# Technical Slide

**Module 3: Common Struggles**

1. Lesson 1: 3.2. Greedy Algorithms
    - Video 1.1: Warm-up
    - Video 1.2: Proving Correctness
    - Video 1.3: Activity Selection
    - Video 1.4: Maximum Scalar Product
    - Video 1.5: Greedy Ordering

# Greedy Algorithm

- Build a solution piece by piece
- At each step, choose the most profitable piece

# Largest Number

## Largest number

Input: A sequence of digits $d_0, \ldots, d_{n-1}$(i.e., integers from 0 to 9).

Output: The largest number that can be obtained by concatenating the given digits in some order.

# Largest Number

## Largest number

Input: A sequence of digits $d_0, \ldots, d_{n-1}$(i.e., integers from 0 to 9).

Output: The largest number that can be obtained by concatenating the given digits in some order.

## Example

Input: 2, 3, 9, 3, 2

Output: 93322

# Idea

- Start with the largest digit

# Idea

- Start with the largest digit
- What is left is the same problem: concatenate the remaining digits to get as large number as possible

# Code

```python
def largest(digits):
  result = []

  while len(digits) > 0:
    max_digit = max(digits)
    digits.remove(max_digit)
    result.append(max_digit)

  return "".join(map(str, result))
```

# Code

```
def largest(digits):
  result = []

  while len(digits) > 0:
    max_digit = max(digits)
    digits.remove(max_digit)
    result.append(max_digit)

  return "".join(map(str, result))
```

Running time: $O(n^2)$

# Money Change

## Money change

**Input:** Non-negative integer $m$.

**Output:** The minimum number of coins with denominations 1, 5, and 10 that changes $m$.

# Money Change

## Money change

Input: Non-negative integer *m*.

Output: The minimum number of coins with denominations 1, 5, and 10 that changes *m*.

## Example

Input: 28

Output: 6 $(10 + 10 + 5 + 1 + 1 + 1)$

# Idea

- Take a coin $c$ with the largest denomination that does not exceed $m$
- What is left is the same problem: change $(m - c)$ with the minimum number of coins

# Code

```python
def change(m, coins):
  result = []

  while m > 0:
    max_coin = max(c for c in coins
                   if c <= m)
    m -= max_coin
    result.append(max_coin)

  return "+".join(map(str, result))
```

# Code

```
def change(m, coins):
    result = []

    while m > 0:
        max_coin = max(c for c in coins
                       if c <= m)
        m -= max_coin
        result.append(max_coin)

    return "+".join(map(str, result))
```

```
change(28, [1, 5, 10])
```

```
10+10+5+1+1+1
```

# Code

```
def change(m, coins):
    result = []

    while m > 0:
        max_coin = max(c for c in coins
                       if c <= m)
        m -= max_coin
        result.append(max_coin)

    return "+".join(map(str, result))
```

```
change(28, [1, 5, 10])
```

```
10+10+5+1+1+1
```

Running time: $O(m \cdot |coins|)$

# Analysis

- Compact and efficient solutions

# Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!

# Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!
    - `largest([2, 21])` returns 212 instead of 221

# Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!
  - `largest([2, 21])` returns 212 instead of 221
  - `change(8, [1, 4, 6])` returns 6+1+1 instead of 4+4

# Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!
  - `largest([2, 21])` returns 212 instead of 221
  - `change(8, [1, 4, 6])` returns 6+1+1 instead of 4+4
- A priori, there should be no reason why a sequence of locally optimal moves leads to a global optimum

# Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!
  - `largest([2, 21])` returns 212 instead of 221
  - `change(8, [1, 4, 6])` returns 6+1+1 instead of 4+4
- A priori, there should be no reason why a sequence of locally optimal moves leads to a global optimum
- In rare cases when a greedy strategy works, one should be able to prove its correctness

# Technical Slide

**Module 3: Common Struggles**

❶ Lesson 1: 3.2. Greedy Algorithms

- Video 1.1: Warm-up
- Video 1.2: Proving Correctness
- Video 1.3: Activity Selection
- Video 1.4: Maximum Scalar Product
- Video 1.5: Greedy Ordering

