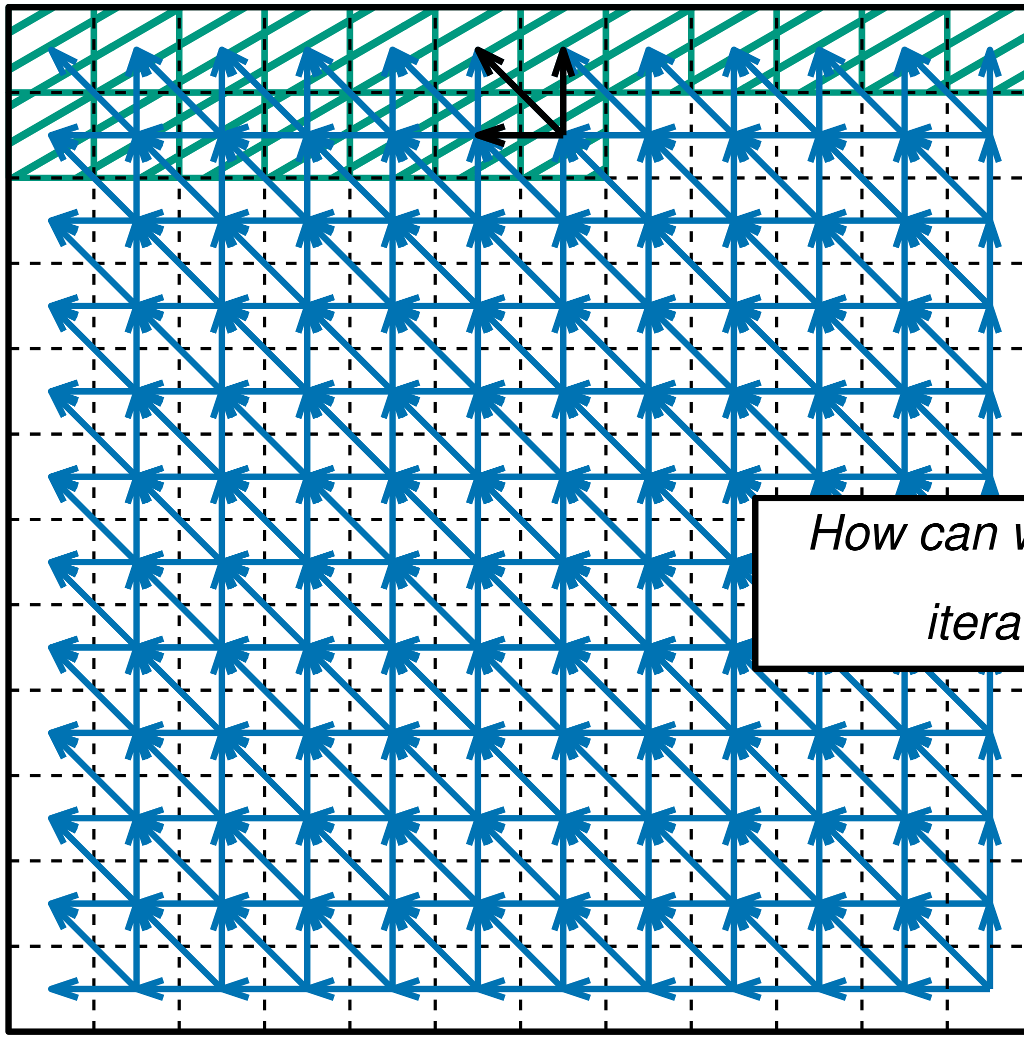
The dependency graph

$$\mathrm{LES}\,[x, y] = \min\left(\textsc{MemLES}(x-1, y-1), \textsc{MemLES}(x-1, y), \textsc{MemLES}(x, y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n, n]$?

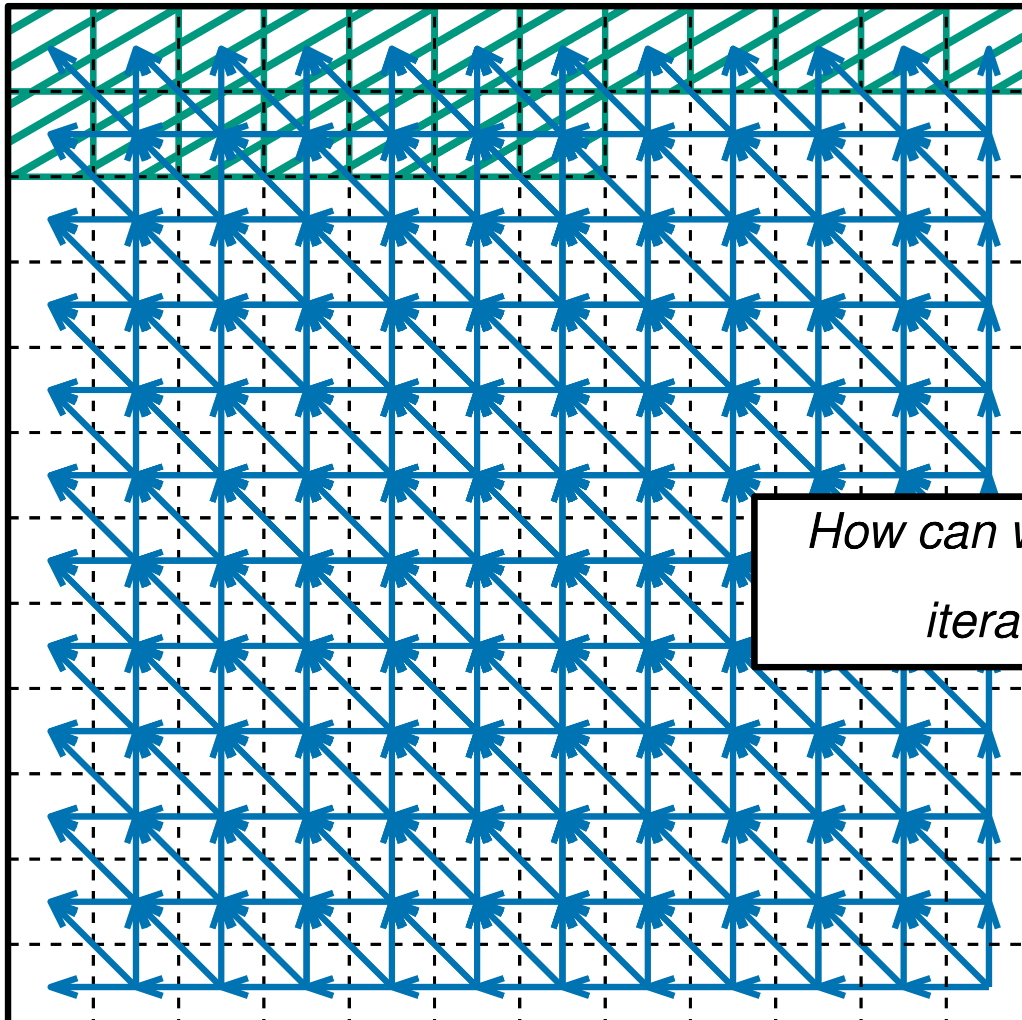
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1),\textsc{MemLES}(x-1,y),\textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n, n]$?

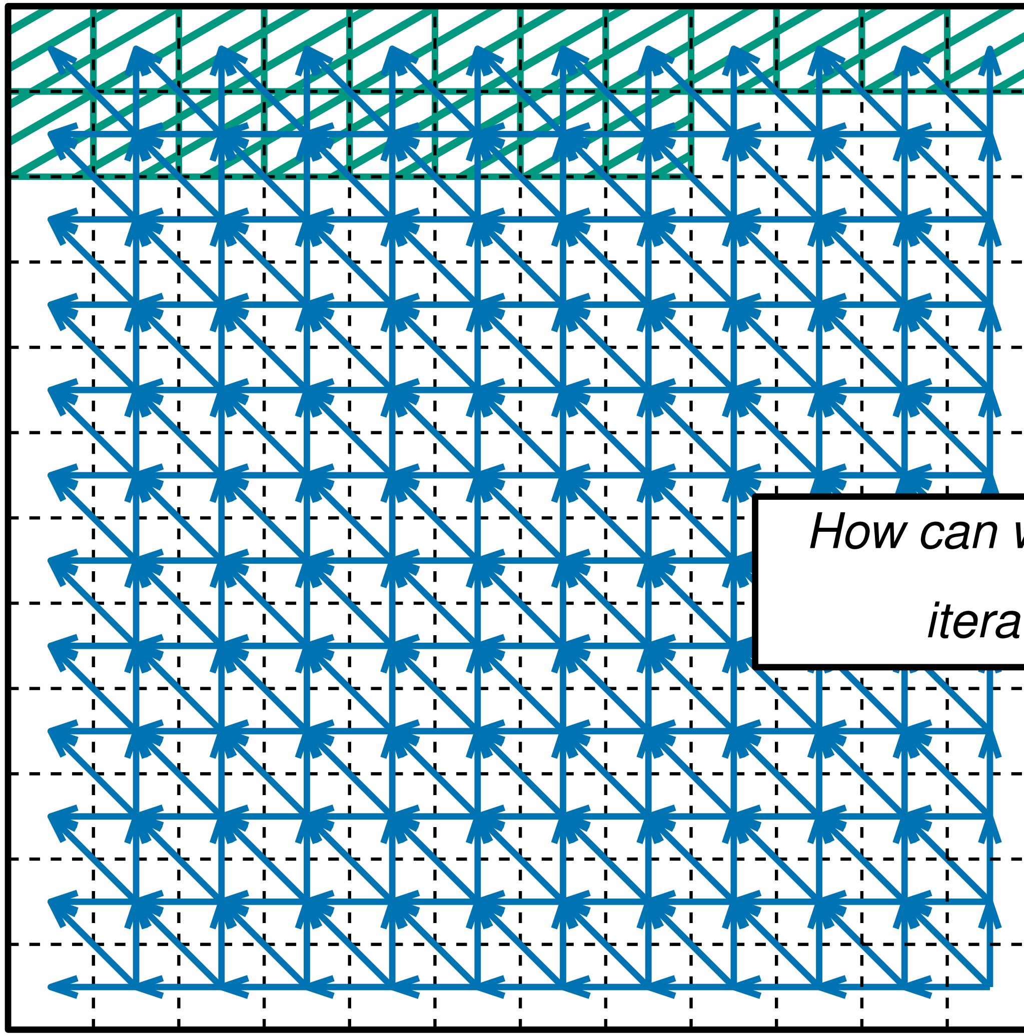
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES:



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

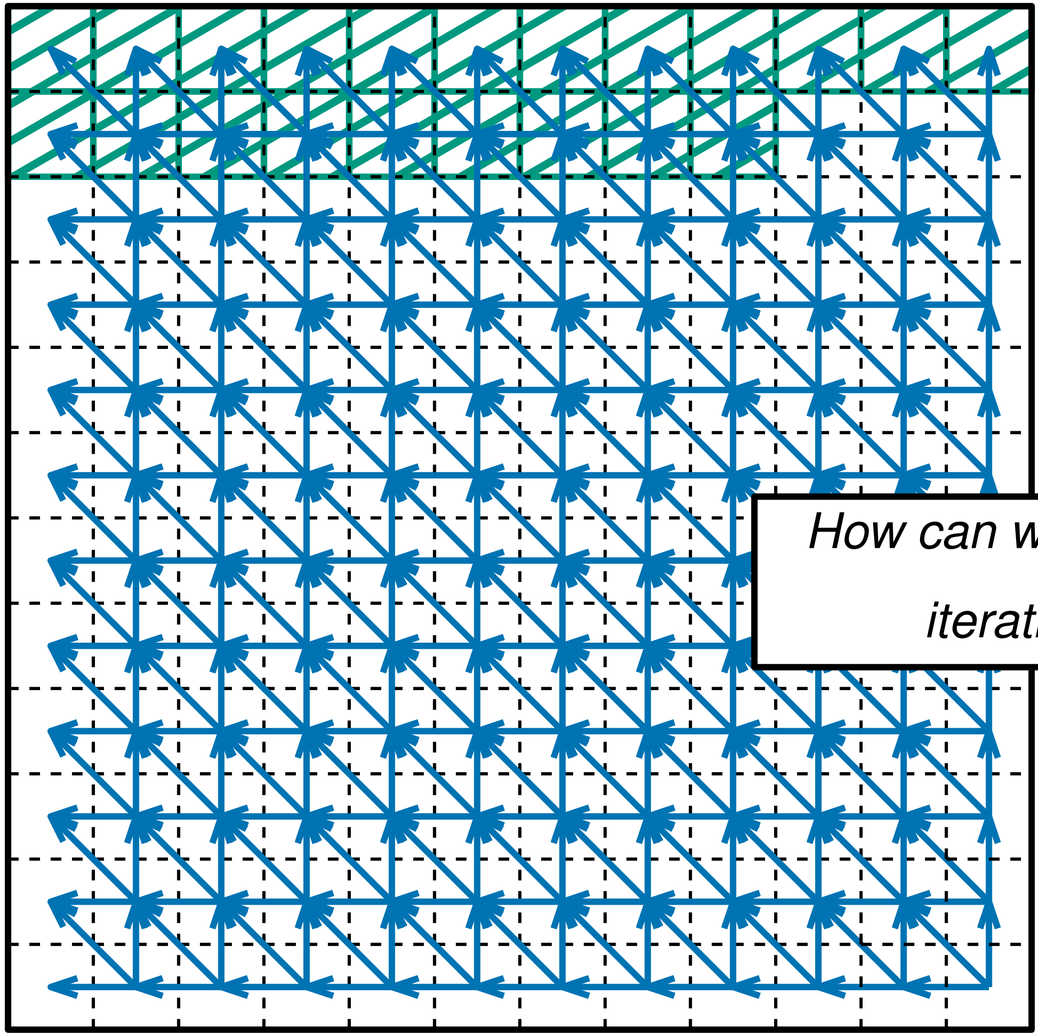
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

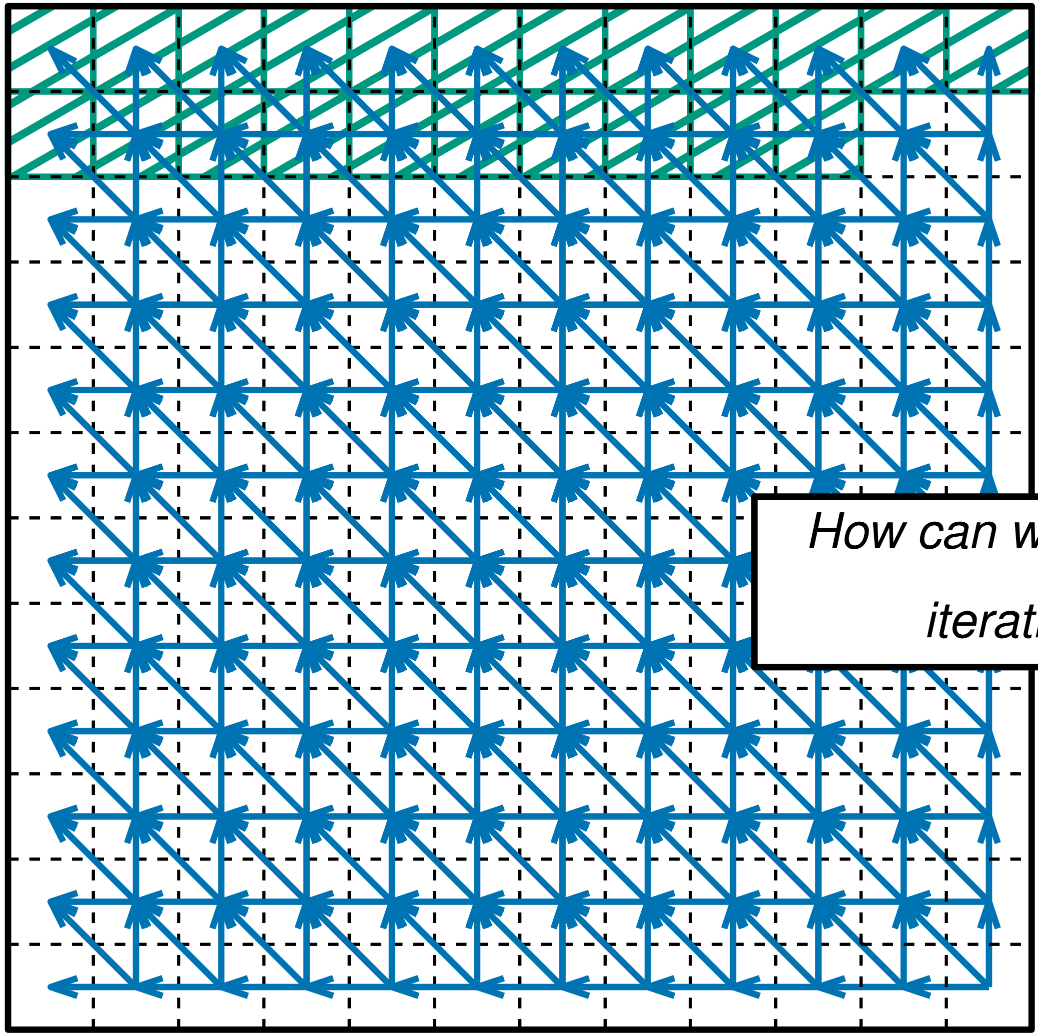
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n, n]$?

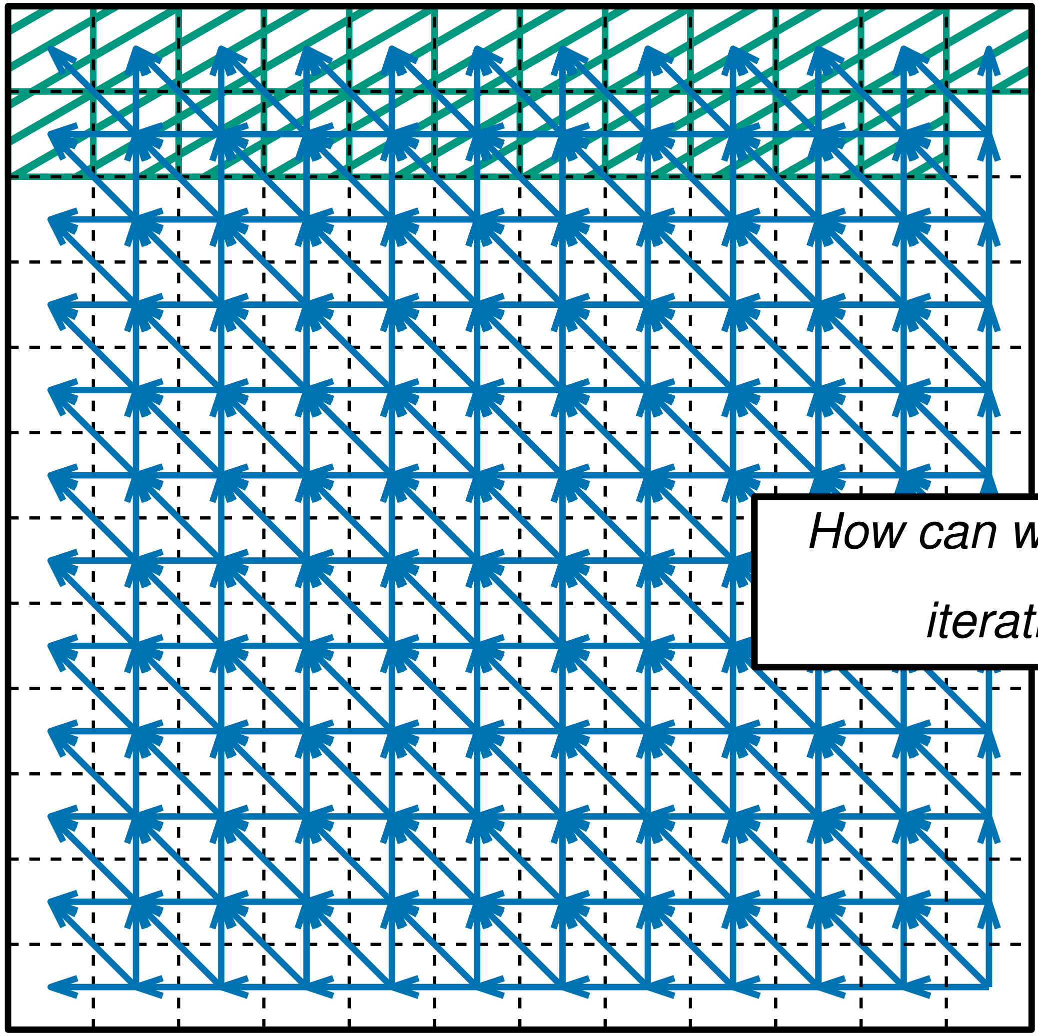
The dependency graph

$$\text{LES}[x,y] = \min\left(\text{MEMLES}(x-1,y-1), \text{MEMLES}(x-1,y), \text{MEMLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\text{LES}[n,n]$?

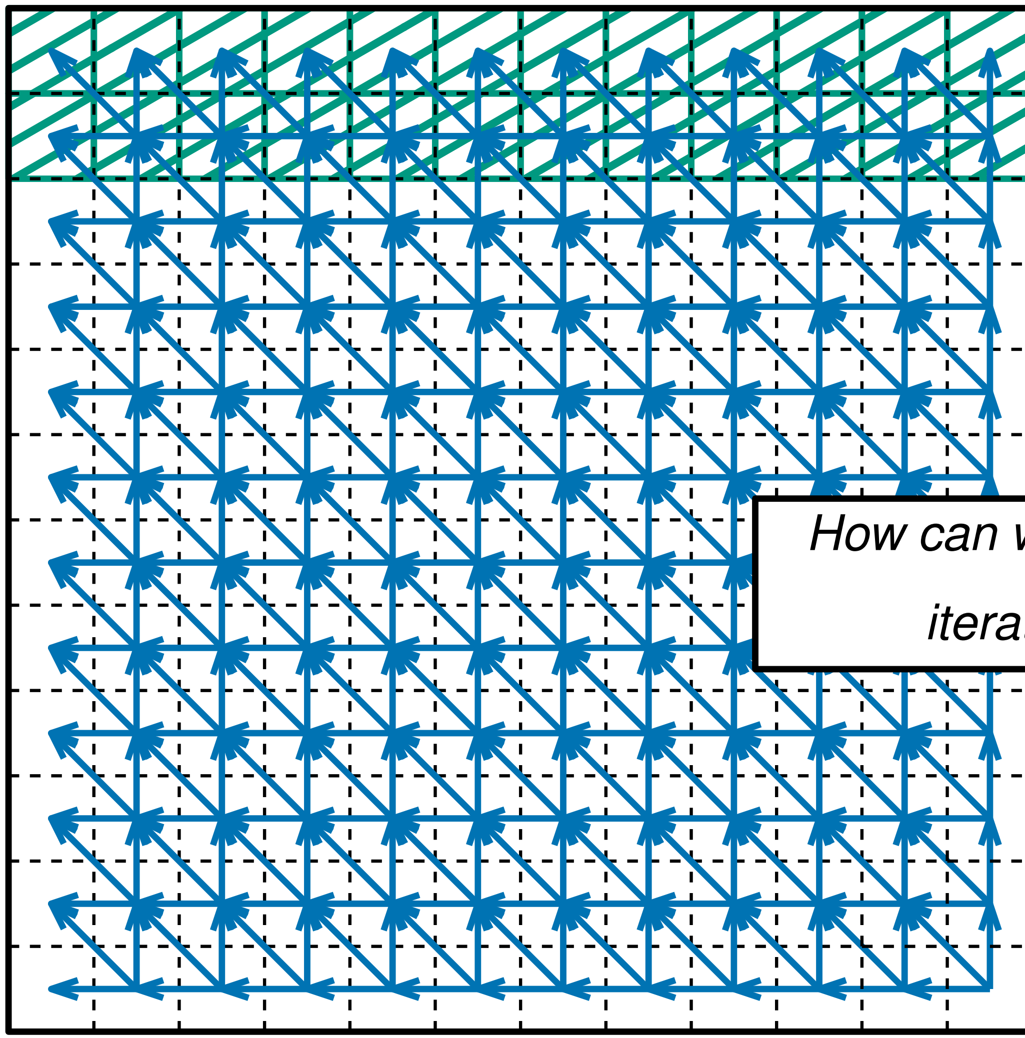
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right) + 1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

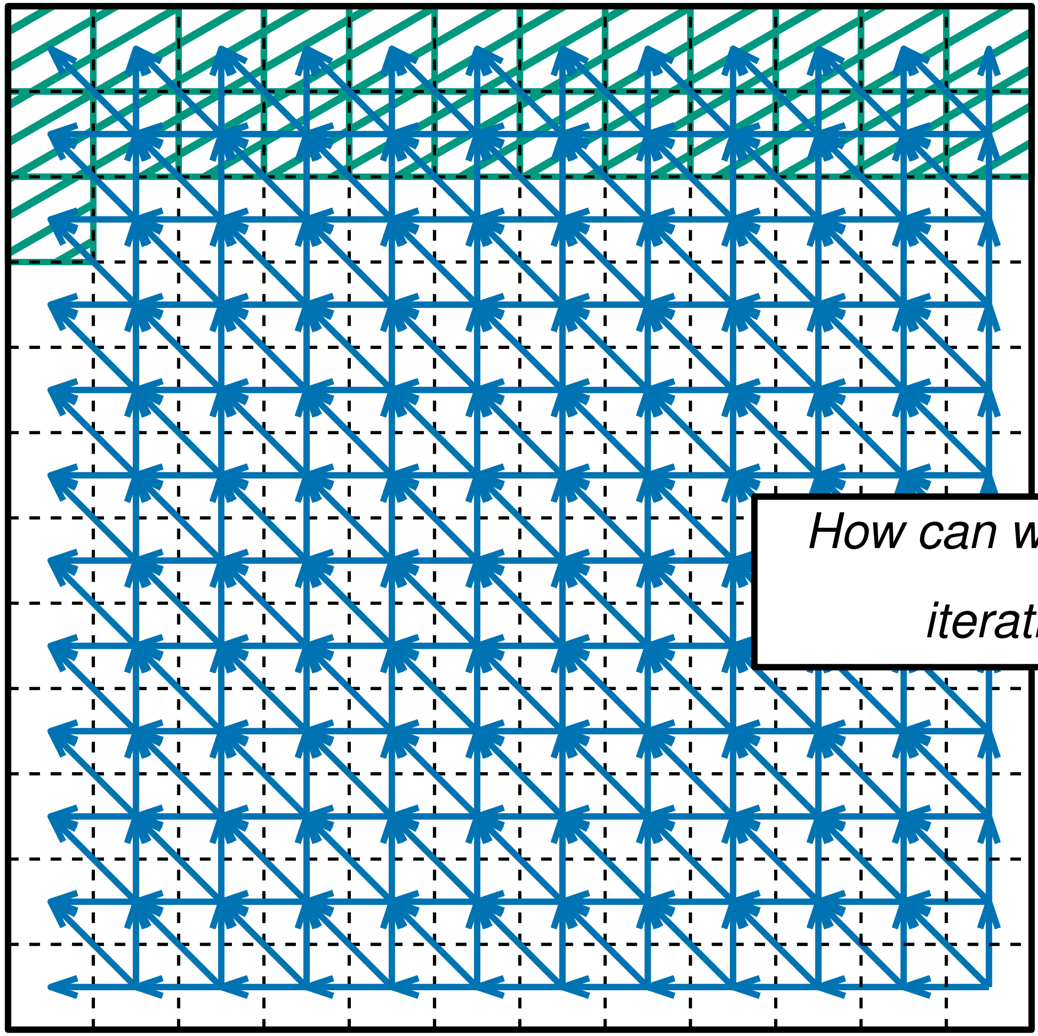
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x, y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n, n]$?

University of BRISTOL

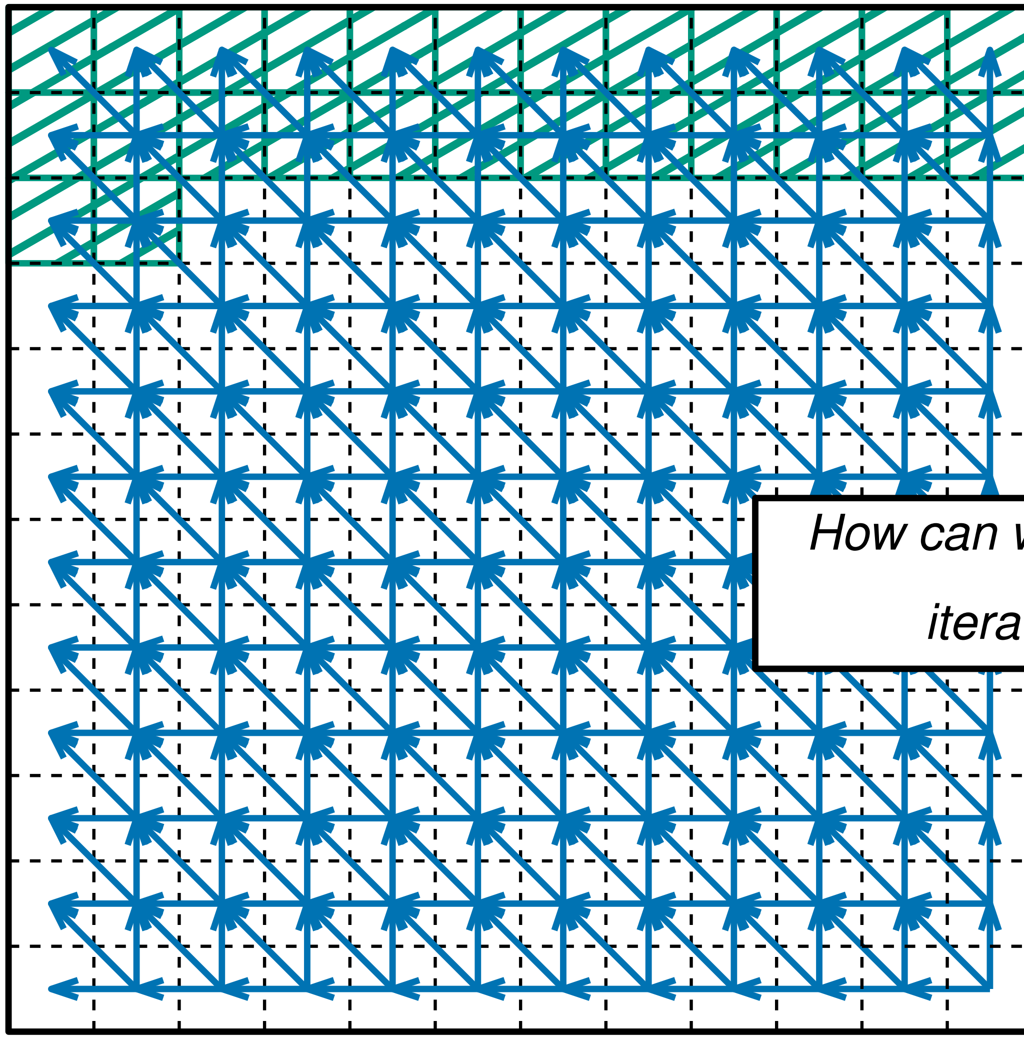
The dependency graph

$$\mathrm{LES}[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

LES :



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}[n,n]$?

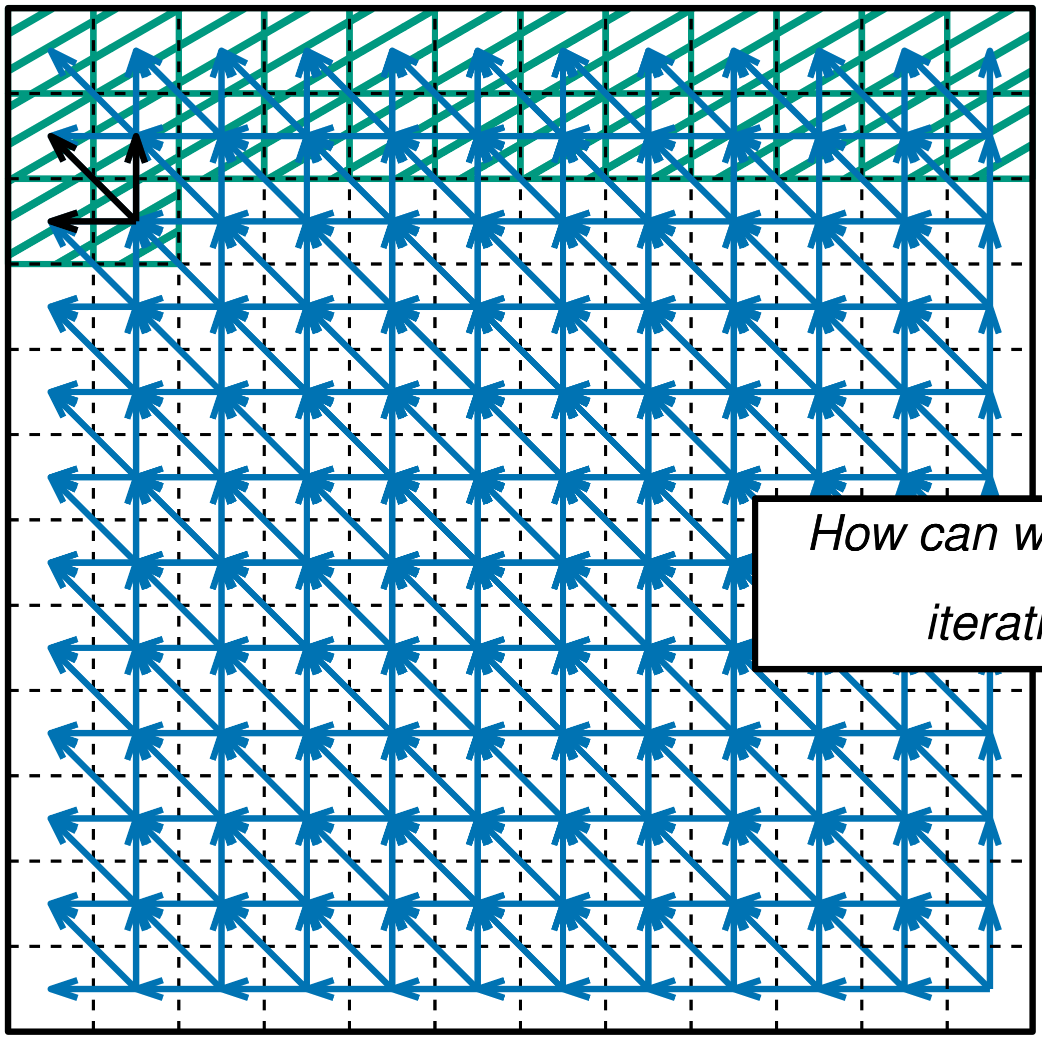
The dependency graph

$$\mathrm{LES}\,[x,y] = \min\left(\textsc{MemLES}(x-1,y-1), \textsc{MemLES}(x-1,y), \textsc{MemLES}(x,y-1)\right)+1$$

(for $x, y > 1$ and (x,y) non empty)

The 2D array

$\mathrm{LES}:$



How can we use this to get an iterative algorithm?

Fill in the array from the top-left!

What information do we need to compute $\mathrm{LES}\,[n,n]$?

4. Derive an iterative algorithm

```
ItLES(n)
```

```
For y = 1 to n
  For x = 1 to n
    If pixel (x, y) is not empty
      LES[x, y] = 0
    Else If (x = 1) or (y = 1)
      Return 1
    Else
      LES[x, y] = min(LES[x − 1, y − 1], LES[x − 1, y], LES[x, y − 1]) + 1
```

This iterative version of the algorithm

runs in $O(n^2)$ time

and avoids making any recursive calls

4. Derive an iterative algorithm

```
ItLES(n)
```

```
For y = 1 to n
  For x = 1 to n
    If pixel (x, y) is not empty
      LES[x, y] = 0
    Else If (x = 1) or (y = 1)
      Return 1
    Else
      LES[x, y] = min (LES[x − 1, y − 1], LES[x − 1, y], LES[x, y − 1]) + 1
```

This iterative version of the algorithm

runs in $O(n^2)$ time

and avoids making any recursive calls

Maximum of $\mathrm{LES}[x, y]$ over all x and y

gives the size of the largest empty square in the whole image

this also takes $O(n^2)$ time

~~Introduction~~ Summary

Dynamic programming is a technique for finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea:

1. Find a recursive formula for the problem
   - in terms of answers to subproblems.

   (typically this is the hard bit)

2. Write down a naive recursive algorithm

   (typically this algorithm will take exponential time)

3. Speed it up by storing the solutions to subproblems (memoization)

   (to avoid recomputing the same thing over and over)

4. Derive an iterative algorithm by solving the subproblems in a good order

   (iterative algorithms are often better in practice, easier to analyse and prettier)

in other words. . .

Dynamic programming is *recursion without repetition*

End of part one

Part two

Weighted Interval Scheduling

Dynamic programming is a technique for finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea:

1. Find a recursive formula for the problem
   - in terms of answers to subproblems.

   (typically this is the hard bit)

2. Write down a naive recursive algorithm

   (typically this algorithm will take exponential time)

3. Speed it up by storing the solutions to subproblems (memoization)

   (to avoid recomputing the same thing over and over)

4. Derive an iterative algorithm by solving the subproblems in a good order

   (iterative algorithms are often better in practice, easier to analyse and prettier)

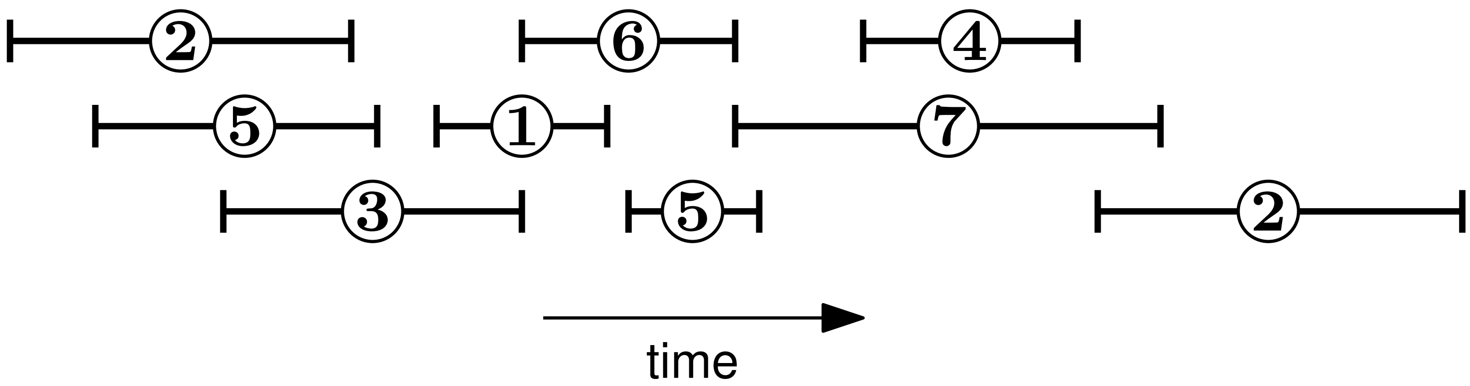
in other words. . .

