Technical Slide

Module 5: Dynamic Programming

1. Lesson 1: Longest Increasing Subsequence
   - Video 1.1: Warm-up
   - Video 1.2: Subproblems and Recurrence Relation
   - Video 1.3: Reconstructing a Solution
   - Video 1.4: Subproblems Revisited

2. Lesson 2: Edit Distance
   - Video 2.1: Algorithm
   - Video 2.2: Reconstructing a Solution
   - Video 2.3: Final Remarks

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems
- A solution is usually not so easy to find, but when found, is easily implementable

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems
- A solution is usually not so easy to find, but when found, is easily implementable
- Need a lot of practice!

Fibonacci numbers

Fibonacci numbers

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Computing Fibonacci Numbers

Computing F_n

Input: An integer $n \geq 0$.

Output: The n-th Fibonacci number F_n.

Computing Fibonacci Numbers

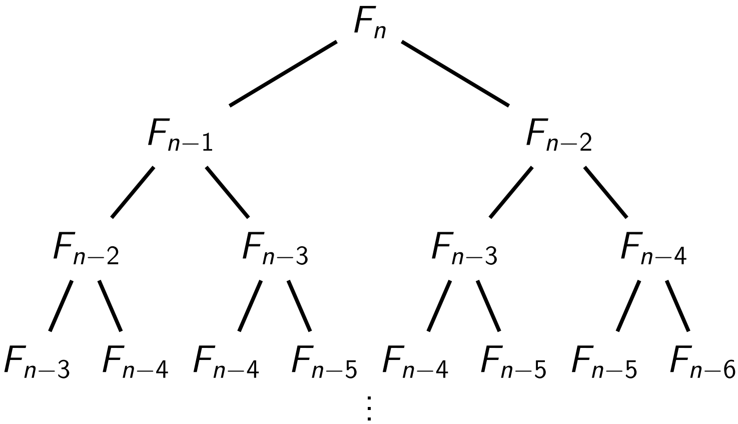
Computing F_n

Input: An integer $n \geq 0$.

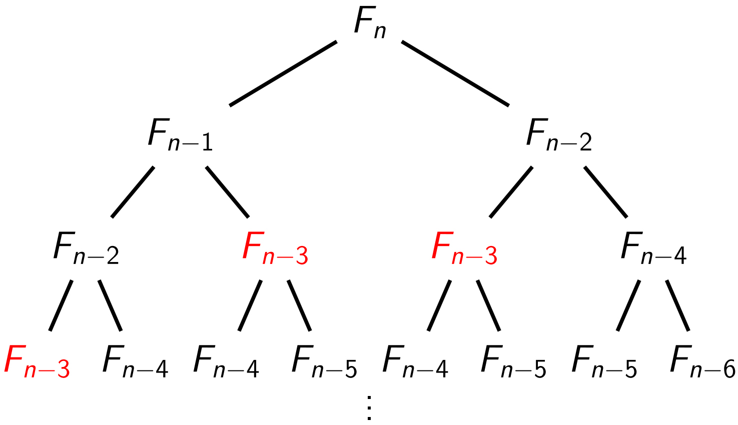
Output: The n-th Fibonacci number F_n.

```
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
```

Recursion Tree



Recursion Tree



Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \phi^n$, where $\phi = 1.618\ldots$ is the golden ratio

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \phi^n$, where $\phi = 1.618\ldots$ is the golden ratio
- E.g., F_{150} is already 31 decimal digits long

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \phi^n$, where $\phi = 1.618\ldots$ is the golden ratio
- E.g., F_{150} is already 31 decimal digits long
- The Sun may die before your computer returns F_{150}

Reason

- Many computations are repeated

Reason

- Many computations are repeated
- *"Those who cannot remember the past are condemned to repeat it."* (George Santayana)

Reason

- Many computations are repeated
- *"Those who cannot remember the past are condemned to repeat it."* (George Santayana)
- A simple, but crucial idea: instead of recomputing the intermediate results, let's store them once they are computed

Memoization

```
def fib(n):
  if n <= 1:
    return n
  return fib(n - 1) + fib(n - 2)
```

