Dynamic Programming

Largest Empty Square and Weighted Interval Scheduling

Benjamin Sach
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
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 \textit{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

\textbf{Proof:} (by picture)

![Diagram](image)

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 \textit{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

\textbf{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?
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 \textit{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)

![Diagram](image.png)
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?
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)**

![Proof Diagram](image)

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?
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 $\square$ 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 \textit{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

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

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 $(x, y)$.

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 $(x, y)$.

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 $(x, y)$.

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 $(x, y)$.

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 \((x, y)\).

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 $(x, y)$.

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 $(x, y)$

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 \((x, y)\).

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,
  \[ \text{LES}(x, y) \leq \text{LES}(x - 1, y - 1) + 1 \]

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

\[
\text{LES}(x, y) \leq \text{LES}(x - 1, y - 1) + 1
\]

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

$(x, y)$ can’t be bigger than this.
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:

1. If the pixel \( (x, y) \) is not empty then \( \text{LES}(x, y) = 0 \).
2. If \( (x, y) \) is empty and in the first row or column, \( \text{LES}(x, y) = 1 \).
3. 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 $\text{LES}(x, y)$ 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 \( \text{LES}(x, y) \) 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 $\text{LES}(x, y)$ 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 $\text{LES}(x, y)$ 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 \( \text{LES}(x, y) \) be the size (i.e. side length) of the largest empty square whose bottom right is at \((x, y)\).

- 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 $(x, y)$.

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 $(x, y)$

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

Is this square always empty?

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

is this square always empty?

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

is this square always empty?

Yes, by the proof on the previous slide.

✓
2. Write down a recursive algorithm

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

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

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

Therefore, the maximum of \(\text{LES}(x, y)\) 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...

\[
\text{LES}(x, y)
\]

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

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

Therefore, the maximum of \(\text{LES}(x, y)\) 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?

\[
\text{LES}(x, y)
\]

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

Let’s compute \(\text{LES}(4, 4)\)...
How efficient is the recursive algorithm?

\[ \text{LES}(x, y) \]

If pixel \((x, y)\) is not empty
  
  Return 0

If \((x = 1)\) or \((y = 1)\)
  
  Return 1

Return \(\text{min}(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let's compute \(\text{LES}(4, 4)\)… *(and consider the recursive calls)*
How efficient is the recursive algorithm?

\[ \text{LES}(x, y) \]

If pixel \((x, y)\) is not empty
  \begin{align*}
  & \text{Return } 0 \\
  \text{If } (x = 1) \text{ or } (y = 1) \\
  & \text{Return } 1 \\
  \text{Return } \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1
  \end{align*}

Let’s compute \(\text{LES}(4, 4)\)… *(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 \(\min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let’s compute LES\((4, 4)\)... * (and consider the recursive calls)
How efficient is the recursive algorithm?

\[
\text{LES}(x, y)
\]

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

Let’s compute \(\text{LES}(4, 4)\)… *(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 \(\min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let’s compute \(\text{LES}(4, 4)\)… *(and consider the recursive calls)*
How efficient is the recursive algorithm?

\[
\text{LES}(x, y)
\]

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

Let’s compute \(\text{LES}(4, 4)\)… \((\text{and consider the recursive calls})\)
How efficient is the recursive algorithm?

\[
\text{LES}(x, y)
\]

If pixel \((x, y)\) is not empty
  
  Return 0

If \((x = 1)\) or \((y = 1)\)
  
  Return 1

Return \(\min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let’s compute \(\text{LES}(4, 4)\)… *(and consider the recursive calls)*
How efficient is the recursive algorithm?

\[ \text{LES}(x, y) \]

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

Let’s compute \( \text{LES}(4, 4) \)… \( \text{(and consider the recursive calls)} \)
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 \(\min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let's compute \(\text{LES}(4, 4)\)... (and consider the recursive calls)

\((2, 2)\) is computed three times just while computing this \((3, 3)\)
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?

\[ \text{LES}(x, y) \]

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

Let's compute \(\text{LES}(4, 4)\)… (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

\[ \text{Return } 0 \]

If \((x = 1)\) or \((y = 1)\)

\[ \text{Return } 1 \]

\[ \text{Return } \min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1 \]

Let’s compute \(\text{LES}(4, 4)\)… *(and consider the recursive calls)*

This doesn’t look good!

In fact the running time of \(\text{LES}(n, n)\) is exponential in \(n\)
How efficient is the recursive algorithm?

Let’s compute \( \text{LES}(4, 4) \)… *(and consider the recursive calls)*

If \( T(n) \) is the runtime of \( \text{LES}(n, n) \) then \( T(n) > 3T(n - 1) \)

This doesn’t look good!

In fact the running time of \( \text{LES}(n, n) \) is *exponential* in \( n \)
How efficient is the recursive algorithm?