Dynamic programming is *recursion without repetition*

Weighted Interval Scheduling

Problem Given an n weighted intervals,

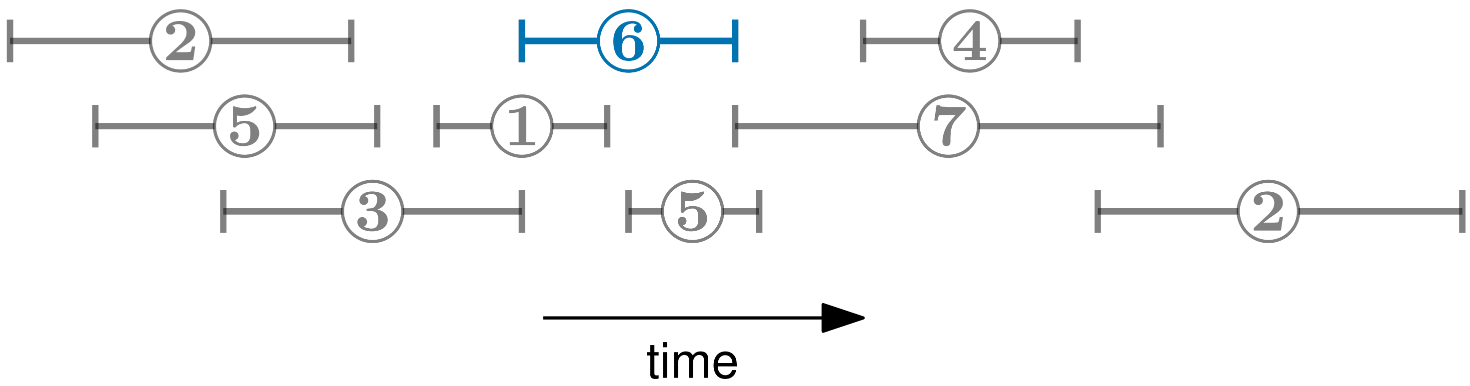
find the *schedule* with largest total weight



Weighted Interval Scheduling

Problem Given an n weighted intervals,

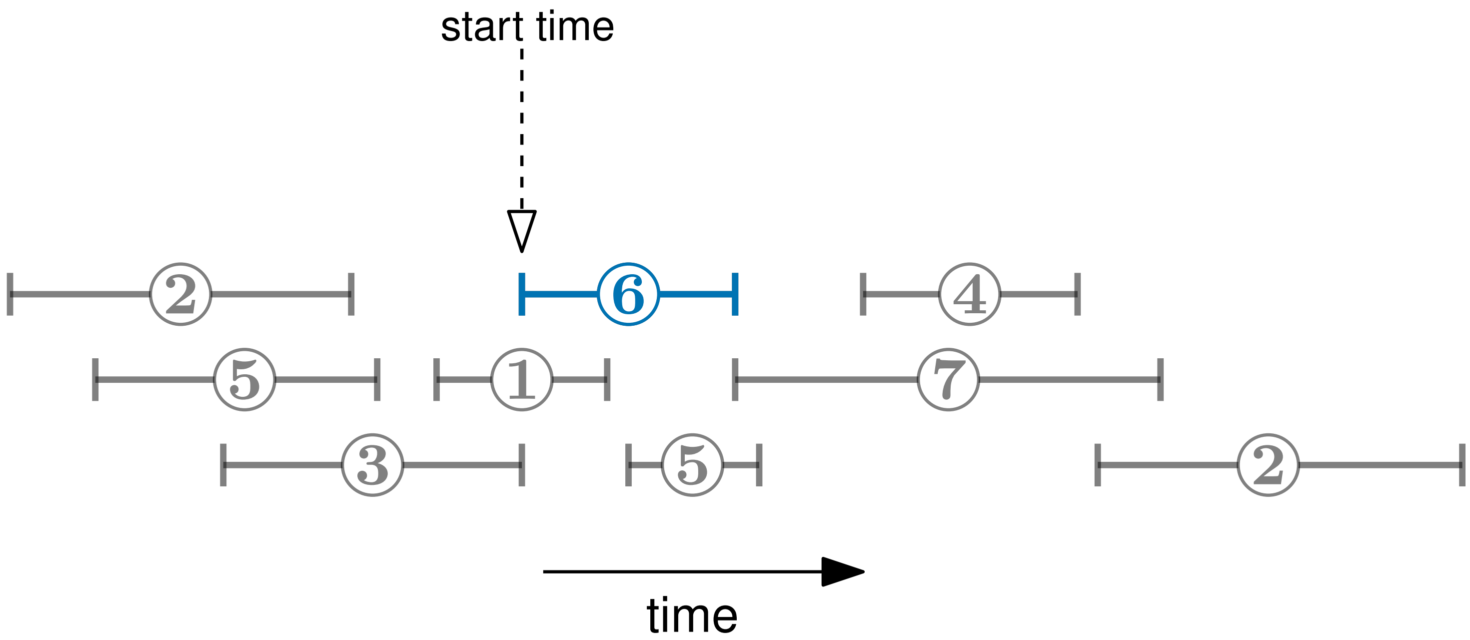
find the *schedule* with largest total weight



Weighted Interval Scheduling

Problem Given an n weighted intervals,

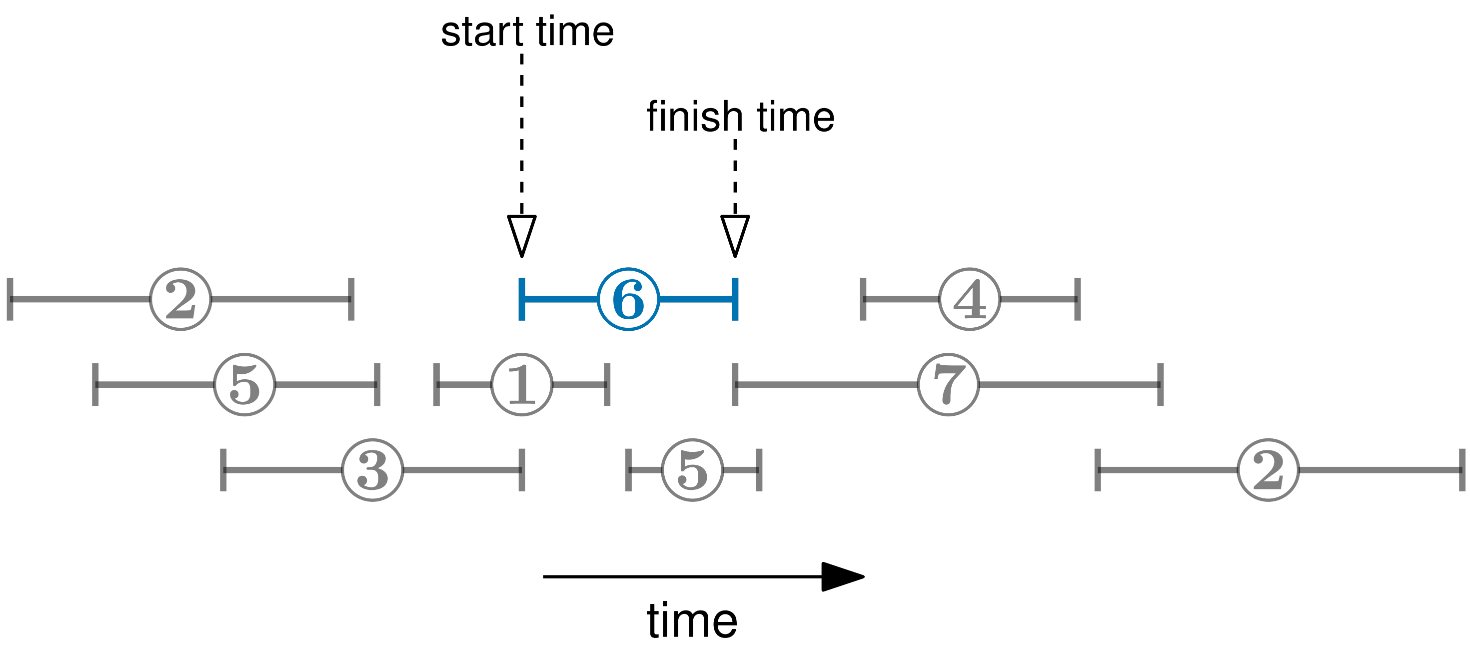
find the *schedule* with largest total weight



Weighted Interval Scheduling

Problem Given an n weighted intervals,

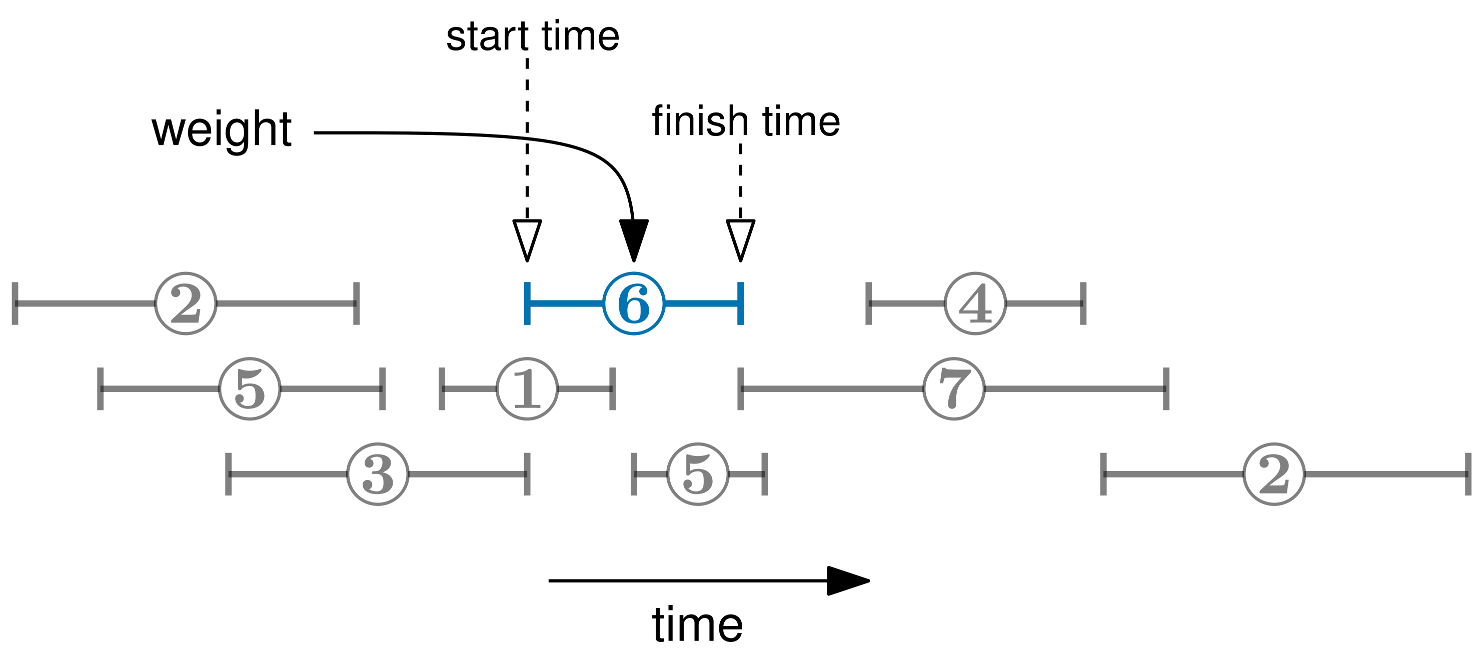
find the *schedule* with largest total weight



Weighted Interval Scheduling

Problem Given an n weighted intervals,

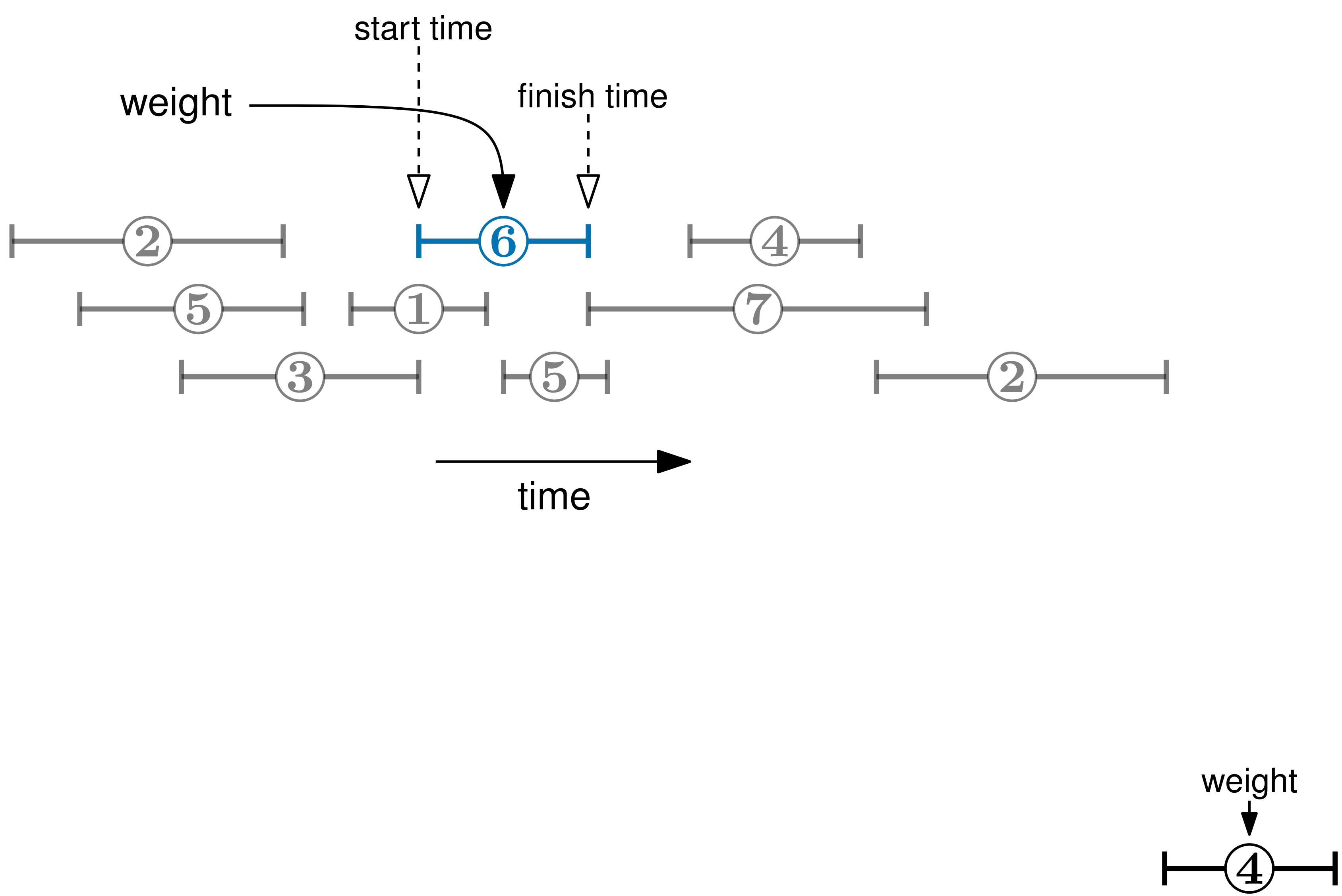
find the *schedule* with largest total weight



Weighted Interval Scheduling

Problem Given an n weighted intervals,

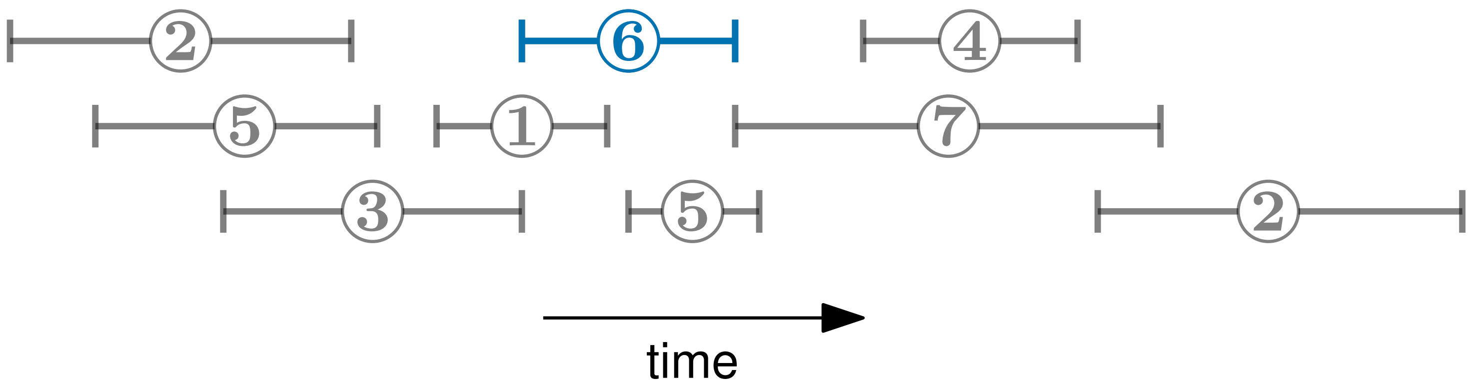
find the *schedule* with largest total weight

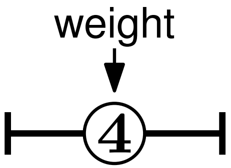


Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

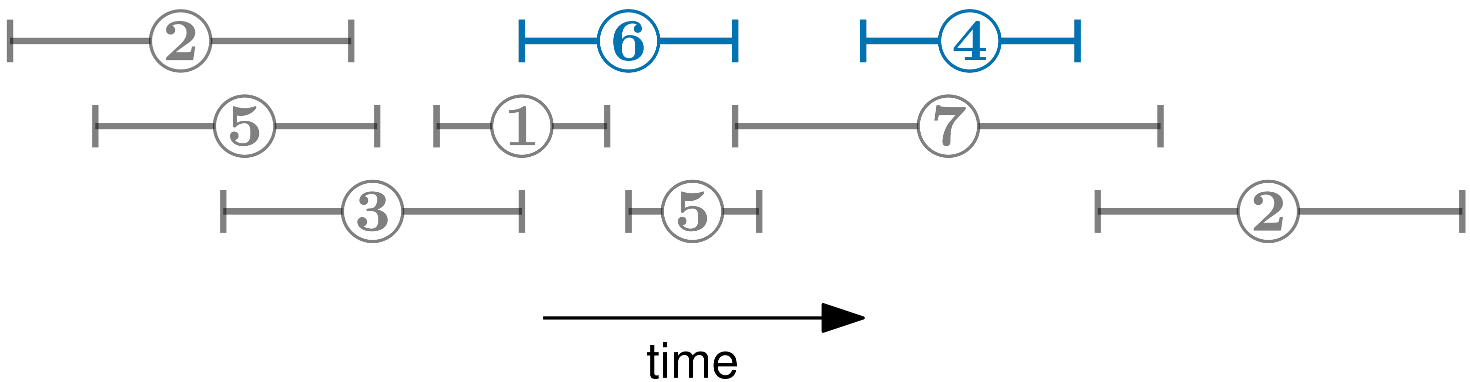


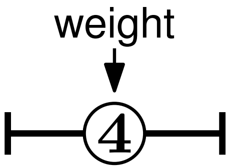


Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

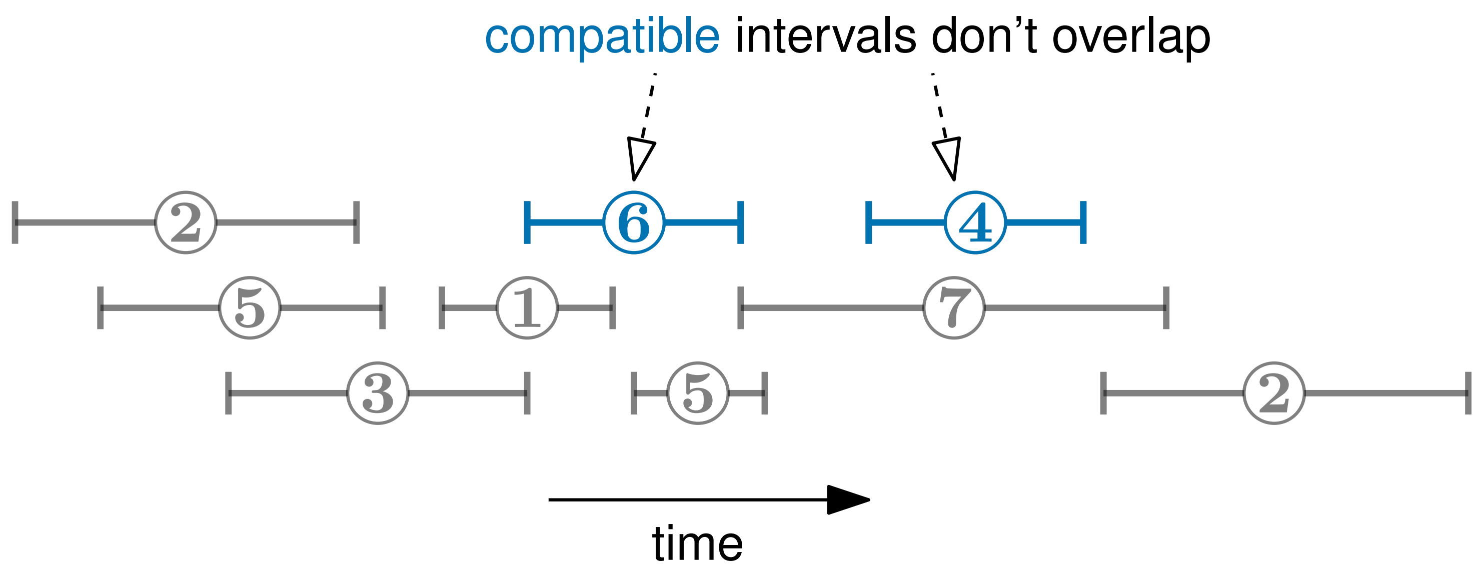


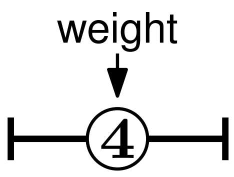


Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

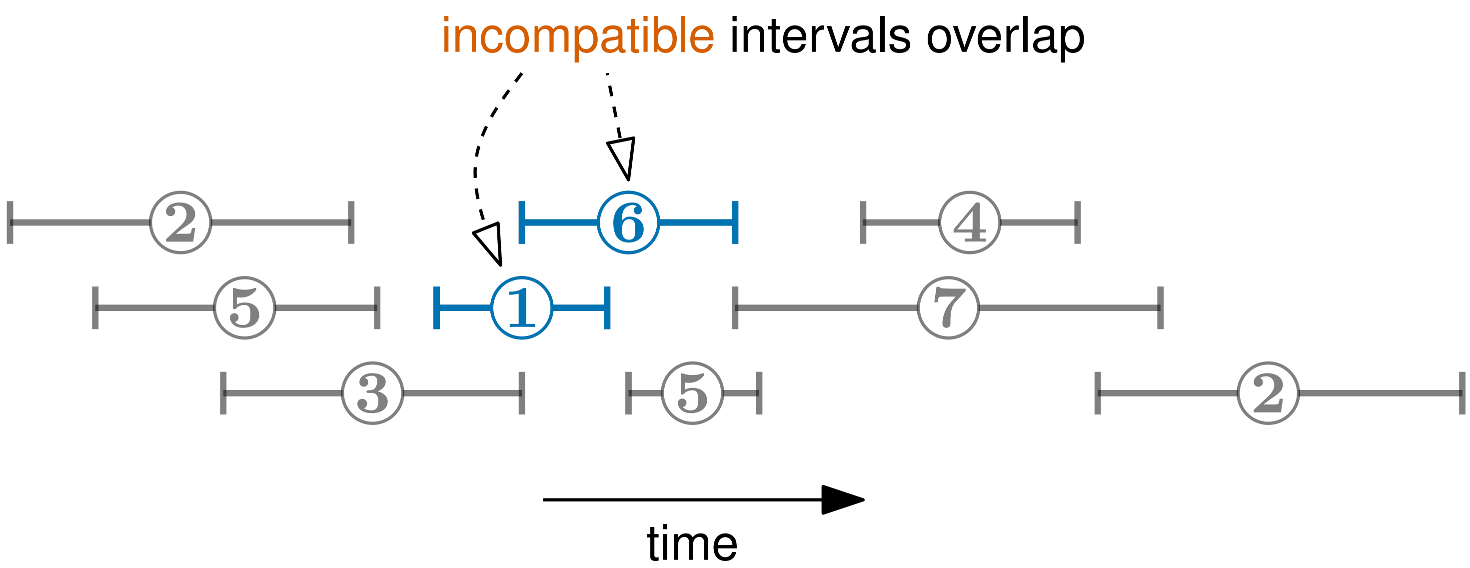


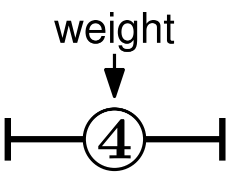


Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

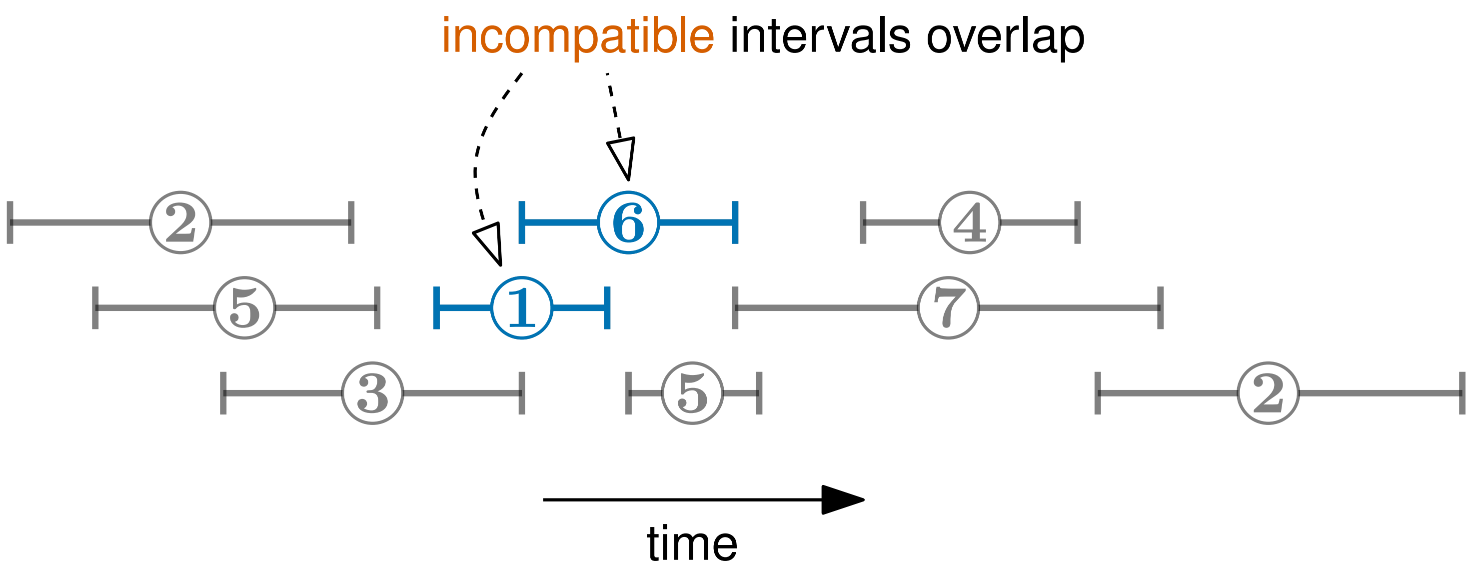




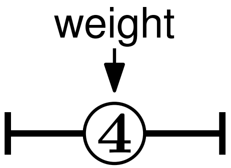
Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight



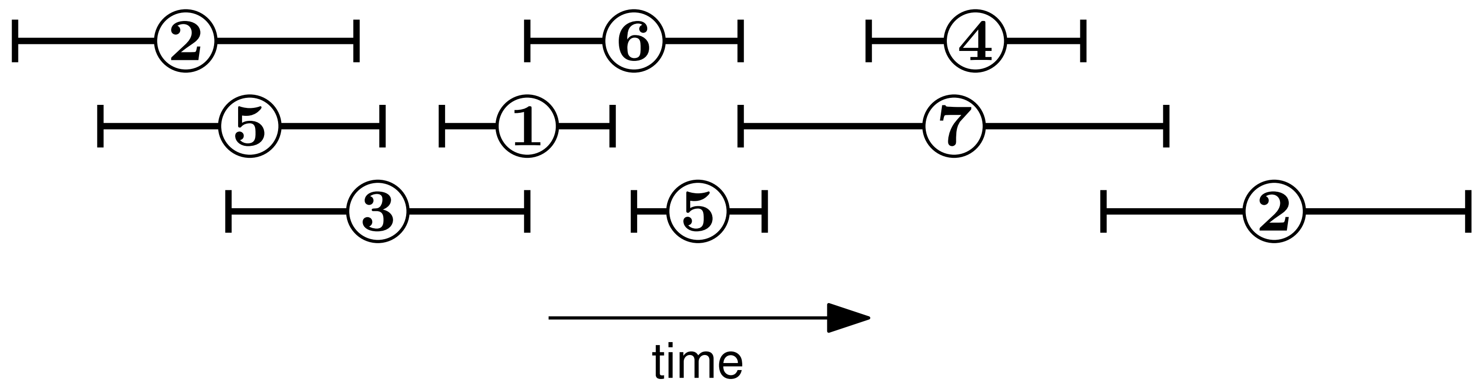
Two intervals are compatible if they don't overlap



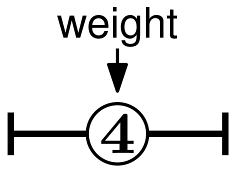
Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight



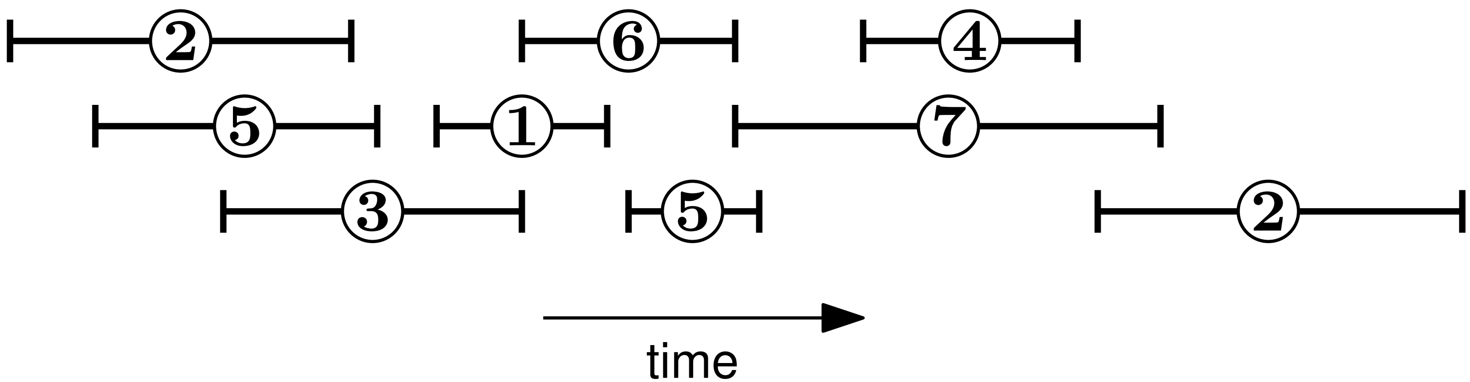
Two intervals are compatible if they don't overlap



Weighted Interval Scheduling

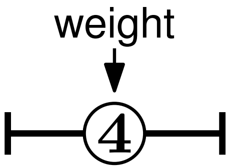
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

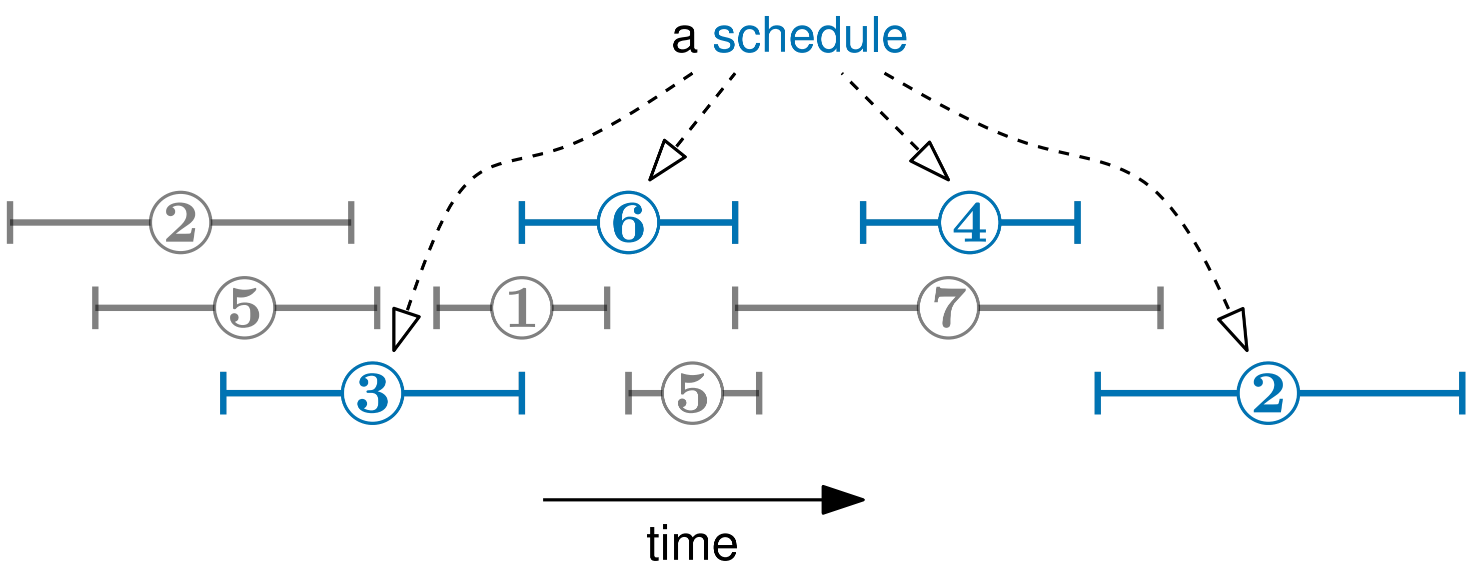
A schedule is a set of compatible intervals



Weighted Interval Scheduling

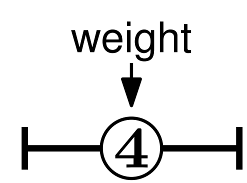
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

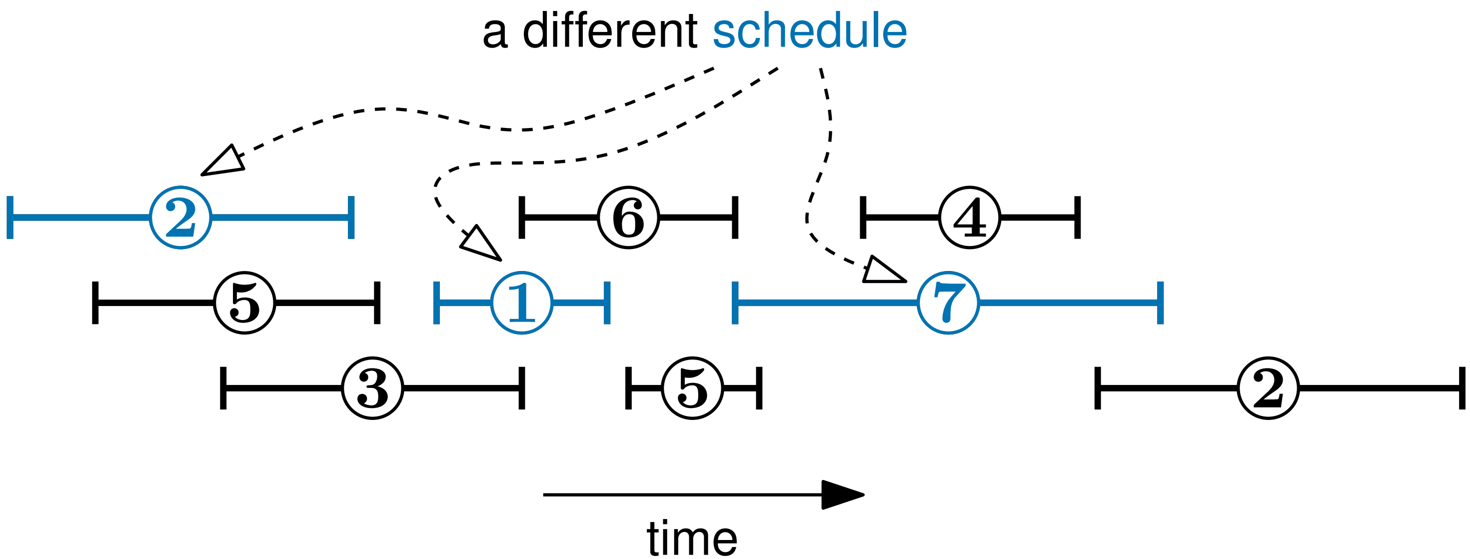
A schedule is a set of compatible intervals



Weighted Interval Scheduling

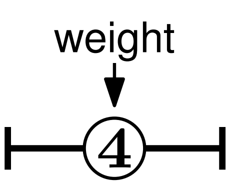
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

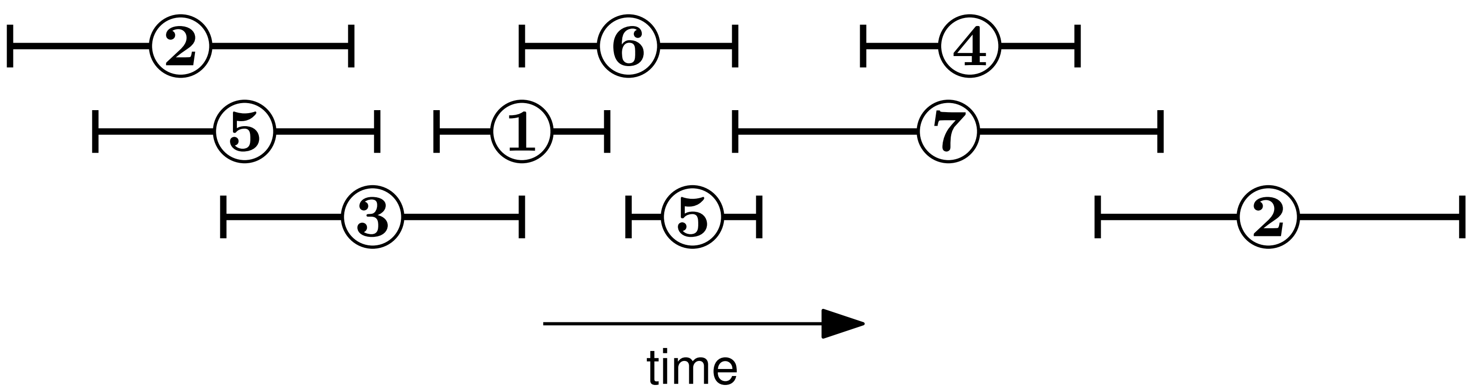
A schedule is a set of compatible intervals



Weighted Interval Scheduling

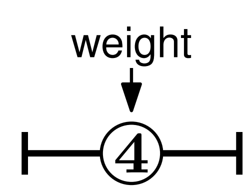
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



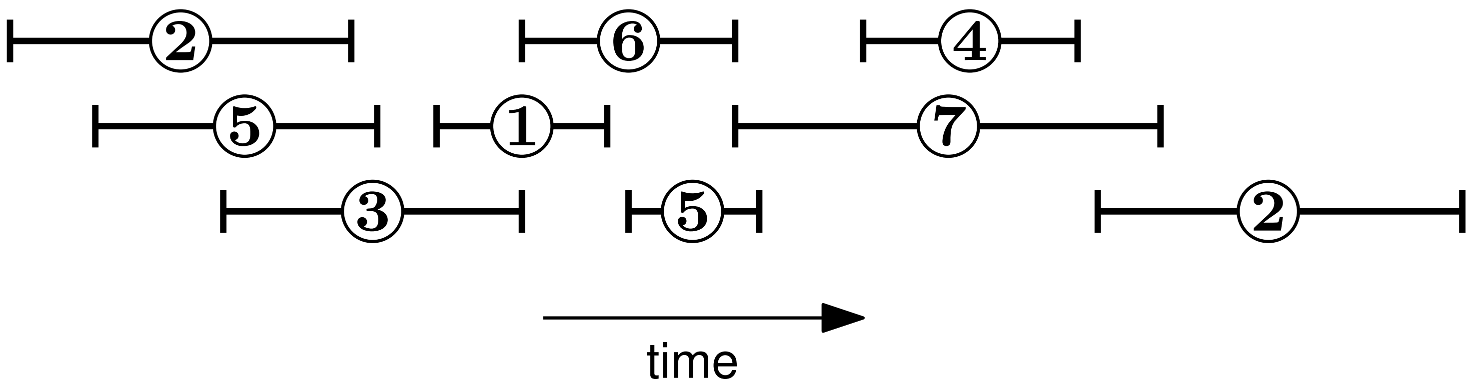
Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals



Weighted Interval Scheduling

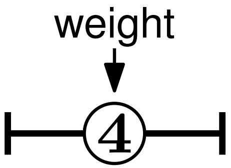
Problem Given an n weighted intervals,
find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

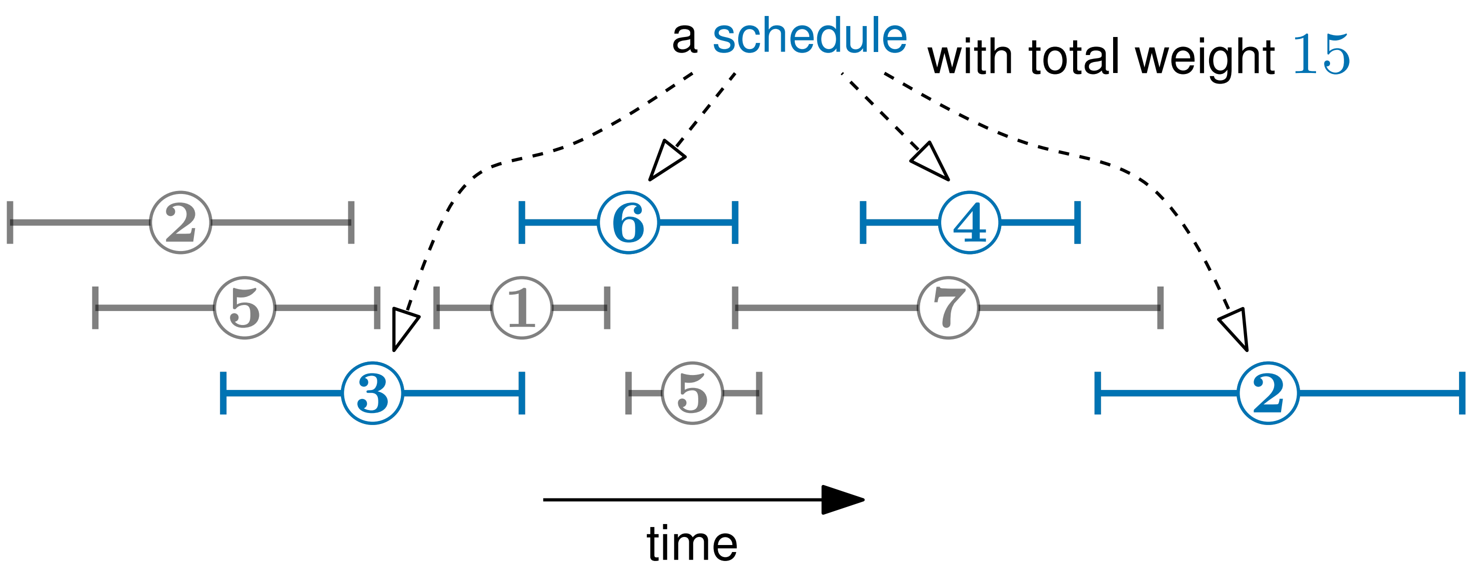
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

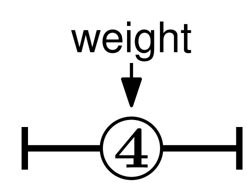
find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

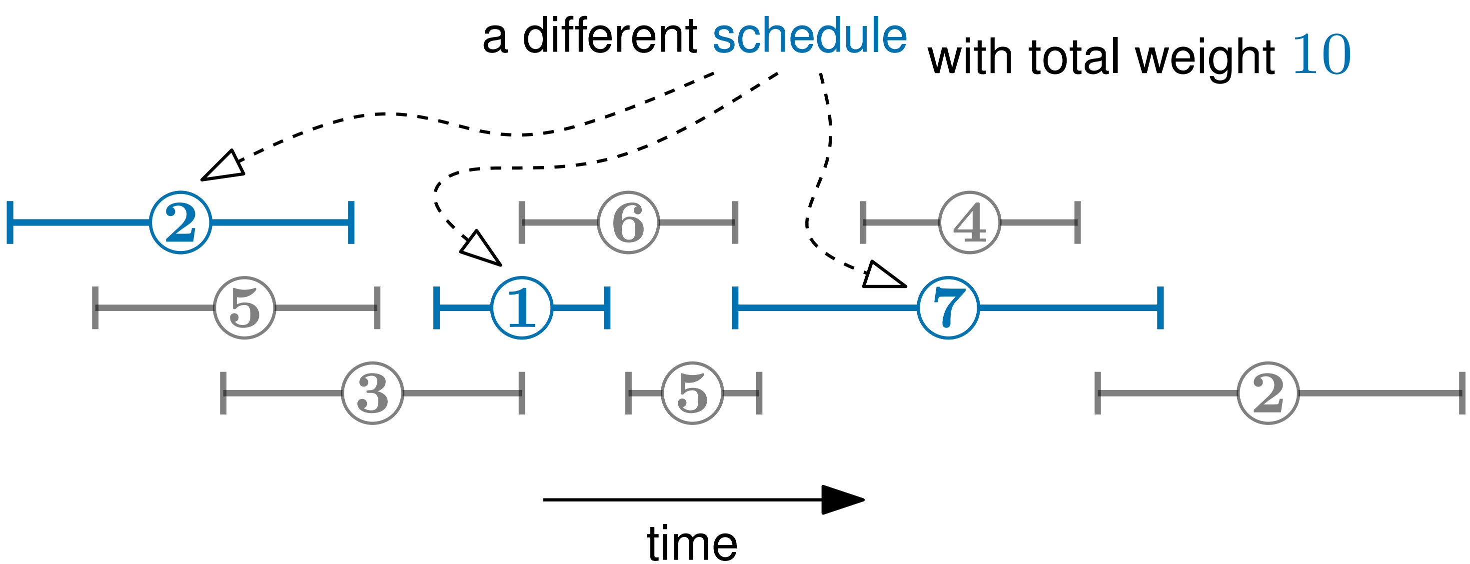
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

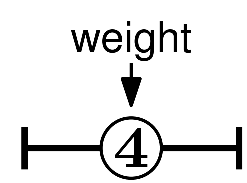
find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

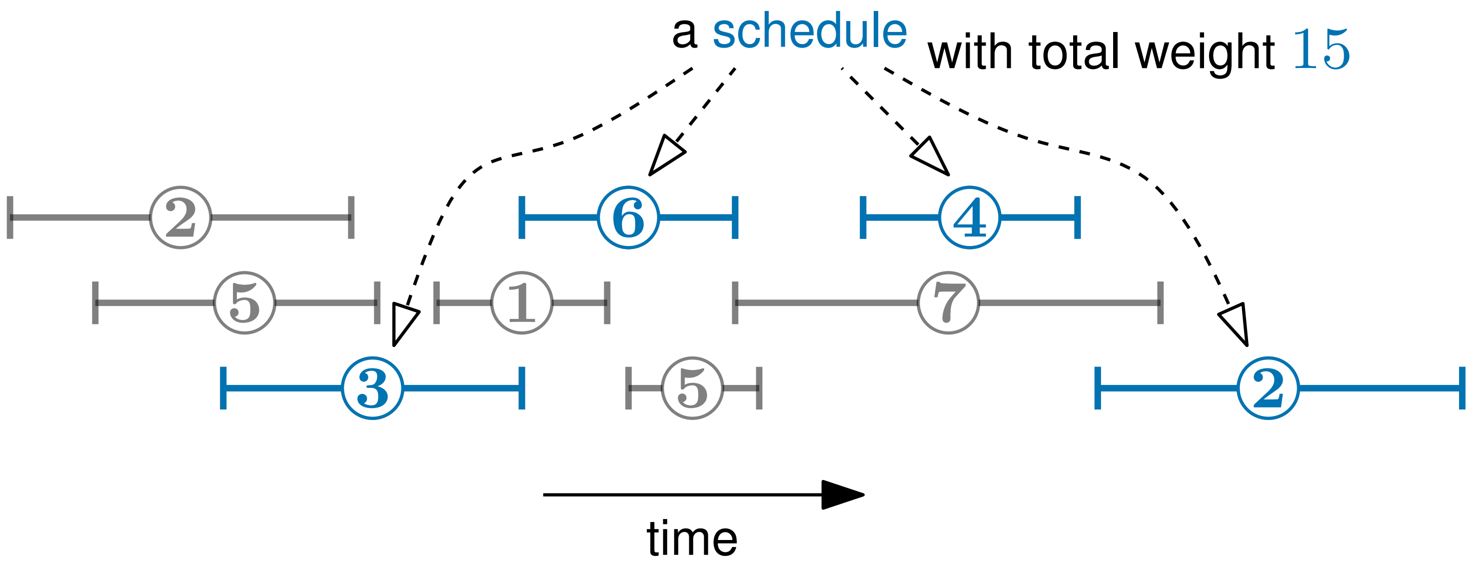
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

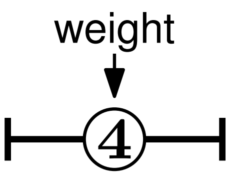
find the *schedule* with largest total weight



Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

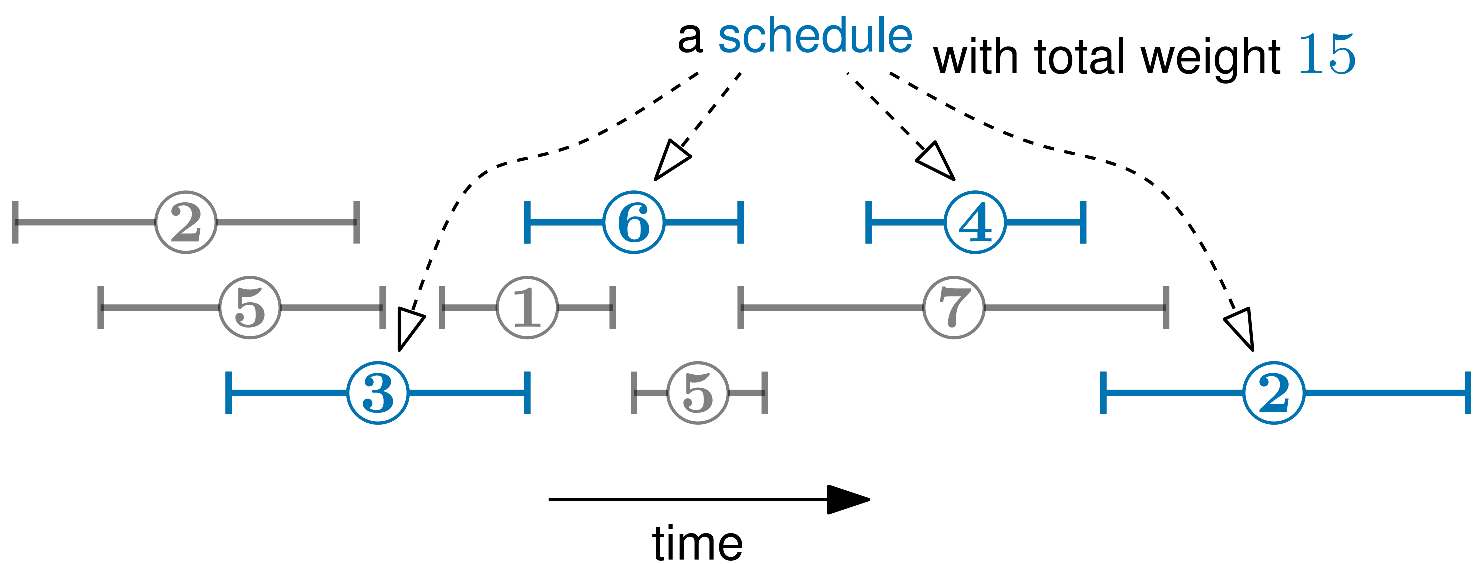
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

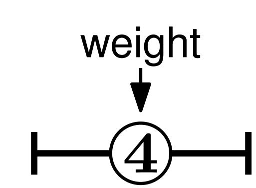


is this the best possible?