Memoization

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

```
T = dict()

def fib(n):
    if n not in T:
        if n <= 1:
            T[n] = n
        else:
            T[n] = fib(n - 1) + fib(n - 2)

    return T[n]
```

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
    1. $F_0 = 0$, $F_1 = 1$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0$, $F_1 = 1$
  2. $F_2 = 0 + 1 = 1$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0$, $F_1 = 1$
  2. $F_2 = 0 + 1 = 1$
  3. $F_3 = 1 + 1 = 2$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
    1. $F_0 = 0$, $F_1 = 1$
    2. $F_2 = 0 + 1 = 1$
    3. $F_3 = 1 + 1 = 2$
    4. $F_4 = 1 + 2 = 3$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0$, $F_1 = 1$
  2. $F_2 = 0 + 1 = 1$
  3. $F_3 = 1 + 1 = 2$
  4. $F_4 = 1 + 2 = 3$
  5. $F_5 = 2 + 3 = 5$

Iterative Algorithm

```
def fib(n):
  T = [None] * (n + 1)
  T[0], T[1] = 0, 1

  for i in range(2, n + 1):
    T[i] = T[i - 1] + T[i - 2]

  return T[n]
```

Hm Again...

But do we really need to waste so much space?

Hm Again...

But do we really need to waste so much space?

```
def fib(n):
  if n <= 1:
    return n

  previous, current = 0, 1
  for _ in range(n - 1):
    new_current = previous + current
    previous, current = current, new_current

  return current
```

Running Time

- $O(n)$ additions

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$
- In theory: we should not treat such additions as basic operations

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$
- In theory: we should not treat such additions as basic operations
- In practice: just F_{100} does not fit into a 64-bit integer type anymore, hence we need bignum arithmetic

Summary

- The key idea of dynamic programming: avoid recomputing the same thing again!

Summary

- The key idea of dynamic programming: avoid recomputing the same thing again!
- At the same time, the case of Fibonacci numbers is a slightly artificial example of dynamic programming since it is clear from the very beginning what intermediate results we need to compute the final result

Technical Slide

Module 5: Dynamic Programming

1. Lesson 1: Longest Increasing Subsequence
   - Video 1.1: Warm-up
   - Video 1.2: Subproblems and Recurrence Relation
   - Video 1.3: Reconstructing a Solution
   - Video 1.4: Subproblems Revisited
2. Lesson 2: Edit Distance
   - Video 2.1: Algorithm
   - Video 2.2: Reconstructing a Solution
   - Video 2.3: Final Remarks

Longest Increasing Subsequence

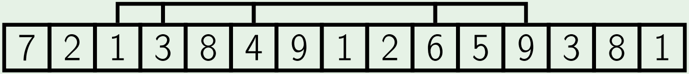
Longest increasing subsequence

Input: An array $A = [a_0, a_1, \ldots, a_{n-1}]$.

Output: A longest increasing subsequence (LIS), i.e., $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $i_1 < i_2 < \ldots < i_k$, $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$, and k is maximal.

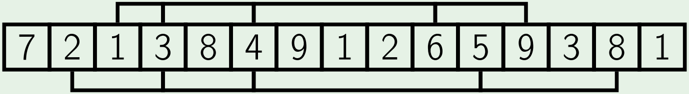
Example

Example



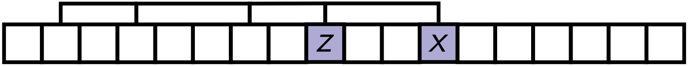
Example

Example



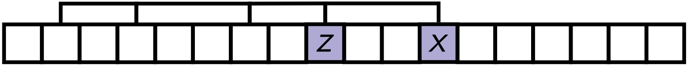
Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:



Analyzing an Optimal Solution

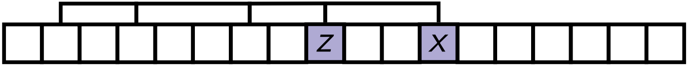
- Consider the last element x of an optimal increasing subsequence and its previous element z:



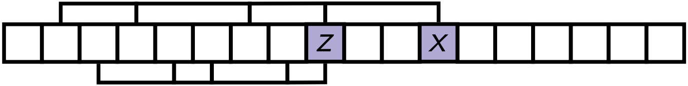
- First of all, $z < x$

Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:

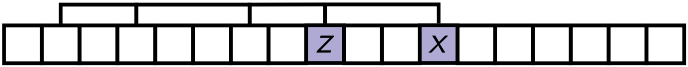


- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:

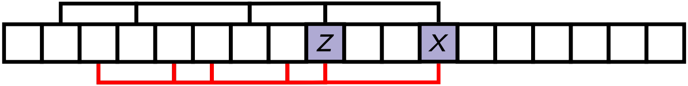


Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:

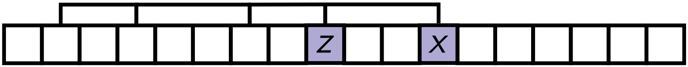


- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:

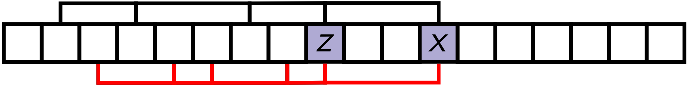


Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:



- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:



- Optimal substructure by "cut-and-paste" trick

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1 + \max\{LIS(j) \colon j < i \text{ and } A[j] < A[i]\}$$

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1 + \max\{LIS(j) \colon j < i \text{ and } A[j] < A[i]\}$$

- Convention: maximum of an empty set is equal to zero

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1 + \max\{LIS(j) \colon j < i \text{ and } A[j] < A[i]\}$$

- Convention: maximum of an empty set is equal to zero
- Base case: $LIS(0) = 1$

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$
- Initially, T is empty. When $LIS(i)$ is computed, we store its value at $T[i]$ (so that we will never recompute $LIS(i)$ again)

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$
- Initially, T is empty. When $LIS(i)$ is computed, we store its value at $T[i]$ (so that we will never recompute $LIS(i)$ again)
- The exact data structure behind T is not that important at this point: it could be an array or a hash table

Memoization

```
T = dict()

def lis(A, i):
  if i not in T:
    T[i] = 1

    for j in range(i):
      if A[j] < A[i]:
        T[i] = max(T[i], lis(A, j) + 1)

  return T[i]

A = [7, 2, 1, 3, 8, 4, 9, 1, 2, 6, 5, 9, 3]
print(max(lis(A, i) for i in range(len(A))))
```

Running Time

The running time is quadratic ($O(n^2)$): there are n "serious" recursive calls (that are not just table look-ups), each of them needs time $O(n)$ (not counting the inner recursive calls)

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n-1$

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n-1$
- Reasonable choice of a data structure for T: an array of size n

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n-1$
- Reasonable choice of a data structure for T: an array of size n
- Moreover, one can fill in this array iteratively instead of recursively

Iterative Algorithm

```
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

Iterative Algorithm

```
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

- Crucial property: when computing $T[i]$, $T[j]$ for all $j < i$ have already been computed

Iterative Algorithm

```
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

- Crucial property: when computing $T[i]$, $T[j]$ for all $j < i$ have already been computed
- Running time: $O(n^2)$

Technical Slide

Module 5: Dynamic Programming

1. Lesson 1: Longest Increasing Subsequence
   - Video 1.1: Warm-up
   - Video 1.2: Subproblems and Recurrence Relation
   - Video 1.3: Reconstructing a Solution
   - Video 1.4: Subproblems Revisited

2. Lesson 2: Edit Distance
   - Video 2.1: Algorithm
   - Video 2.2: Reconstructing a Solution
   - Video 2.3: Final Remarks

Reconstructing a Solution

- How to reconstruct an optimal IS?

Reconstructing a Solution

- How to reconstruct an optimal IS?
- In order to reconstruct it, for each subproblem we will keep its optimal value and a choice leading to this value

Adjusting the Algorithm

```
def lis(A):
    T = [None] * len(A)
    prev = [None] * len(A)

    for i in range(len(A)):
        T[i] = 1
        prev[i] = -1
        for j in range(i):
            if A[j] < A[i] and T[i] < T[j] + 1:
                T[i] = T[j] + 1
                prev[i] = j
```

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
`prev`	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Unwinding Solution