Let's compute $\text{LES}(4, 4)$... *(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 \(\min(\text{LES}(x - 1, y - 1), \text{LES}(x - 1, y), \text{LES}(x, y - 1)) + 1\)

Let’s compute \(\text{LES}(4, 4)\)… *(and consider the recursive calls)*

What should we do about all this repeated computation?
3. **Store the solutions to subproblems**

\[ \text{MEMLES}(x, y) \]

If pixel \((x, y)\) is not empty
   
   Return 0

If \((x = 1)\) or \((y = 1)\)
   
   Return 1

If \(\text{LES}[x, y]\) undefined
   
   \[ \text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1 \]

Return \(\text{LES}[x, y]\)

In the \text{MEMLES} version of the algorithm

we store solutions to previously computed subproblems

in an \((n \times n)\) 2D array called \text{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 \(\text{LES}[x, y]\) undefined
   - \(\text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1\)
Return \(\text{LES}[x, y]\)

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

\[ \text{MEMLES}(x, y) \]

If pixel \( (x, y) \) is not empty
  Return 0
If \( (x = 1) \) or \( (y = 1) \)
  Return 1
If \( \text{LES}[x, y] \) undefined
  \[ \text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1 \]
Return \( \text{LES}[x, y] \)

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

This is called \text{memoization} \text{(not memorization)}

Crucially, now each entry \( \text{LES}[x, y] \) is only computed \text{once}
3. Store the solutions to subproblems

\[
\text{MEMLES}(x, y)
\]

If pixel \((x, y)\) is not empty
   - Return 0
If \((x = 1)\) or \((y = 1)\)
   - Return 1
If \(\text{LES}[x, y]\) undefined
   - \(\text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1\)
Return \(\text{LES}[x, y]\)

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

This is called \textit{memoization} \((\textit{not memorization})\)

Crucially, now each entry \(\text{LES}[x, y]\) is only computed \textit{once}

The time complexity of computing \text{MEMLES}(n, n) is now \(O(n^2)\)
3. Store the solutions to subproblems

\[
\text{MEMLES}(x, y)
\]

If pixel \((x, y)\) is not empty
  Return 0
If \((x = 1)\) or \((y = 1)\)
  Return 1
If \(\text{LES}[x, y]\) undefined
  \[
  \text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1
  \]
Return \(\text{LES}[x, y]\)

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

This is called \textit{memoization} (\textit{not memorization})

Crucially, now each entry \(\text{LES}[x, y]\) is only computed \textit{once}

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

(in fact, computing \(\max_{x,y} \text{MEMLES}(x, y)\) takes \(O(n^2)\) time too)
The dependency graph

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

(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

\[ \text{LES}[x, y] = \min (\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1 \]  
(for \( x, y > 1 \) and \( (x, y) \) non empty)

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

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

The 2D array

\[
\text{LES}:
\]

to compute \( \text{LES}[n, n] \) we need
The dependency graph

\[
\text{LES}[x, y] = \min(\text{MEMLES}(x-1, y-1), \text{MEMLES}(x-1, y), \text{MEMLES}(x, y-1)) + 1
\]
(for \(x, y > 1\) and \((x, y)\) non empty)

The 2D array \(\text{LES}\):

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

We need:
- \(\text{LES}[n-1, n-1]\)
- \(\text{LES}[n-1, n]\)
- \(\text{LES}[n, n-1]\)
The dependency graph

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

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

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

The 2D array

\[ \text{LES} : \]

we need

\( \text{LES}[n - 1, n - 1] \)
\( \text{LES}[n - 1, n] \)
and \( \text{LES}[n, n - 1] \)
The dependency graph

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

(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

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

(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

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

(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

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

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

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

The 2D array \(\text{LES}\):

to compute

\(\text{LES}[n, n-1]\)

\(\text{LES}[n-1, n-2]\)

\(\text{LES}[n-1, n-1]\)

and \(\text{LES}[n, n-2]\)
The dependency graph

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

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

The 2D array

\[ \text{LES} : \]

to compute

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

What information do we need to compute \(\text{LES}[n, n]\)?
The dependency graph

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

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

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

The 2D array \( \text{LES} : \)

- to compute \( \text{LES}[n - 1, n] \)
- we need \( \text{LES}[n - 2, n - 1] \)
- \( \text{LES}[n - 2, n] \)
- and \( \text{LES}[n - 1, n - 1] \)
The dependency graph

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

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

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

The 2D array

\[ \text{LES} : \]

to compute

\[ \text{LES}[n - 1, n], \text{LES}[n - 2, n - 1], \text{LES}[n - 2, n] \]

and \( \text{LES}[n - 1, n - 1] \)
The dependency graph

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

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

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

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

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

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

\[
\text{LES}[x, y] = \min(\text{MemLES}(x-1, y-1), \text{MemLES}(x-1, y), \text{MemLES}(x, y-1)) + 1 \\
\text{(for } x, y > 1 \text{ and } (x, y) \text{ non empty)}
\]

What information do we need to compute \(\text{LES}[n, n]\)?
The dependency graph

\[
\text{LES}[x, y] = \min (\text{MemLES}(x - 1, y - 1), \text{MemLES}(x - 1, y), \text{MemLES}(x, y - 1)) + 1
\]
(for \(x, y > 1\) and \((x, y)\) non empty)

What information do we need to compute \(\text{LES}[n, n]\)?
The dependency graph

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

(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

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

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

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

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

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

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

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

(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

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

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

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

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

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

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

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)

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

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

$$\text{LES}[x, y] = \min(\text{MemLES}(x - 1, y - 1), \text{MemLES}(x - 1, y), \text{MemLES}(x, y - 1)) + 1$$
(for $x, y > 1$ and $(x, y)$ non empty)

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

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

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

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

What information do we need to compute \(\text{LES}[n, n]\)?
The dependency graph

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

(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 $\text{LES}[n, n]$?
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)