# Proving Correctness

- At each step a greedy algorithm restricts the search space by selecting the most profitable piece of a solution

# Proving Correctness

- At each step a greedy algorithm restricts the search space by selecting the most profitable piece of a solution
  - Largest number: instead of considering all numbers that can be obtained by concatenating the given digits, let's consider only number starting with the maximum digit

# Proving Correctness

- At each step a greedy algorithm restricts the search space by selecting the most profitable piece of a solution
    - Largest number: instead of considering all numbers that can be obtained by concatenating the given digits, let's consider only number starting with the maximum digit
    - Change problem: instead of considering all ways of changing the given amount, let's consider only ways including a coin with the largest denomination

# Proving Correctness

- At each step a greedy algorithm restricts the search space by selecting the most profitable piece of a solution
    - Largest number: instead of considering all numbers that can be obtained by concatenating the given digits, let's consider only number starting with the maximum digit
    - Change problem: instead of considering all ways of changing the given amount, let's consider only ways including a coin with the largest denomination
- One needs to show that the restricted search space contains at least one optimum solution

# Template for Proving Correctness

- Take some optimum solution

# Template for Proving Correctness

- Take some optimum solution
- If it belongs to the restricted search space, then we are done

# Template for Proving Correctness

- Take some optimum solution
- If it belongs to the restricted search space, then we are done
- If it does not belong to the restricted search space, tweak it so that it is still optimum (or even better) and belongs to the restricted search space

# Largest Number: Correctness

## Lemma

Let $N$ be the largest number that can be obtained by concatenating digits $d_0, \ldots, d_{n-1}$ in some order. Then $N$ starts with the largest digit $d_i$.

# Largest Number: Correctness

## Lemma

Let $N$ be the largest number that can be obtained by concatenating digits $d_0, \ldots, d_{n-1}$ in some order. Then $N$ starts with the largest digit $d_i$.

## Proof

Assume the contrary: $N = d_j \alpha d_i \beta$, where $d_j < d_i$ and $\alpha, \beta$ are sequences of digits. But then $N' = d_j \alpha d_i \beta$ is greater than $N$, a contradiction. (It is essential here that $d_i$ and $d_j$ are single digit integers!) □

# Money Change: Correctness

**Lemma**

For any positive integer $m$, there exists an optimal way of changing $m$ using a coin with the largest denomination $D \in \{1, 5, 10\}$ that does not exceed $m$.

# Proof

$m < 5$, $D = 1$: $m$ is changed using 1's only

## Proof

$m < 5$, $D = 1$: $m$ is changed using 1's only

$5 \leq m < 10$, $D = 5$: if 5 is not used, then there are at least five 1's; replace them with 5

## Proof

$m < 5$, $D = 1$: $m$ is changed using 1's only

$5 \leq m < 10$, $D = 5$: if 5 is not used, then there are at least five 1's; replace them with 5

$10 \leq m$, $D = 10$: if there are at least two 5's, replace them with 10; if there is just one 5, then there must be at least five 1's, replace them with 10; if there are no 5's, there must be ten 1's, replace them with 10 □

## Proof

$m < 5$, $D = 1$: $m$ is changed using 1's only

$5 \leq m < 10$, $D = 5$: if 5 is not used, then there are at least five 1's; replace them with 5

$10 \leq m$, $D = 10$: if there are at least two 5's, replace them with 10; if there is just one 5, then there must be at least five 1's, replace them with 10; if there are no 5's, there must be ten 1's, replace them with 10 □

## Observation

It is the last case where the analysis breaks for $\{1, 4, 6\}$

# Technical Slide

**Module 3: Common Struggles**

① Lesson 1: 3.2. Greedy Algorithms

- Video 1.1: Warm-up
- Video 1.2: Proving Correctness
- Video 1.3: Activity Selection
- Video 1.4: Maximum Scalar Product
- Video 1.5: Greedy Ordering

# Activity Selection

## Activity selection

Input: A set of $n$ segments on a line.