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

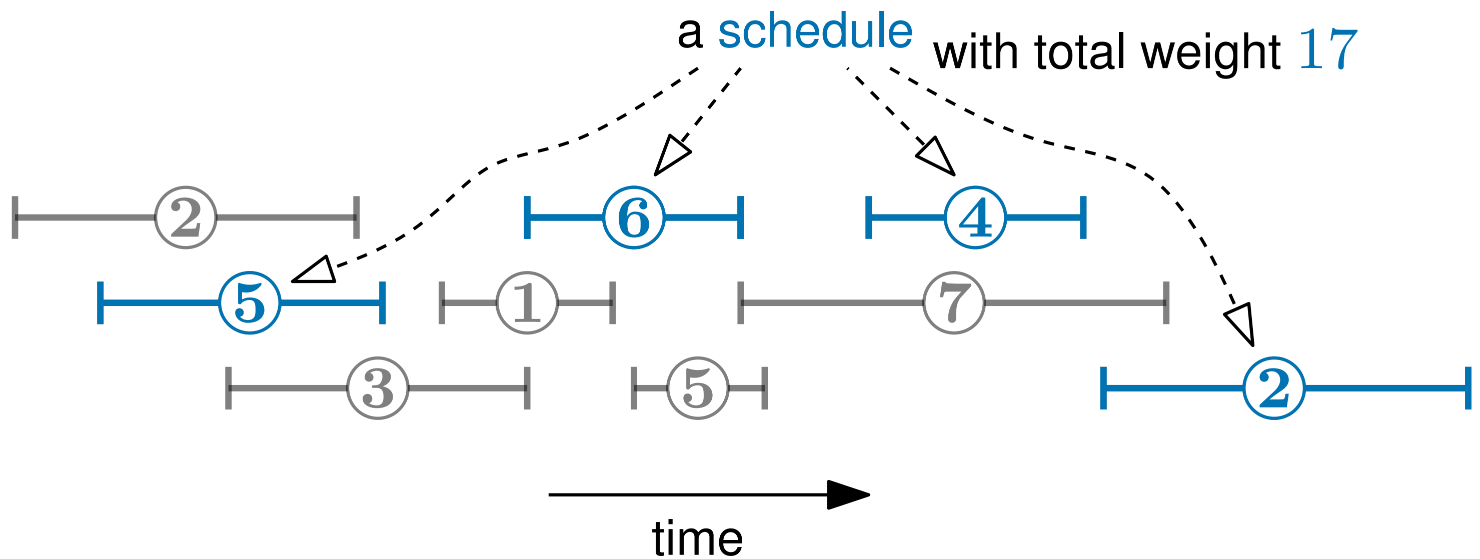
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

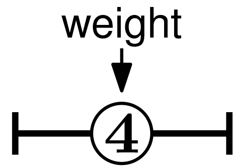


is this the best possible?

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

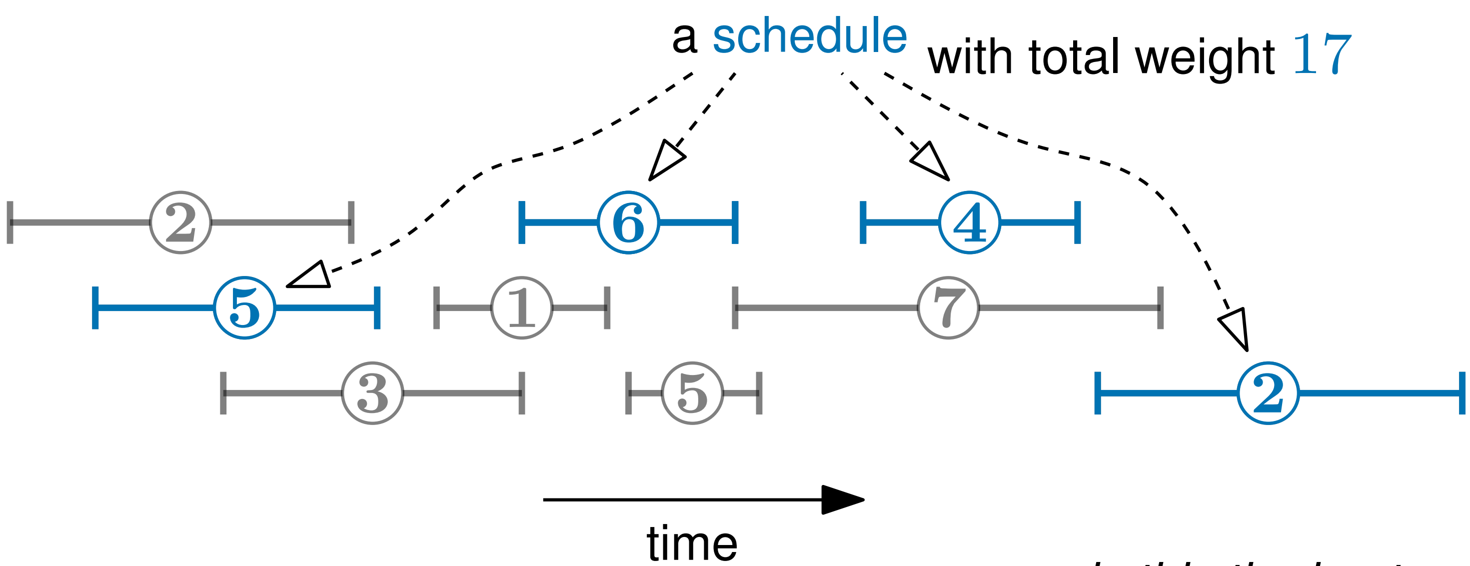
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

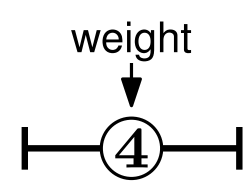


is this the best possible?

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

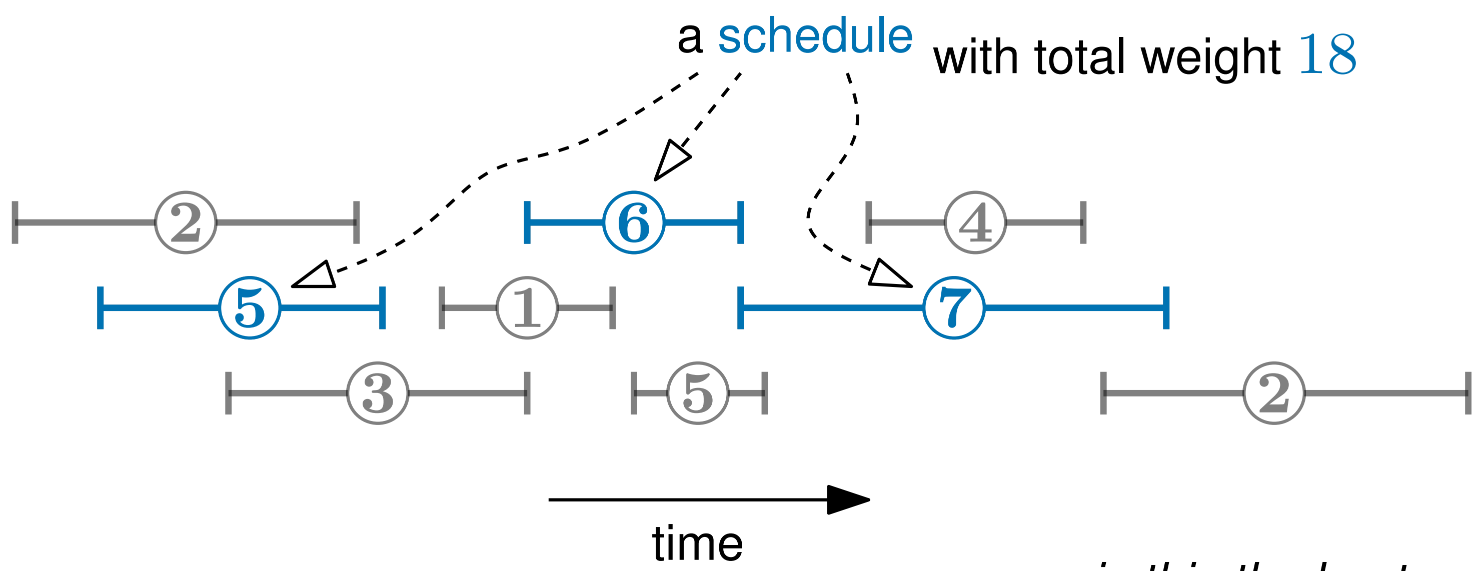
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

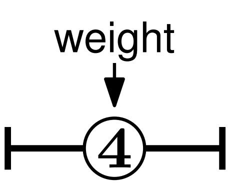


is this the best possible?

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

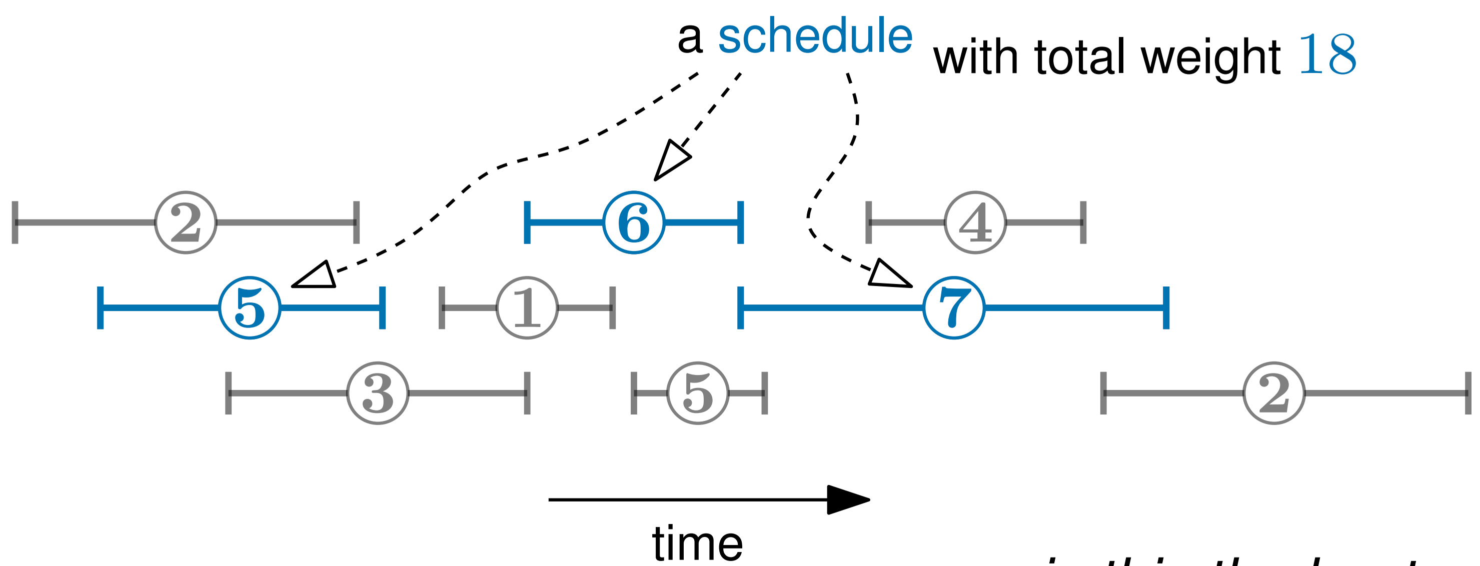
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

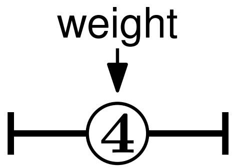


is this the best possible?

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

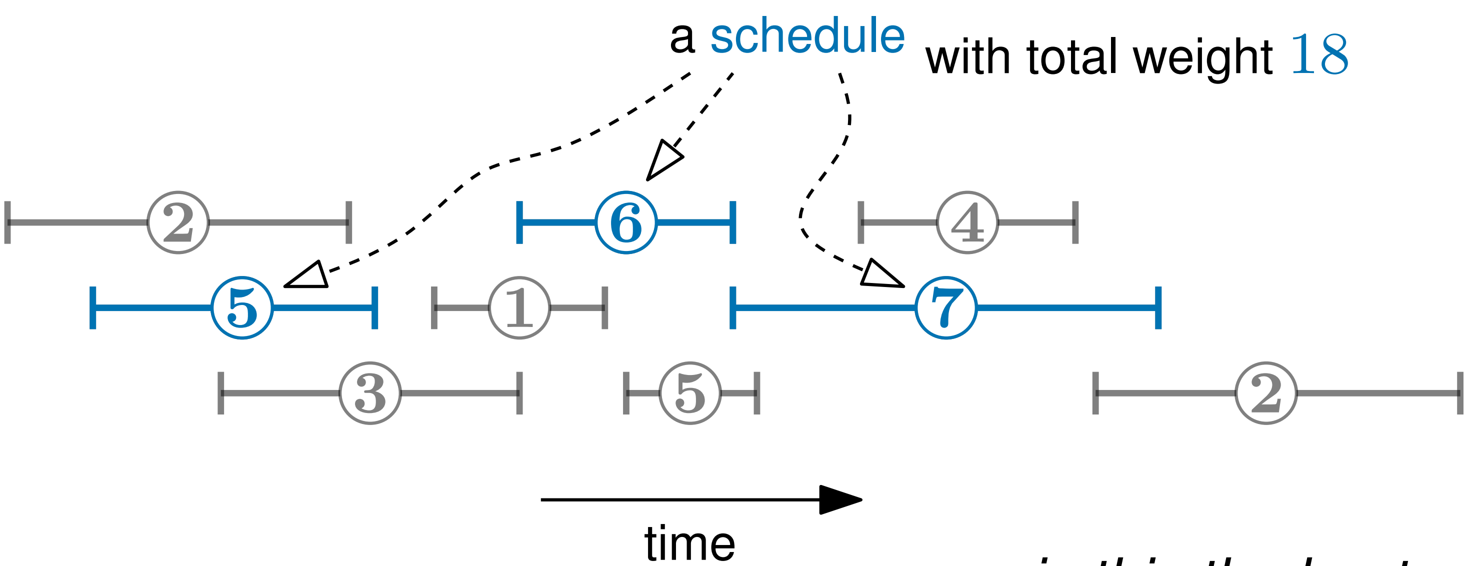
The weight of a schedule is the sum of
the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight



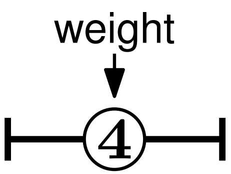
is this the best possible?

yes

Two intervals are compatible if they don't overlap

A schedule is a set of compatible intervals

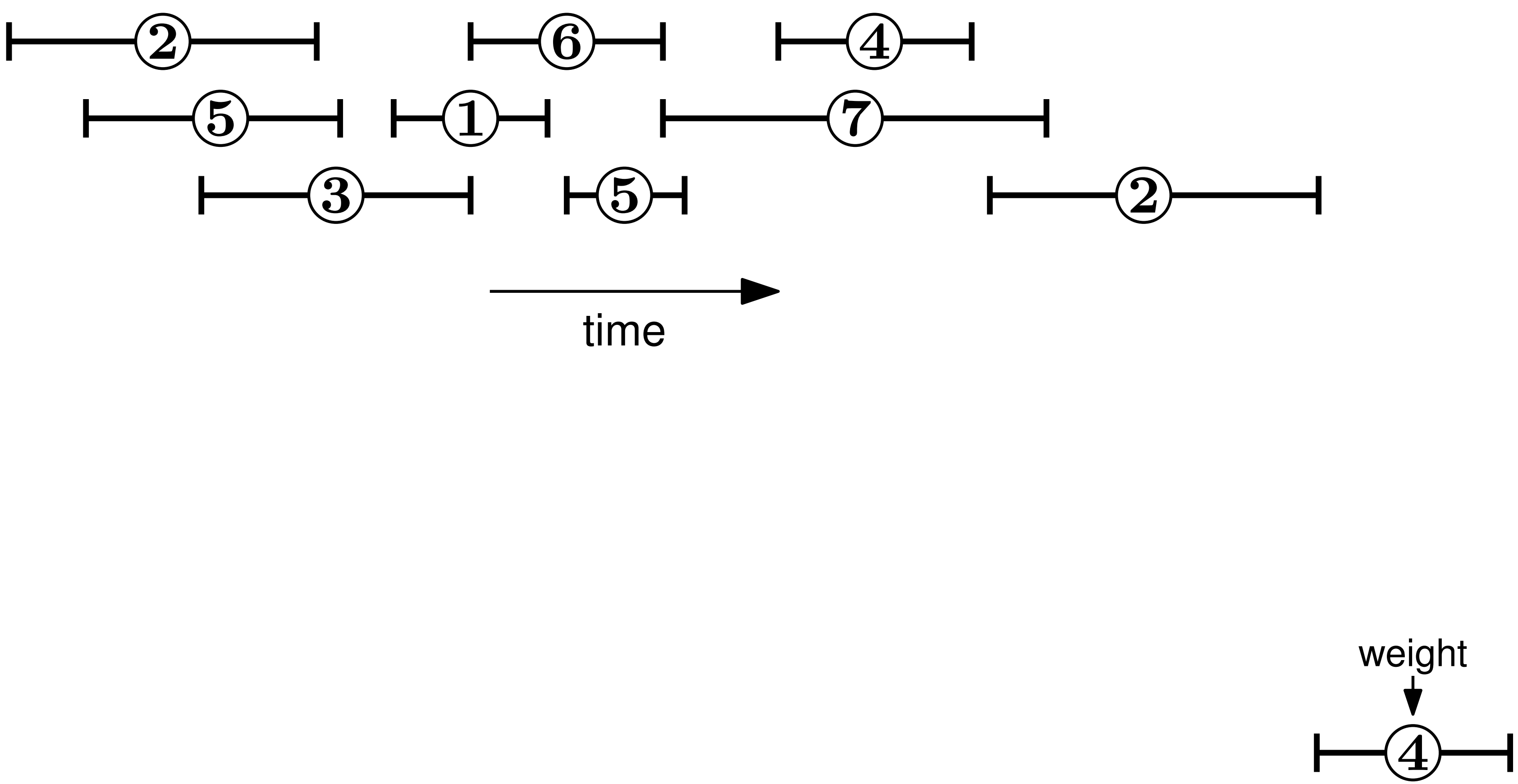
The weight of a schedule is the sum of the weight of the intervals it contains



Weighted Interval Scheduling

Problem Given an n weighted intervals,

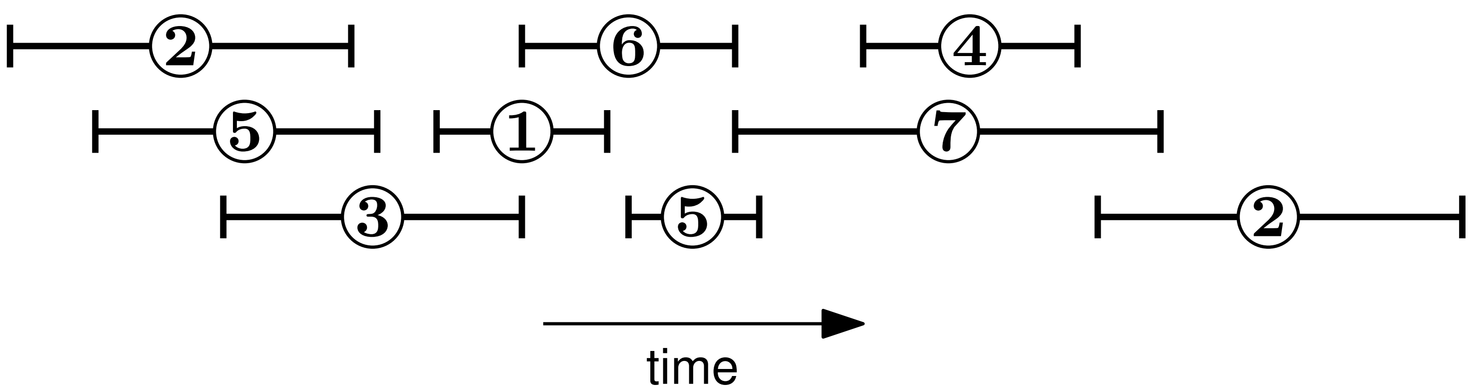
find the *schedule* with largest total weight



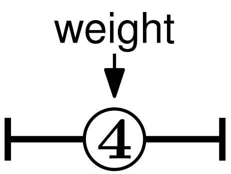
Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight



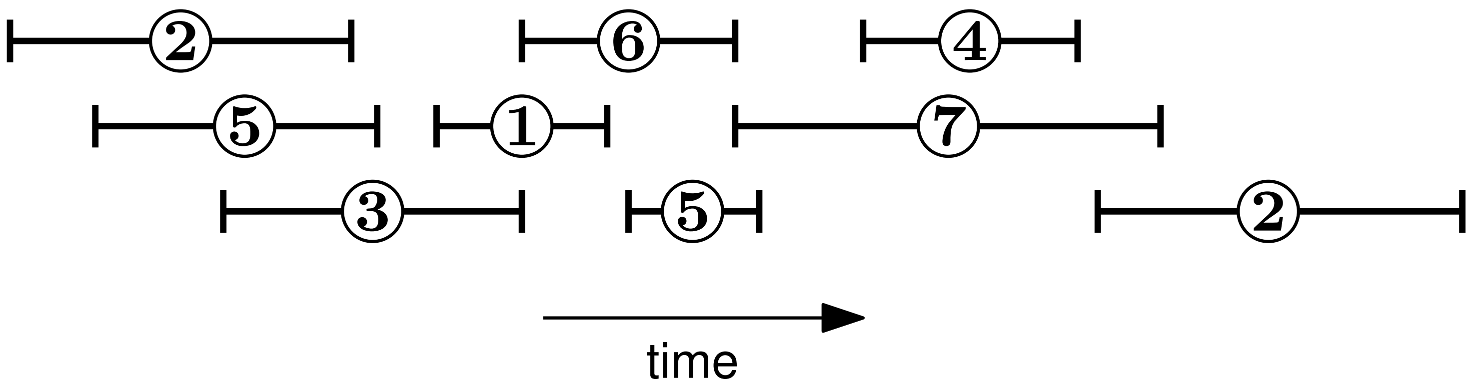
How is the input provided?



Weighted Interval Scheduling

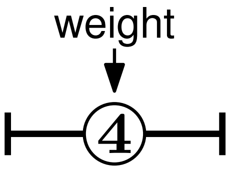
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



How is the input provided?

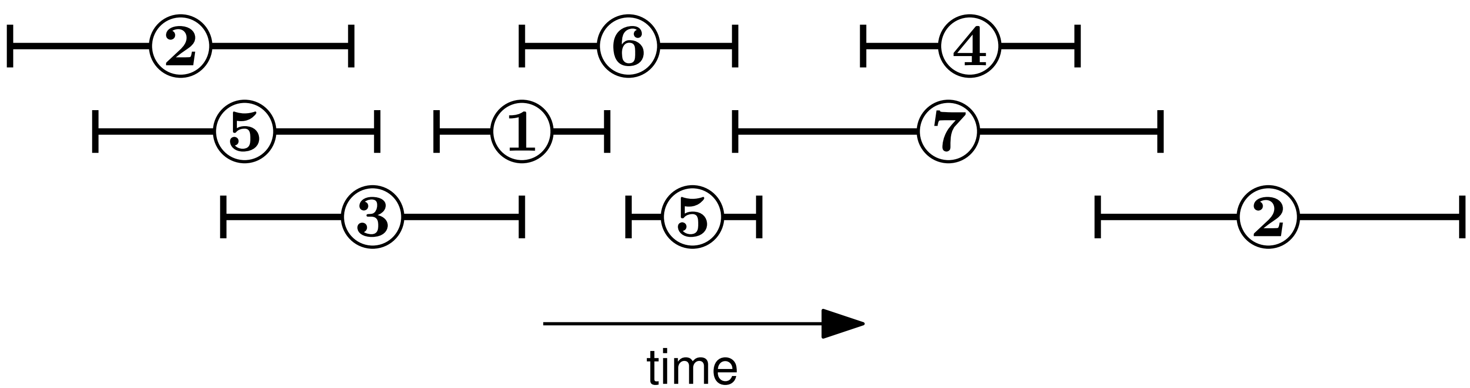
The intervals are given in an array A of length n



Weighted Interval Scheduling

Problem Given an n weighted intervals,

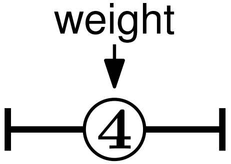
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

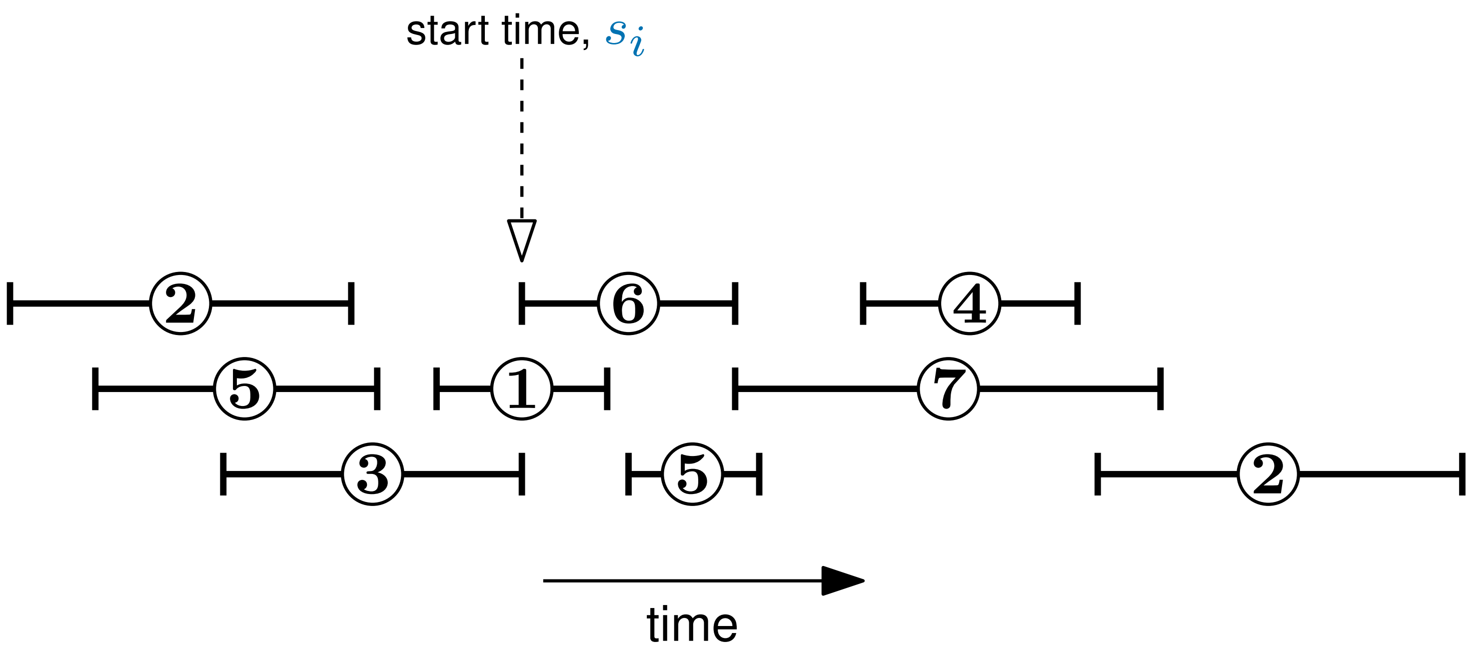
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval



Weighted Interval Scheduling

Problem Given an n weighted intervals,

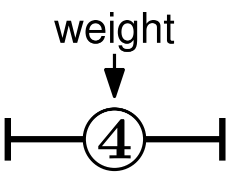
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

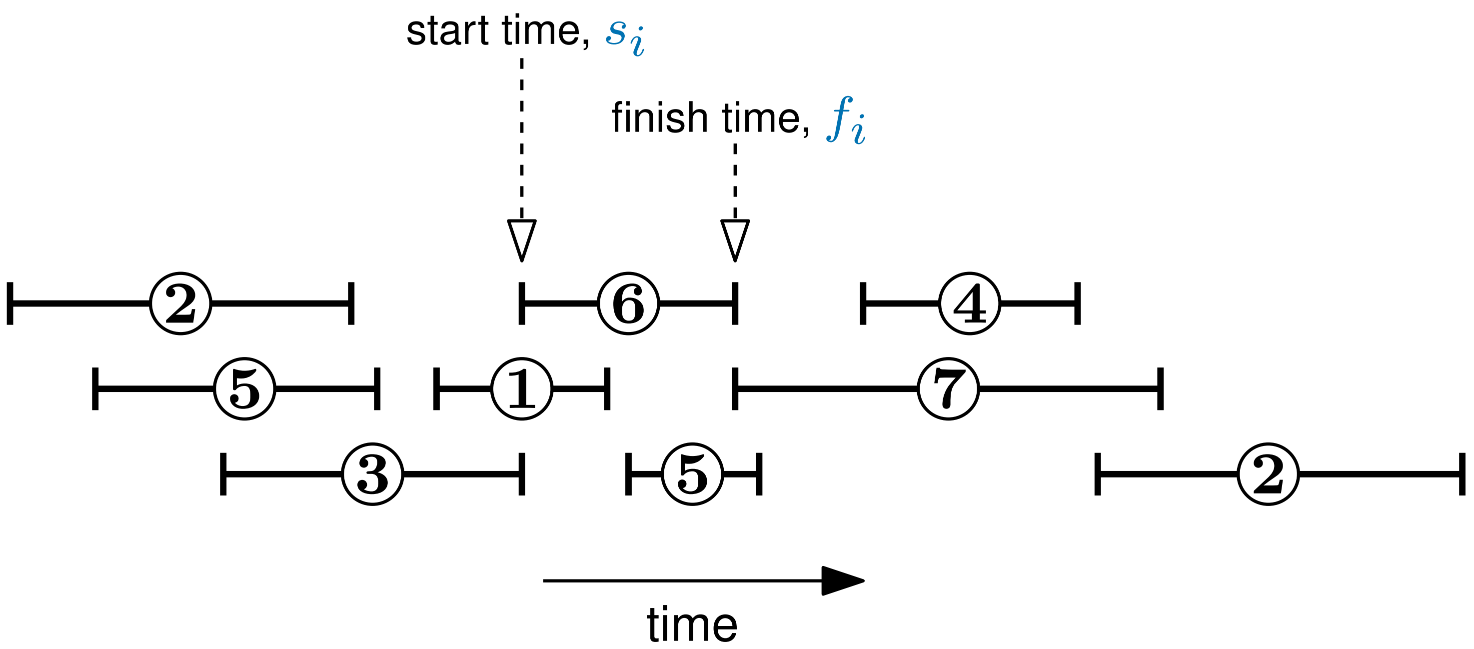
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval



Weighted Interval Scheduling

Problem Given an n weighted intervals,

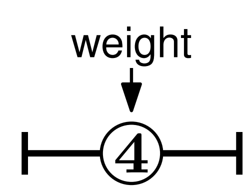
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

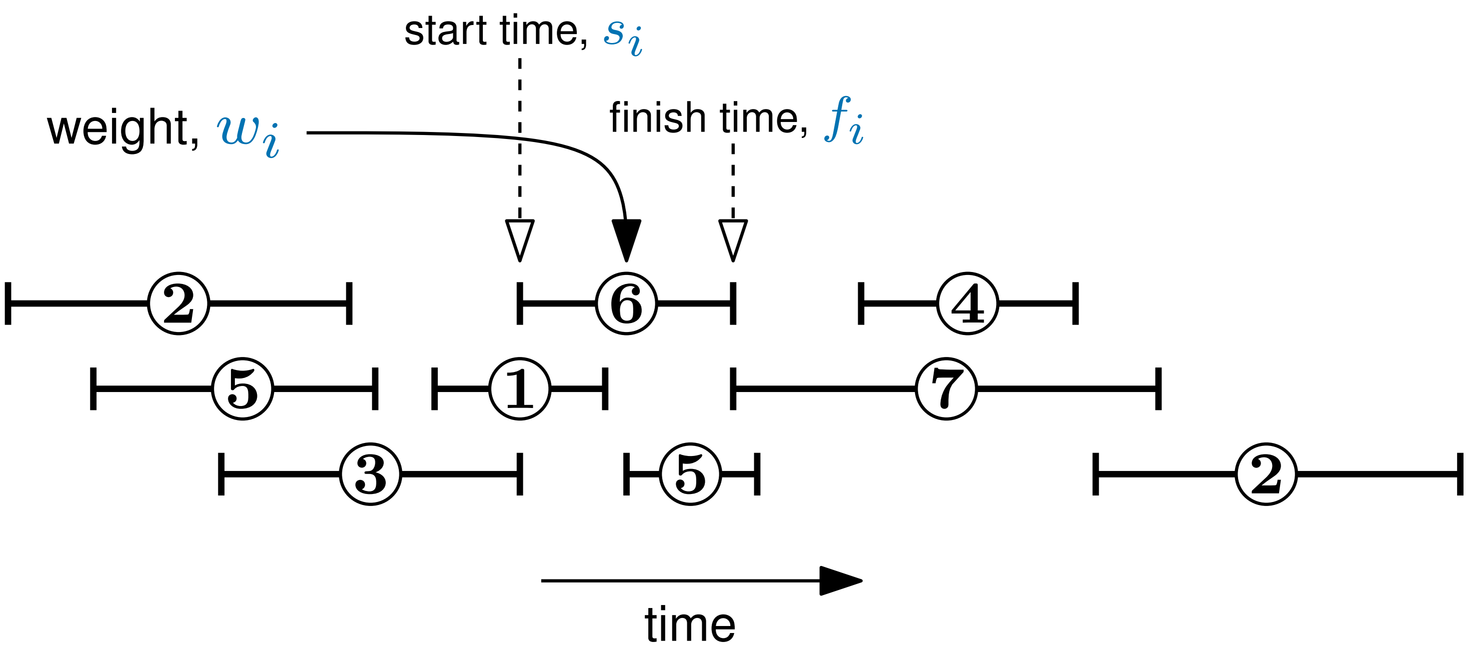
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval



Weighted Interval Scheduling

Problem Given an n weighted intervals,

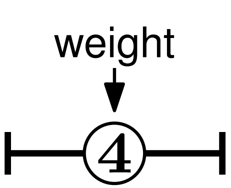
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

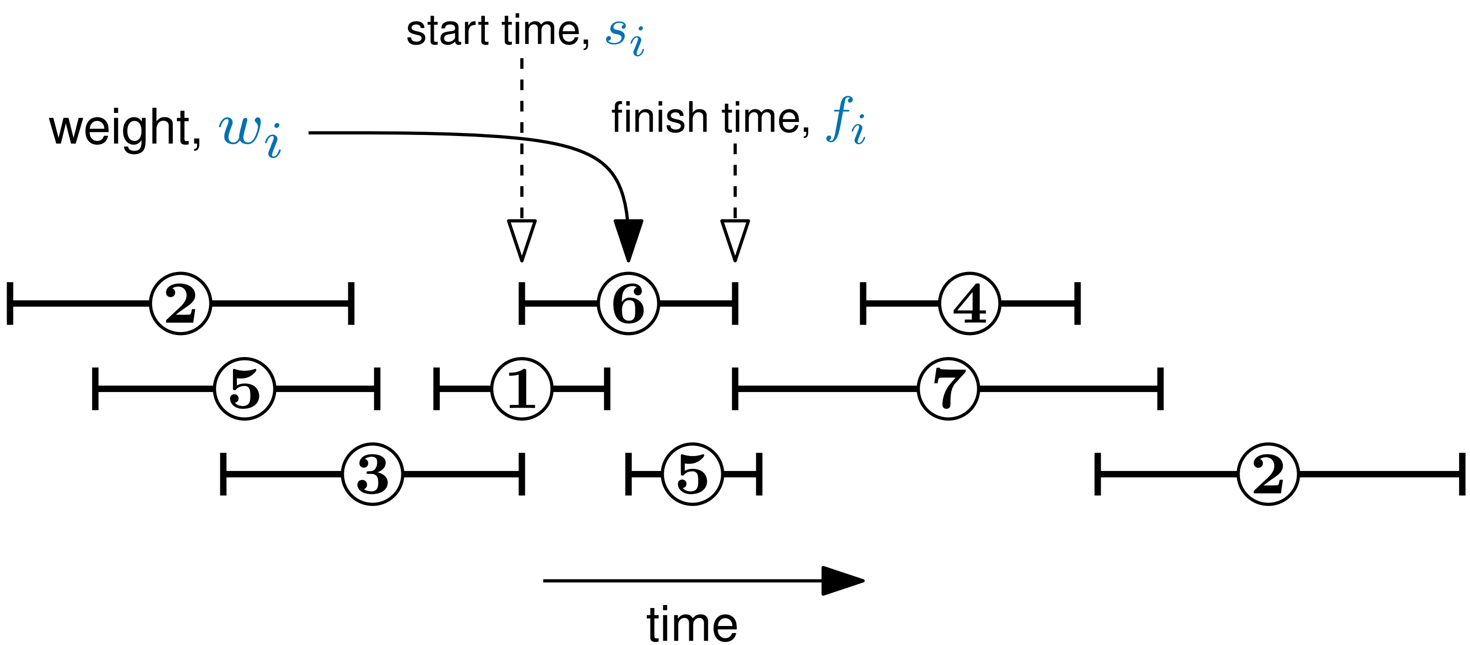
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval



Weighted Interval Scheduling

Problem Given an n weighted intervals,

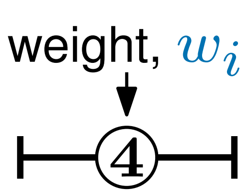
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

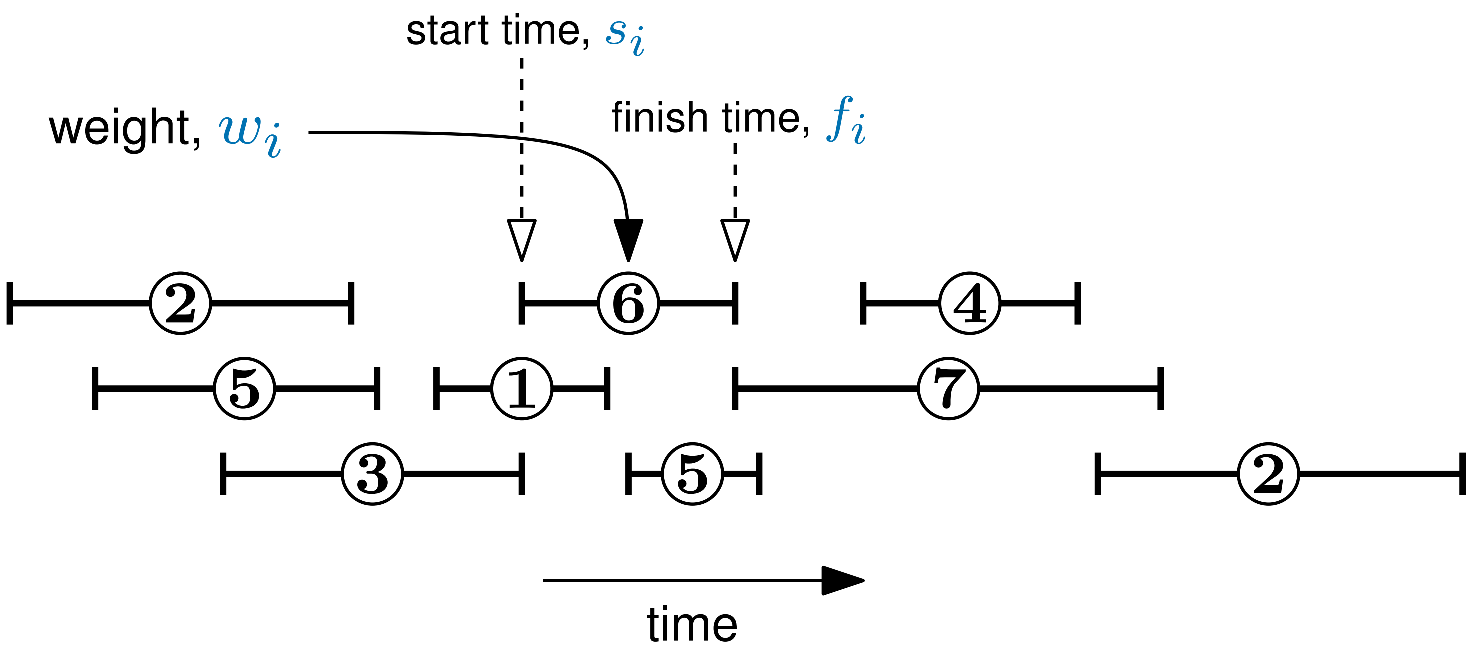
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval



Weighted Interval Scheduling

Problem Given an n weighted intervals,

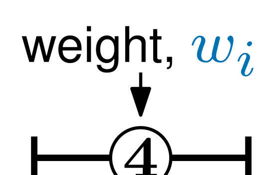
find the *schedule* with largest total weight



How is the input provided?