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

The 2D array \( \text{LES} \):

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

(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

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

(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

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

(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

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

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

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

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

\[
\text{LES}[x, y] = \min (\text{MemLES}(x - 1, y - 1), \text{MemLES}(x - 1, y), \text{MemLES}(x, y - 1)) + 1
\]
(for \(x, y > 1\) and \((x, y)\) non empty)

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

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

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

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

The 2D array

$\text{LES}$:

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array

\[ \text{LES} : \]

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array

\[ \text{LES} : \]

\textbf{How can we use this to get an iterative algorithm?}

\textbf{Fill in the array from the top-left!}
The dependency graph

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

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

(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

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

(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

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

(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

\[ \text{LES}[x, y] = \min\left(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)\right) + 1 \]
(for \(x, y > 1\) and \((x, y)\) non empty)

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

The 2D array

\(\text{LES}\):

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

(for \( x, y > 1 \) and \( (x, y) \) non-empty)

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

The 2D array \( \text{LES} \):

- How can we use this to get an iterative algorithm?
- Fill in the array from the top-left!
The dependency graph

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

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

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

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

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

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

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

The 2D array

$$ \text{LES} : $$

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array

\[ \text{LES} : \]

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

(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

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

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

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

The 2D array

\[ \text{LES} : \]

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

**The 2D array \(\text{LES}\):**

*How can we use this to get an iterative algorithm?*

*Fill in the array from the top-left!*
The dependency graph

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

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

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

The 2D array

\( \text{LES} \):

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

What information do we need to compute \(\text{LES}[n, n]\)?
The dependency graph

\[ \text{LES}[x, y] = \min(\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1 \]
(for \( x, y > 1 \) and \( (x, y) \) non empty)

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

The 2D array 
\[ \text{LES} : \]

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array \( \text{LES} : \)

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array \( \text{LES} \):

- How can we use this to get an iterative algorithm?
- Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array

\( \text{LES} : \)

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

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

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

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

The 2D array $\text{LES}$:

How can we use this to get an iterative algorithm?

Fill in the array from the top-left!
The dependency graph

\[ \text{LES}[x, y] = \min (\text{MEMLES}(x - 1, y - 1), \text{MEMLES}(x - 1, y), \text{MEMLES}(x, y - 1)) + 1 \]
(for \( x, y > 1 \) and \( (x, y) \) non-empty)

What information do we need to compute \( \text{LES}[n, n] \)?
The dependency graph

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

(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 \( \text{LES}[n, n] \)?
4. Derive an iterative algorithm

\textbf{ItLES}(n)

\begin{itemize}
  \item For $y = 1$ to $n$
    \begin{itemize}
      \item For $x = 1$ to $n$
        \begin{itemize}
          \item If pixel $(x, y)$ is not empty
            \begin{itemize}
              \item LES$[x, y] = 0$
            \end{itemize}
          \item Else if $(x = 1)$ or $(y = 1)$
            \begin{itemize}
              \item LES$[x, y] = 1$
            \end{itemize}
          \item Else
            \begin{itemize}
              \item LES$[x, y] = \min(\text{LES}[x - 1, y - 1], \text{LES}[x - 1, y], \text{LES}[x, y - 1]) + 1$
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

This iterative version of the algorithm runs in $O(n^2)$ time and avoids making any recursive calls.
4. Derive an iterative algorithm

\[
\text{ItLES}(n)
\]

For \( y = 1 \) to \( n \)
  For \( x = 1 \) to \( n \)
    If pixel \((x, y)\) is not empty
      \[ \text{LES}[x, y] = 0 \]
    Else If \((x = 1)\) or \((y = 1)\)
      \[ \text{LES}[x, y] = 1 \]
    Else
      \[ \text{LES}[x, y] = \min(\text{LES}[x-1, y-1], \text{LES}[x-1, y], \text{LES}[x, y-1]) + 1 \]

This iterative version of the algorithm runs in \(O(n^2)\) time and avoids making any recursive calls.

Maximum of \(\text{LES}[x, y]\) 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

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

```
   1  2  3  4  5  6  7
  ___________  ___________  ___________
      |       |       |       |
    2  5  1  4  3  6  7
  ___________  ___________
      |       |       |
    2  5  7  2
```

weight

---

University of Bristol
Weighted Interval Scheduling

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

**Diagram:**
- Compatible intervals don't overlap.
- Intervals are labeled with numbers corresponding to their duration.
- The diagram shows intervals overlapping and non-overlapping, with time and weight axes indicated.
Weighted Interval Scheduling

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

[Diagram showing intervals and incompatible overlaps]
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.
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
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.

![Diagram showing weighted intervals and schedules](image-url)
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.

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

*is this the best possible?*
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

Is this the best possible?
Weighted Interval Scheduling

**Problem** Given an \( n \) weighted intervals, find the *schedule* with largest total weight.

A schedule with total weight 18 is illustrated. 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

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

---

A schedule with total weight 18

is this the best possible?
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.

\[
\begin{aligned}
&\text{a schedule with total weight 18} \\
&\text{is this the best possible? yes}
\end{aligned}
\]
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.

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

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

---

**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 \( n \).

\[ A[i] \text{ stores a triple } (s_i, f_i, w_i) \text{ which defines the } i\text{-th interval} \]
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. $f_i \leq f_{i+1}$.
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

weight, $w_i$
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

weight, $w_i$
**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

interval \( p(7) \)

interval 7
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.
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

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

What is \( p(2) \)?
Compatible Intervals

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

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, $p(i) = 0$.