```
last = 0
for i in range(1, len(A)):
    if T[i] > T[last]:
        last = i

lis= []
current = last
while current >= 0:
    lis.append(current)
    current = prev[current]
lis.reverse()
return [A[i] for i in lis]
```

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization
- A recursive algorithm, in turn, can be converted into an iterative one

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization
- A recursive algorithm, in turn, can be converted into an iterative one
- An optimal solution can be recovered either by using an additional bookkeeping info or by using the computed solutions to all subproblems

Technical Slide

Module 5: Dynamic Programming

1. Lesson 1: Longest Increasing Subsequence
   - Video 1.1: Warm-up
   - Video 1.2: Subproblems and Recurrence Relation
   - Video 1.3: Reconstructing a Solution
   - Video 1.4: Subproblems Revisited

2. Lesson 2: Edit Distance
   - Video 2.1: Algorithm
   - Video 2.2: Reconstructing a Solution
   - Video 2.3: Final Remarks

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm
- In the previous video, we arrived at a reasonable subproblem by analyzing the structure of an optimal solution

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm
- In the previous video, we arrived at a reasonable subproblem by analyzing the structure of an optimal solution
- In this video, we'll provide an alternative way of arriving at subproblems: implement a naive brute force solution, then optimize it

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
  - Start with an empty sequence

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
  - Start with an empty sequence
  - Extend it element by element recursively

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
  - Start with an empty sequence
  - Extend it element by element recursively
  - Keep track of the length of the sequence

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
  - Start with an empty sequence
  - Extend it element by element recursively
  - Keep track of the length of the sequence
- This is going to be slow, but not to worry: we will optimize it later

Brute Force: Code

```python
def lis(A, seq):
    result = len(seq)

    if len(seq) == 0:
        last_index = -1
        last_element = float("-inf")
    else:
        last_index = seq[-1]
        last_element = A[last_index]

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result, lis(A, seq + [i]))

    return result

print(lis(A=[7, 2, 1, 3, 8, 4, 9], seq=[]))
```

Optimizing

- At each step, we are trying to extend the current sequence

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call
- At the same time, code inspection reveals that we are not using all of the sequence: we are only interested in its last element and its length

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call
- At the same time, code inspection reveals that we are not using all of the sequence: we are only interested in its last element and its length
- Let's optimize!

Optimized Code

```python
def lis(A, seq_len, last_index):
    if last_index == -1:
        last_element = float("-inf")
    else:
        last_element = A[last_index]

    result = seq_len

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result,
                         lis(A, seq_len + 1, i))

    return result

print(lis([3, 2, 7, 8, 9, 5, 8], 0, -1))
```

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any x, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any x, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`
- Hence, can optimize the code as follows: `max(result, 1 + seq_len + extend(A, 0, i))`

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any x, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`
- Hence, can optimize the code as follows: `max(result, 1 + seq_len + extend(A, 0, i))`
- Excludes `seq_len` from the list of parameters!

Resulting Code

```
def lis(A, last_index):
  if last_index == -1:
    last_element = float("-inf")
  else:
    last_element = A[last_index]

  result = 0

  for i in range(last_index + 1, len(A)):
    if A[i] > last_element:
      result = max(result, 1 + lis(A, i))

  return result

print(lis([8, 2, 3, 4, 5, 6, 7], -1))
```

Resulting Code

```
def lis(A, last_index):
    if last_index == -1:
        last_element = float("-inf")
    else:
        last_element = A[last_index]

    result = 0

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result, 1 + lis(A, i))