The intervals are given in an array A of length n

$A[i]$ stores a triple (s_i, f_i, w_i) which defines the i-th interval

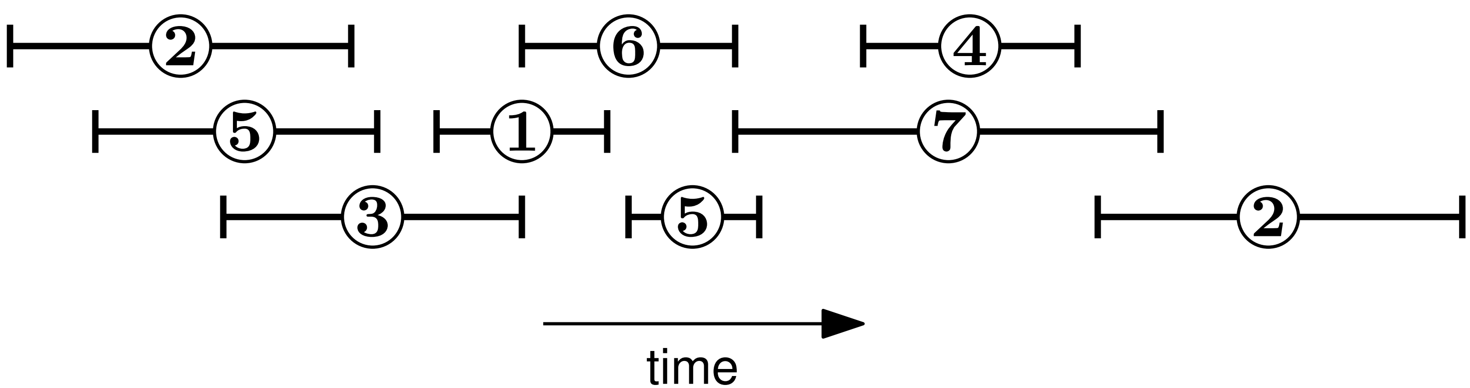


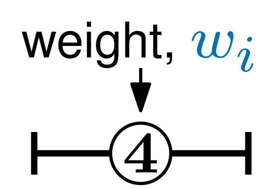
The intervals are sorted by *finish time* i.e. $f_i \leqslant f_{i+1}$

Weighted Interval Scheduling

Problem Given an n weighted intervals,

find the *schedule* with largest total weight

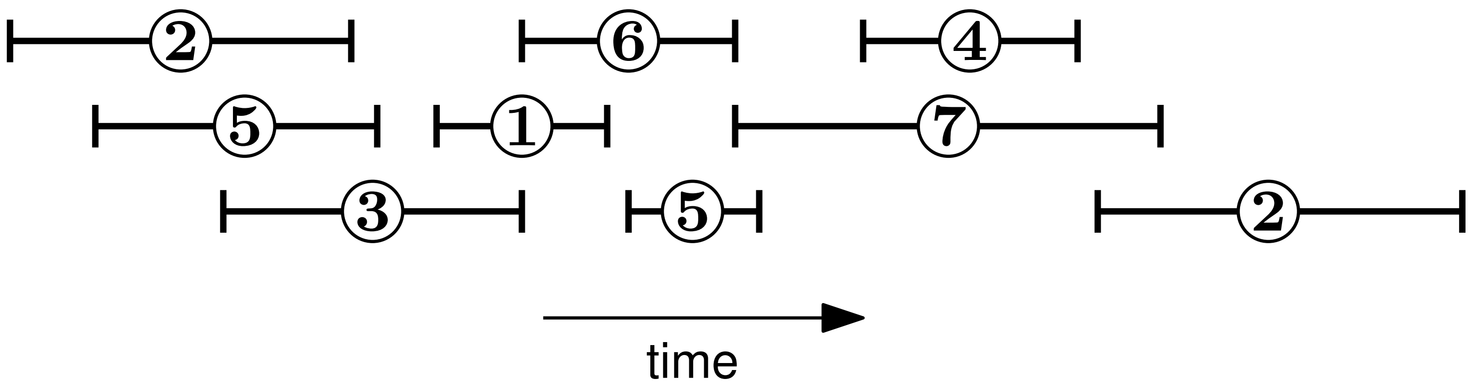




Weighted Interval Scheduling

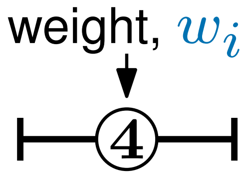
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

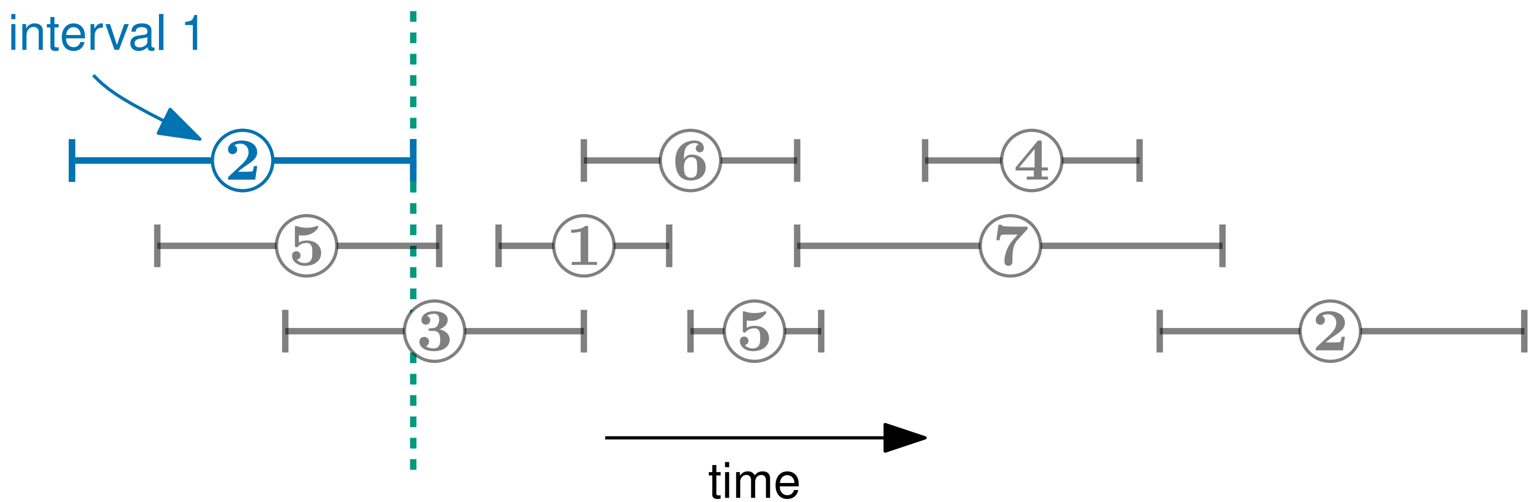
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

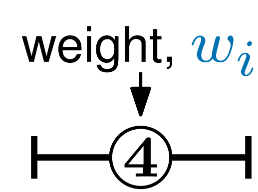
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

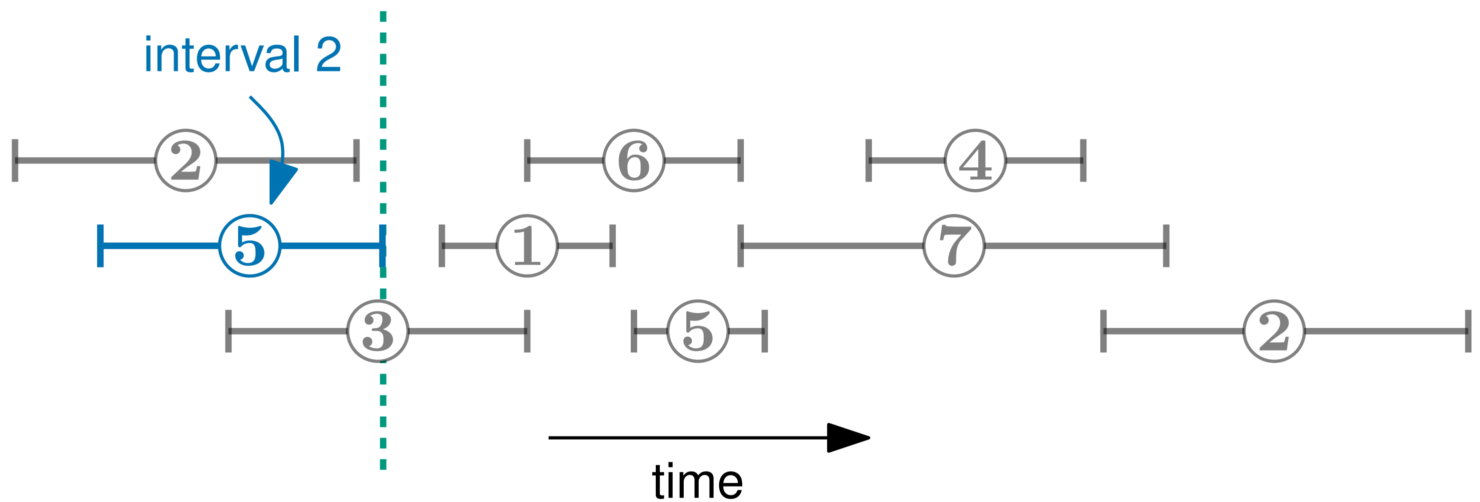
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

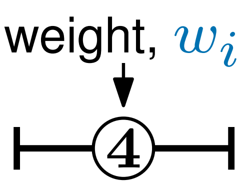
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

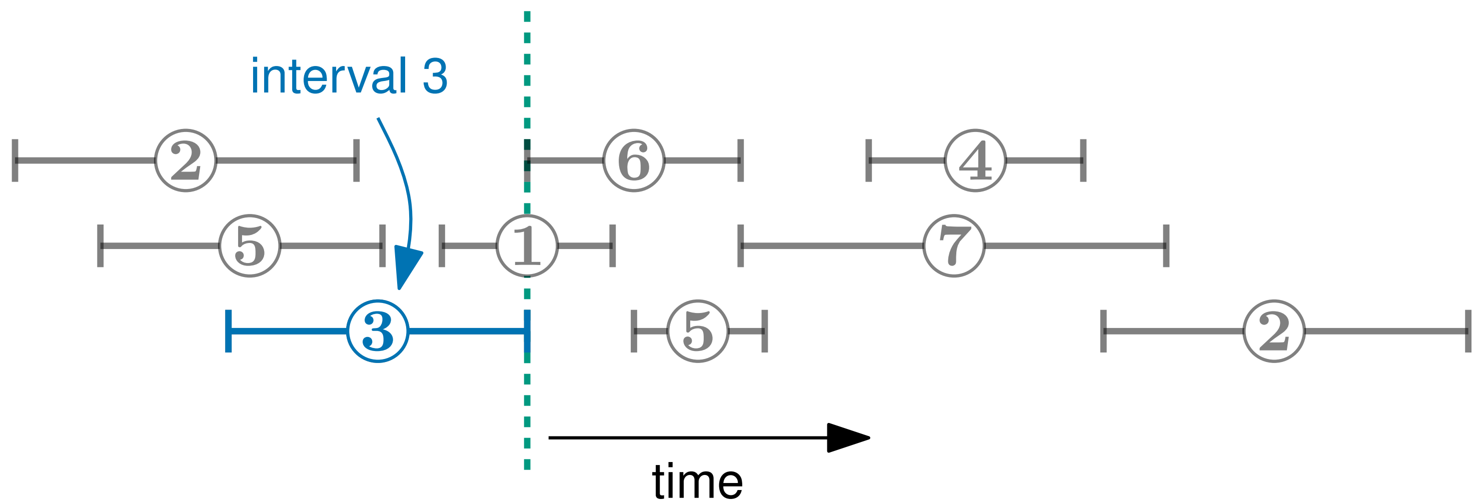
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

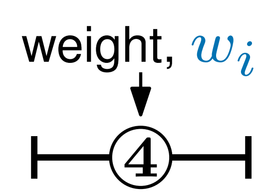
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

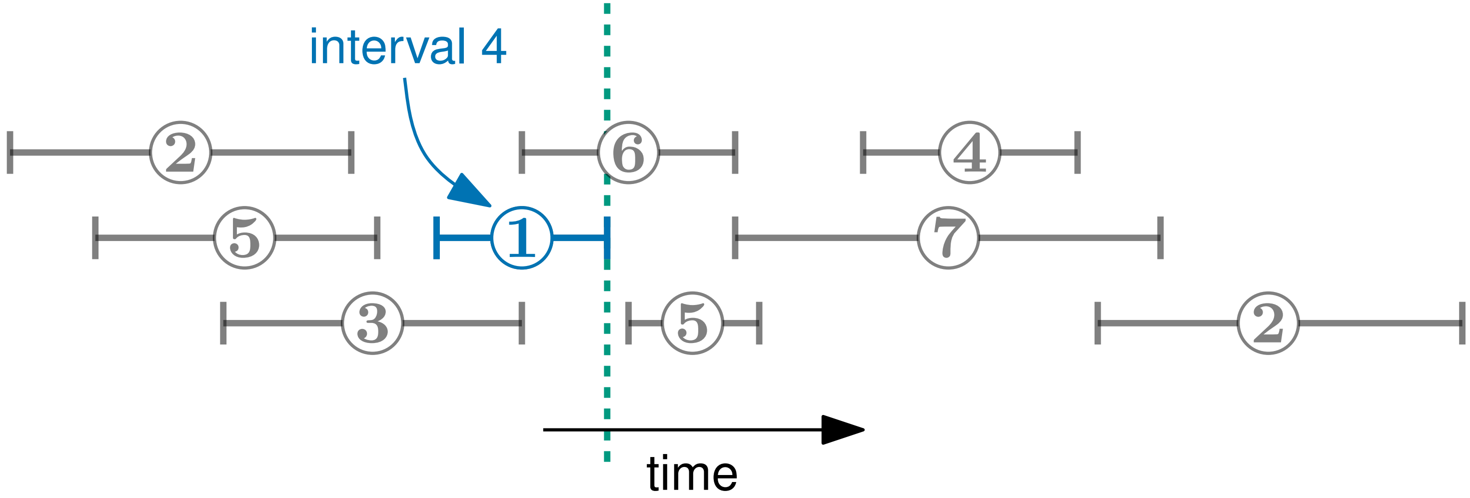
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

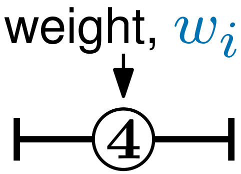
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

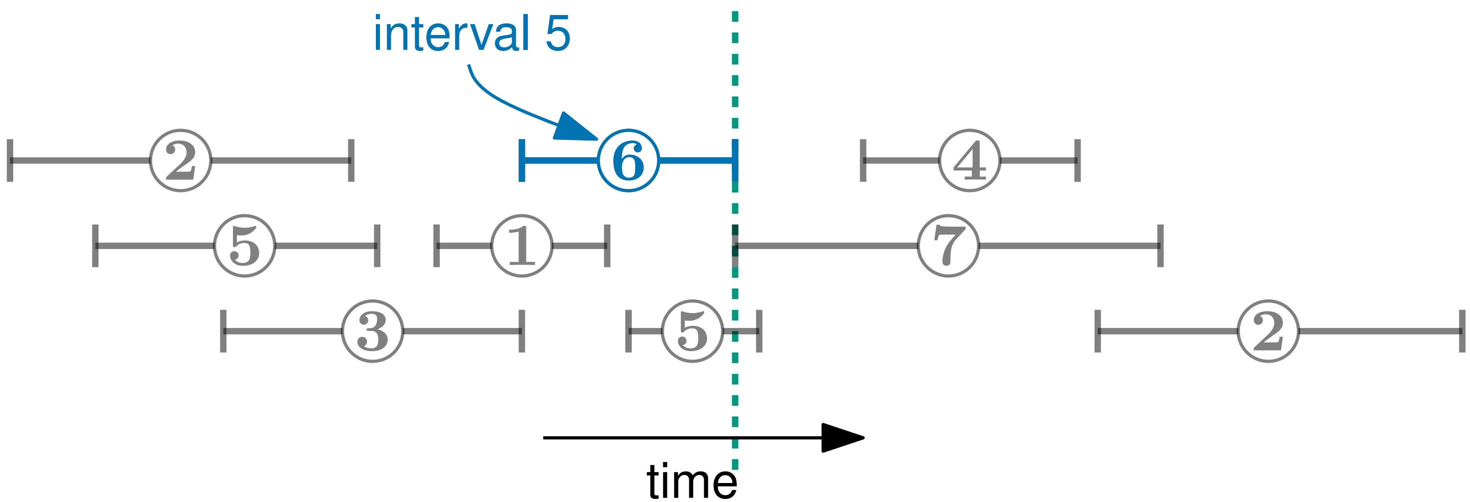
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

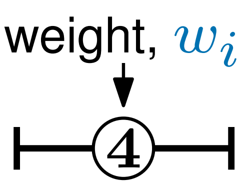
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

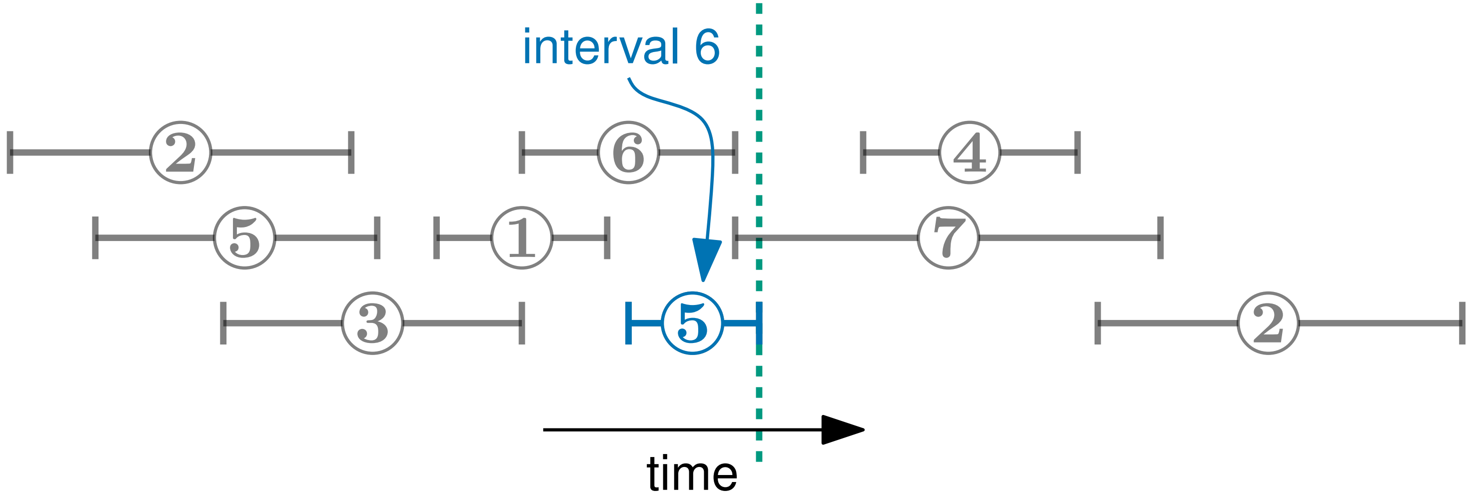
interval i finishes before interval $i + 1$ finishes



Weighted Interval Scheduling

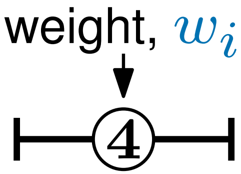
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

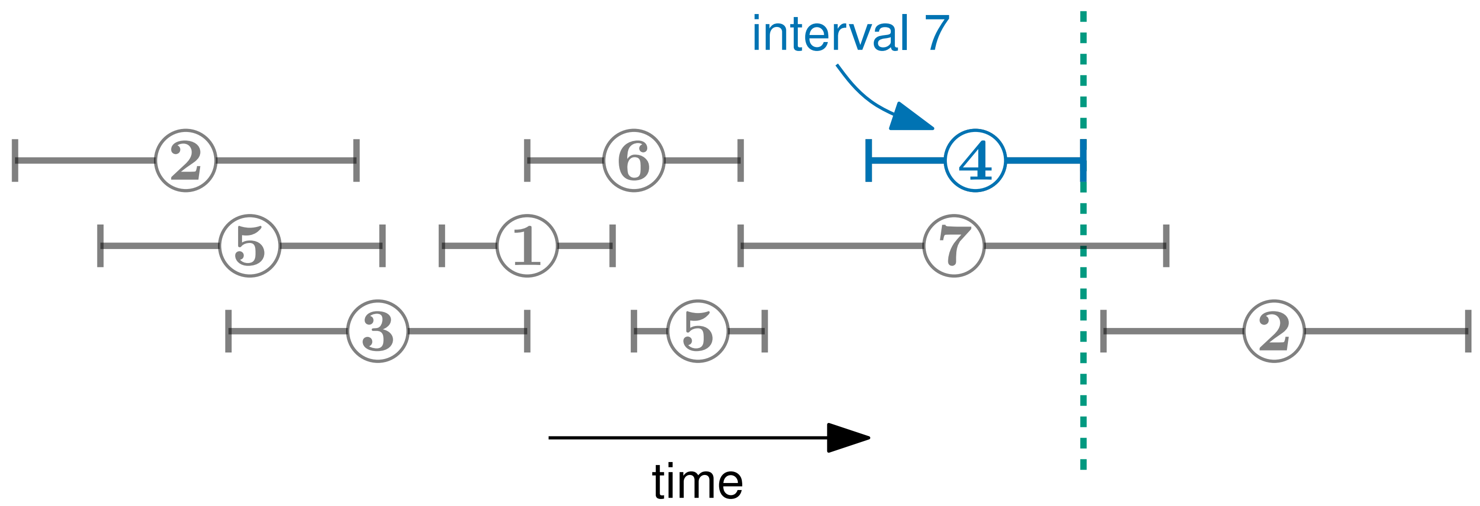
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

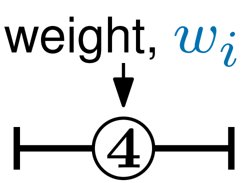
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

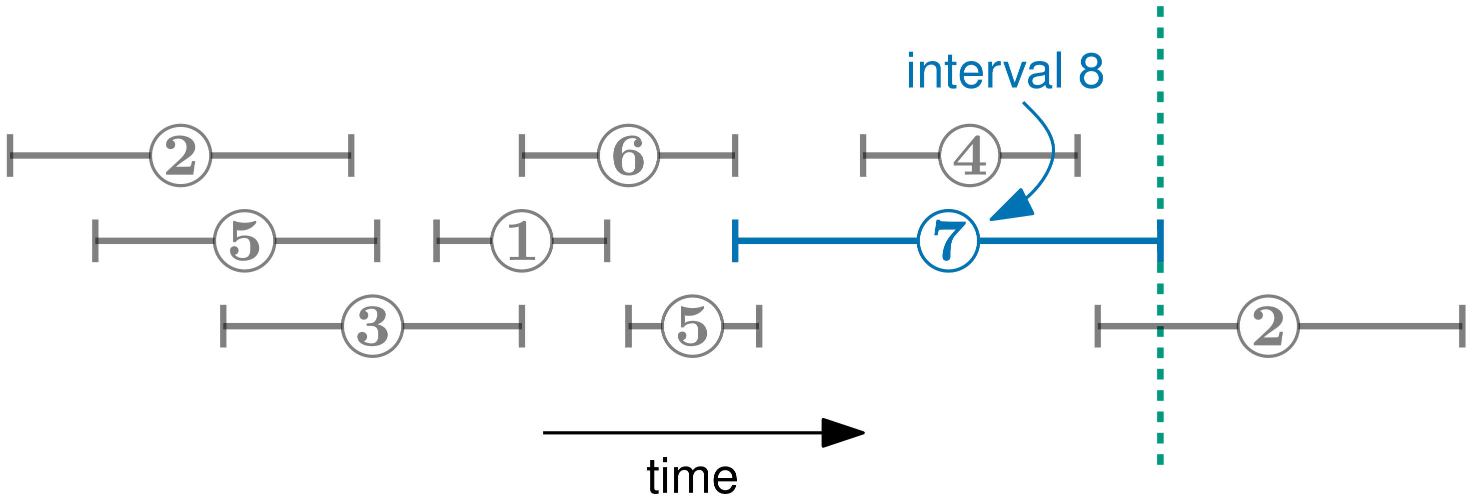
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

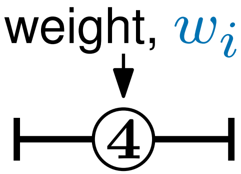
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

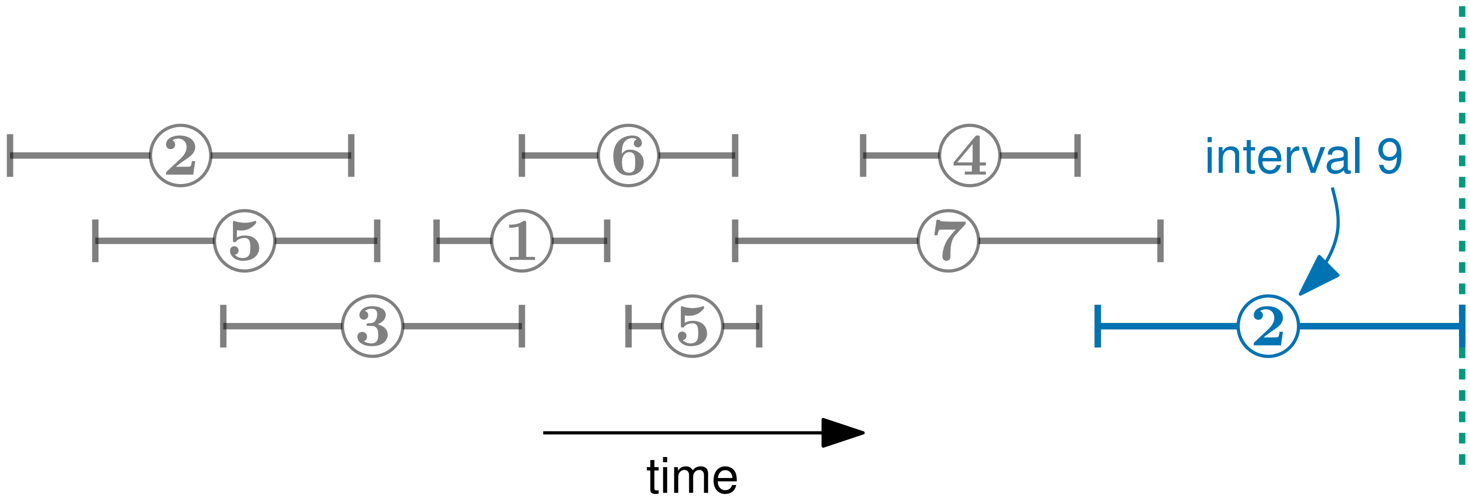
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

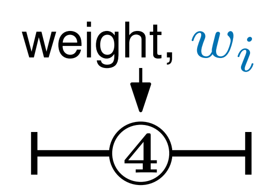
Problem Given an n weighted intervals,

find the *schedule* with largest total weight



The intervals in the input are sorted by *finish time*

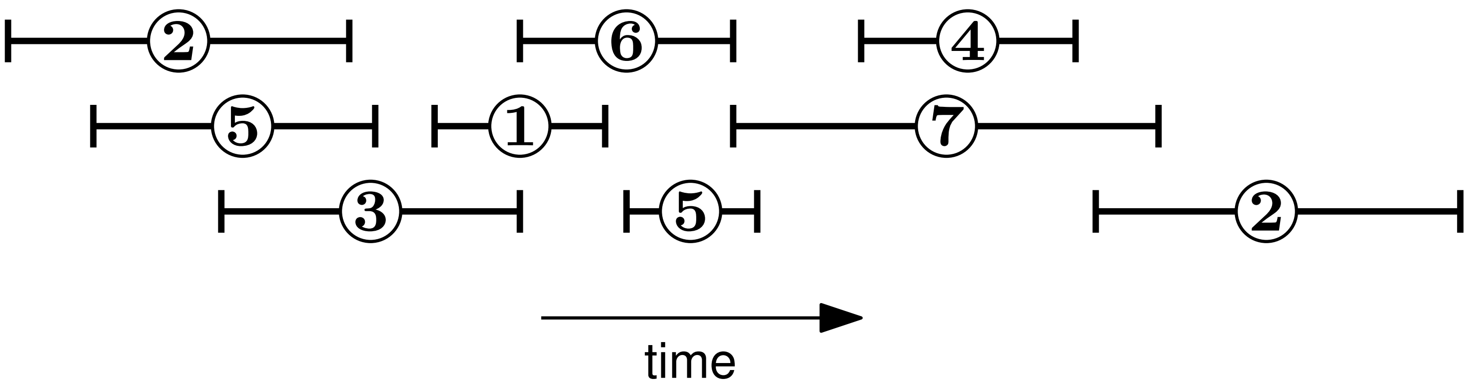
interval i finishes before interval $i+1$ finishes



Weighted Interval Scheduling

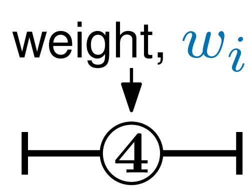
Problem Given an n weighted intervals,

find the *schedule* with largest total weight

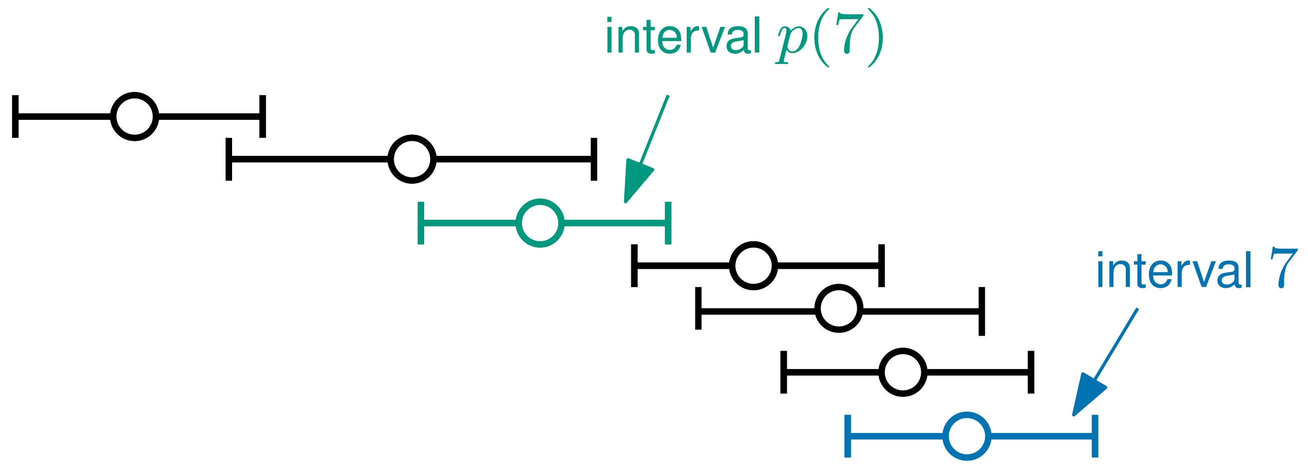


The intervals in the input are sorted by *finish time*

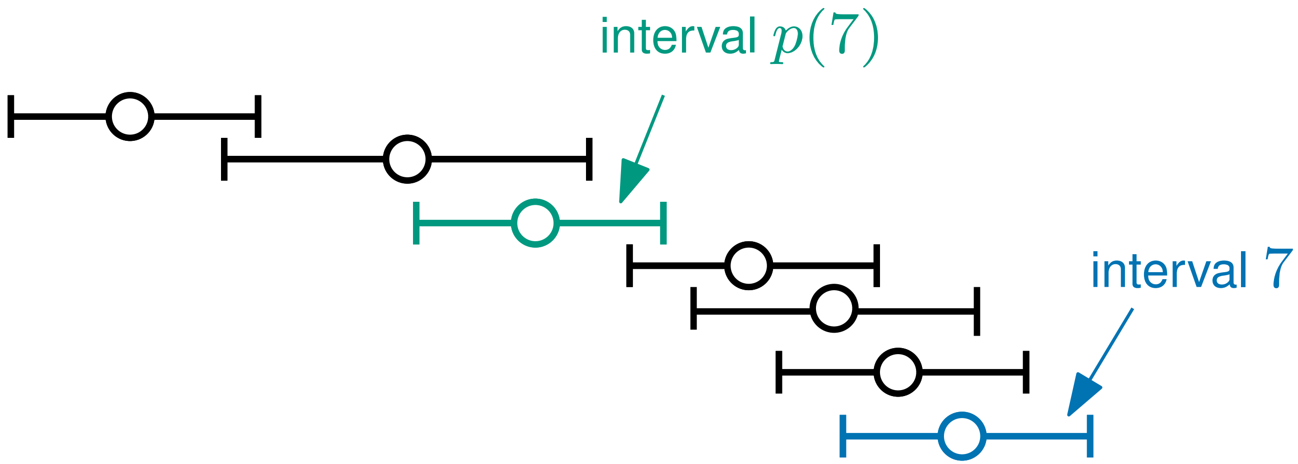
interval i finishes before interval $i+1$ finishes



Compatible Intervals



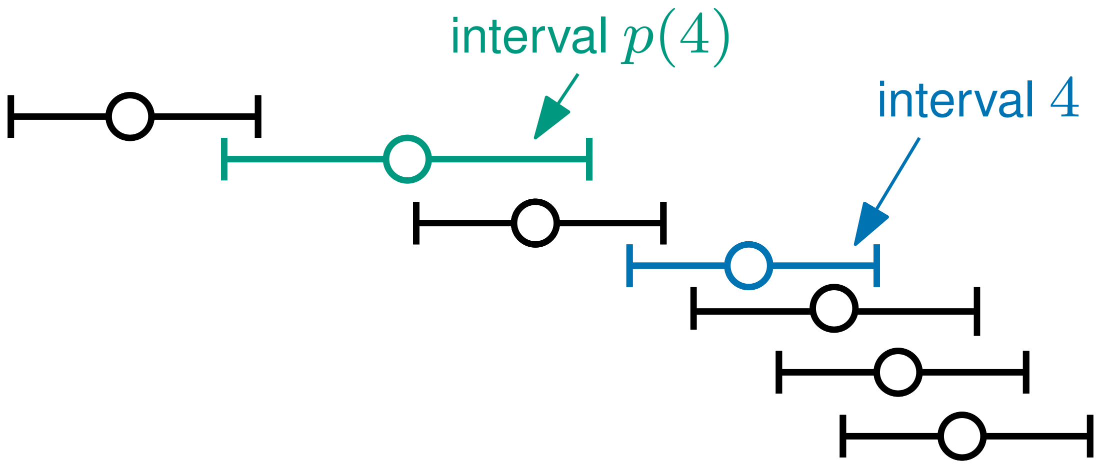
Compatible Intervals



For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

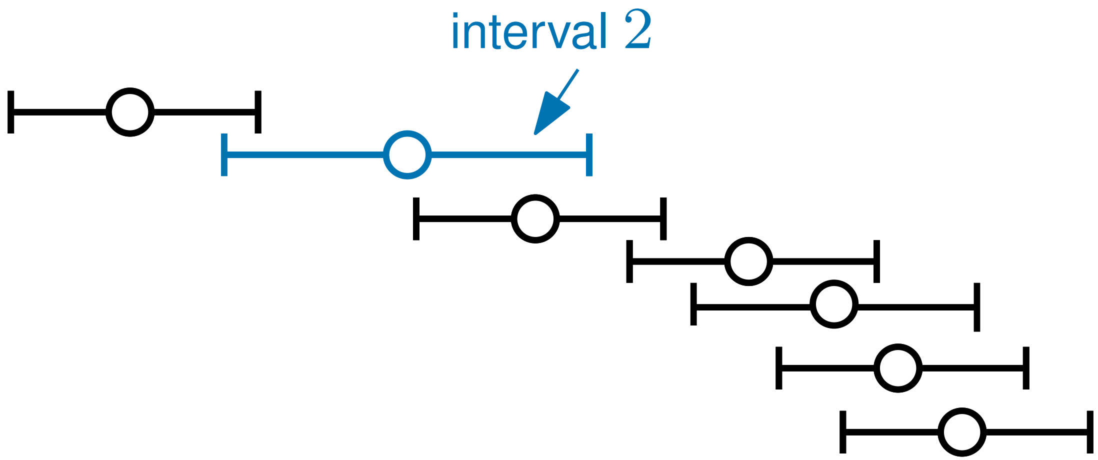
Compatible Intervals



For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

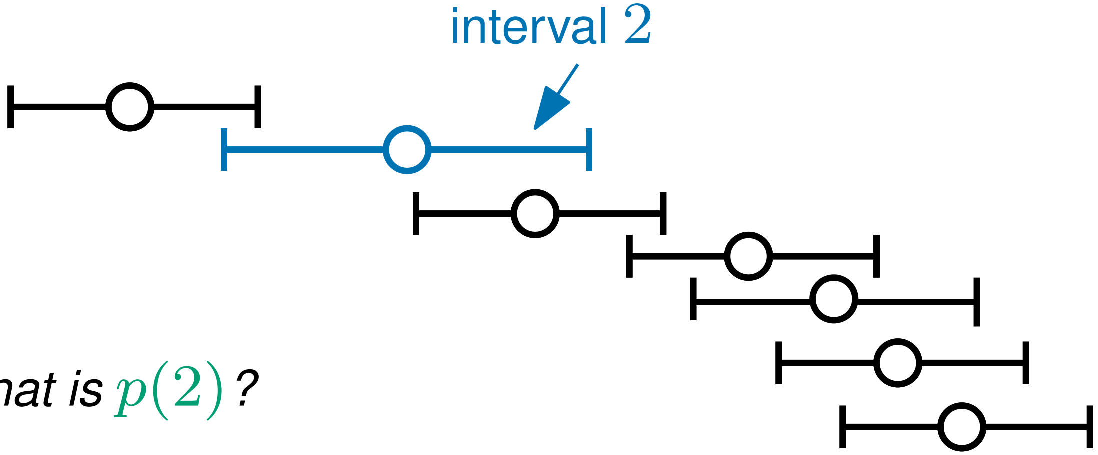
Compatible Intervals



For all i,

Let $p(i)$ be the rightmost interval (in order of finish time)
which finishes before the i-th interval but doesn't overlap it