What is $p(2)$?
Compatible Intervals

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

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

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

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

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.

\[ \text{if no such interval exists, } p(i) = 0 \]

**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 \( \text{OPT} \).

more intervals not shown
1. Find a recursive formula

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

In particular, consider the $n$-th interval \ldots

more intervals not shown
1. Find a recursive formula

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

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 $\text{OPT}$. . .

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

more intervals not shown

$n$-th interval
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 \( \text{OPT} \).

**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 $O$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight $\text{OPT}$. …

Case 1: The $n$-th interval is not in $O$

- schedule $O$ 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 $O$ for intervals $\{1, 2, 3 \ldots, n\}$ with weight $OPT$.

**Case 1:** The $n$-th interval is *not* in $O$

- schedule $O$ 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 \( \text{OPT} \)...

**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 $\text{OPT}$.

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 $\text{OPT}$. . .

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 $\text{OPT}$... 

\[ \text{more intervals not shown} \]

**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 $\text{OPT}$...

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 $\text{OPT}$.

**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 $\text{OPT}$. 

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

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

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

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

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

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

Interval $p(n)$

Case 2: The $n$-th interval is in $O$

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

*(the ones which don’t overlap the $n$-th interval)*

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

Notation: $OPT(i)$ 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 $\text{OPT}$. . .

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

$(w_n$ is the weight of interval $n$)

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

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 $\text{OPT}$.

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

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

($\text{the ones which don't overlap the } n\text{-th interval}$)

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 $\text{OPT}$.

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

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 $\mathcal{O}$

$\text{OPT} = \text{OPT}(p(n)) + w_n$
1. Find a recursive formula

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

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?

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

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 $\text{OPT}$.

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

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

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 $\text{OPT}$.

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 \( \text{OPT}(i) \)...

**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...

\[
\text{WIS}(i) = \begin{cases} 
0 & \text{if } i = 0 \\
\max \left( \text{WIS}(i-1), \text{WIS}(p(i)) + w_i \right) & \text{otherwise}
\end{cases}
\]

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

Therefore, \(\text{WIS}(n)\) 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...

\[
\text{WIS}(i) =
\begin{align*}
&\text{If } (i = 0) \\
&\quad \text{Return } 0 \\
&\text{Return } \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\end{align*}
\]

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

Therefore, \(\text{WIS}(n)\) 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?

\[ \text{WIS}(i) \]

If \((i = 0)\)  
Return 0  
Return \( \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i) \)

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 \( n = 6 \)

(\textit{the best schedule has weight 3})
How efficient is the recursive algorithm?

\[
\text{WIS}(i) = \begin{cases} 
0 & \text{if } (i = 0) \\
\max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i) & \text{otherwise}
\end{cases}
\]

consider this simple input with \( n = 6 \)

(\text{the best schedule has weight 3})

further, for all \( i, p(i) = i - 2 \)
How efficient is the recursive algorithm?

\[
WIS(i) =
\]

\[
\text{If } (i = 0) \\
\quad \text{Return } 0
\]

\[
\quad \text{Return } \max(WIS(i - 1), WIS(p(i)) + w_i)
\]

Consider this simple input with \( n = 6 \)

\[
\begin{array}{c}
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
1 \\
\hline
(\text{the best schedule has weight 3})
\end{array}
\]

Further, for all \( i, p(i) = i - 2 \)

So \( WIS(i) \) makes recursive calls to \( WIS(i - 1) \) and \( WIS(i - 2) \)
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

\[
\text{If } (i = 0) \\
\quad \text{Return } 0 \\
\text{Return } \max(WIS(i - 1), WIS(p(i)) + w_i)
\]

so \(WIS(i)\) makes recursive calls to \(WIS(i - 1)\) and \(WIS(i - 2)\)
How efficient is the recursive algorithm?

\[
\text{WIS}(i) \\
\begin{align*}
\text{If } (i = 0) & \\
\quad \text{Return } 0 & \\
\text{Return } \max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i) &
\end{align*}
\]

so \(\text{WIS}(i)\) makes recursive calls to \(\text{WIS}(i - 1)\) and \(\text{WIS}(i - 2)\)
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

If \((i = 0)\)

\[
\text{Return } 0
\]

\[
\text{Return } \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\]

\[
\text{WIS}(6)
\]

so \(\text{WIS}(i)\) makes recursive calls to \(\text{WIS}(i - 1)\) and \(\text{WIS}(i - 2)\)
How efficient is the recursive algorithm?

\[
WIS(i)
\]

\[
\text{If } (i = 0) \\
\quad \text{Return } 0 \\
\text{Return } \max(WIS(i - 1), WIS(p(i)) + w_i)
\]

so \(WIS(i)\) makes recursive calls to \(WIS(i - 1)\) and \(WIS(i - 2)\)
How efficient is the recursive algorithm?

\[
\text{WIS}(i) = \begin{cases} 
0 & \text{if } i = 0 \\
\max(\text{WIS}(i-1), \text{WIS}(p(i)) + w_i) & \text{otherwise}
\end{cases}
\]

so \( \text{WIS}(i) \) makes recursive calls to \( \text{WIS}(i-1) \) and \( \text{WIS}(i-2) \)
How efficient is the recursive algorithm?