    return result

print(lis([8, 2, 3, 4, 5, 6, 7], -1))
```

It remains to add memoization!

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:
  - Analyze the structure of an optimal solution

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:
  - Analyze the structure of an optimal solution
  - Implement a brute force solution and optimize it

Technical Slide

Module 5: Dynamic Programming

Statement

Edit distance

Input: Two strings $A[0 \dots n-1]$ and $B[0 \dots m-1]$.

Output: The minimal number of insertions, deletions, and substitutions needed to transform A to B. This number is known as edit distance or Levenshtein distance.

Example: EDITING $\rightarrow$ DISTANCE

EDITING

Example: EDITING $\rightarrow$ DISTANCE

EDITING

$\downarrow$ remove E

DITING

Example: EDITING $\to$ DISTANCE

EDITING

$\downarrow$ remove E

DITING

$\downarrow$ insert S

DISTING

Example: EDITING → DISTANCE

EDITING

↓ remove E

DITING

↓ insert S

DISTING

↓ replace I with by A

DISTANG

Example: EDITING $\rightarrow$ DISTANCE

$$\begin{array}{c} \text{EDITING} \\ \big\downarrow \text{remove E} \\ \text{DITING} \\ \big\downarrow \text{insert S} \\ \text{DISTING} \\ \big\downarrow \text{replace I with by A} \\ \text{DISTANG} \\ \big\downarrow \text{replace G with C} \\ \text{DISTANC} \end{array}$$

Example: EDITING → DISTANCE

EDITING

↓ remove E

DITING

↓ insert S

DISTING

↓ replace I with by A

DISTANG

↓ replace G with C

DISTANC

↓ insert E

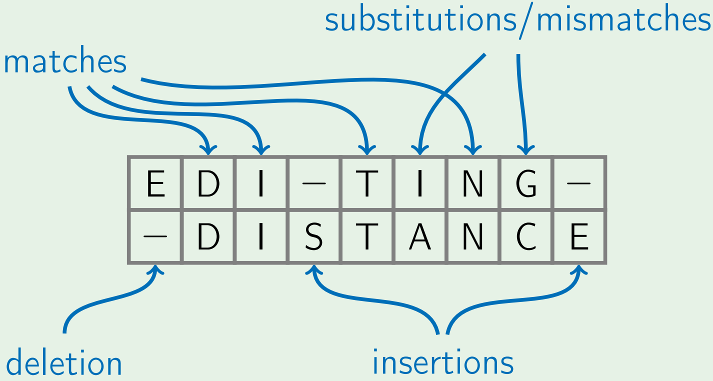
DISTANCE

Example: alignment

E	D	I	–	T	I	N	G	–
–	D	I	S	T	A	N	C	E

cost: 5

Example: alignment



cost: 5

Analyzing an Optimal Alignment

$A[0 \ldots n-1]$
$B[0 \ldots m-1]$

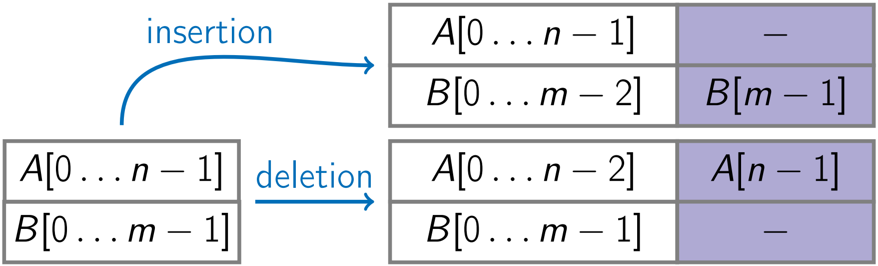
Analyzing an Optimal Alignment

$A[0 \dots n-1]$
$B[0 \dots m-1]$

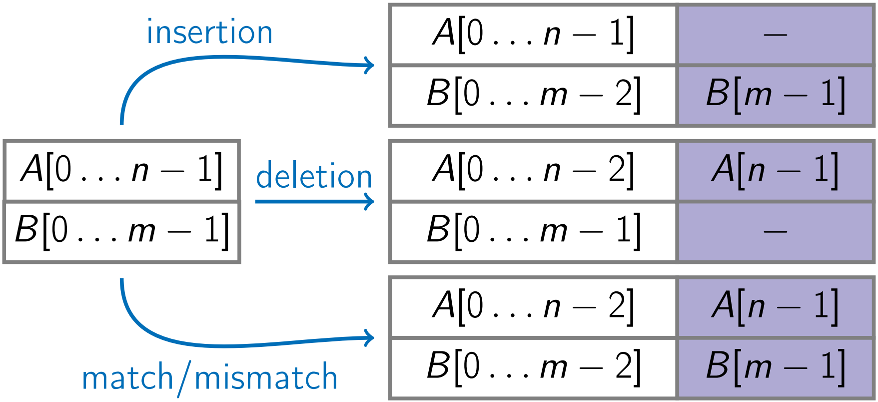
insertion

$A[0 \dots n-1]$	$-$
$B[0 \dots m-2]$	$B[m-1]$

Analyzing an Optimal Alignment



Analyzing an Optimal Alignment



Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.

Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.
- We know for sure that the last column of an optimal alignment is either an insertion, a deletion, or a match/mismatch.

Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.
- We know for sure that the last column of an optimal alignment is either an insertion, a deletion, or a match/mismatch.
- What is left is an optimal alignment of the corresponding two prefixes (by cut-and-paste).

Recurrence Relation

$$ED(i,j) = \min \begin{cases} ED(i, j-1) + 1 \\ ED(i-1, j) + 1 \\ ED(i-1, j-1) + \text{diff}(A[i], B[j]) \end{cases}$$

Recurrence Relation

$$ED(i,j) = \min \begin{cases} ED(i, j-1) + 1 \\ ED(i-1, j) + 1 \\ ED(i-1, j-1) + \mathrm{diff}(A[i], B[j]) \end{cases}$$

Base case: $ED(i, 0) = i$, $ED(0, j) = j$

Recursive Algorithm

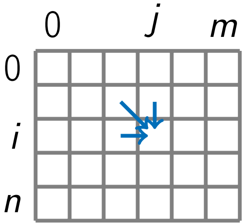
```python
T = dict()