Compatible Intervals

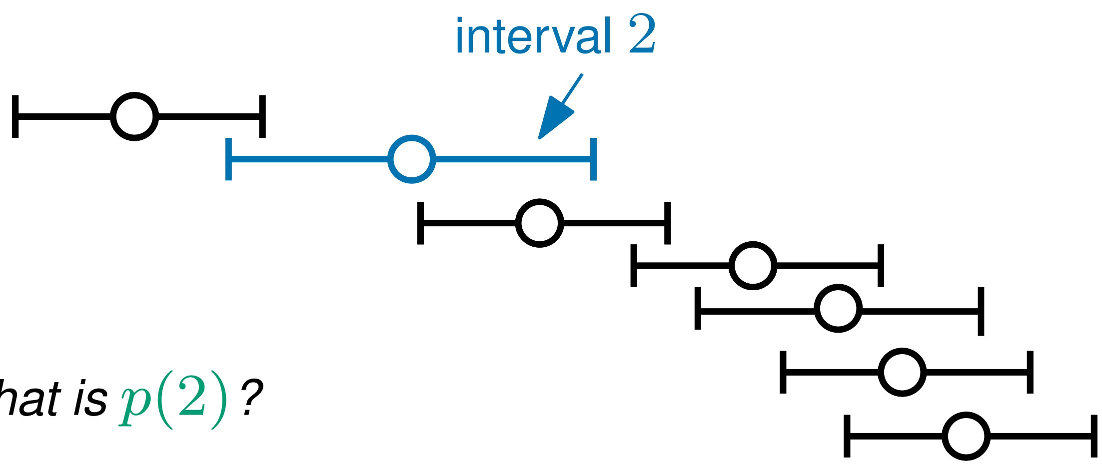


What is $p(2)$ *?*

For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

Compatible Intervals



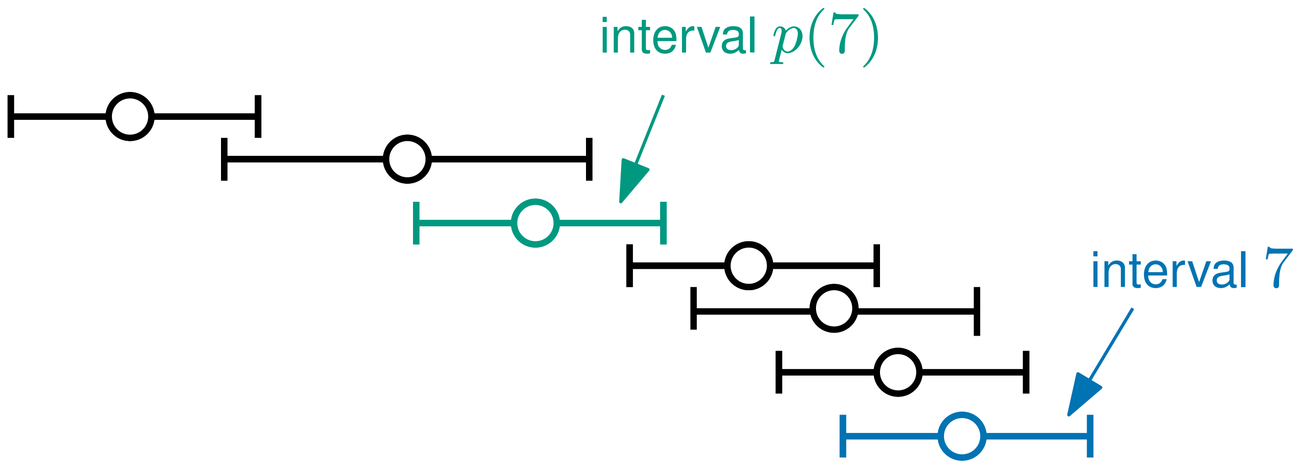
What is $p(2)$ *?*

For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

if no such interval exists, $p(i) = 0$

Compatible Intervals

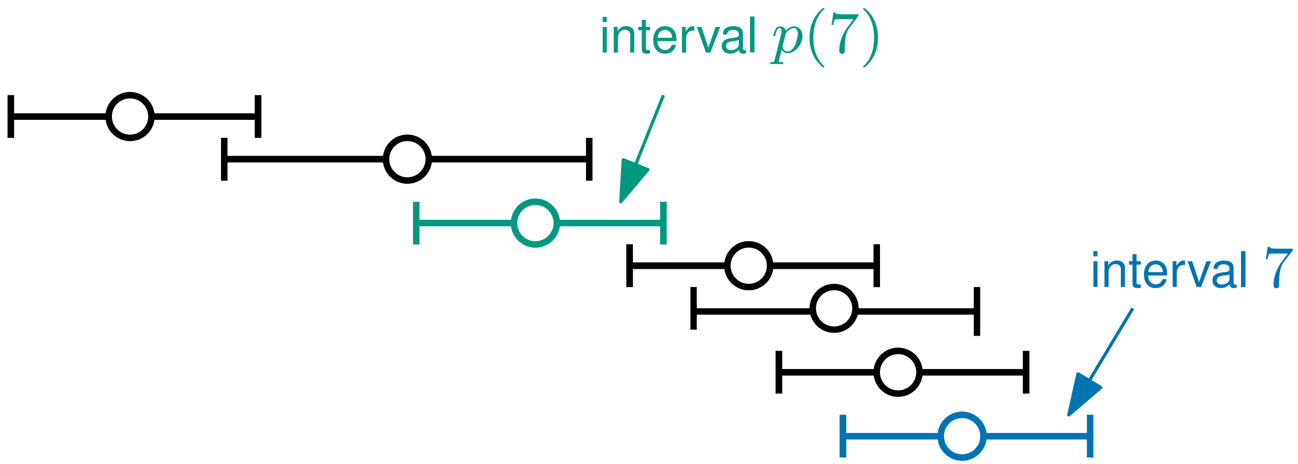


For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

if no such interval exists, $p(i) = 0$

Compatible Intervals



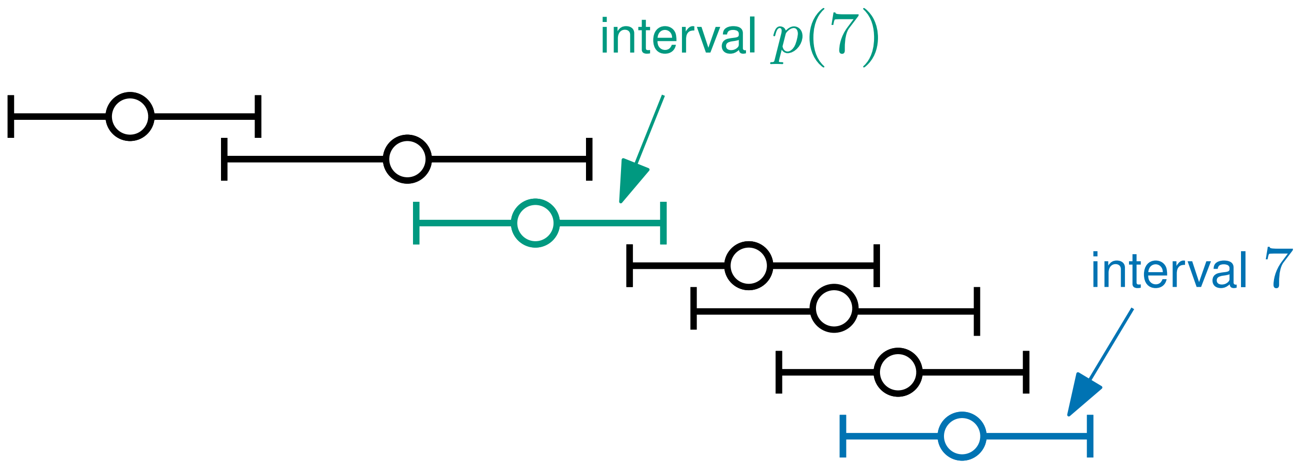
For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

if no such interval exists, $p(i) = 0$

Claim: We can precompute all $p(i)$ in $O(n \log n)$ time

Compatible Intervals



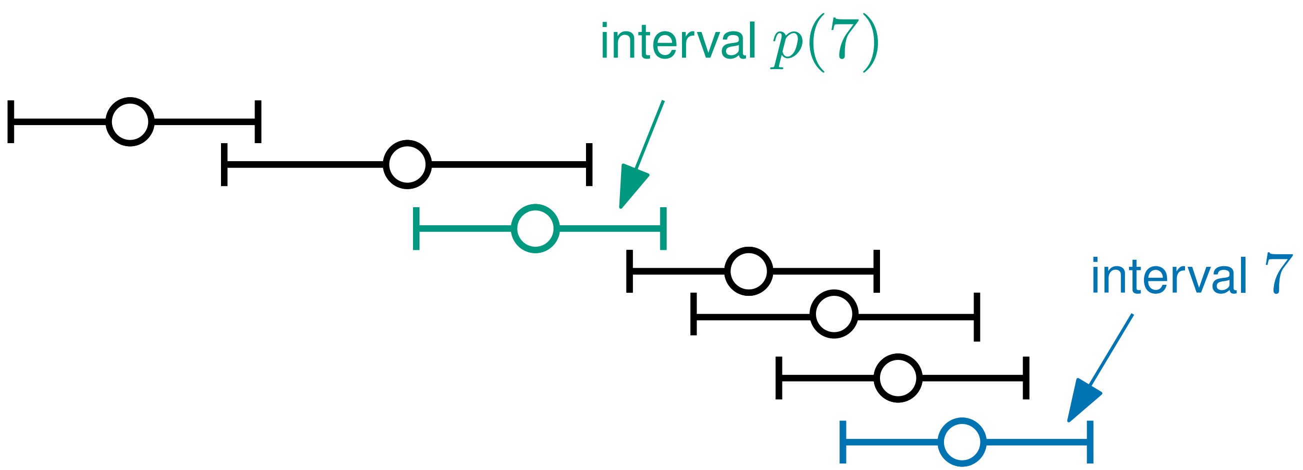
For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

if no such interval exists, $p(i) = 0$

Claim: We can precompute all $p(i)$ in $O(n \log n)$ time (and we'll assume we did this already)

Compatible Intervals



For all i,

Let $p(i)$ be the rightmost interval (in order of finish time) which finishes before the i-th interval but doesn't overlap it

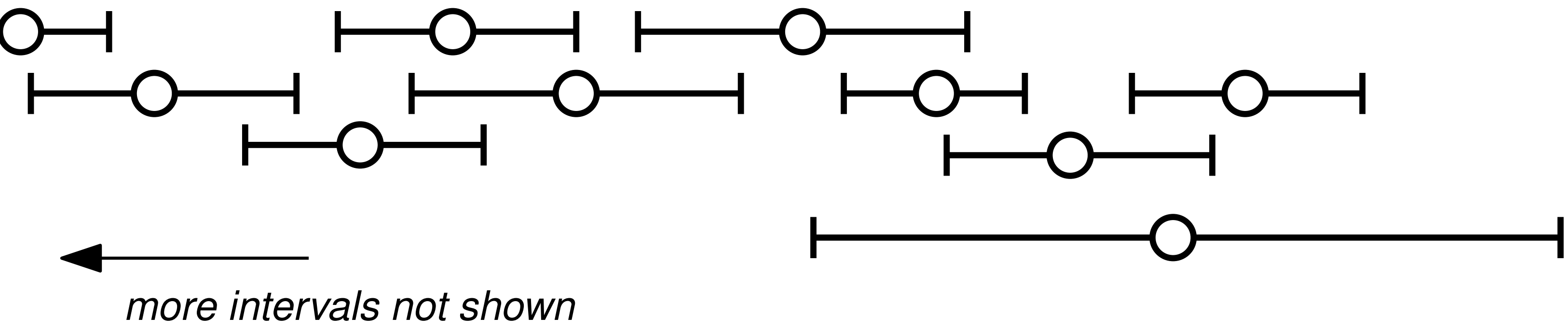
if no such interval exists, $p(i) = 0$

Claim: We can precompute all $p(i)$ in $O(n \log n)$ time (and we'll assume we did this already)

- we'll come back to this at the end

1. Find a recursive formula

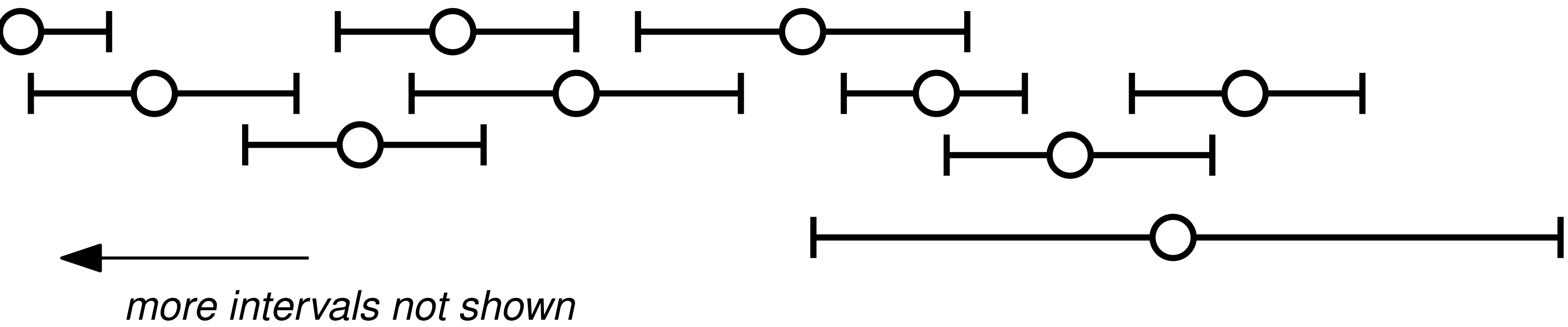
Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...

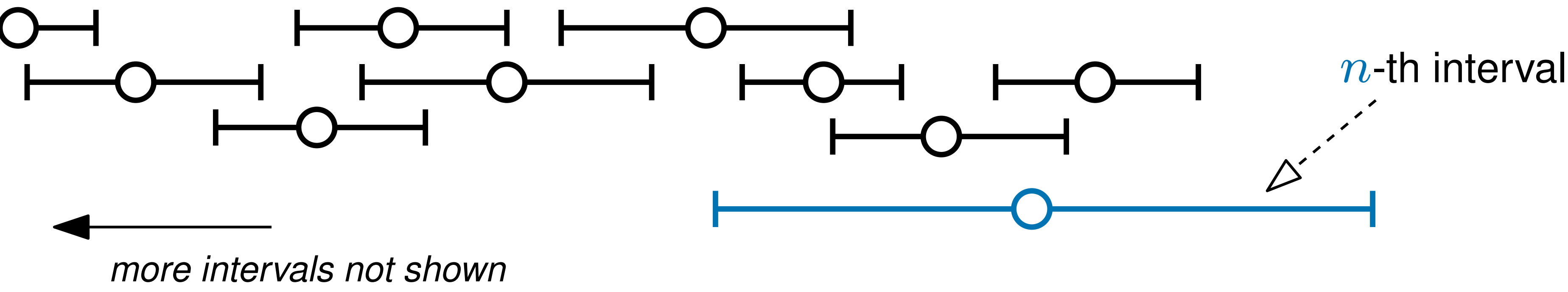
In particular, consider the n-th interval ...



1. Find a recursive formula

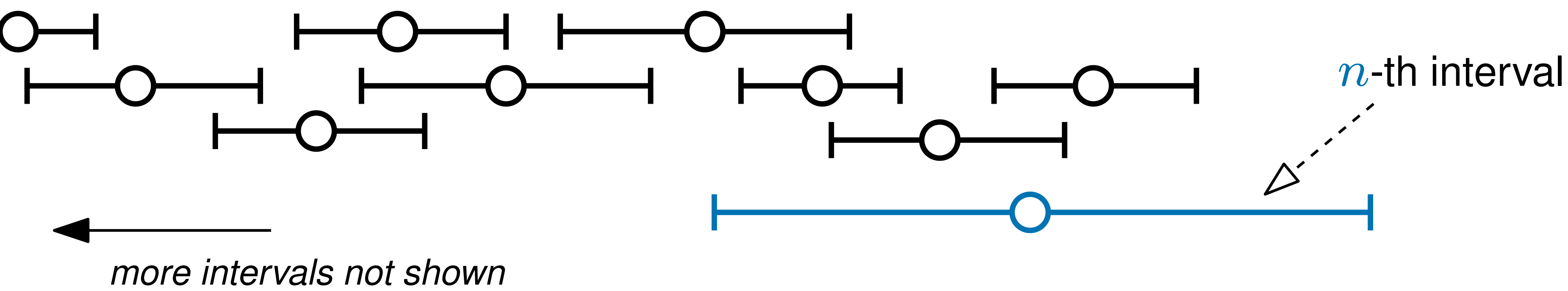
Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...

In particular, consider the n-th interval ...



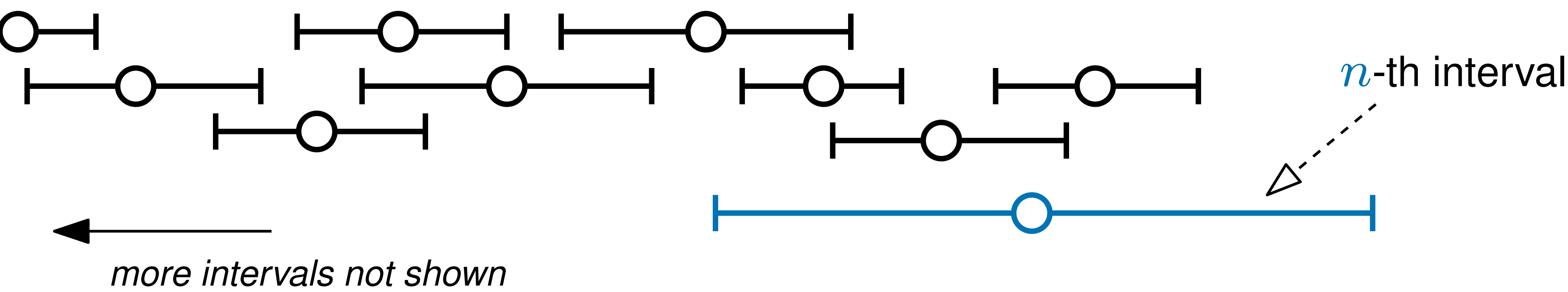
1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



1. Find a recursive formula

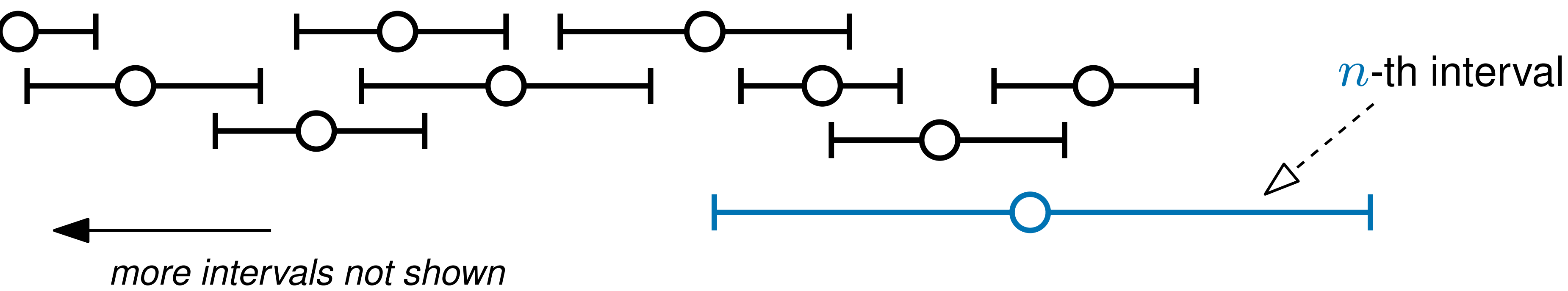
Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Either the n-th interval is in schedule $\mathcal{O}$... or it isn't

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT. . .

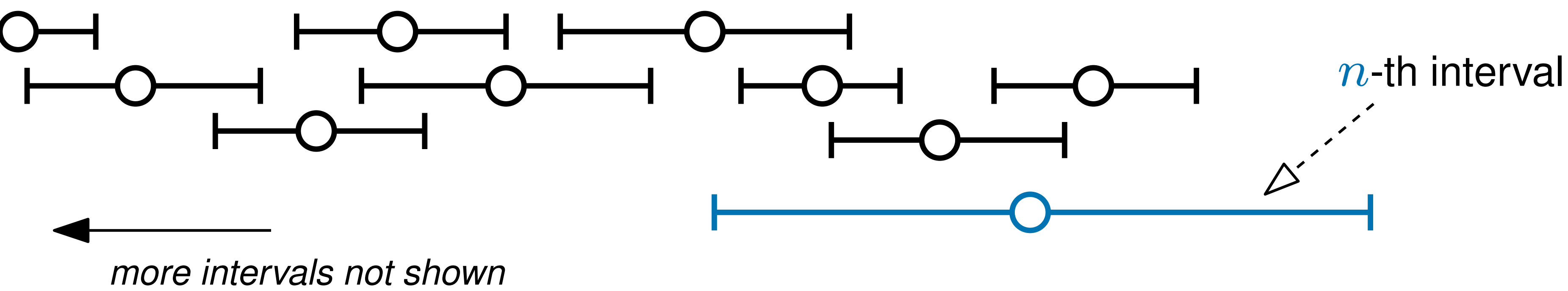


Either the n-th interval is in schedule $\mathcal{O}$. . . or it isn't

this gives us two cases to consider:

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Either the n-th interval is in schedule $\mathcal{O}$. . . or it isn't

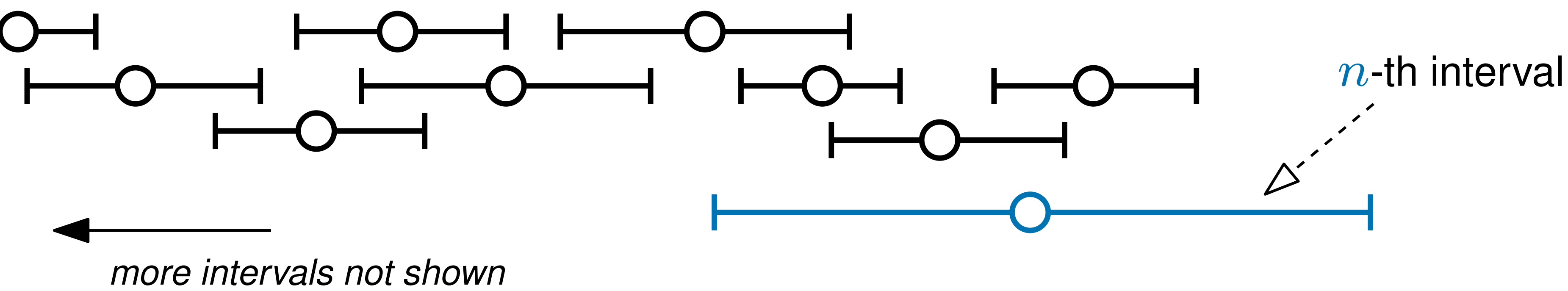
this gives us two cases to consider:

Case 1: The n-th interval is *not* in $\mathcal{O}$

Case 2: The n-th interval is in $\mathcal{O}$

1. Find a recursive formula

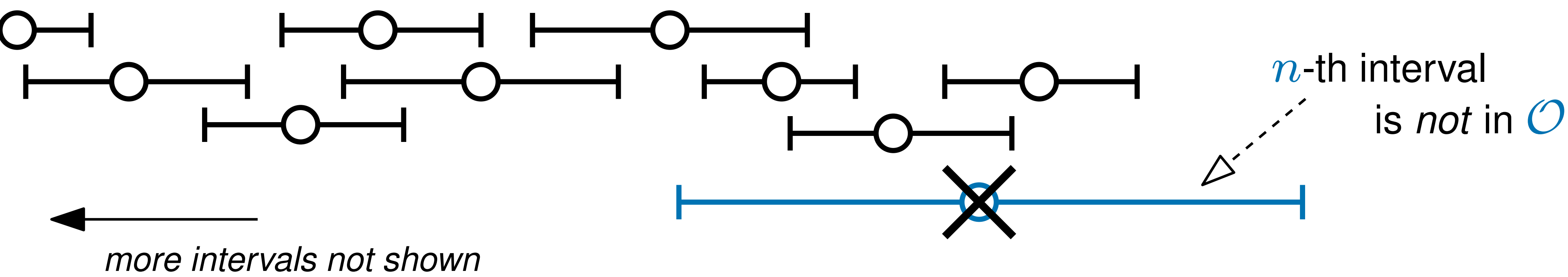
Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 1: The n-th interval is *not* in $\mathcal{O}$

1. Find a recursive formula

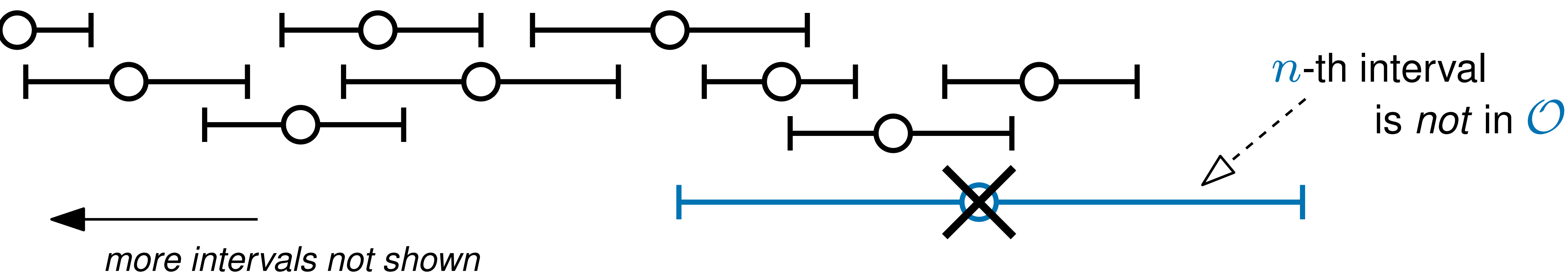
Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 1: The n-th interval is *not* in $\mathcal{O}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...

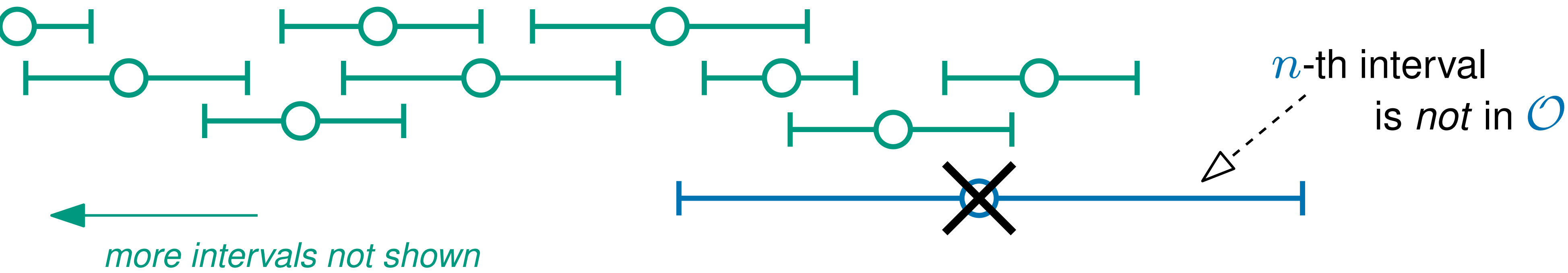


Case 1: The n-th interval is *not* in $\mathcal{O}$

- schedule $\mathcal{O}$ is also an optimal schedule for the problem
with the input consisting of intervals $\{1, 2, 3 \ldots, n-1\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...

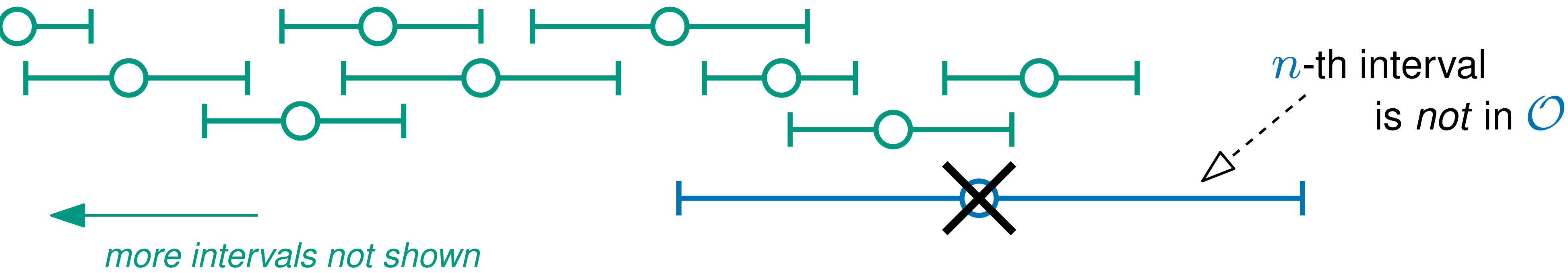


Case 1: The n-th interval is *not* in $\mathcal{O}$

- schedule $\mathcal{O}$ is also an optimal schedule for the problem with the input consisting of intervals $\{1, 2, 3 \ldots, n-1\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



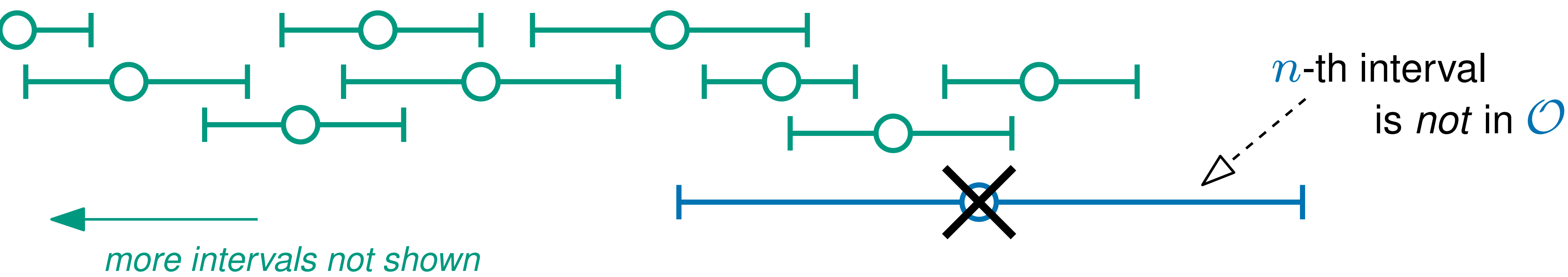
Case 1: The n-th interval is *not* in $\mathcal{O}$

- schedule $\mathcal{O}$ is also an optimal schedule for the problem with the input consisting of intervals $\{1, 2, 3 \ldots, n-1\}$

so, in this case we have that $\text{OPT} = \text{OPT}(n-1)$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 1: The n-th interval is *not* in $\mathcal{O}$

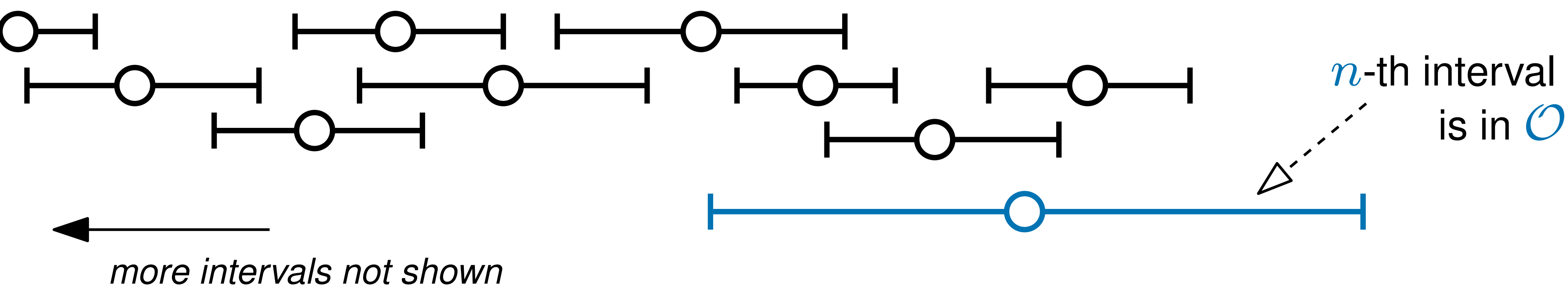
- schedule $\mathcal{O}$ is also an optimal schedule for the problem with the input consisting of intervals $\{1, 2, 3 \ldots, n-1\}$

so, in this case we have that $\text{OPT} = \text{OPT}(n-1)$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...

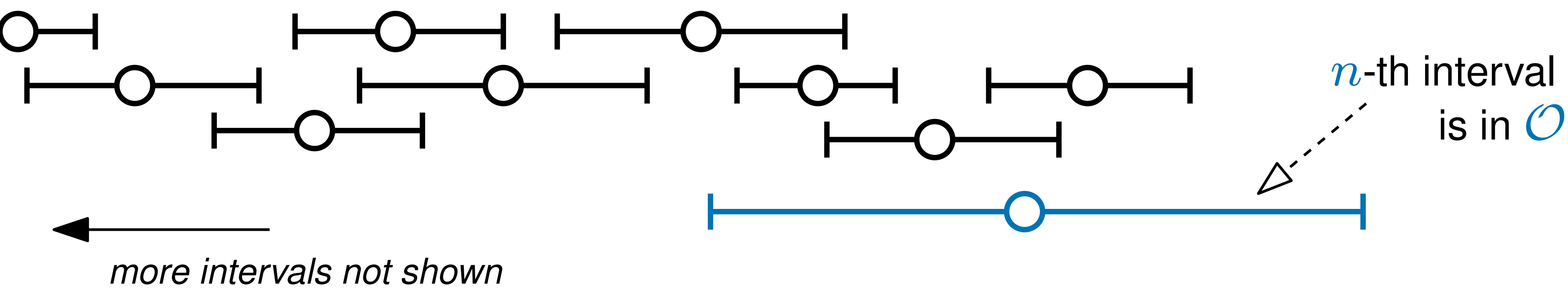


Case 2: The n-th interval is in $\mathcal{O}$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



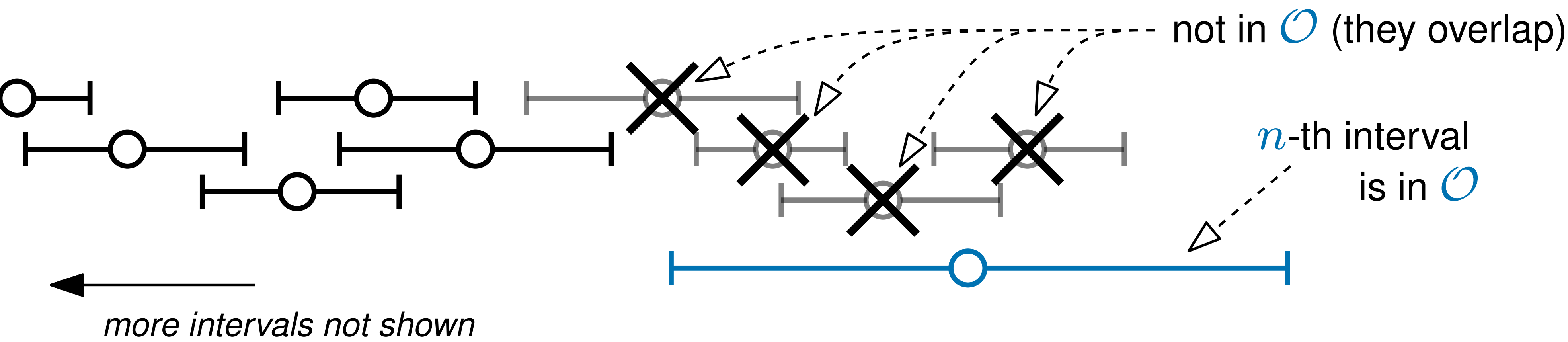
Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

Notation: $\mathrm{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT…



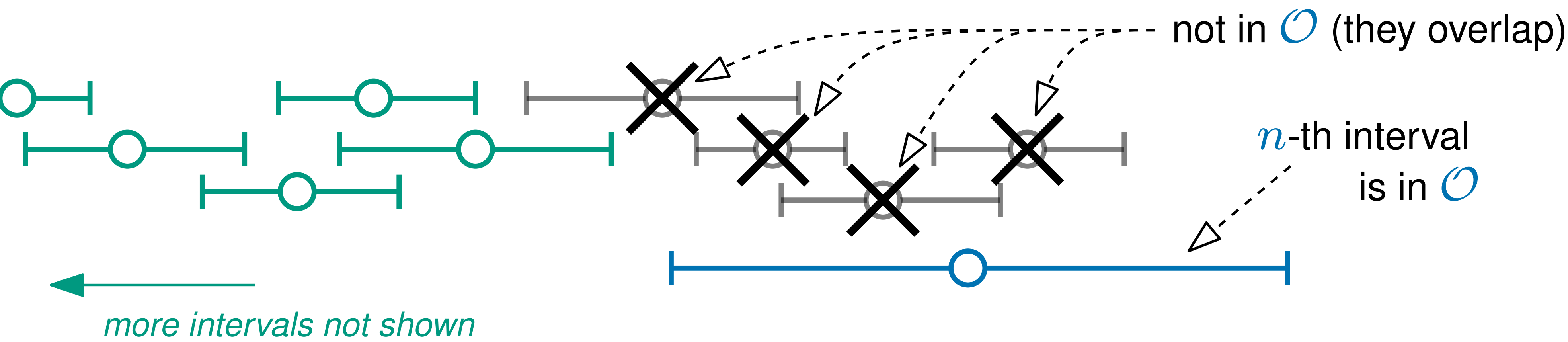
Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



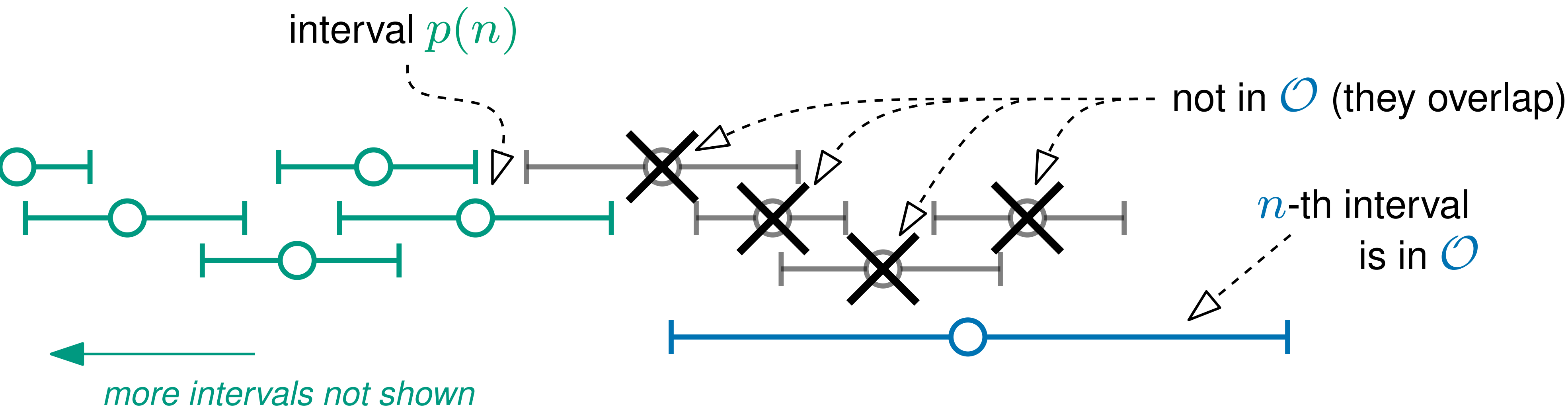
Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

Notation: $\mathrm{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



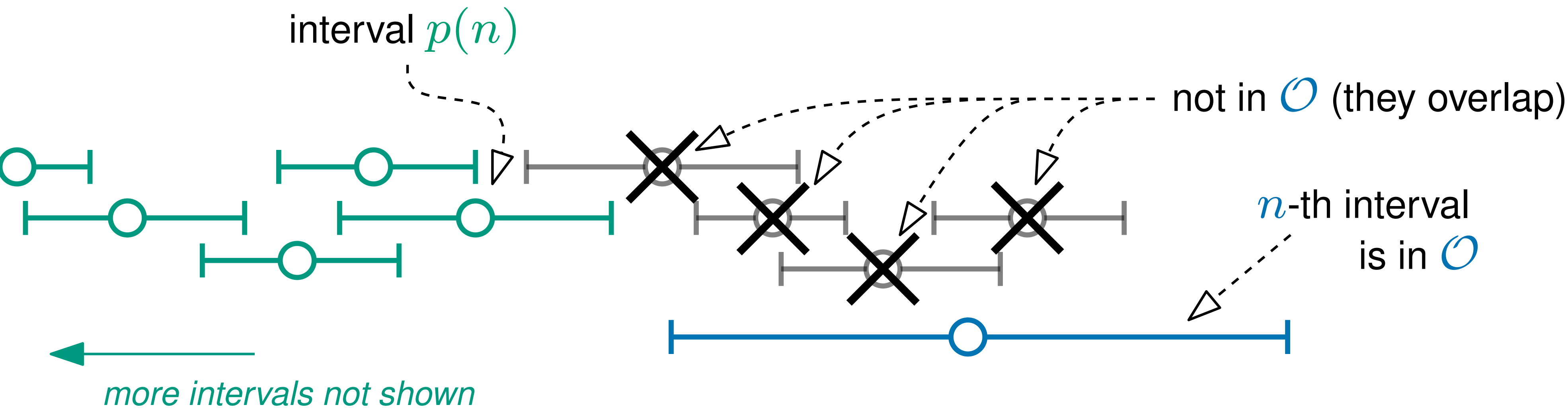
Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

Notation: $\mathsf{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 2: The n-th interval is in $\mathcal{O}$

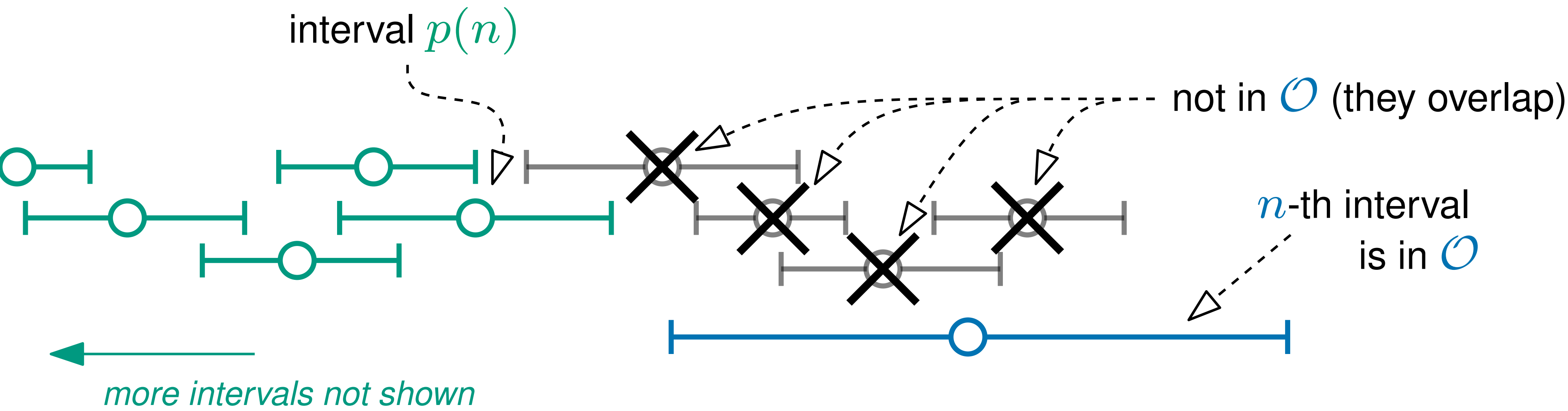
The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

(the ones which don't overlap the n-th interval)