\[
\begin{align*}
\text{WIS}(i) \\
\text{If } (i = 0) \\
\quad \text{Return } 0 \\
\text{Return } \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\end{align*}
\]

so \( \text{WIS}(i) \) makes recursive calls to \( \text{WIS}(i - 1) \) and \( \text{WIS}(i - 2) \)
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

If \((i = 0)\)

\[
\text{Return } 0
\]

\[
\text{Return } \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\]

so \(\text{WIS}(i)\) makes recursive calls to \(\text{WIS}(i - 1)\) and \(\text{WIS}(i - 2)\)
How efficient is the recursive algorithm?

\[
WIS(i)
\]

If \(i = 0\)
  \[
  \text{Return } 0
  \]
  \[
  \text{Return } \max(WIS(i-1), WIS(p(i)) + w_i)
  \]

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 \(\max(WIS(i-1), WIS(p(i)) + w_i)\)

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?

\[ WIS(i) \]

If \((i = 0)\)  
\[
\text{Return } 0
\]
\[
\text{Return } \max(WIS(i - 1), WIS(p(i)) + w_i)
\]

if we extend this input in the same way...
How efficient is the recursive algorithm?

\[
\text{WIS}(i) = \begin{cases} 
0 & \text{if } i = 0 \\
\max(\text{WIS}(i-1), \text{WIS}(p(i)) + w_i) & \text{otherwise}
\end{cases}
\]

if we extend this input in the same way...
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

If \( i = 0 \)
\[
\text{Return 0}
\]

Return \( \max(\text{WIS}(i - 1), \text{WIS}(\text{p}(i)) + w_i) \)

if we extend this input in the same way...
How efficient is the recursive algorithm?

WIS\( (i) \)

If \((i = 0)\)
  
  Return 0

Return \( \max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i) \)

if we extend this input in the same way...
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

If \( i = 0 \)
Return 0
Return \( \max(\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i) \)

if we extend this input in the same way…
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

\[
\text{If } (i = 0) \\
\text{Return } 0 \\
\text{Return } \max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\]

if we extend this input in the same way...

Given \( n \) intervals set out in this manner,
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

\[
\text{If } (i = 0) \quad \text{Return } 0 \\
\text{Return } \max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)
\]

if we extend this input in the same way...

Given \( n \) intervals set out in this manner, \( \text{WIS}(n) \) runs in *exponential* time
How efficient is the recursive algorithm?

\[
\text{WIS}(i)
\]

- If \((i = 0)\)
  - Return 0
- Return \(\max (\text{WIS}(i - 1), \text{WIS}(p(i)) + w_i)\)

if we extend this input in the same way...

Given \(n\) intervals set out in this manner,

\(\text{WIS}(n)\) runs in \textit{exponential} time

If \(T(n)\) is the run time of \(\text{WIS}(n)\) using these intervals then \(T(n) > 2T(n - 2)\)
3. Store the solutions to subproblems

\[ \text{MEMWIS}(i) \]

\[
\begin{align*}
\text{If } (i = 0) & \\
\quad \text{Return } 0 & \\
\text{If } \text{WIS}[i] \text{ undefined} & \\
\quad \text{WIS}[i] &= \max (\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i) & \\
\text{Return } \text{WIS}[i] &
\end{align*}
\]
3. Store the solutions to subproblems

\[ \text{MEMWIS}(i) \]

If \( (i = 0) \)
   
   Return 0

If WIS\([i]\) undefined
   
   \[ \text{WIS}[i] = \max (\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i) \]

Return \( \text{WIS}[i] \)

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

\[
\text{MEMWIS}(i)
\]

If \(i = 0\)
- Return 0
If \(\text{WIS}[i]\) undefined
  - \(\text{WIS}[i] = \max(\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i)\)
Return \(\text{WIS}[i]\)

In the \text{MEMWIS} version of the algorithm
we store solutions to previously computed subproblems
in an \(n\) length array called \text{WIS}
(we have memoized the algorithm)
3. Store the solutions to subproblems

\[
\text{MEMWIS}(i)
\]

If \( i = 0 \)

Return 0

If \( \text{WIS}[i] \) undefined

\[
\text{WIS}[i] = \max (\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i)
\]

Return \( \text{WIS}[i] \)

In the \text{MEMWIS} version of the algorithm

we store solutions to previously computed subproblems

in an \( n \) length array called \text{WIS}

(we have memoized the algorithm)

Each entry \( \text{WIS}[i] \) is only computed \textit{once}
3. Store the solutions to subproblems

\[
\text{MEMWIS}(i)\\
\begin{array}{l}
\text{If } (i = 0) \\
\quad \text{Return 0}\\
\text{If } \text{WIS}[i] \text{ undefined} \\
\quad \text{WIS}[i] = \max (\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i)\\
\text{Return } \text{WIS}[i]
\end{array}
\]

In the \textbf{MEMWIS} version of the algorithm
we store solutions to previously computed subproblems
in an \( n \) length array called \textbf{WIS}
(we have memoized the algorithm)

Each entry \textbf{WIS}[i] is only computed \textit{once}

The time complexity of computing \textbf{MEMWIS}(n) is now \( O(n) \)
3. Store the solutions to subproblems

\[
\text{MEMWIS}(i)
\]

If \((i = 0)\)