def edit_distance(a, b, i, j):
  if not (i, j) in T:
    if i == 0: T[i, j] = j
    elif j == 0: T[i, j] = i
    else:
      diff = 0 if a[i - 1] == b[j - 1] else 1
      T[i, j] = min(
        edit_distance(a, b, i - 1, j) + 1,
        edit_distance(a, b, i, j - 1) + 1,
        edit_distance(a, b, i - 1, j - 1) + diff)

  return T[i, j]


print(edit_distance(a="editing", b="distance",
                    i=7, j=8))
```

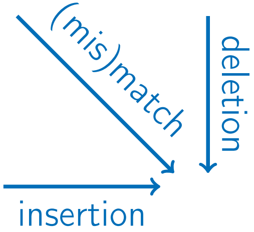
Converting to a Recursive Algorithm

- Use a 2D table to store the intermediate results

Converting to a Recursive Algorithm

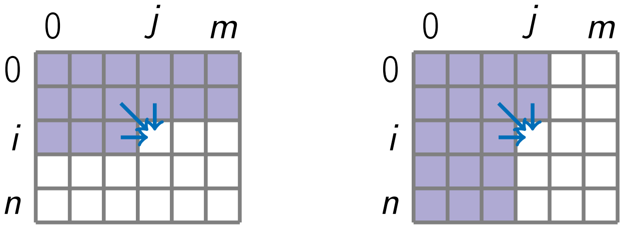
- Use a 2D table to store the intermediate results
- $ED(i, j)$ depends on $ED(i-1, j-1)$, $ED(i-1, j)$, and $ED(i, j-1)$:





Filling the Table

Fill in the table row by row or column by column:



Iterative Algorithm

```
def edit_distance(a, b):
  T = [[float("inf")] * (len(b) + 1)
       for _ in range(len(a) + 1)]
  for i in range(len(a) + 1):
    T[i][0] = i
  for j in range(len(b) + 1):
    T[0][j] = j

  for i in range(1, len(a) + 1):
    for j in range(1, len(b) + 1):
      diff = 0 if a[i - 1] == b[j - 1] else 1
      T[i][j] = min(T[i - 1][j] + 1,
                    T[i][j - 1] + 1,
                    T[i - 1][j - 1] + diff)

  return T[len(a)][len(b)]