Notation: $\mathrm{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

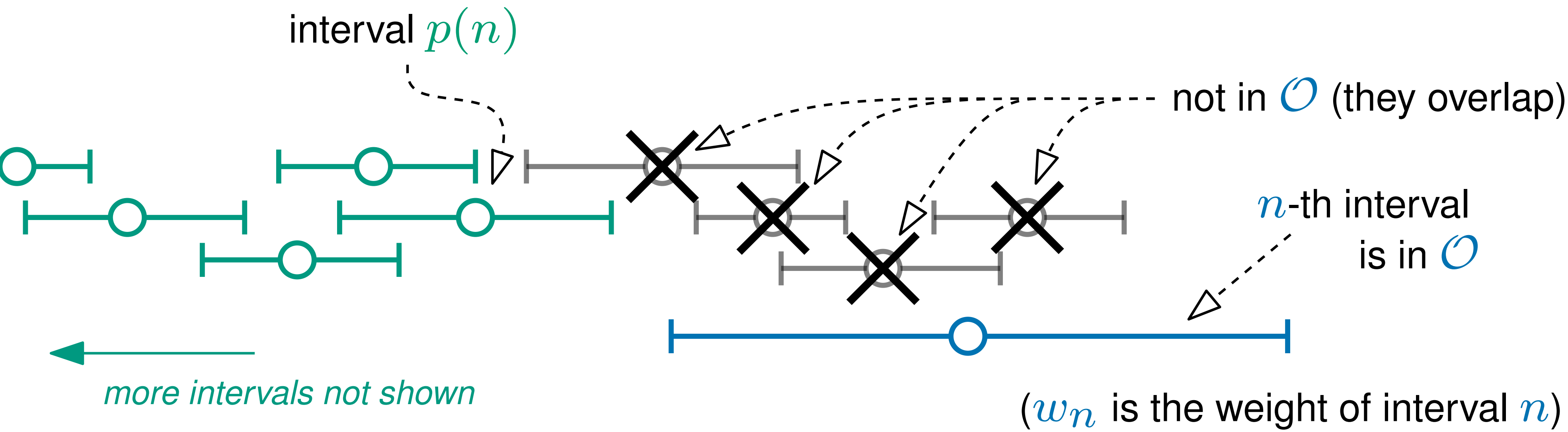
(the ones which don't overlap the n-th interval)

Schedule $\mathcal{O}$ with interval n removed gives an optimal schedule for the intervals $\{1, 2, 3 \ldots, p(n)\}$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

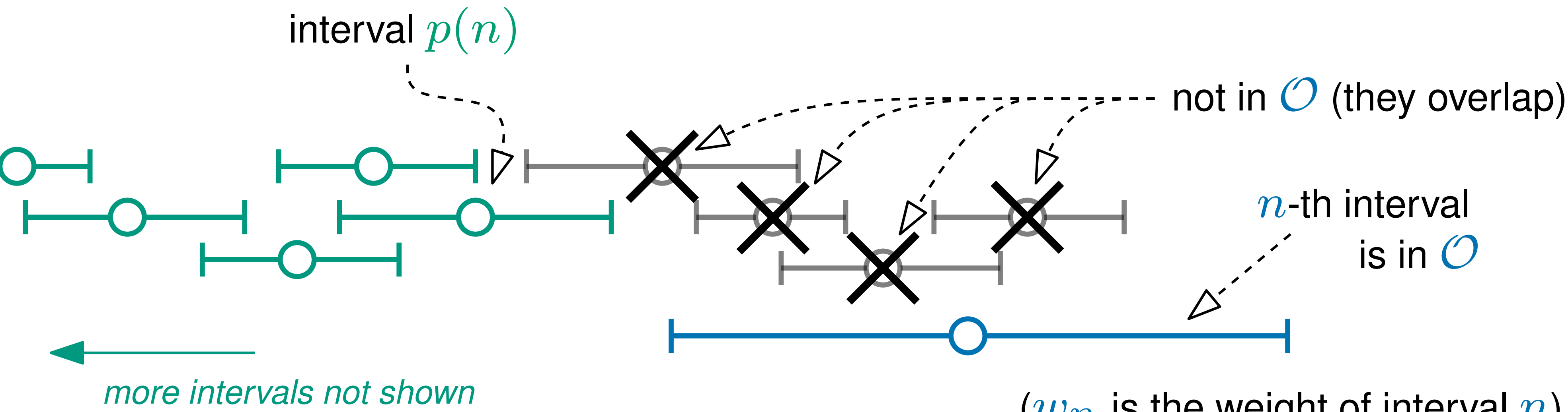
(the ones which don't overlap the n-th interval)

Schedule $\mathcal{O}$ with interval n removed gives an optimal schedule for the intervals $\{1, 2, 3 \ldots, p(n)\}$

Notation: $\mathrm{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



(w_n is the weight of interval n)

Case 2: The n-th interval is in $\mathcal{O}$

The only other intervals which could be in $\mathcal{O}$ are $\{1, 2, 3, \ldots p(n)\}$

(the ones which don't overlap the n-th interval)

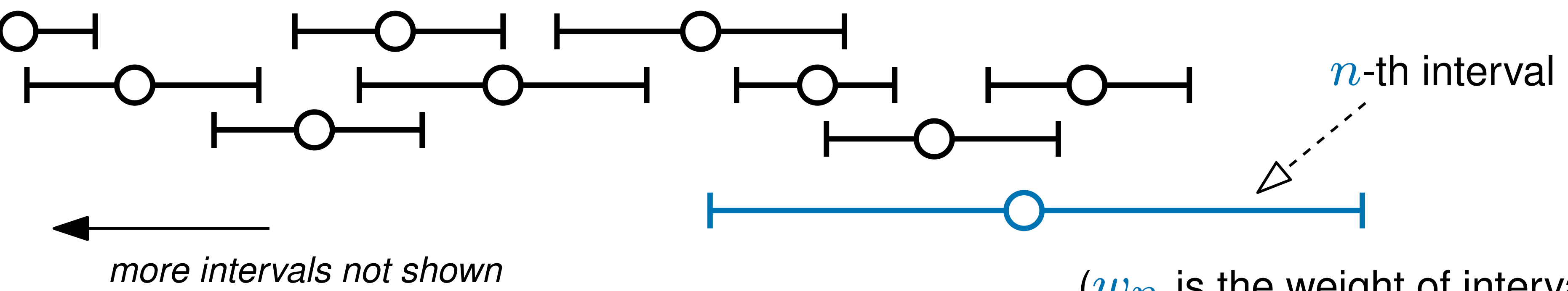
Schedule $\mathcal{O}$ with interval n removed gives an optimal schedule for the intervals $\{1, 2, 3 \ldots, p(n)\}$

so we have that $\text{OPT} = \text{OPT}(p(n)) + w_n$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



(w_n is the weight of interval n)

Case 1: The n-th interval is *not* in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(n-1)$$

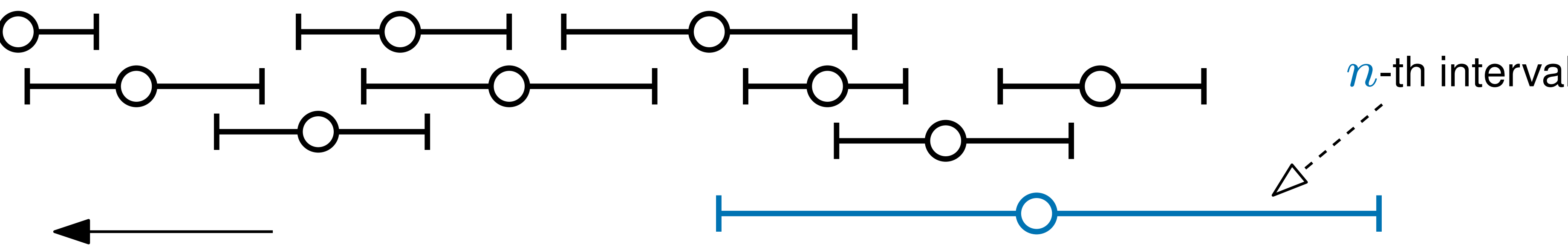
Case 2: The n-th interval is in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(p(n)) + w_n$$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



more intervals not shown

(w_n is the weight of interval n)

Case 1: The n-th interval is *not* in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(n-1)$$

Case 2: The n-th interval is in $\mathcal{O}$

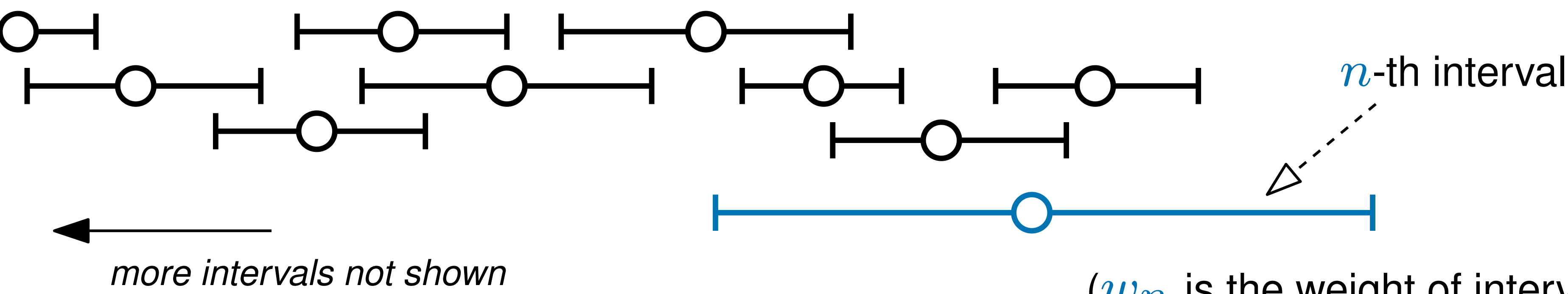
$$\text{OPT} = \text{OPT}(p(n)) + w_n$$

Well, which is it?

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



(w_n is the weight of interval n)

Case 1: The n-th interval is *not* in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(n-1)$$

Case 2: The n-th interval is in $\mathcal{O}$

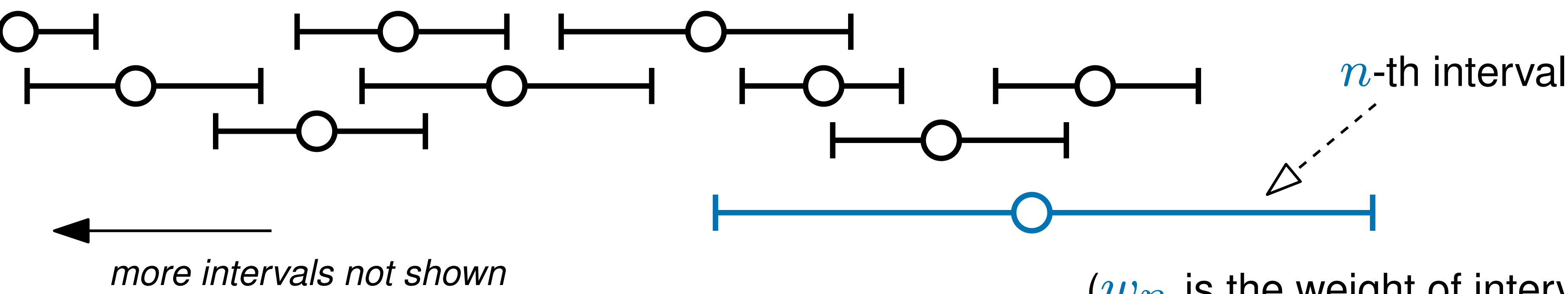
$$\text{OPT} = \text{OPT}(p(n)) + w_n$$

Well, which is it? *It's the bigger one*

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



(w_n is the weight of interval n)

Case 1: The n-th interval is *not* in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(n-1)$$

Case 2: The n-th interval is in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(p(n)) + w_n$$

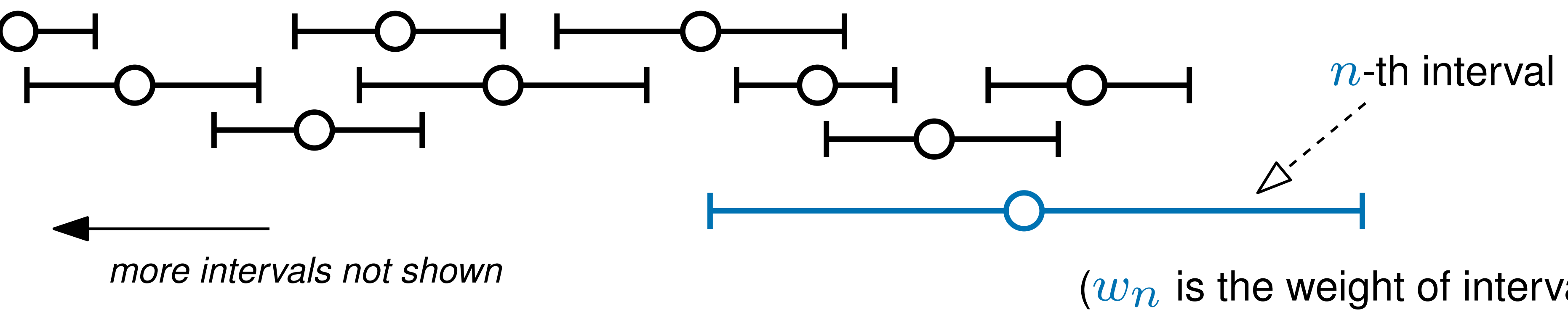
Well, which is it? *It's the bigger one*

$$\text{OPT} = \max(\text{OPT}(n-1), \text{OPT}(p(n)) + w_n)$$

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight OPT...



Case 1: The n-th interval is *not* in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(n-1)$$

Case 2: The n-th interval is in $\mathcal{O}$

$$\text{OPT} = \text{OPT}(p(n)) + w_n$$

Well, which is it? *It's the bigger one*

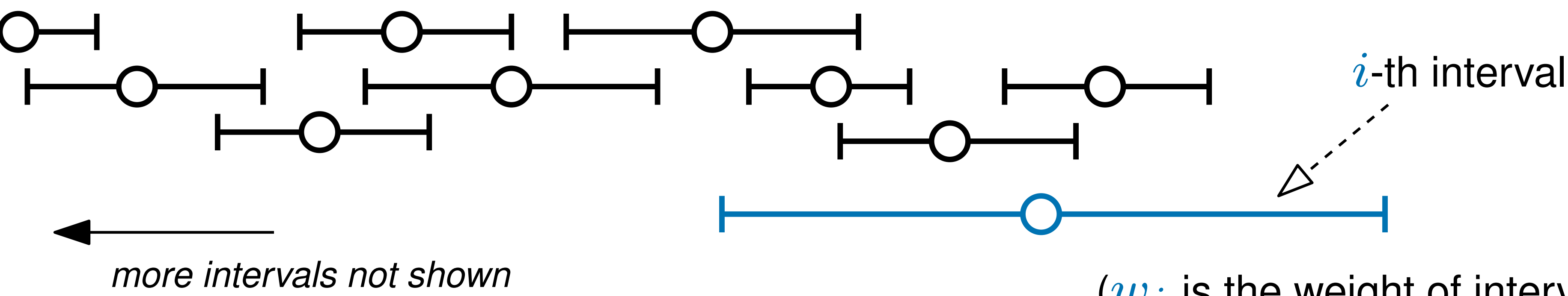
$$\text{OPT} = \max(\text{OPT}(n-1), \text{OPT}(p(n)) + w_n)$$

(they both always give viable schedules)

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

1. Find a recursive formula

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3 \ldots, i\}$ with weight $\text{OPT}(i)$...



(w_i is the weight of interval i)

Case 1: The i-th interval is *not* in $\mathcal{O}$

$$\text{OPT}(i) = \text{OPT}(i-1)$$

Case 2: The i-th interval is in $\mathcal{O}$

$$\text{OPT}(i) = \text{OPT}(p(i)) + w_i$$

Well, which is it? *It's the bigger one*

$$\text{OPT}(i) = \max(\text{OPT}(i-1), \text{OPT}(p(i)) + w_i)$$

(they both always give viable schedules)

Notation: $\text{OPT}(i)$ is the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

2. Write down a recursive algorithm

Once again, we can use the recursive formula to get a recursive algorithm...

```
WIS(i)
```

```
If (i = 0)
   Return 0
Return max(WIS(i - 1), WIS(p(i)) + w_i)
```

$\texttt{WIS}(i)$ computes the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

Therefore, $\texttt{WIS}(n)$ gives the weight of the optimal schedule (for the full problem)

2. Write down a recursive algorithm

Once again, we can use the recursive formula to get a recursive algorithm...

```
WIS(i)
```

```
If (i = 0)
  Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

$\texttt{WIS}(i)$ computes the weight of an optimal schedule for intervals $\{1, 2, 3, \ldots, i\}$

Therefore, $\texttt{WIS}(n)$ gives the weight of the optimal schedule (for the full problem)

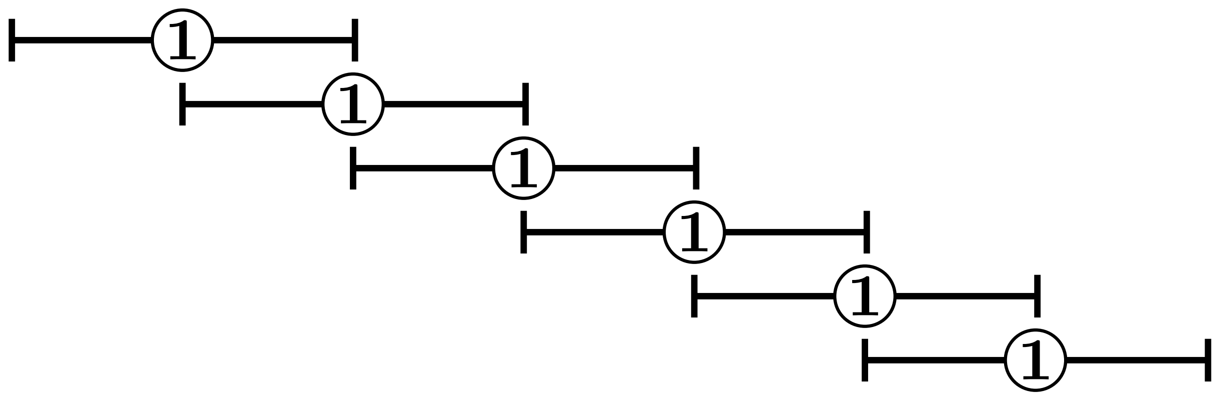
What is the time complexity of this algorithm?

How efficient is the recursive algorithm?

$\texttt{WIS}(i)$

$\texttt{If}\ (i = 0)$
$\quad\texttt{Return}\ 0$
$\texttt{Return}\ \max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

consider this simple input with $n = 6$

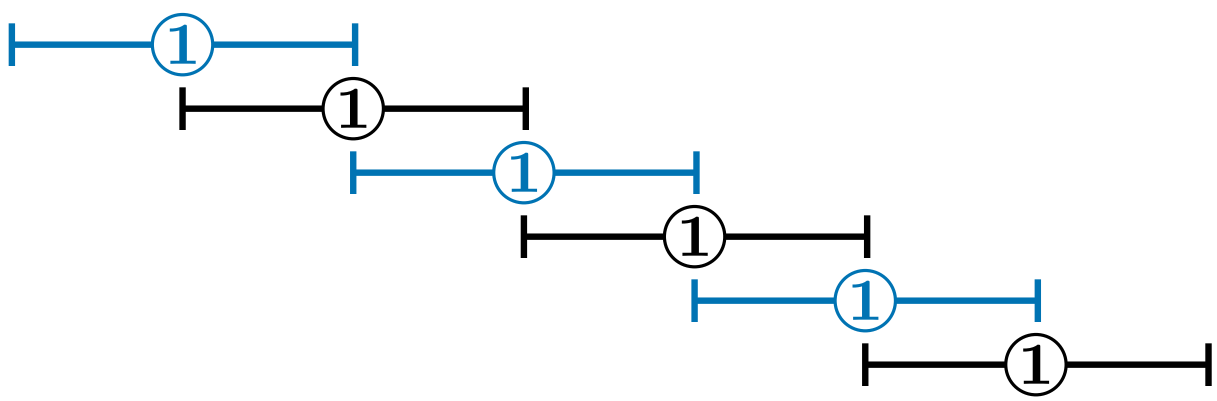


How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
   Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

consider this simple input with $n = 6$



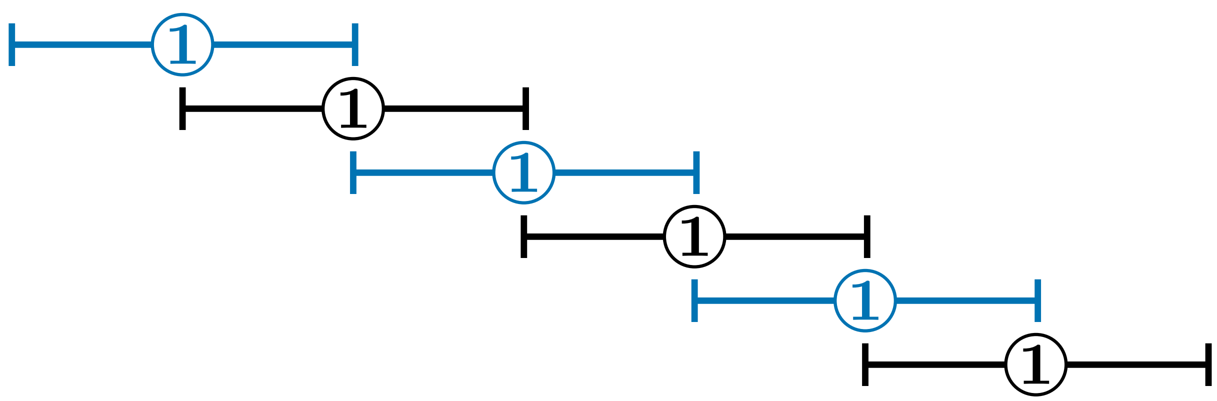
(the best schedule has weight 3)

How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
    Return 0
Return max (WIS(i − 1), WIS(p(i)) + w_i)
```

consider this simple input with $n = 6$



(the best schedule has weight 3)

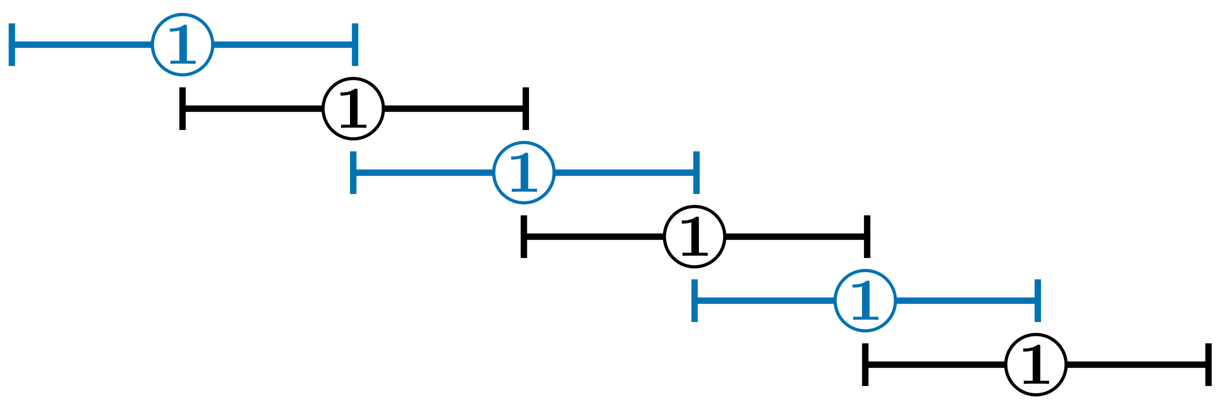
further, for all i, $p(i) = i - 2$

How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
  Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

consider this simple input with $n = 6$



(the best schedule has weight 3)

further, for all i, $p(i) = i - 2$

so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

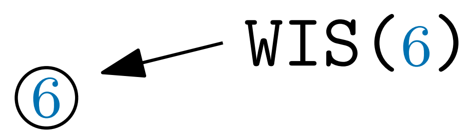
$\texttt{WIS}(i)$

$\texttt{If}\ (i = 0)$
$\quad\texttt{Return}\ 0$
$\texttt{Return}\ \max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

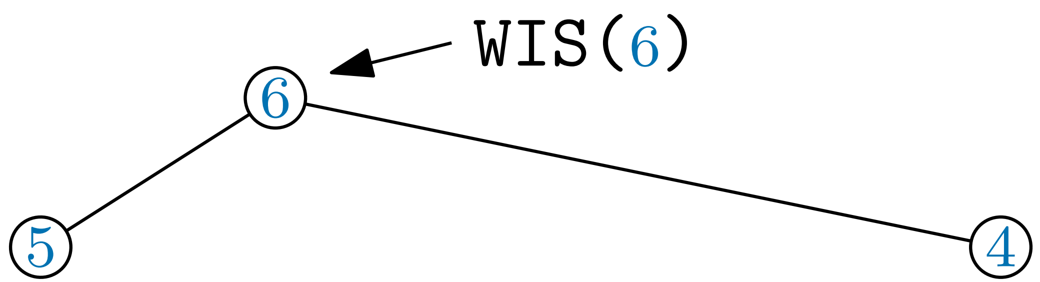
$\texttt{WIS}(i)$
$\texttt{If}\ \ (i = 0)$ $\quad\texttt{Return}\ \ 0$ $\texttt{Return}\ \ \max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$



so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

$\texttt{WIS}(i)$
$\texttt{If}\ (i = 0)$
$\quad\texttt{Return}\ 0$
$\texttt{Return}\ \max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

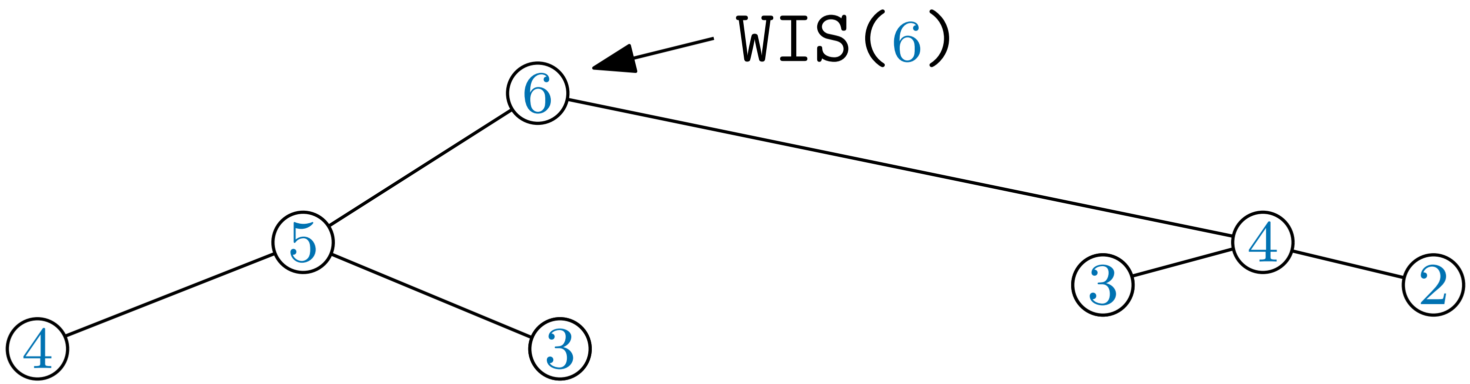


so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

```
WIS(i)
```

If $(i = 0)$
  Return 0
Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

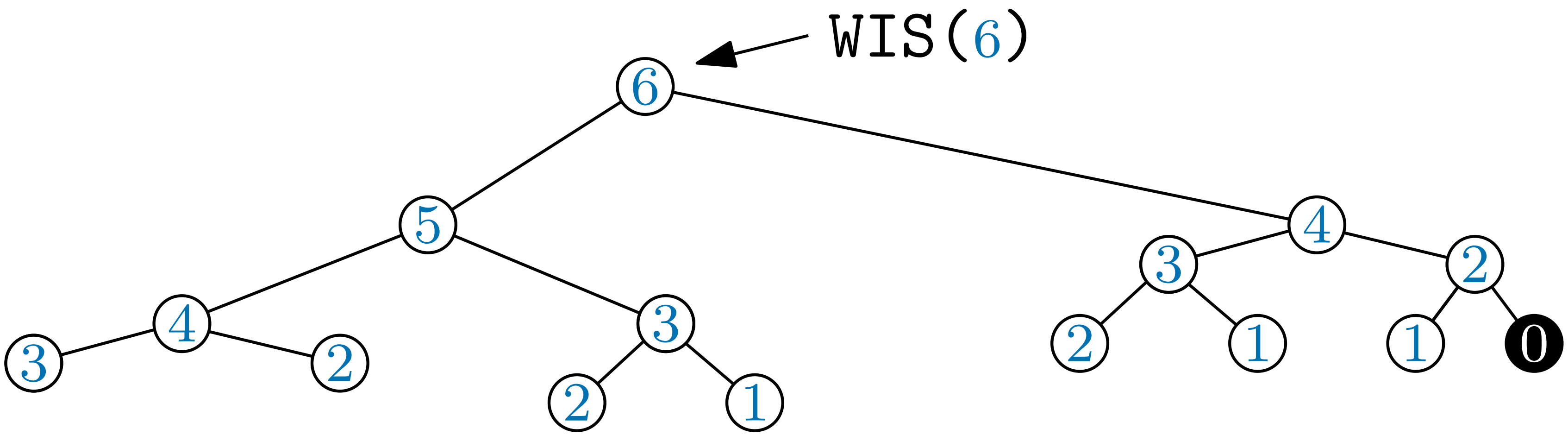


so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

```
WIS(i)
```

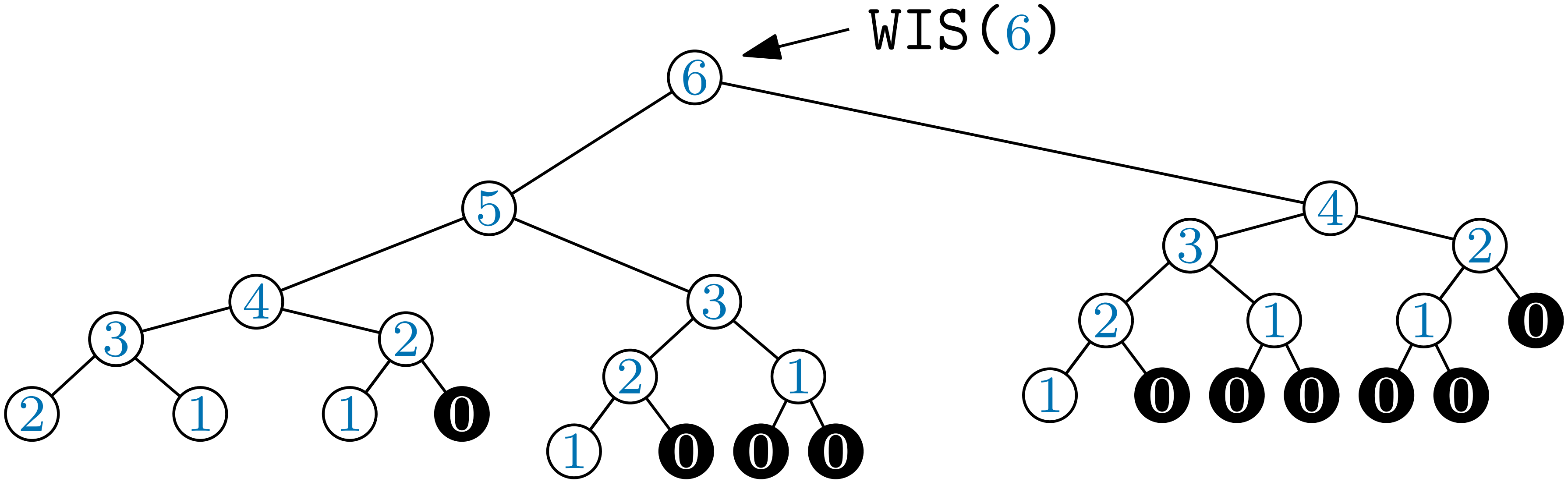
```
If (i = 0)
  Return 0
Return max (WIS(i − 1), WIS(p(i)) + w_i)
```



so WIS(i) makes recursive calls to WIS($i-1$) and WIS($i-2$)

How efficient is the recursive algorithm?

`WIS(`i`)`
`If` $(i = 0)$
`Return` 0
`Return` $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

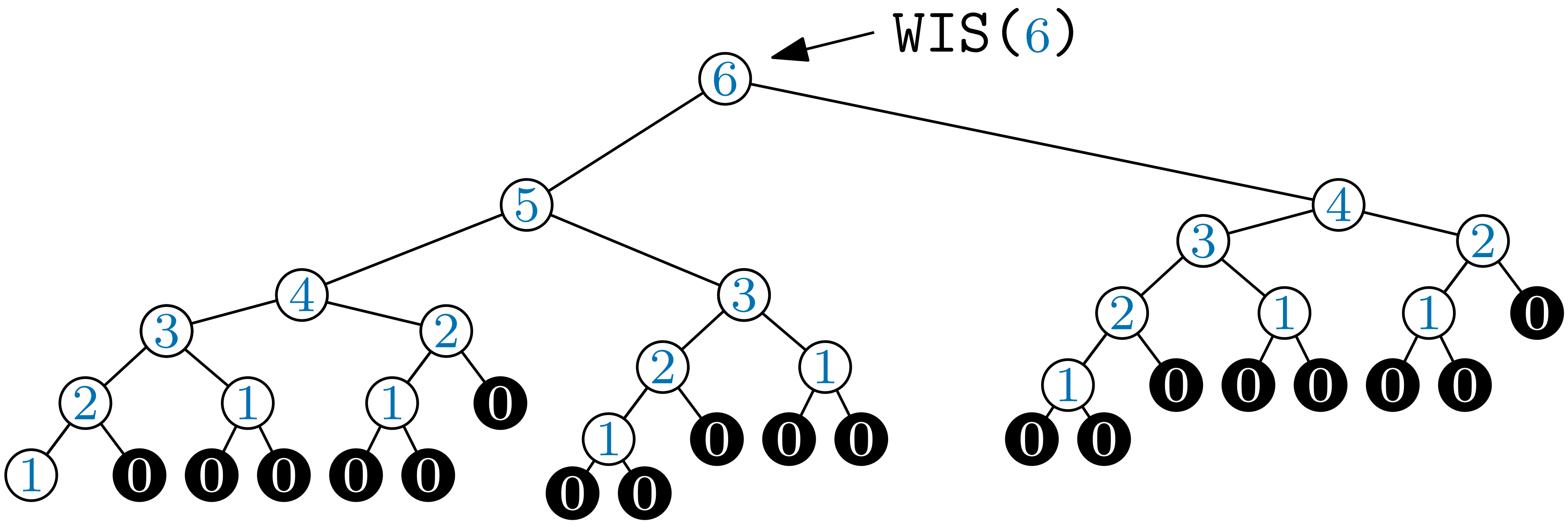


so `WIS(`i`)` makes recursive calls to `WIS(`$i-1$`)` and `WIS(`$i-2$`)`

How efficient is the recursive algorithm?

```
WIS(i)
```

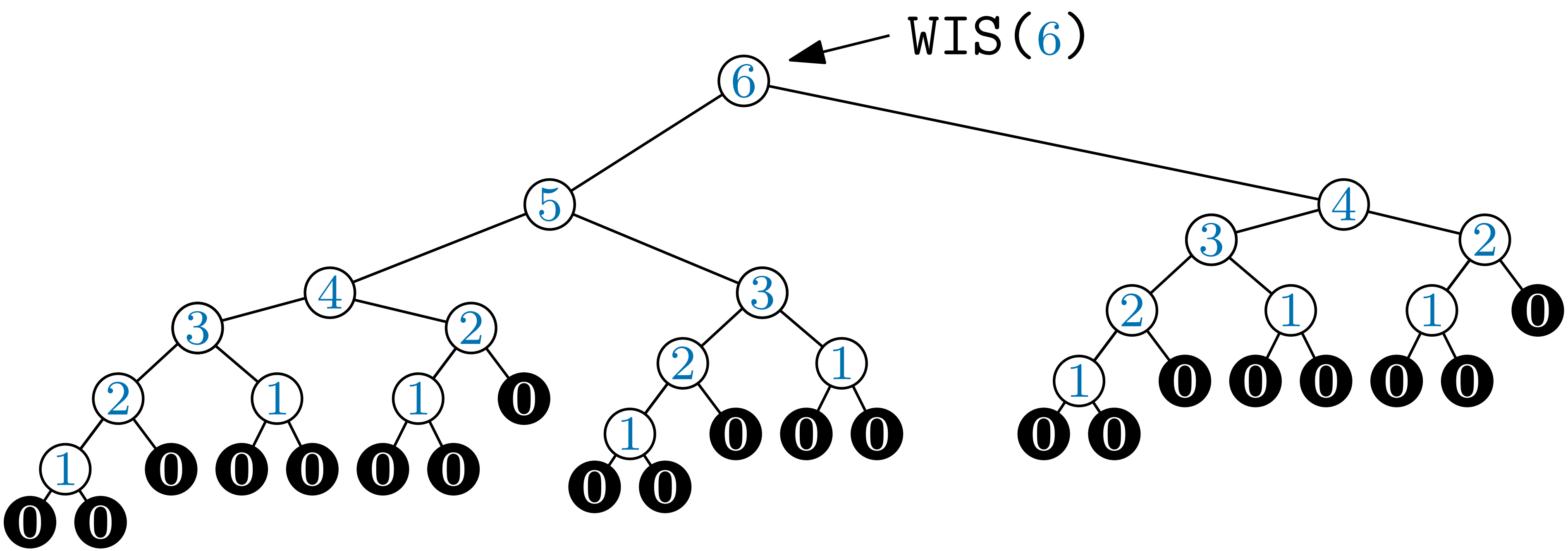
If $(i = 0)$
  Return 0
Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$



so $\texttt{WIS}(i)$ makes recursive calls to $\texttt{WIS}(i-1)$ and $\texttt{WIS}(i-2)$

How efficient is the recursive algorithm?

`WIS`(i)
`If` $(i = 0)$ `Return` 0 `Return` $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

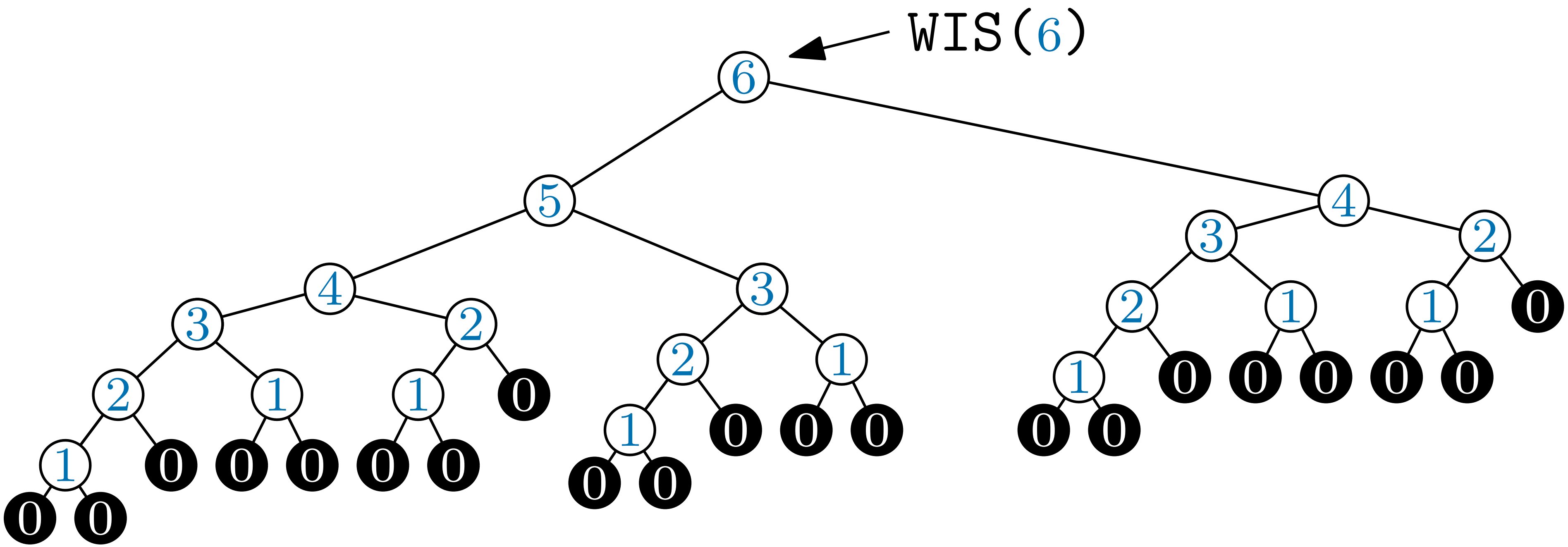


so `WIS`(i) makes recursive calls to `WIS`$(i-1)$ and `WIS`$(i-2)$

How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
    Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```



This doesn't look good (but it does look familiar)

so WIS(i) makes recursive calls to WIS($i-1$) and WIS($i-2$)

How efficient is the recursive algorithm?

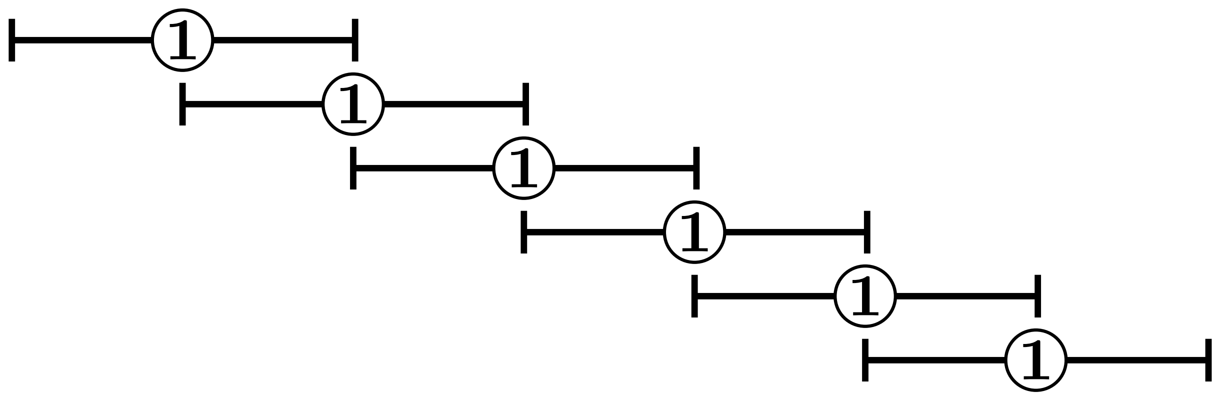
$\texttt{WIS}(i)$

If $(i = 0)$

  Return 0

Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

if we extend this input in the same way...