Return 0

If \(\text{WIS}[i]\) undefined

\[
\text{WIS}[i] = \max(\text{MEMWIS}(i - 1), \text{MEMWIS}(p(i)) + w_i)
\]

Return \(\text{WIS}[i]\)

In the \text{MEMWIS} version of the algorithm
we store solutions to previously computed subproblems
in an \(n\) length array called \text{WIS}
(we have memoized the algorithm)

Each entry \(\text{WIS}[i]\) is only computed \text{once}

The time complexity of computing \text{MEMWIS}(n) is now \(O(n)\)

\text{because every recursion causes an unfilled entry to be filled in the array}
The dependency graph

The array

\texttt{WIS: \underline{\hspace{\textwidth}}} 

What information do we need to compute \texttt{WIS[i]}?
The dependency graph

The array

\[ WIS[i] \]

What information do we need to compute \( WIS[i] \)?
The dependency graph

The array

\[ \text{WIS:} \]

What information do we need to compute \( \text{WIS}[i] \)?

to compute \( \text{WIS}[i] \) we need \( \text{WIS}[i-1] \) and \( \text{WIS}[p(i)] \)
The dependency graph

The array

\[ \text{WIS:} \]

\[ \text{WIS}[i] \]

\[ \text{WIS}[i - 1] \]

What information do we need to compute \( \text{WIS}[i] \)?

to compute \( \text{WIS}[i] \) we need \( \text{WIS}[i - 1] \) and \( \text{WIS}[p(i)] \)
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

\[
WIS: \quad \begin{array}{c}
WIS[p(i)] \\
\downarrow \\
WIS[i] \\
\downarrow \\
WIS[i-1]
\end{array}
\]

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 \( WIS[i] \)?

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

The array

\[ WIS[i] \]

\[ WIS[p(i)] \]

\[ WIS[i - 1] \]

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

\[ \text{WIS} : \]

\[ \text{WIS}[p(i)] \quad \text{WIS}[i] \]

\[ \text{WIS}[i-1] \]

What information do we need to compute \( \text{WIS}[i] \)?

to compute \( \text{WIS}[i] \) we need \( \text{WIS}[i-1] \) and \( \text{WIS}[p(i)] \)

both of which are to the *left* of \( \text{WIS}[i] \)

*(somewhere)*
The dependency graph

all of the dependencies go left...

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)

This suggests another iterative algorithm
The dependency graph

all of the dependencies go left...

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

WIS: 

\[ WIS[p(i)] \quad WIS[i] \]

\[ WIS[i - 1] \]

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)

This suggests another iterative algorithm

Fill in the array from the left again
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)

This suggests another iterative algorithm

Fill in the array from the left again

all of the dependencies go left...
The dependency graph

all of the dependencies go left...

What information do we need to compute \( \text{WIS}[i] \)?

to compute \( \text{WIS}[i] \) we need \( \text{WIS}[i-1] \) and \( \text{WIS}[p(i)] \)

both of which are to the left of \( \text{WIS}[i] \)

(somewhere)
The dependency graph

all of the dependencies go left...

The array

WIS:  

\[ WIS[p(i)] \quad WIS[i] \]

\[ WIS[i - 1] \]

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)

This suggests another iterative algorithm

Fill in the array from the left again
The dependency graph

all of the dependencies go left...

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)

This suggests another iterative algorithm

Fill in the array from the left again
The dependency graph

all of the dependencies go left…

The array

\[
\begin{align*}
\text{WIS:} & \quad \text{WIS}[p(i)] \quad \text{WIS}[i] \\
& \quad \text{WIS}[i - 1]
\end{align*}
\]

What information do we need to compute \( \text{WIS}[i] \)?

to compute \( \text{WIS}[i] \) we need \( \text{WIS}[i - 1] \) and \( \text{WIS}[p(i)] \)

both of which are to the left of \( \text{WIS}[i] \) (somewhere)

This suggests another iterative algorithm

Fill in the array from the left again
The dependency graph

all of the dependencies go left...

The array

\[ WIS[p(i)] \quad WIS[i] \]

WIS: 

\[ WIS[i-1] \]

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

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)

This suggests another iterative algorithm

Fill in the array from the left again

all of the dependencies go left...
The dependency graph

all of the dependencies go left...

The array

WIS: 

WIS[\(p(i)\)]    WIS[i]

WIS[i - 1]

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)

This suggests another iterative algorithm

Fill in the array from the left again
4. Derive an iterative algorithm

This is an iterative dynamic programming algorithm for Weighted Interval Scheduling

\[
\text{ITWIS}(n)
\]

\[
\begin{align*}
\text{If} \quad (i = 0) \\
\quad \text{Return} \ 0 \\
\text{For} \quad i = 1 \ \text{to} \ n \\
\quad \text{WIS}[i] = \max \left( \text{WIS}[i-1], \text{WIS}[p(i)] + w_i \right) \\
\text{Return} \ \text{WIS}[i]
\end{align*}
\]

it runs in \(O(n)\) time
4. Derive an iterative algorithm

This is an iterative dynamic programming algorithm for Weighted Interval Scheduling. It runs in $O(n)$ time, but it requires that you precomputed all the $p(i)$ values.

\[
\text{ITWIS}(n) =
\begin{align*}
\text{If } (i = 0) & \quad \text{Return 0} \\
\text{For } i = 1 \text{ to } n & \\
\quad \text{WIS}[i] = \max(WIS[i - 1], WIS[p(i)] + w_i) \\
\text{Return WIS}[i]
\end{align*}
\]
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

$$f_j < s_i < f_{j+1}.$$
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?