Output: The maximum number of non-overlapping segments.

# Activity Selection

## Activity selection

Input: A set of $n$ segments on a line.

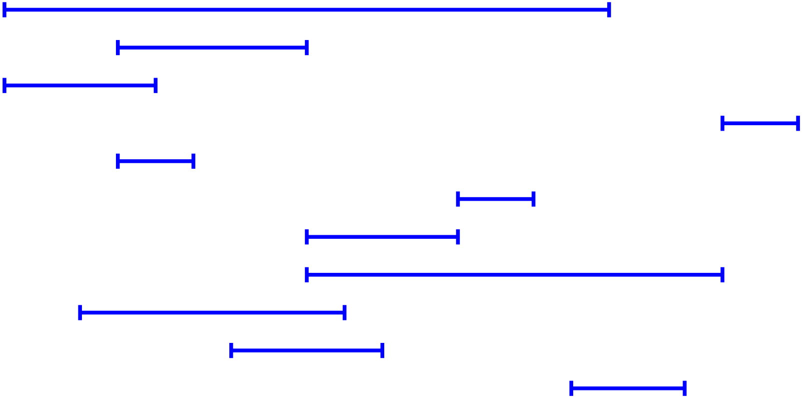
Output: The maximum number of non-overlapping segments.

## Example

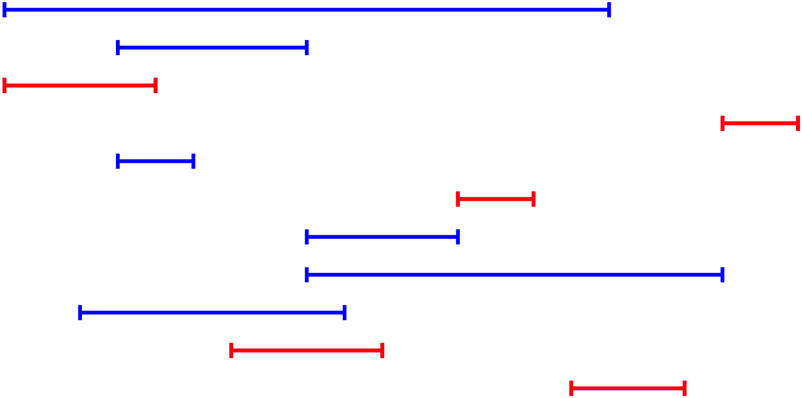
Input: $[2, 6]$, $[1, 4]$, $[7, 9]$, $[3, 8]$

Output: 2 ($[1, 4]$, $[7, 9]$)

# Example



# Example



# Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is

# Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is
- Wild guesses:

# Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is
- Wild guesses:
    - Take the shortest segment

# Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is
- Wild guesses:
    - Take the shortest segment
    - Take the segment with the minimal left endpoint

# Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is
- Wild guesses:
    - Take the shortest segment
    - Take the segment with the minimal left endpoint
    - Take the segment with the minimal right endpoint

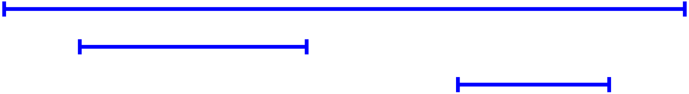
# Counterexamples

- Taking the shortest segment does not work:

# Counterexamples

- Taking the shortest segment does not work:

- Taking the segment with the minimal left endpoint does not work:



# Correct Greedy Strategy

### Lemma

There exists an optimal solution containing the segment with the smallest right endpoint.

# Proof

## Proof

- Let $[a, b]$ be a segment with the smallest right endpoint and let $S$ be an optimal solution such that $[c, d]$ is its segment with the minimal right endpoint.