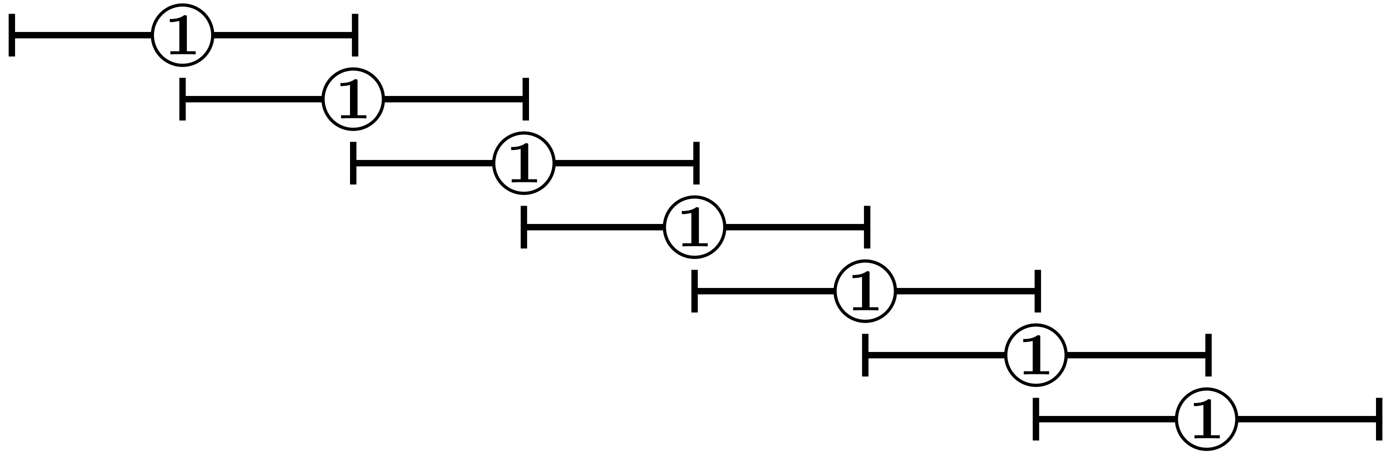


How efficient is the recursive algorithm?

WIS(i)

```
If (i = 0)
    Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

if we extend this input in the same way. . .



How efficient is the recursive algorithm?

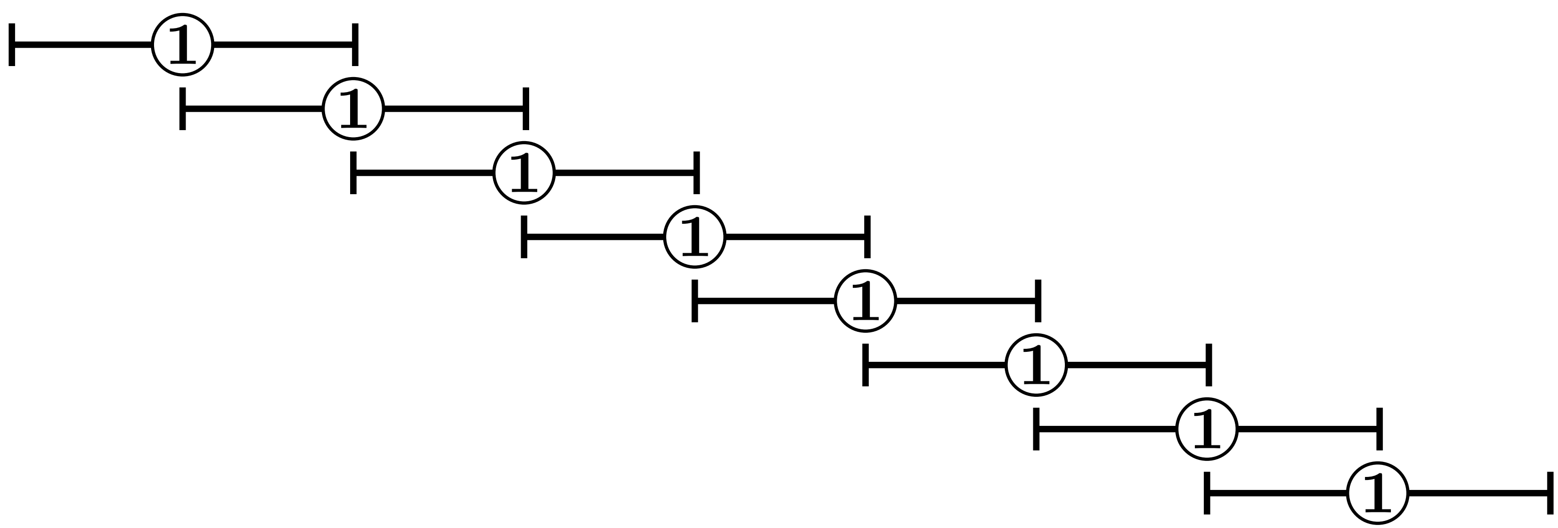
WIS(i)

If $(i = 0)$

  Return 0

Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

if we extend this input in the same way. . .

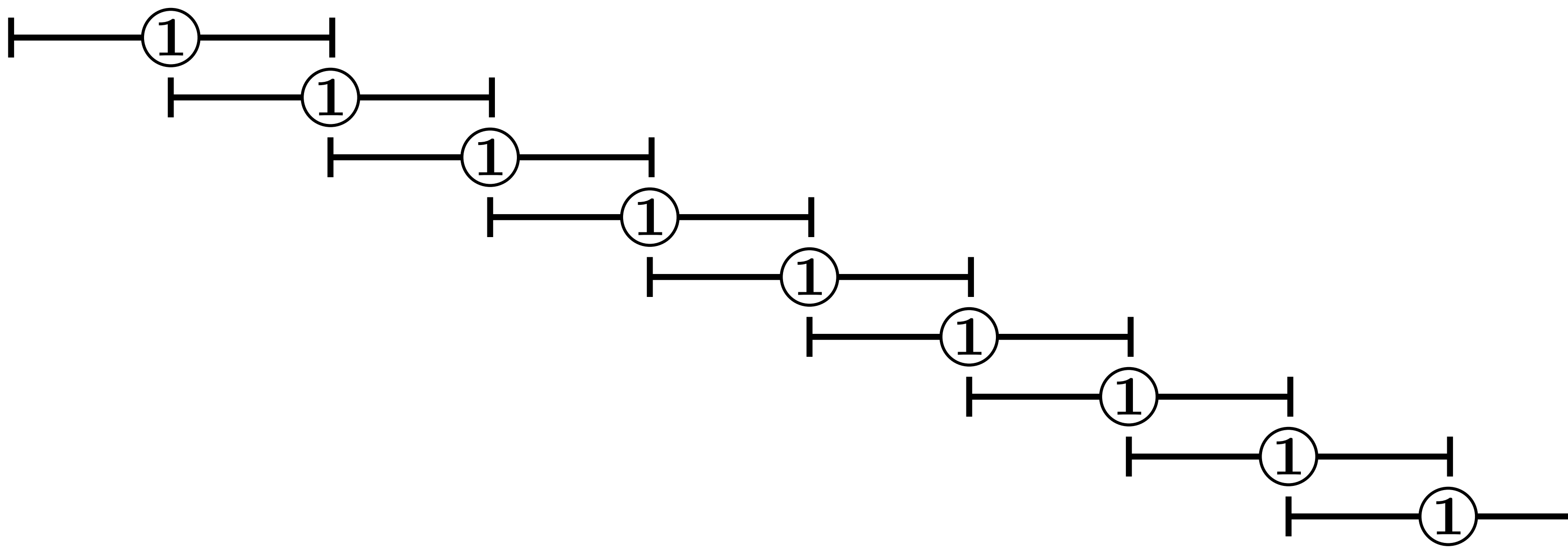


How efficient is the recursive algorithm?

$\texttt{WIS}(i)$

If $(i = 0)$
  Return 0
Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

if we extend this input in the same way...

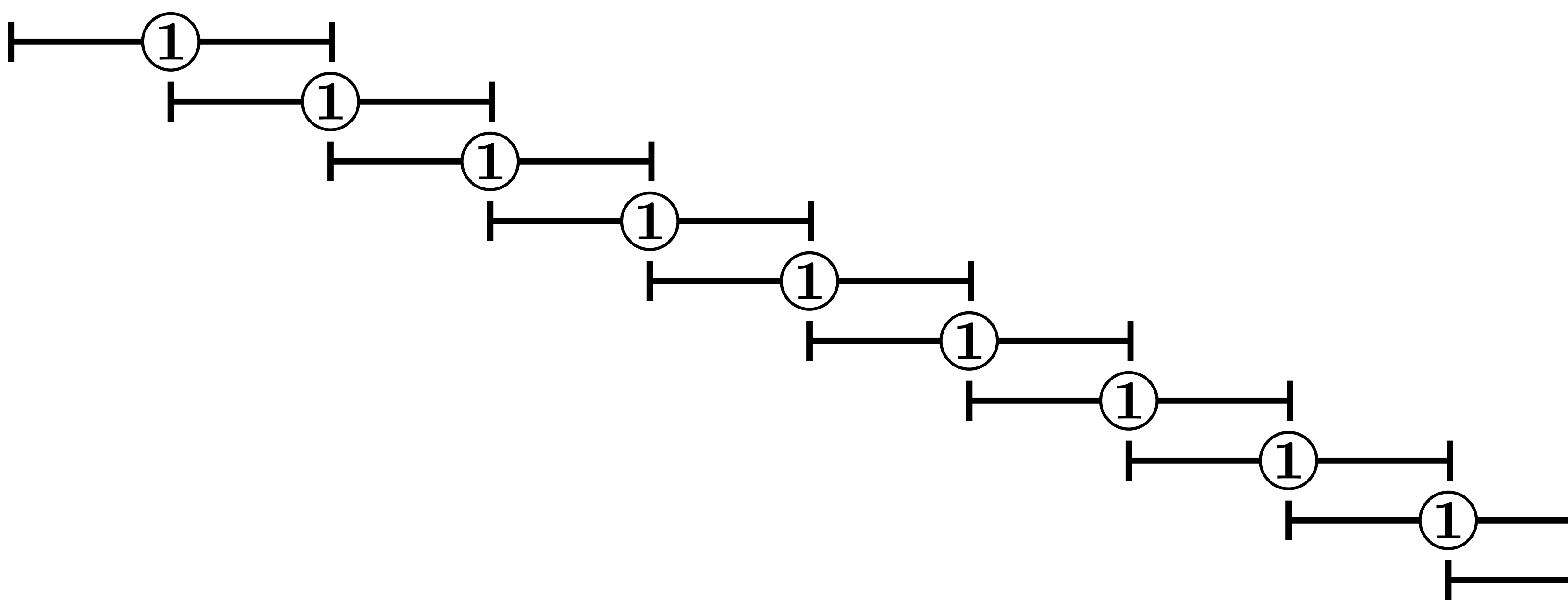


How efficient is the recursive algorithm?

$\texttt{WIS}(i)$

If $(i = 0)$
  Return 0
Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

if we extend this input in the same way...

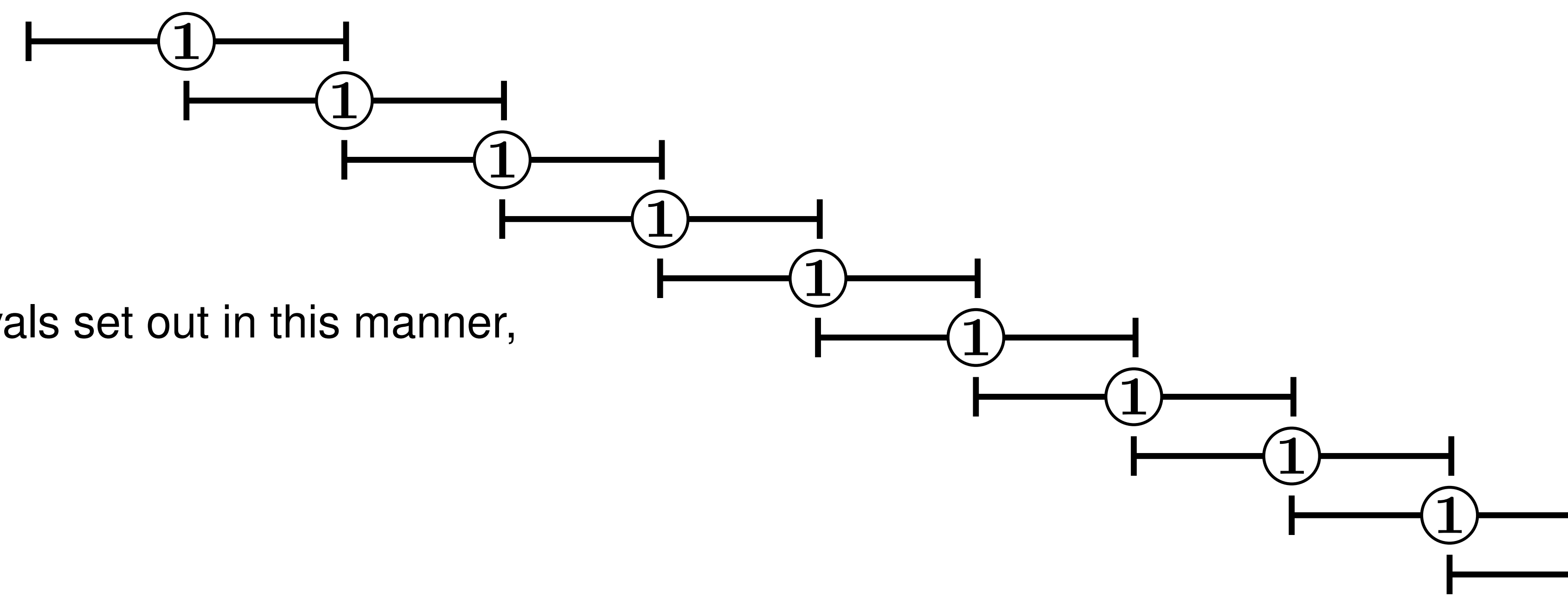


How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
  Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

if we extend this input in the same way...



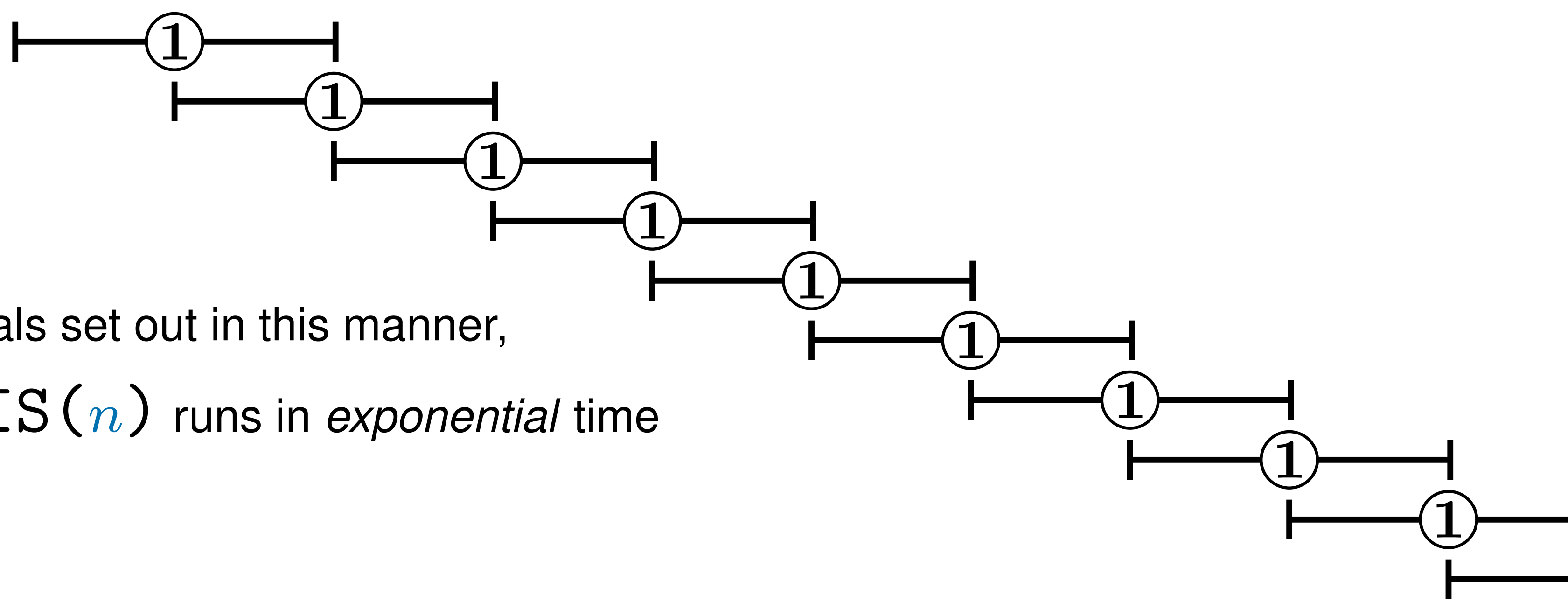
Given n intervals set out in this manner,

How efficient is the recursive algorithm?

WIS(i)

If $(i = 0)$
  Return 0
Return $\max\left(\texttt{WIS}(i-1), \texttt{WIS}(p(i)) + w_i\right)$

if we extend this input in the same way...



Given n intervals set out in this manner,

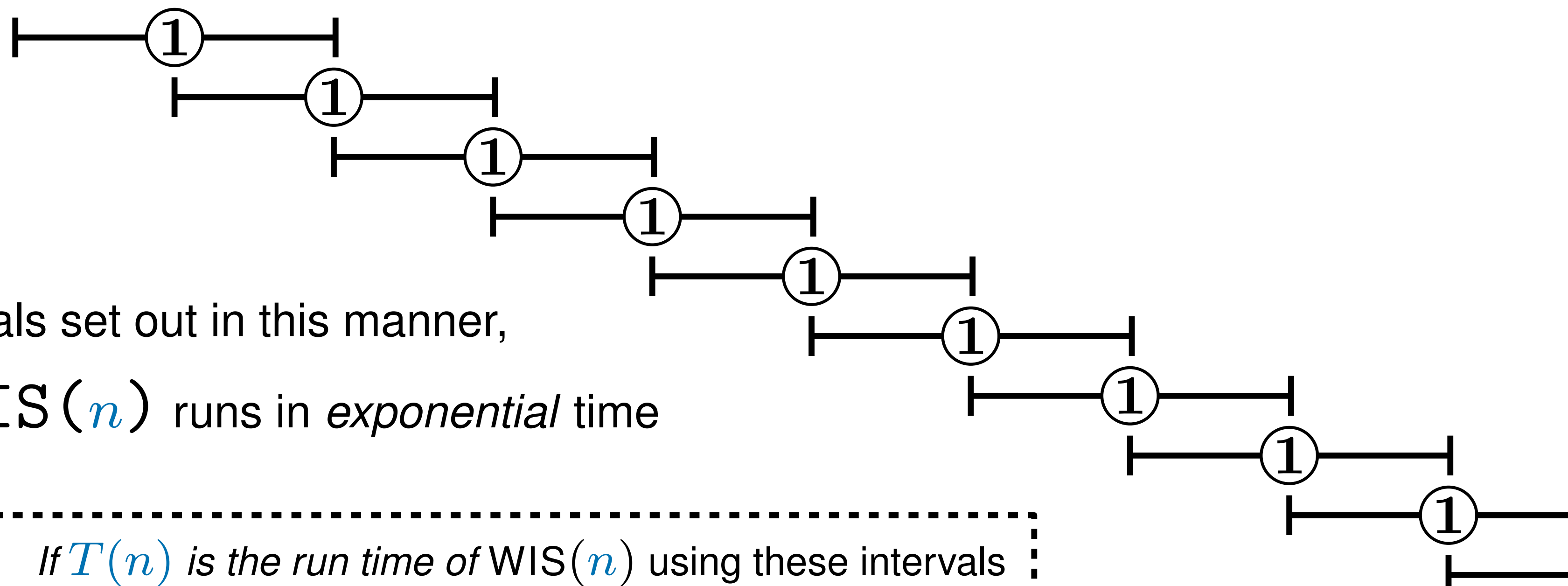
WIS(n) runs in *exponential* time

How efficient is the recursive algorithm?

```
WIS(i)
```

```
If (i = 0)
  Return 0
Return max(WIS(i − 1), WIS(p(i)) + w_i)
```

if we extend this input in the same way...



Given n intervals set out in this manner,

$\texttt{WIS}(n)$ runs in *exponential* time

If $T(n)$ *is the run time of* WIS(n) using these intervals
then $T(n) > 2T(n-2)$

3. Store the solutions to subproblems

```
MemWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max (MemWIS(i − 1), MemWIS(p(i)) + w_i)
Return WIS[i]
```

3. Store the solutions to subproblems

```
MEMWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max (MEMWIS(i − 1), MEMWIS(p(i)) + w_i)
Return WIS[i]
```

In the MEMWIS version of the algorithm we store solutions to previously computed subproblems in an n length array called WIS

3. Store the solutions to subproblems

```
MemWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max (MemWIS(i − 1), MemWIS(p(i)) + w_i)
Return WIS[i]
```

In the MemWIS version of the algorithm
we store solutions to previously computed subproblems
in an n length array called WIS

(we have memoized the algorithm)

3. Store the solutions to subproblems

```
MemWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max (MemWIS(i − 1), MemWIS(p(i)) + w_i)
Return WIS[i]
```

In the MemWIS version of the algorithm

we store solutions to previously computed subproblems

in an n length array called WIS

(we have memoized the algorithm)

Each entry WIS[i] is only computed *once*

3. Store the solutions to subproblems

```
MEMWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max(MEMWIS(i − 1), MEMWIS(p(i)) + w_i)
Return WIS[i]
```

In the $\textsc{MemWIS}$ version of the algorithm

we store solutions to previously computed subproblems

in an n length array called $\texttt{WIS}$

(we have memoized the algorithm)

Each entry $\texttt{WIS}[i]$ is only computed *once*

The time complexity of computing $\textsc{MemWIS}(n)$ is now $O(n)$

3. Store the solutions to subproblems

```
MemWIS(i)
```

```
If (i = 0)
  Return 0
If WIS[i] undefined
  WIS[i] = max (MemWIS(i − 1), MemWIS(p(i)) + w_i)
Return WIS[i]
```

In the MemWIS version of the algorithm
we store solutions to previously computed subproblems
in an n length array called WIS

(we have memoized the algorithm)

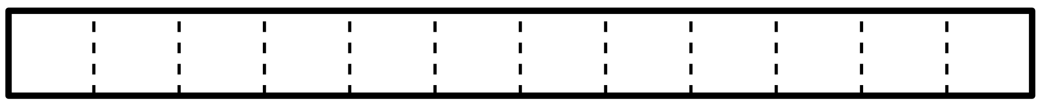
Each entry WIS$[i]$ is only computed *once*

The time complexity of computing MemWIS(n) is now $O(n)$

because every recursion causes an unfilled entry to be filled in the array

The dependency graph

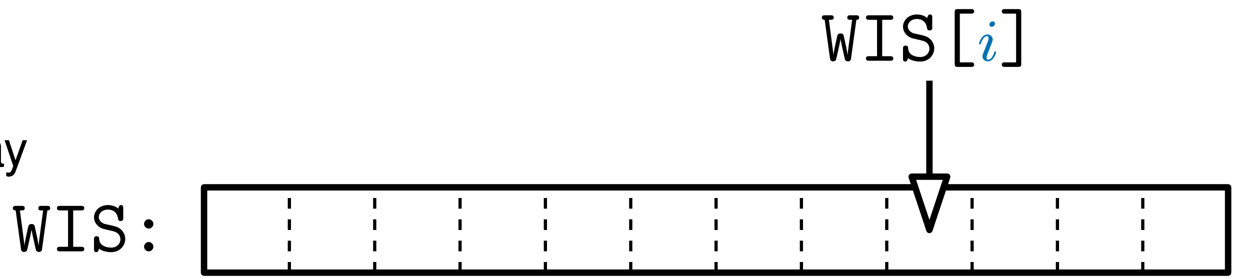
The array

WIS: 

What information do we need to compute $\texttt{WIS}[i]$?

The dependency graph

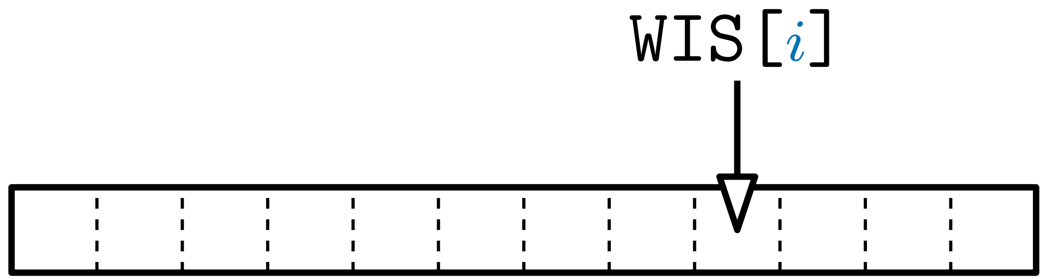
The array



What information do we need to compute WIS[i] ?

The dependency graph

The array
WIS:

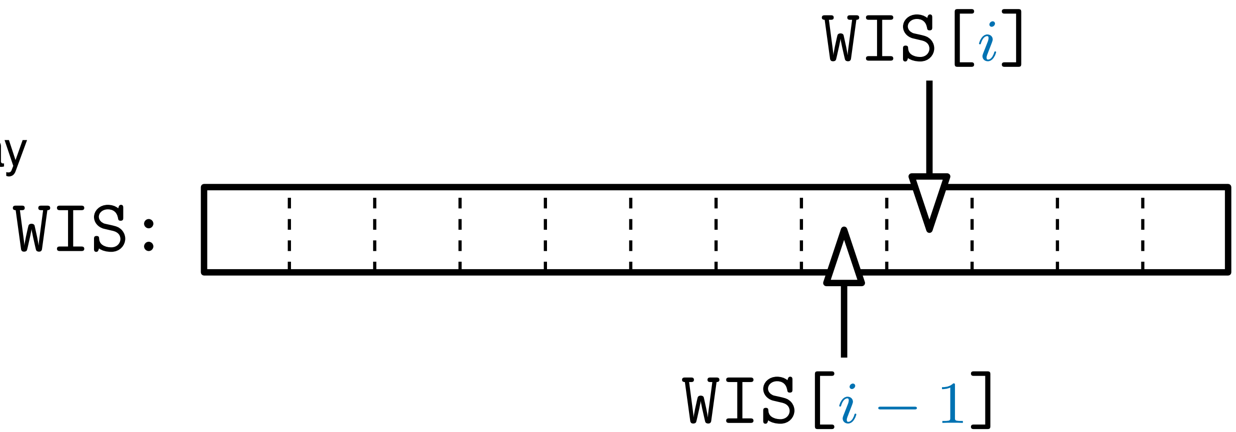


What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

The dependency graph

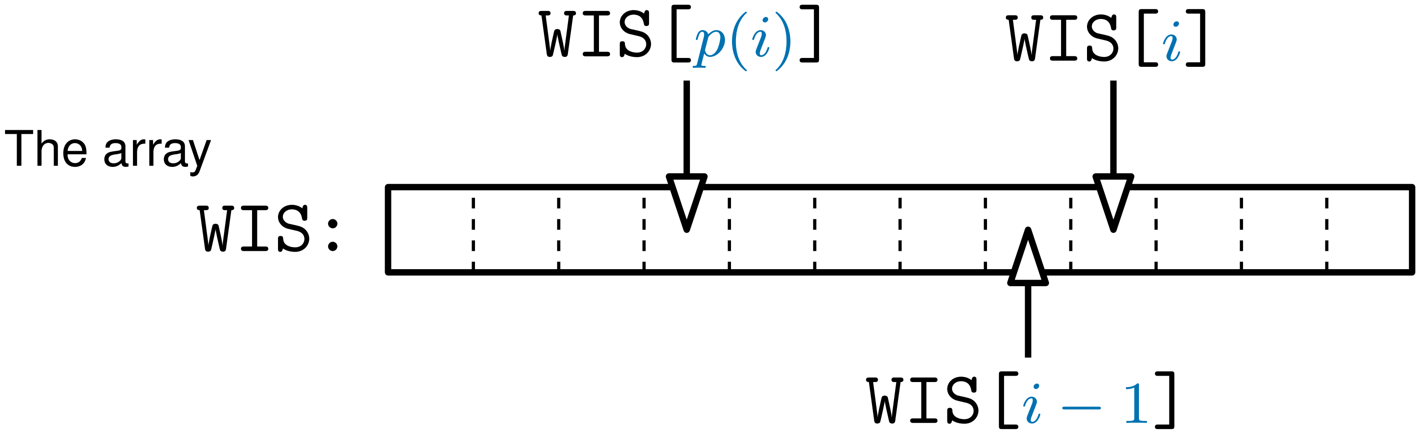
The array



What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

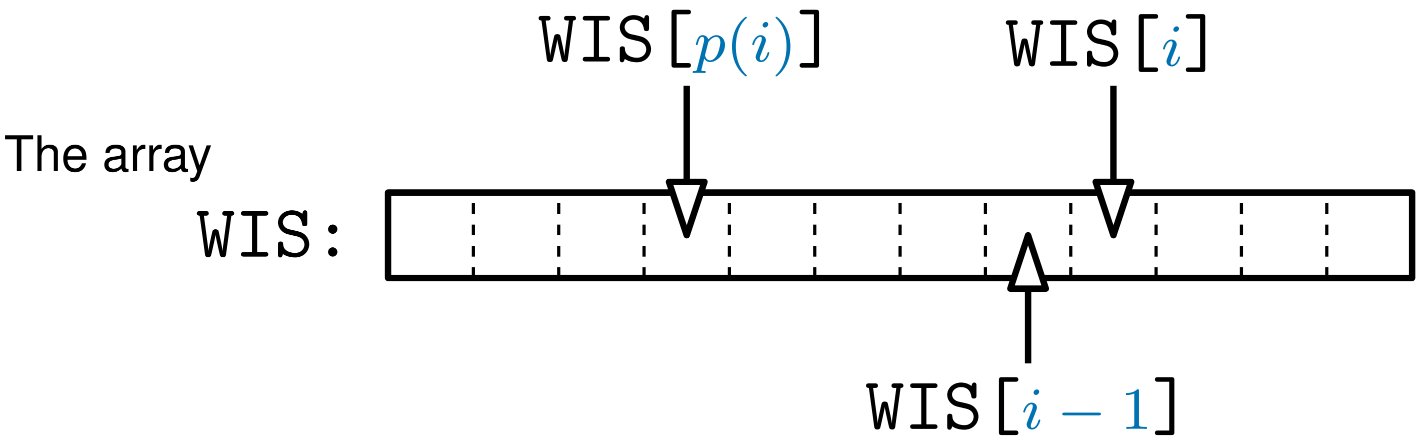
The dependency graph



What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

The dependency graph

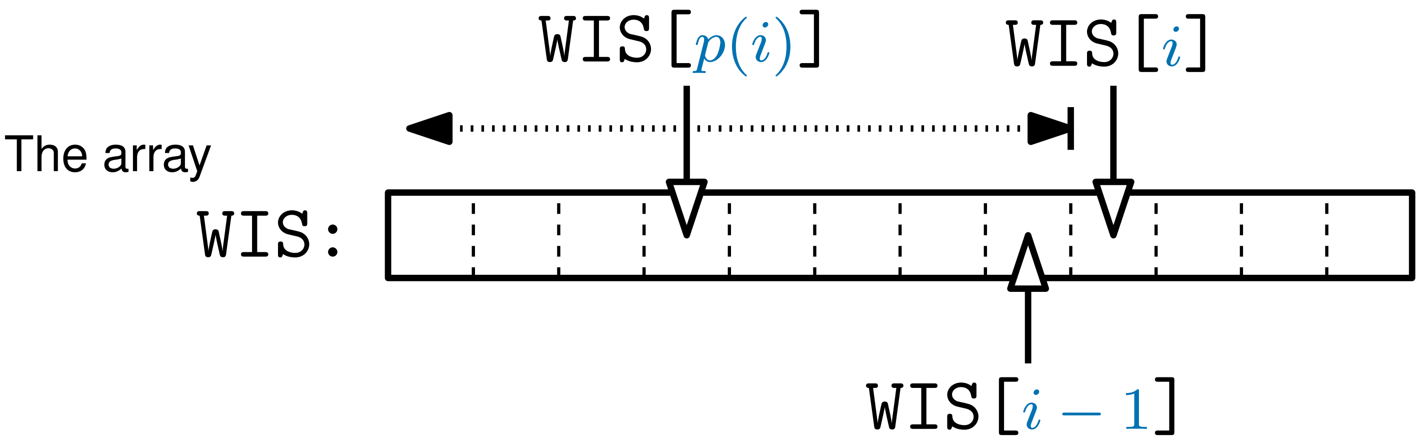


What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

The dependency graph



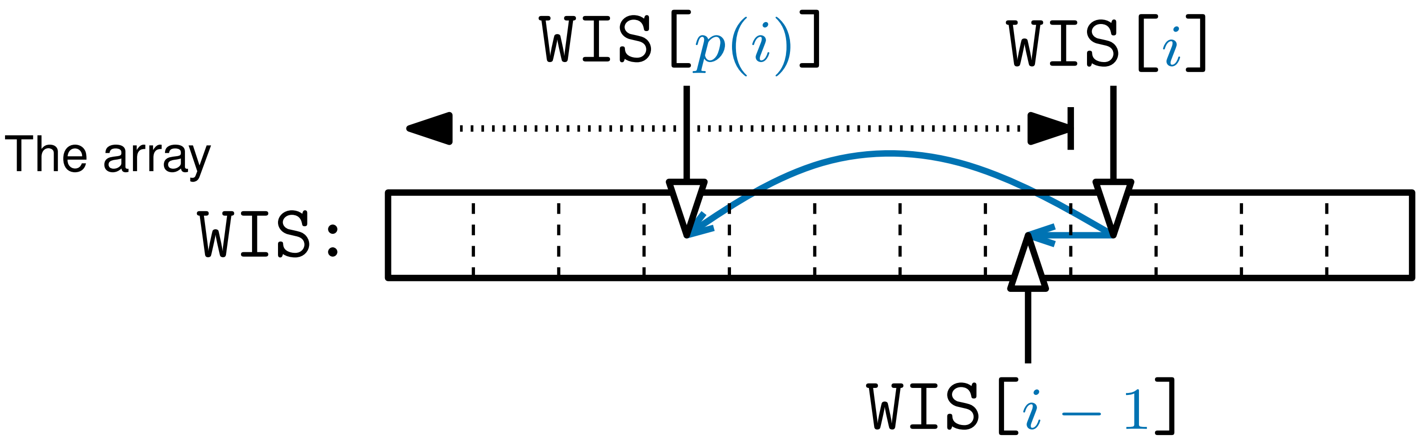
What information do we need to compute $\texttt{WIS}[i]$?

to compute $\texttt{WIS}[i]$ we need $\texttt{WIS}[i-1]$ and $\texttt{WIS}[p(i)]$

both of which are to the *left* of $\texttt{WIS}[i]$

(somewhere)

The dependency graph



What information do we need to compute WIS[i] ?

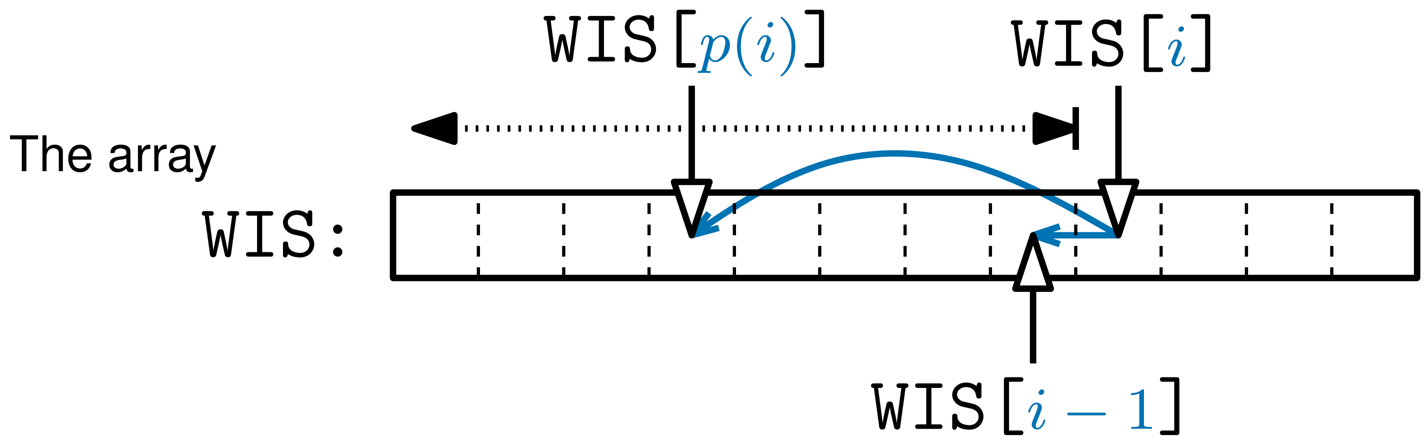
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .



What information do we need to compute $\texttt{WIS}[i]$?

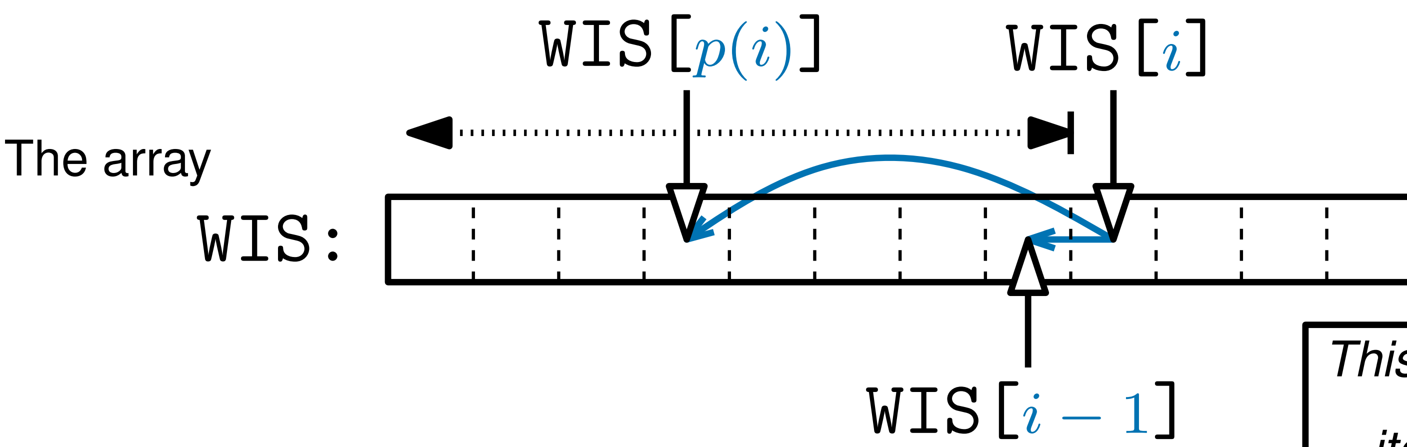
to compute $\texttt{WIS}[i]$ we need $\texttt{WIS}[i-1]$ and $\texttt{WIS}[p(i)]$

both of which are to the *left* of $\texttt{WIS}[i]$

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

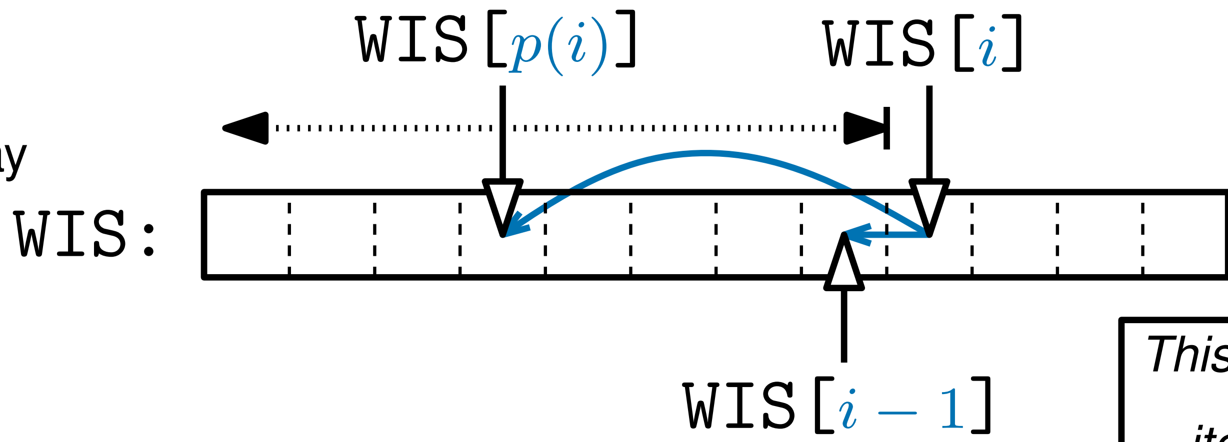
both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left...

The array



This suggests another iterative algorithm

Fill in the array from the left again

What information do we need to compute WIS[i] ?

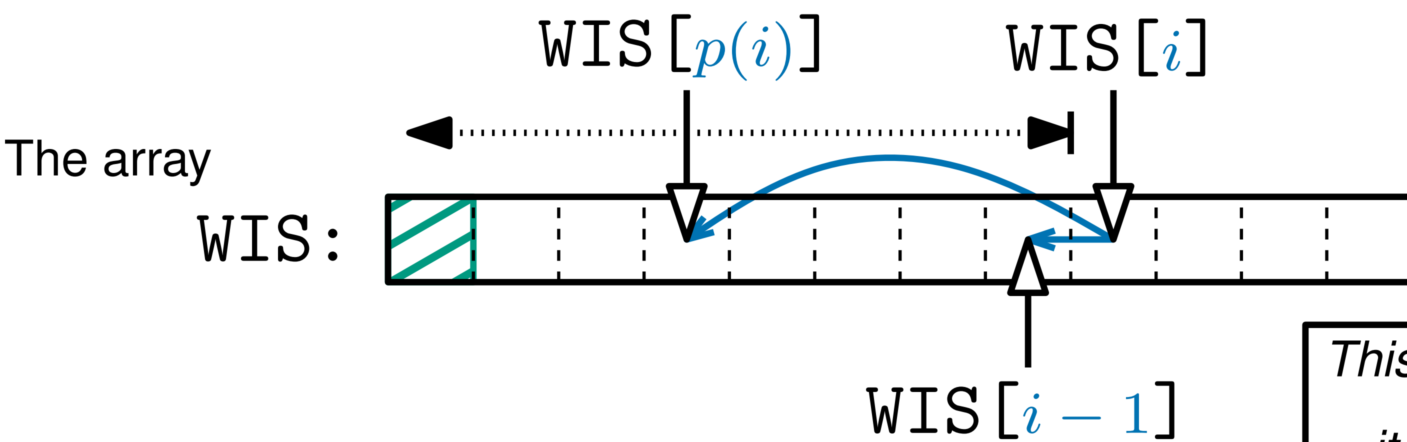
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left...



This suggests another iterative algorithm

What information do we need to compute $\texttt{WIS}[i]$?

Fill in the array from the left again

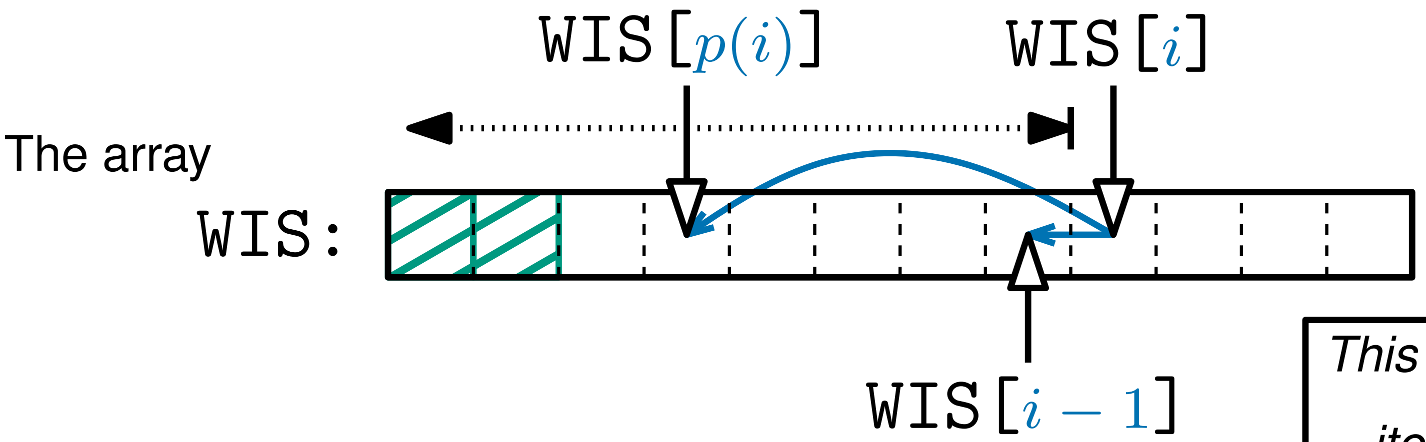
to compute $\texttt{WIS}[i]$ we need $\texttt{WIS}[i-1]$ and $\texttt{WIS}[p(i)]$

both of which are to the *left* of $\texttt{WIS}[i]$

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

What information do we need to compute WIS[i] ?

Fill in the array from the left again

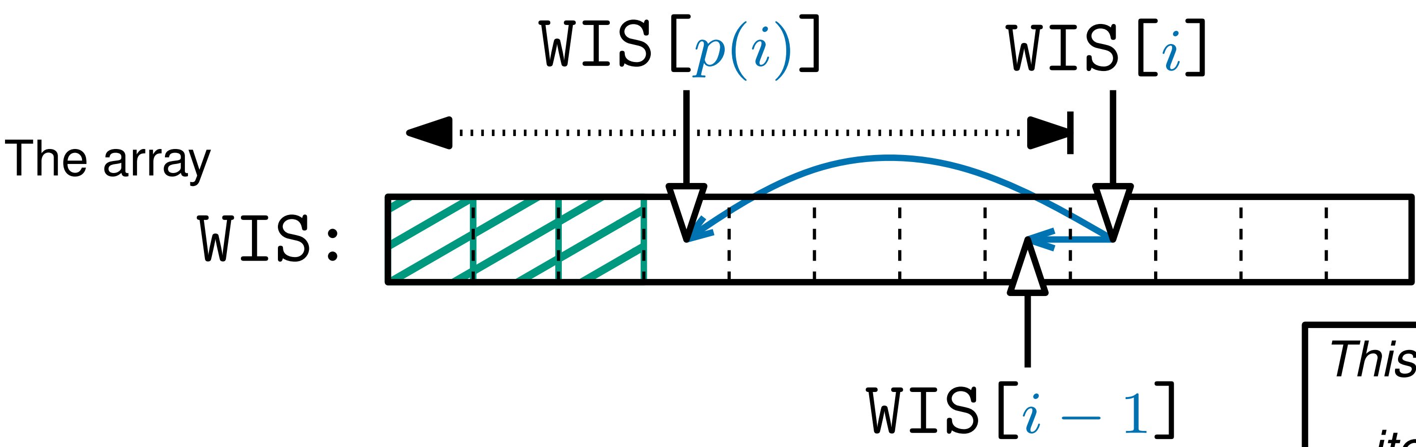
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

What information do we need to compute $\texttt{WIS}[i]$?

Fill in the array from the left again

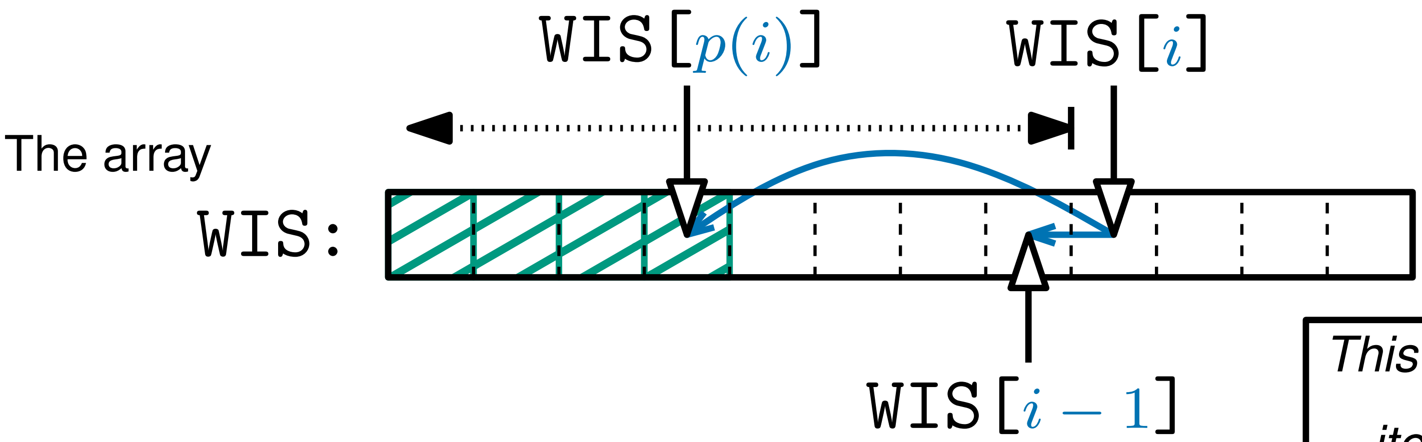
to compute $\texttt{WIS}[i]$ we need $\texttt{WIS}[i-1]$ and $\texttt{WIS}[p(i)]$

both of which are to the *left* of $\texttt{WIS}[i]$

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

What information do we need to compute WIS[i] ?

Fill in the array from the left again

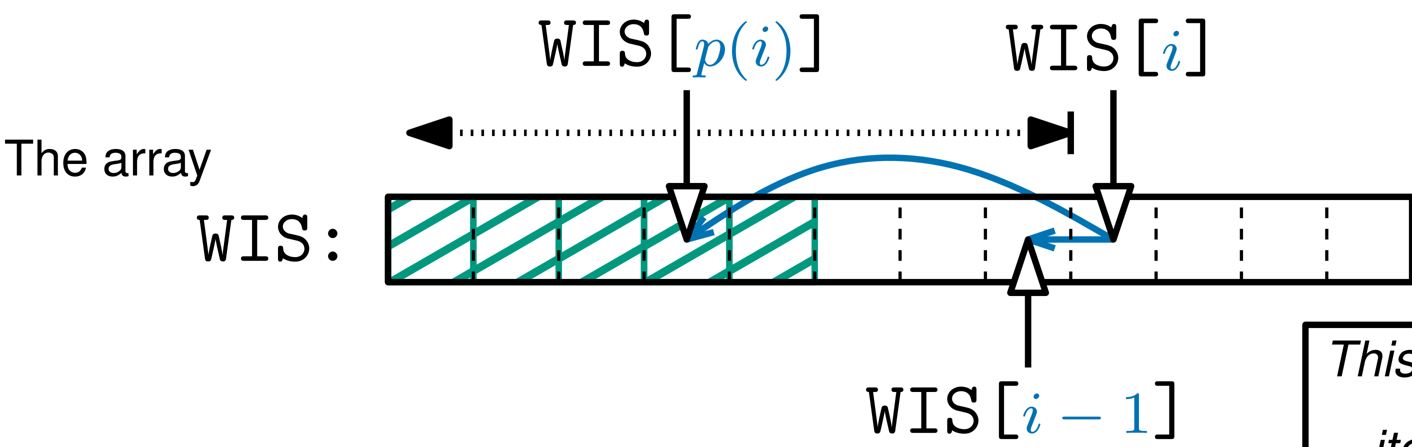
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

What information do we need to compute WIS[i] ?

Fill in the array from the left again

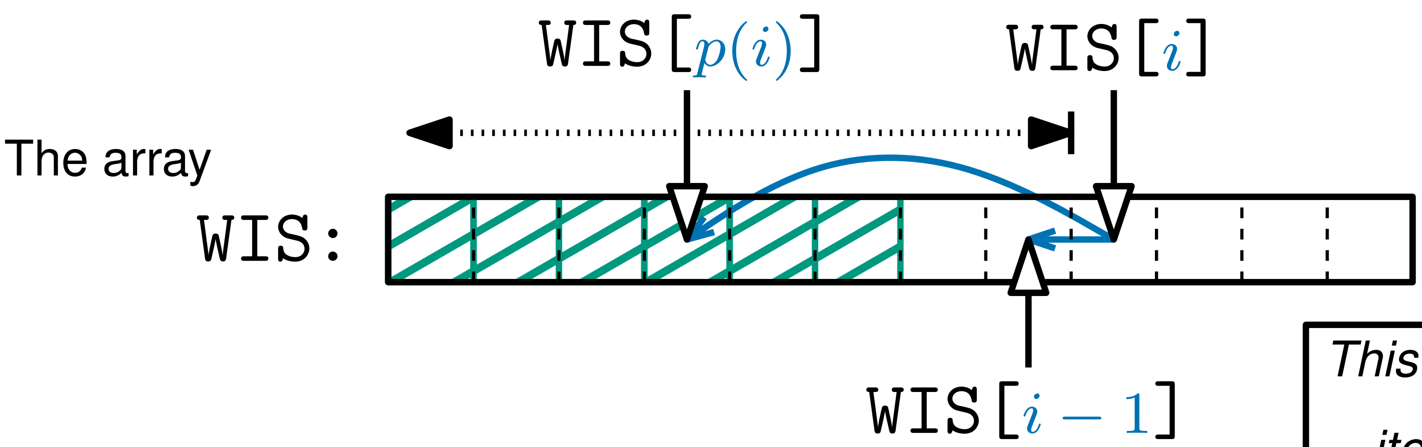
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

Fill in the array from the left again

What information do we need to compute WIS[i] ?

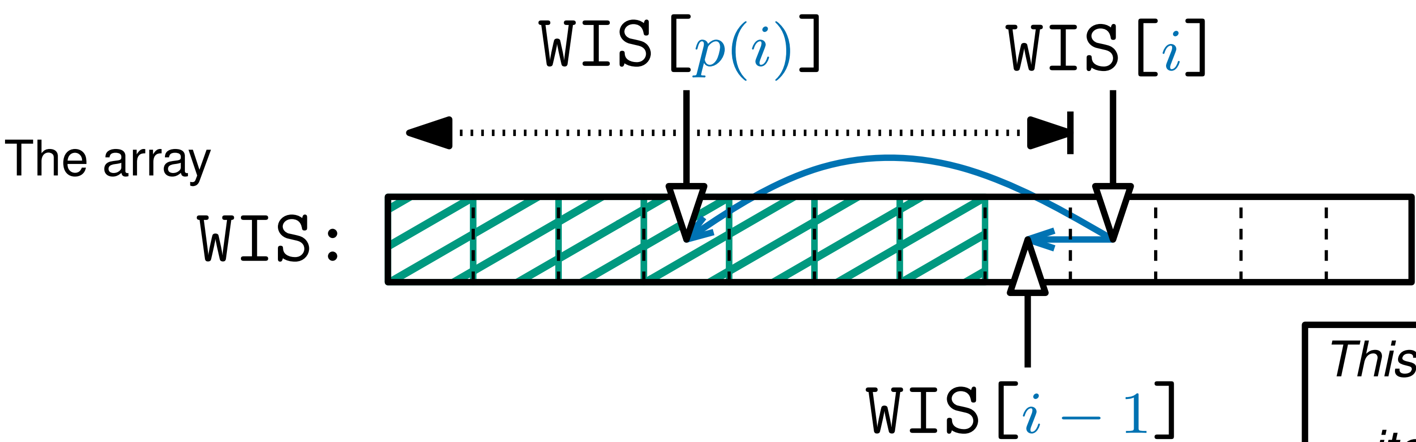
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .



This suggests another iterative algorithm

Fill in the array from the left again

What information do we need to compute WIS[i] ?

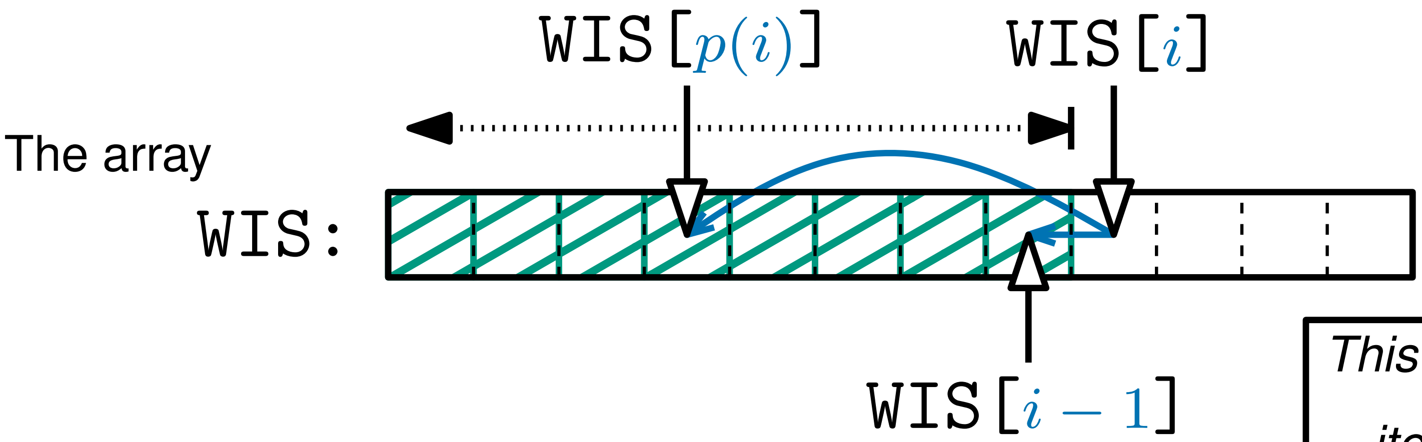
to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left...



This suggests another iterative algorithm

Fill in the array from the left again

What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

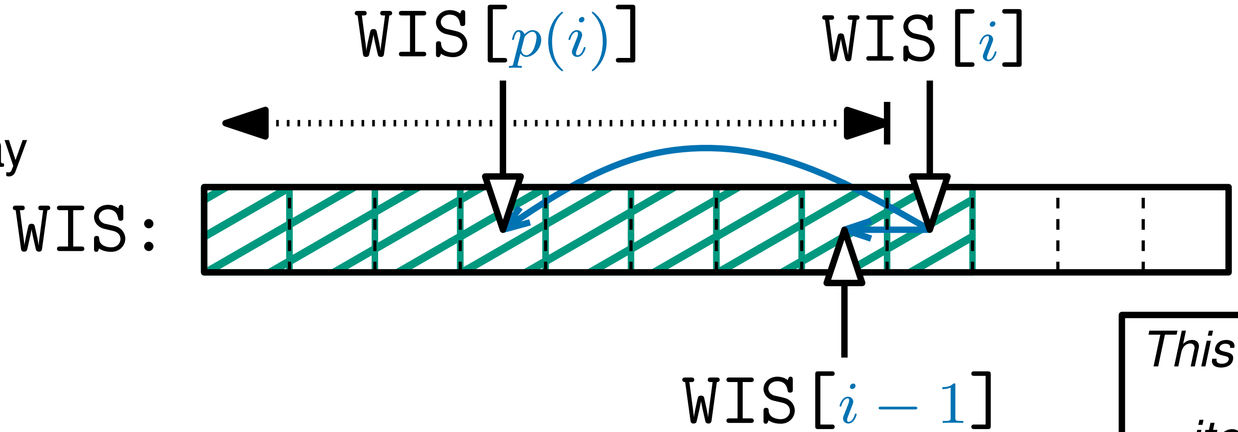
both of which are to the *left* of WIS[i]

(somewhere)

The dependency graph

all of the dependencies go left. . .

The array



This suggests another iterative algorithm

Fill in the array from the left again

What information do we need to compute WIS[i] ?

to compute WIS[i] we need WIS[$i-1$] and WIS[$p(i)$]

both of which are to the *left* of WIS[i]

(somewhere)

4. Derive an iterative algorithm