\text{ItWIS}(n)$ finds the weight of the optimal schedule but doesn’t find the actual schedule

\begin{center}
\begin{tabular}{|p{0.9\textwidth}|}
\hline
\textbf{ItWIS}(n) \\
\hline
If \(i = 0\) \\
\quad Return 0 \\
For \(i = 1\) to \(n\) \\
\quad \text{WIS}[i] = \max (\text{WIS}[i - 1], \text{WIS}[p(i)] + w_i) \\
\quad Return \text{WIS}[i] \\
\hline
\end{tabular}
\end{center}
Wait, did you want the actual schedule?

\( \text{ITWIS}(n) \) finds the weight of the optimal schedule but doesn’t find the actual schedule

\[
\begin{align*}
\text{ITWIS}(n) & \\
\text{If } (i = 0) & \quad \text{Return } 0 \\
\text{For } i = 1 \text{ to } n & \\
\text{WIS}[i] & = \max (\text{WIS}[i - 1], \text{WIS}[p(i)] + w_i) \\
\text{Return } \text{WIS}[i]
\end{align*}
\]

There is an optimal schedule for \( \{1, 2, \ldots, i\} \) containing interval \( i \) if and only if

\[ \text{WIS}[i - 1] \leq \text{WIS}[p(i)] + w_i \]
Wait, did you want the actual schedule?

\( \text{ItWIS}(n) \) finds the weight of the optimal schedule but doesn’t find the actual schedule

\[
\text{ItWIS}(n)
\]

\[
\text{If } (i = 0) \\
\text{Return 0} \\
\text{For } i = 1 \text{ to } n \\
\quad \text{WIS}[i] = \max (\text{WIS}[i - 1], \text{WIS}[p(i)] + w_i) \\
\text{Return WIS}[i]
\]

There is an optimal schedule for \( \{1, 2, \ldots, i\} \) containing interval \( i \) if and only if

\[
\text{WIS}[i - 1] \leq \text{WIS}[p(i)] + w_i
\]

(by the argument we saw earlier)
Wait, did you want the actual schedule?

**ItWIS**($n$) finds the weight of the optimal schedule and **FINDWIS**($n$) finds the actual schedule.

<table>
<thead>
<tr>
<th>ItWIS($n$)</th>
<th>FINDWIS($i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $(i = 0)$</td>
<td>If $(i = 0)$</td>
</tr>
<tr>
<td>Return 0</td>
<td>Return nothing</td>
</tr>
<tr>
<td>For $i = 1$ to $n$</td>
<td>If $WIS[i - 1] \leq WIS[p(i)] + w_i$</td>
</tr>
<tr>
<td>$WIS[i] = \max (WIS[i - 1], WIS[p(i)] + w_i)$</td>
<td>Return FINDWIS($p(i)$) then $i$</td>
</tr>
<tr>
<td>Return $WIS[i]$</td>
<td>Return FINDWIS($i - 1$)</td>
</tr>
</tbody>
</table>

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing interval $i$ if and only if

$$WIS[i - 1] \leq WIS[p(i)] + w_i$$

(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?

**I**TWIS($n$) finds the weight of the optimal schedule
and **FINDWIS($n$)** finds the actual schedule

<table>
<thead>
<tr>
<th>ITWIS($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ($i = 0$)</td>
</tr>
<tr>
<td>Return 0</td>
</tr>
<tr>
<td>For $i = 1$ to $n$</td>
</tr>
<tr>
<td>[ WIS[i] = \max(WIS[i - 1], WIS[p(i)] + w_i) ]</td>
</tr>
<tr>
<td>Return $WIS[i]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FINDWIS($i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ($i = 0$)</td>
</tr>
<tr>
<td>Return nothing</td>
</tr>
<tr>
<td>If $WIS[i - 1] \leq WIS[p(i)] + w_i$</td>
</tr>
<tr>
<td>Return FINDWIS($p(i)$) then $i$</td>
</tr>
<tr>
<td>Return FINDWIS($i - 1$)</td>
</tr>
</tbody>
</table>

There is an optimal schedule for $\{1, 2, \ldots, i\}$ containing

interval $i$ if and only if

\[ WIS[i - 1] \leq WIS[p(i)] + w_i \]

(by the argument we saw earlier)

This is called **backtracking** and works for lots of Dynamic Programming algorithms
The final algorithm

\textbf{ItWIS}(n)\ finds\ the\ weight\ of\ the\ optimal\ schedule
and \textbf{FINDWIS}(n)\ finds\ the\ actual\ schedule

\begin{tabular}{|l|}
\hline
\textbf{ItWIS}(n) \vspace{1em} \\
\hline
\text{If } (i = 0) \\
\quad \text{Return 0} \\
\text{For } i = 1 \text{ to } n \\
\quad \text{WIS}[i] = \max (\text{WIS}[i - 1], \text{WIS}[p(i)] + w_i) \\
\text{Return WIS}[i] \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\textbf{FINDWIS}(i) \vspace{1em} \\
\hline
\text{If } (i = 0) \\
\quad \text{Return nothing} \\
\text{If } \text{WIS}[i - 1] \leq \text{WIS}[p(i)] + w_i \\
\quad \text{Return FINDWIS}(p(i)) \text{ then } i \\
\text{Return FINDWIS}(i - 1) \\
\hline
\end{tabular}