# Proof

## Proof

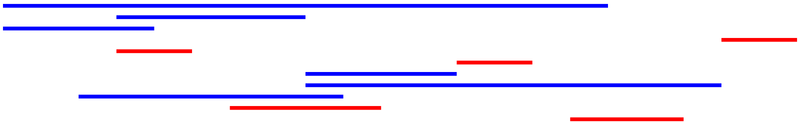
- Let $[a, b]$ be a segment with the smallest right endpoint and let $S$ be an optimal solution such that $[c, d]$ is its segment with the minimal right endpoint.
- $b \leq d$. If $b = d$, nothing needs to be done, so assume that $b < d$

# Proof

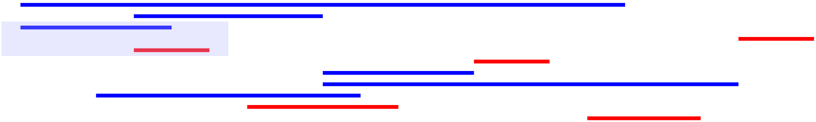
## Proof

- Let $[a, b]$ be a segment with the smallest right endpoint and let $S$ be an optimal solution such that $[c, d]$ is its segment with the minimal right endpoint.
- $b \leq d$. If $b = d$, nothing needs to be done, so assume that $b < d$
- Replace $[c, d]$ with $[a, b]$ in $S$. Then, it is still a solution (if $[c, d]$ does not intersect any other segment in $S$, then neither does $[a, b]$) and it is optimal $\square$

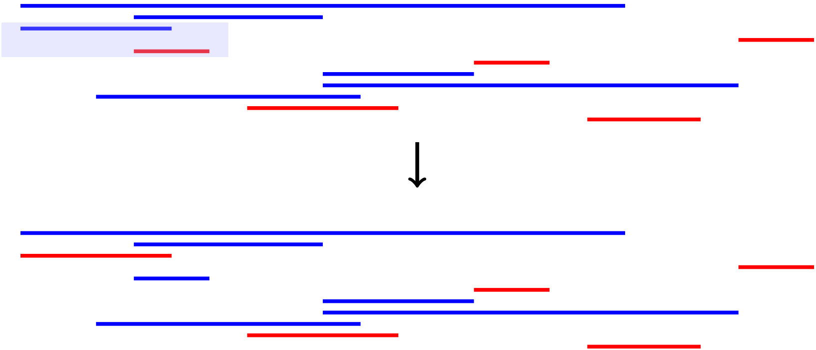
# Visually



# Visually



# Visually



# Algorithm

## Algorithm

- Take the segment with the minimal right endpoint into a solution

# Algorithm

## Algorithm

- Take the segment with the minimal right endpoint into a solution
- Remove all segments that intersect it

# Algorithm

## Algorithm

- Take the segment with the minimal right endpoint into a solution
- Remove all segments that intersect it
- Repeat

# Algorithm

## Algorithm

- Take the segment with the minimal right endpoint into a solution
- Remove all segments that intersect it
- Repeat

## Running time

$O(n^2)$

# Technical Slide

**Module 3: Common Struggles**

❶ Lesson 1: 3.2. Greedy Algorithms
  Video 1.1: Warm-up
  Video 1.2: Proving Correctness
  Video 1.3: Activity Selection
  Video 1.4: Maximum Scalar Product
  Video 1.5: Greedy Ordering

# Maximum Scalar Product

## Maximum Scalar Product

| | |
|---|---|
| Input: | Two sequences of $n$ integers:<br>$A = [a_0, \ldots, a_{n-1}]$ and<br>$B = [b_0, \ldots, b_{n-1}]$. |
| Output: | The maximum value of<br>$a_0 c_0 + \ldots + a_{n-1} c_{n-1}$, where $c_0, \ldots, c_{n-1}$<br>is a permutation of $b_0, \ldots, b_{n-1}$. |

# Maximum Scalar Product

## Maximum Scalar Product

Input: Two sequences of $n$ integers: $A = [a_0, \ldots, a_{n-1}]$ and $B = [b_0, \ldots, b_{n-1}]$.

Output: The maximum value of $a_0 c_0 + \ldots + a_{n-1} c_{n-1}$, where $c_0, \ldots, c_{n-1}$ is a permutation of $b_0, \ldots, b_{n-1}$.