```
ItWIS(n)
```

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i-1], WIS[p(i)] + w_i)
Return WIS[i]
```

This is an iterative dynamic programming algorithm
for Weighted Interval Scheduling

it runs in $O(n)$ time

4. Derive an iterative algorithm

```
ItWIS(n)
```

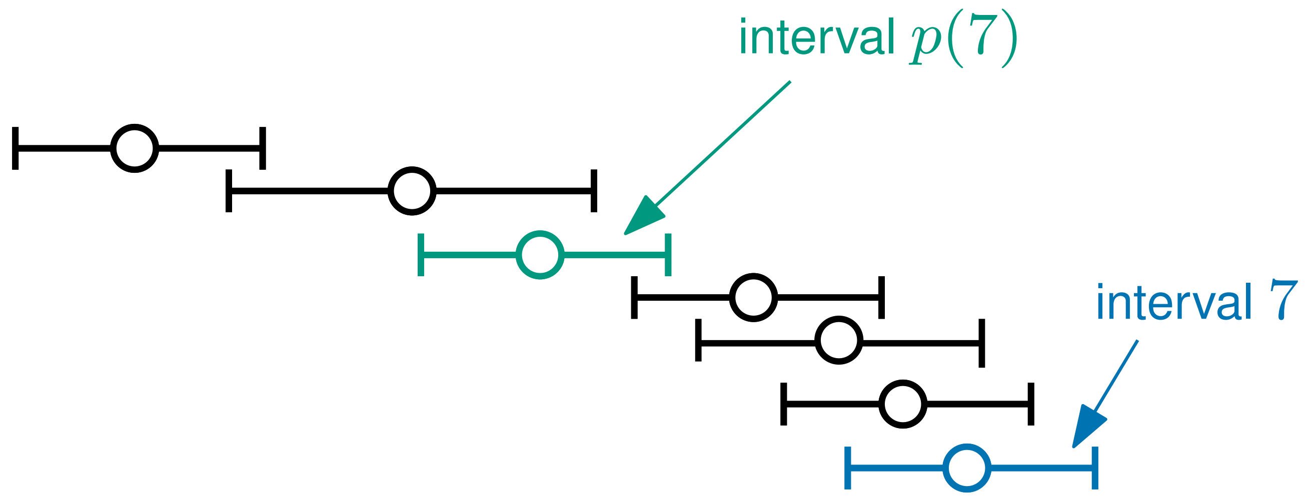
```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i − 1],WIS[p(i)] + w_i)
Return WIS[i]
```

This is an iterative dynamic programming algorithm
for Weighted Interval Scheduling

it runs in $O(n)$ time

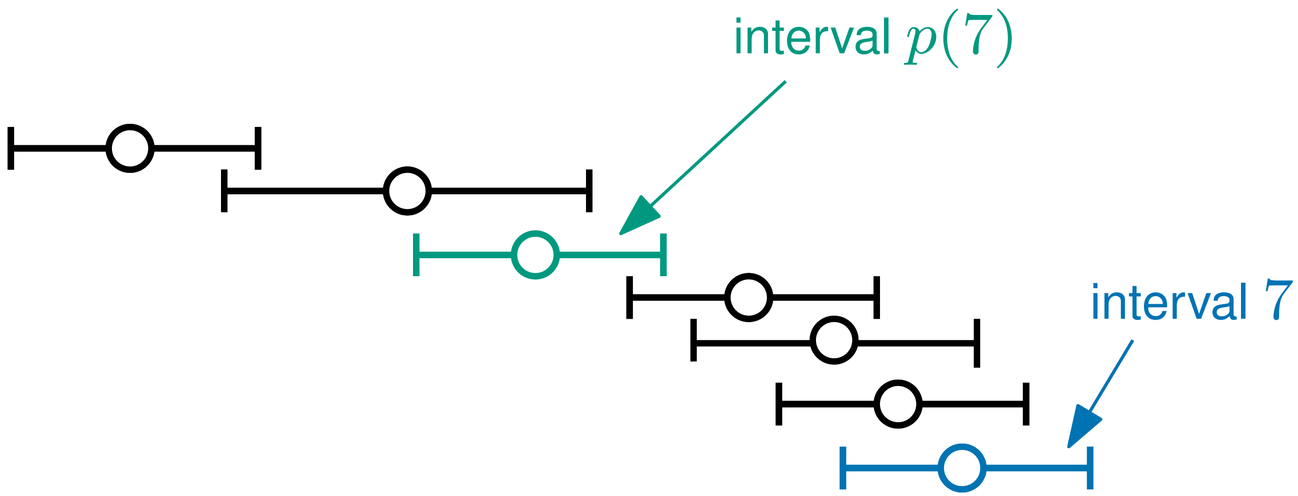
…but it requires than you precomputed all the $p(i)$ values

How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

How do you find all those $p(i)$ values?

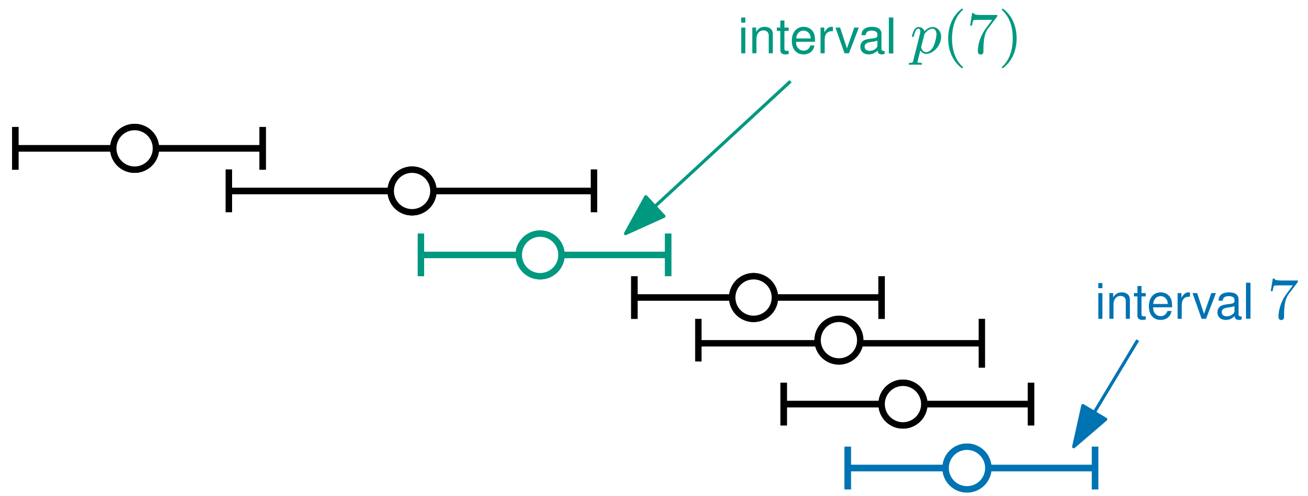


Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

Recall that s_i is the start time of interval i

and f_i is the finish time of interval i

How do you find all those $p(i)$ values?



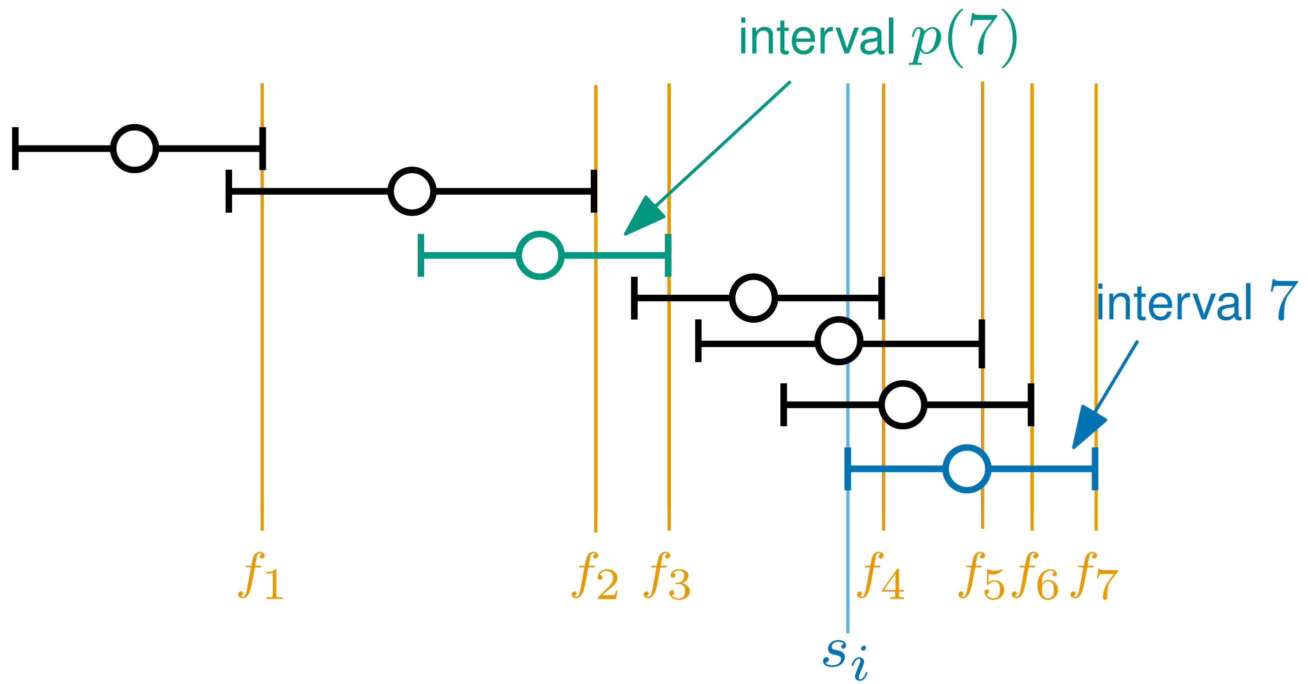
Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

Recall that s_i is the start time of interval i
and f_i is the finish time of interval i

We want to find the unique value $j = p(i)$ such that

$$f_j < s_i < f_{j+1}.$$

How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

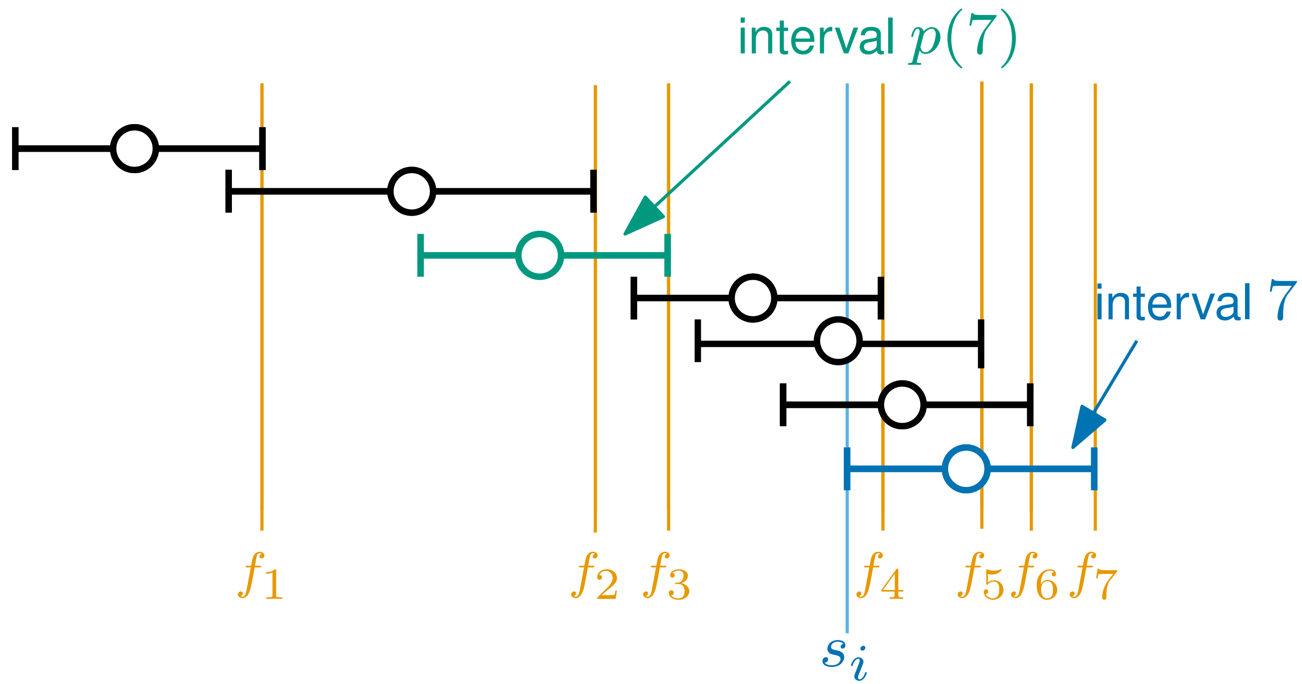
Recall that s_i is the start time of interval i

and f_i is the finish time of interval i

We want to find the unique value $j = p(i)$ such that

$$f_j < s_i < f_{j+1}.$$

How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

Recall that s_i is the start time of interval i

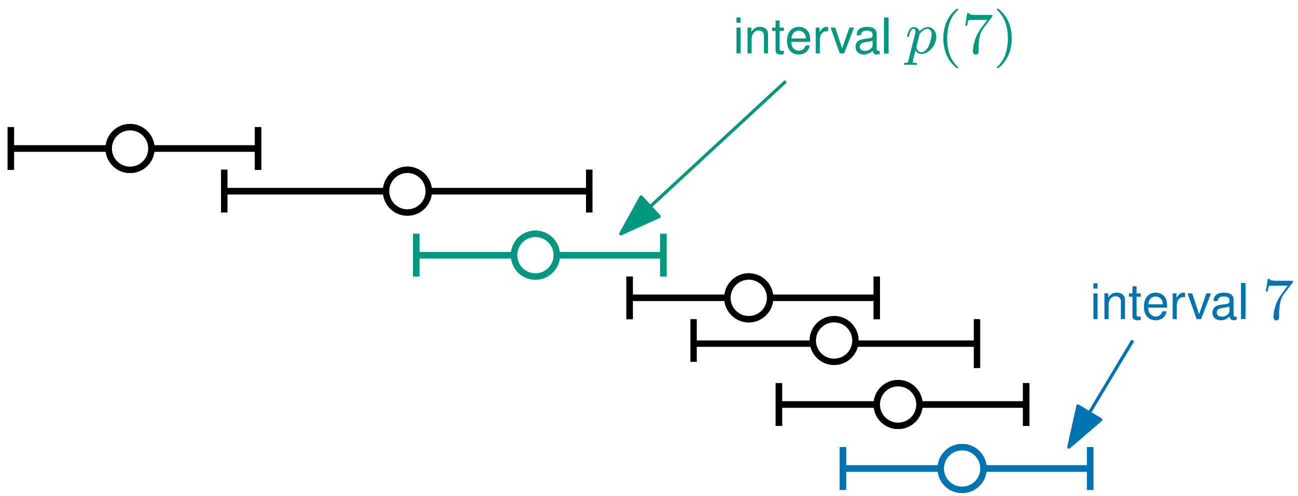
and f_i is the finish time of interval i

We want to find the unique value $j = p(i)$ such that

$$f_j < s_i < f_{j+1}.$$

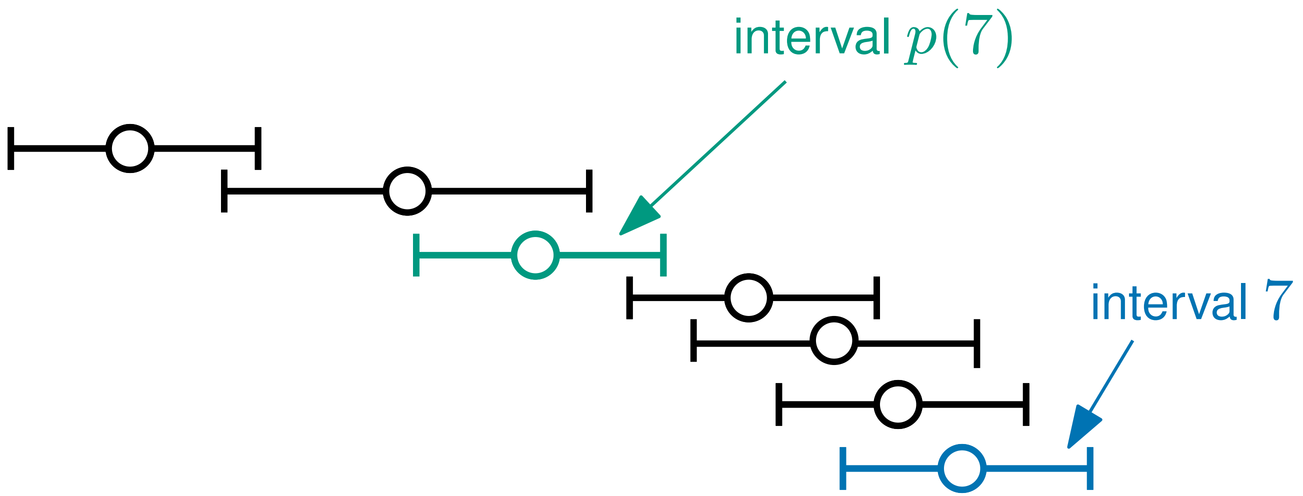
As the input is sorted by finish times, we can find j by binary search in $O(\log n)$ time

How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

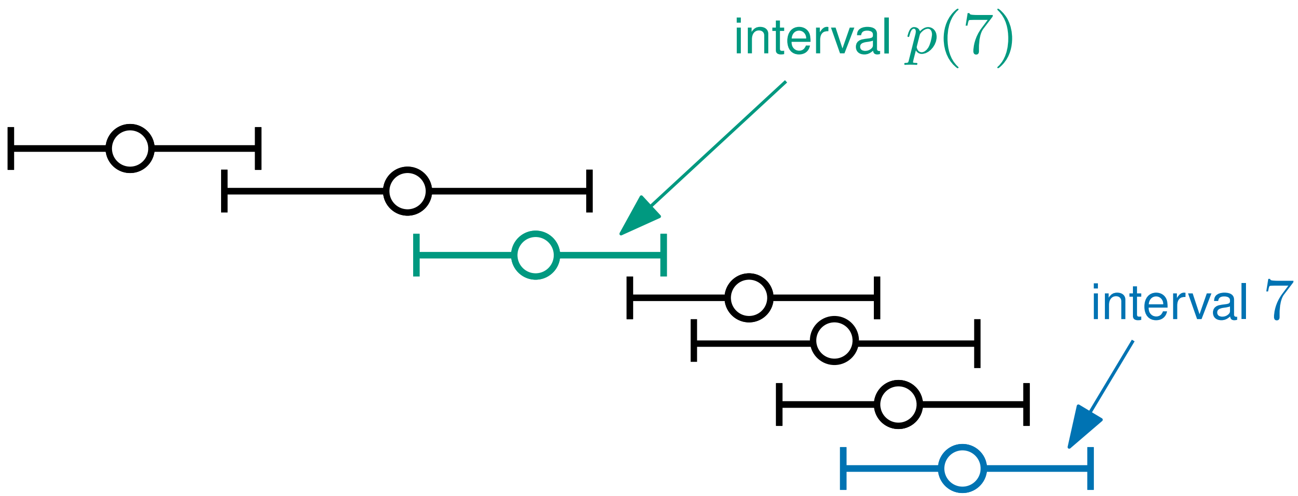
How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

Original Claim: We can precompute all $p(i)$ in $O(n \log n)$ time

How do you find all those $p(i)$ values?



Revised Claim: We can precompute any $p(i)$ in $O(\log n)$ time

Original Claim: We can precompute all $p(i)$ in $O(n \log n)$ time

(by using the revised claim n times)

Wait, did you want the actual schedule?

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
but doesn't find the actual schedule

$\texttt{ItWIS}(n)$

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i-1], WIS[p(i)] + w_i)
Return WIS[i]
```

Wait, did you want the actual schedule?

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
but doesn't find the actual schedule

$\texttt{ItWIS}(n)$

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i-1], WIS[p(i)] + w_i)
Return WIS[i]
```

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing
interval i if and only if

$$\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$$

Wait, did you want the actual schedule?

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
but doesn't find the actual schedule

```
ItWIS(n)
```

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i-1], WIS[p(i)] + w_i)
Return WIS[i]
```

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing
interval i if and only if

$$\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$$

(by the argument we saw earlier)

Wait, did you want the actual schedule?

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
and $\textsc{FindWIS}(n)$ finds the actual schedule

$\texttt{ItWIS}(n)$	$\textsc{FindWIS}(i)$

If $(i = 0)$
  Return 0
For $i = 1$ to n
  $\texttt{WIS}[i] = \max(\texttt{WIS}[i-1], \texttt{WIS}[p(i)] + w_i)$
Return $\texttt{WIS}[i]$
	If $(i = 0)$
	  Return nothing
	If $\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$
	  Return $\texttt{FindWIS}(p(i))$ then i
	Return $\texttt{FindWIS}(i-1)$

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing
interval i if and only if

$$\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$$

(by the argument we saw earlier)

This is called *backtracking* and works for lots of Dynamic Programming algorithms

Wait, did you want the actual schedule?

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
and $\textsc{FindWIS}(n)$ finds the actual schedule

$\texttt{ItWIS}(n)$	$\texttt{FindWIS}(i)$

If $(i=0)$
  Return 0
For $i=1$ to n
  $\texttt{WIS}[i] = \max(\texttt{WIS}[i-1], \texttt{WIS}[p(i)] + w_i)$
Return $\texttt{WIS}[i]$
	If $(i=0)$
	  Return nothing
	If $\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$
	  Return FindWIS$(p(i))$ then i
	Return FindWIS$(i-1)$

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing
interval i if and only if

$$\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$$

(by the argument we saw earlier)

This is called *backtracking* and works for lots of Dynamic Programming algorithms

The final algorithm

ITWIS(n) finds the weight of the optimal schedule
and FINDWIS(n) finds the actual schedule

ITWIS(n)

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max (WIS[i − 1],WIS[p(i)] + w_i)
Return WIS[i]
```

FINDWIS(i)

```
If (i = 0)
  Return nothing
If WIS[i − 1] ⩽ WIS[p(i)] + w_i
  Return FindWIS(p(i)) then i
Return FindWIS(i − 1)
```

The final algorithm:

Step 1: Find all the $p(i)$ values

Step 2: Run ITWIS(n) to find the optimal weight

Step 3: Run FINDWIS(n) to find the schedule

The final algorithm

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
and $\text{FindWIS}(n)$ finds the actual schedule

$\texttt{ItWIS}(n)$	$\texttt{FindWIS}(i)$

If $(i=0)$
  Return 0
For $i=1$ to n
  $\texttt{WIS}[i] = \max(\texttt{WIS}[i-1], \texttt{WIS}[p(i)] + w_i)$
Return $\texttt{WIS}[i]$
	If $(i=0)$
	  Return nothing
	If $\texttt{WIS}[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$
	  Return FindWIS$(p(i))$ then i
	Return FindWIS$(i-1)$

The final algorithm:

$O(n \log n)$ time

Step 1: Find all the $p(i)$ values

Step 2: Run $\texttt{ItWIS}(n)$ to find the optimal weight

Step 3: Run $\texttt{FindWIS}(n)$ to find the schedule

The final algorithm

$\texttt{ITWIS}(n)$ finds the weight of the optimal schedule and $\textsc{FindWIS}(n)$ finds the actual schedule

ITWIS(n)

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max(WIS[i-1], WIS[p(i)] + w_i)
Return WIS[i]
```

FINDWIS(i)

```
If (i = 0)
  Return nothing
If WIS[i-1] ≤ WIS[p(i)] + w_i
  Return FindWIS(p(i)) then i
Return FindWIS(i-1)
```

The final algorithm:

$O(n \log n)$ time

Step 1: Find all the $p(i)$ values

$O(n)$ time

Step 2: Run $\texttt{ITWIS}(n)$ to find the optimal weight

Step 3: Run $\textsc{FindWIS}(n)$ to find the schedule

The final algorithm

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
and $\textsc{FindWIS}(n)$ finds the actual schedule

ItWIS(n)

```
If (i = 0)
  Return 0
For i = 1 to n
  WIS[i] = max (WIS[i − 1],WIS[p(i)] + w_i)
Return WIS[i]
```

FindWIS(i)

```
If (i = 0)
  Return nothing
If WIS[i − 1] ⩽ WIS[p(i)] + w_i
  Return FindWIS(p(i)) then i
Return FindWIS(i − 1)
```

The final algorithm:

Step 1: Find all the $p(i)$ values ← $O(n \log n)$ time

Step 2: Run $\texttt{ItWIS}(n)$ to find the optimal weight ← $O(n)$ time

Step 3: Run $\textsc{FindWIS}(n)$ to find the schedule ← $O(n)$ time

The final algorithm

$\texttt{ItWIS}(n)$ finds the weight of the optimal schedule
and $\text{FindWIS}(n)$ finds the actual schedule

$\texttt{ItWIS}(n)$	$\texttt{FindWIS}(i)$

`If` $(i = 0)$
  `Return` 0
`For` $i = 1$ `to` n
  `WIS`$[i] = \max(\texttt{WIS}[i-1], \texttt{WIS}[p(i)] + w_i)$
`Return WIS`$[i]$
	`If` $(i = 0)$
	  `Return nothing`
	`If WIS`$[i-1] \leqslant \texttt{WIS}[p(i)] + w_i$
	  `Return FindWIS`$(p(i))$ `then` i
	`Return FindWIS`$(i-1)$

The final algorithm:

$O(n \log n)$ time

Step 1: Find all the $p(i)$ values

$O(n)$ time

Step 2: Run $\texttt{ItWIS}(n)$ to find the optimal weight

$O(n)$ time

Step 3: Run $\texttt{FindWIS}(n)$ to find the schedule

Overall this takes $O(n \log n)$ time

[Dynamic programming] is a technique for finding efficient algorithms for problems which can be broken down into simpler, overlapping subproblems.

The basic idea:

1. Find a recursive formula for the problem
- in terms of answers to subproblems.

(typically this is the hard bit)

2. Write down a naive recursive algorithm

(typically this algorithm will take exponential time)

3. Speed it up by storing the solutions to subproblems (memoization)

(to avoid recomputing the same thing over and over)

4. Derive an iterative algorithm by solving the subproblems in a good order

(iterative algorithms are often better in practice, easier to analyse and prettier)

in other words. . .

Dynamic programming is *recursion without repetition*