The final algorithm:

\textbf{Step 1:} Find all the \( p(i) \)\ values

\textbf{Step 2:} Run \textbf{ITWIS}(n)\ to find the optimal weight

\textbf{Step 3:} Run \textbf{FINDWIS}(n)\ to find the schedule
The final algorithm

\textbf{\textsc{ItWIS}}(n) finds the weight of the optimal schedule
and \textbf{\textsc{FindWIS}}(n) finds the actual schedule

\begin{align*}
\textbf{\textsc{ItWIS}}(n) & \\
\text{If (}i = 0) & \quad \text{Return 0} \\
\text{For } i = 1 \text{ to } n & \\
\quad & \quad \text{WIS}[i] = \max (\text{WIS}[i - 1], \text{WIS}[p(i)] + w_i) \\
\text{Return } \text{WIS}[i] & \\
\end{align*}

\begin{align*}
\textbf{\textsc{FindWIS}}(i) & \\
\text{If (}i = 0) & \quad \text{Return nothing} \\
\text{If } \text{WIS}[i - 1] \leq \text{WIS}[p(i)] + w_i & \\
\quad & \quad \text{Return } \text{FindWIS}(p(i)) \text{ then } i \\
\text{Return } \text{FindWIS}(i - 1) & \\
\end{align*}

The final algorithm:

\begin{align*}
\textbf{Step 1:} & \text{ Find all the } p(i) \text{ values} \\
\textbf{Step 2:} & \text{ Run } \textbf{\textsc{ItWIS}}(n) \text{ to find the optimal weight} \\
\textbf{Step 3:} & \text{ Run } \textbf{\textsc{FindWIS}}(n) \text{ to find the schedule} \\
\end{align*}

\textit{O}(n \log n) \text{ time}
The final algorithm

\( \text{ITWIS}(n) \) finds the weight of the optimal schedule
and \( \text{FINDWIS}(n) \) finds the actual schedule

**ITWIS(n)**

<table>
<thead>
<tr>
<th>If ( i = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return 0</td>
</tr>
<tr>
<td>For ( i = 1 ) to ( n )</td>
</tr>
<tr>
<td>( \text{WIS}[i] = \max(\text{WIS}[i-1], \text{WIS}[p(i)] + w_i) )</td>
</tr>
<tr>
<td>Return ( \text{WIS}[i] )</td>
</tr>
</tbody>
</table>

**FINDWIS(i)**

<table>
<thead>
<tr>
<th>If ( i = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return nothing</td>
</tr>
<tr>
<td>If ( \text{WIS}[i-1] \leq \text{WIS}[p(i)] + w_i )</td>
</tr>
<tr>
<td>Return ( \text{FINDWIS}(p(i)) ) then ( i )</td>
</tr>
<tr>
<td>Return ( \text{FINDWIS}(i-1) )</td>
</tr>
</tbody>
</table>

The final algorithm:

**Step 1:** Find all the \( p(i) \) values \( O(n \log n) \) time

**Step 2:** Run \( \text{ITWIS}(n) \) to find the optimal weight \( O(n) \) time

**Step 3:** Run \( \text{FINDWIS}(n) \) to find the schedule
The final algorithm

\[ \text{ITWIS}(n) \] finds the weight of the optimal schedule
and \[ \text{FINDWIS}(n) \] finds the actual schedule

**ITWIS**(\(n\))

- If \( i = 0 \)
  - Return 0
- For \( i = 1 \) to \( n \)
  - \( \text{WIS}[i] = \max (\text{WIS}[i-1], \text{WIS}[p(i)] + w_i) \)
- Return \( \text{WIS}[i] \)

**FINDWIS**(\(i\))

- If \( i = 0 \)
  - Return nothing
- If \( \text{WIS}[i-1] \leq \text{WIS}[p(i)] + w_i \)
  - Return \( \text{FINDWIS}(p(i)) \)
- Then \( i \)
  - Return \( \text{FINDWIS}(i-1) \)

The final algorithm:

- **Step 1:** Find all the \( p(i) \) values \( O(n \log n) \) time
- **Step 2:** Run \( \text{ITWIS}(n) \) to find the optimal weight \( O(n) \) time
- **Step 3:** Run \( \text{FINDWIS}(n) \) to find the schedule \( O(n) \) time
The final algorithm

\( \text{ITWIS}(n) \) finds the weight of the optimal schedule and \( \text{FINDWIS}(n) \) finds the actual schedule.

**ITWIS(\( n \))**

If \( (i = 0) \)
   Return 0
For \( i = 1 \) to \( n \)
   \( \text{WIS}[i] = \max(\text{WIS}[i-1], \text{WIS}[p(i)] + w_i) \)
Return \( \text{WIS}[i] \)

**FINDWIS(\( i \))**

If \( (i = 0) \)
   Return nothing
If \( \text{WIS}[i - 1] \leq \text{WIS}[p(i)] + w_i \)
   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 \( \text{ITWIS}(n) \) to find the optimal weight \( O(n) \) time

**Step 3:** Run \( \text{FINDWIS}(n) \) to find the schedule \( O(n) \) time

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*