## Example

Input: $[2, 3, 9]$, $[7, 4, 2]$

Output: $79$ $(79 = 2 \cdot 2 + 3 \cdot 4 + 9 \cdot 7)$

# Greedy Strategy

## Lemma

There exists an optimal solution where the maximum element $a_i$ of $A$ is paired with the maximum element $b_j$ of $B$.

# Proof

## Proof

- Consider an optimal solution $S$

# Proof

## Proof

- Consider an optimal solution $S$
- If it pairs $a_i$ and $b_j$, then we are done

# Proof

## Proof

- Consider an optimal solution $S$
- If it pairs $a_i$ and $b_j$, then we are done
- Otherwise $S = a_i b_p + a_q b_j + \cdots$

# Proof

## Proof

- Consider an optimal solution $S$
- If it pairs $a_i$ and $b_j$, then we are done
- Otherwise $S = a_i b_p + a_q b_j + \cdots$
- Let's swap these two pairs:
  $S' = a_i b_j + a_q b_p + \cdots$

# Proof

## Proof

- Consider an optimal solution $S$
- If it pairs $a_i$ and $b_j$, then we are done
- Otherwise $S = a_i b_p + a_q b_j + \cdots$
- Let's swap these two pairs:
  $S' = a_i b_j + a_q b_p + \cdots$
- $S'$ is not worse than $S$:

$$\begin{aligned} S' - S &= a_i b_j + a_q b_p - a_i b_p - a_q b_j \\ &= (a_i - a_q)(b_j - b_p) \geq 0 \end{aligned}$$

□

# Code

```python
def scalar_product(A, B):
  assert len(A) == len(B)
  result = 0
  while len(A) > 0:
    am, bm = max(A), max(B)
    result += am * bm
    A.remove(am)
    B.remove(bm)
  return result
```

# Code

```python
def scalar_product(A, B):
    assert len(A) == len(B)
    result = 0
    while len(A) > 0:
        am, bm = max(A), max(B)
        result += am * bm
        A.remove(am)
        B.remove(bm)
    return result
```

Running time: $O(n^2)$

# Technical Slide

**Module 3: Common Struggles**

1. Lesson 1: 3.2. Greedy Algorithms
   - Video 1.1: Warm-up
   - Video 1.2: Proving Correctness
   - Video 1.3: Activity Selection
   - Video 1.4: Maximum Scalar Product
   - Video 1.5: Greedy Ordering

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
    - Largest number: the larger digit is better

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
  - Largest number: the larger digit is better
  - Money change: the larger denomination is better

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
    - Largest number: the larger digit is better
    - Money change: the larger denomination is better
    - Activity selection: the activity with a smaller ending time is better

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
    - Largest number: the larger digit is better
    - Money change: the larger denomination is better
    - Activity selection: the activity with a smaller ending time is better
    - Scalar product: the larger $b_i$ is better

# Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
    - Largest number: the larger digit is better
    - Money change: the larger denomination is better
    - Activity selection: the activity with a smaller ending time is better
    - Scalar product: the larger $b_i$ is better
- Then, everything boils down to sorting with respect to this ordering

# Compact Code

```
def largest(digits):
    return "".join(map(str, sorted(digits, reverse=True)))
```

```
def change(m):
    return m // 10 + (m % 10) // 5 + (m % 5)
```

```
def scalar_product(A, B):
    assert len(A) == len(B)
    A, B = sorted(A), sorted(B)
    return sum(A[i] * B[i] for i in range(len(A)))
```

# Ordering Correctness

Proving that a specific ordering leads to a correct greedy strategy: if in a solution $a_1, a_2, \ldots, a_n$, $a_i \not\preceq a_{i+1}$ for some $i$, then swapping $a_i$ and $a_{i+1}$ can only improve this solution

# Summary

- Construct a solution piece by piece, always choosing the most profitable piece

# Summary

- Construct a solution piece by piece, always choosing the most profitable piece
- Pros: efficient, easy to implement

# Summary

- Construct a solution piece by piece, always choosing the most profitable piece
- Pros: efficient, easy to implement
- Cons: rarely work, not so easy to prove correctness