print(edit_distance(a="distance", b="editing"))
```

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1							
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1							
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1						
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1						
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1					
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1					
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1	2				
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1	2	3	4	5	6
I	2	2	2	2	1	2	3	4	5
S	3	3	3	3	2	2	3	4	5
T	4	4	4	4	3	2	3	4	5
A	5	5	5	5	4	3	3	4	5
N	6	6	6	6	5	4	4	3	4
C	7	7	7	7	6	5	5	4	4
E	8	8	7	8	7	6	6	5	5

Brute Force

- Recursively construct an alignment column by column

Brute Force

- Recursively construct an alignment column by column
- Then note, that for extending the partially constructed alignment optimally, one only needs to know the already used length of prefix of A and the length of prefix of B

Technical Slide

Module 5: Dynamic Programming

1. Lesson 1: Longest Increasing Subsequence
   - Video 1.1: Warm-up
   - Video 1.2: Subproblems and Recurrence Relation
   - Video 1.3: Reconstructing a Solution
   - Video 1.4: Subproblems Revisited

2. Lesson 2: Edit Distance
   - Video 2.1: Algorithm
   - Video 2.2: Reconstructing a Solution
   - Video 2.3: Final Remarks

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$
- If $ED(i, j) = ED(i-1, j) + 1$, then there exists an optimal alignment whose last column is a deletion

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$
- If $ED(i, j) = ED(i - 1, j) + 1$, then there exists an optimal alignment whose last column is a deletion
- If $ED(i, j) = ED(i, j - 1) + 1$, then there exists an optimal alignment whose last column is an insertion

Reconstructing a Solution

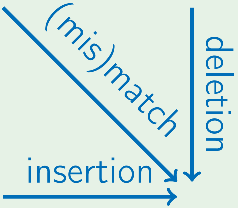
- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$
- If $ED(i, j) = ED(i-1, j) + 1$, then there exists an optimal alignment whose last column is a deletion
- If $ED(i, j) = ED(i, j-1) + 1$, then there exists an optimal alignment whose last column is an insertion
- If $ED(i, j) = ED(i-1, j-1) + \text{diff}(A[i], B[j])$, then match (if $A[i] = B[j]$) or mismatch (if $A[i] \neq B[j]$)

Example

		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5

Example

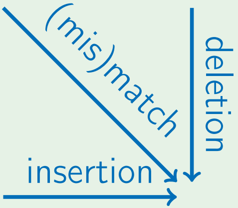
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



E
G

Example

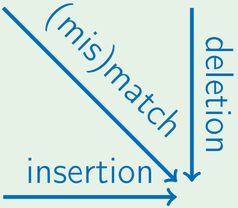
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



C	E
-	G

Example

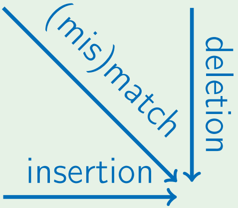
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



N	C	E
N	-	G

Example

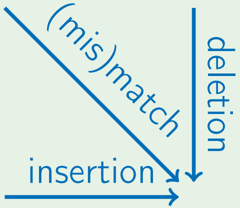
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



A	N	C	E
I	N	-	G

Example

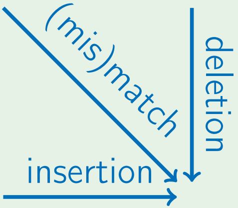
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



T	A	N	C	E
T	I	N	-	G

Example

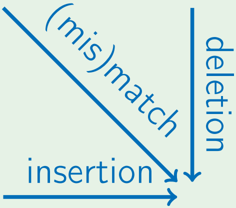
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



S	T	A	N	C	E
-	T	I	N	-	G

Example

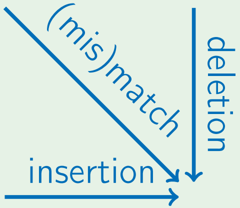
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



I	S	T	A	N	C	E
I	-	T	I	N	-	G

Example

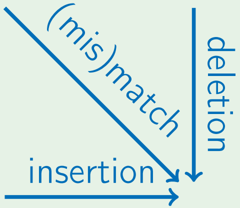
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



D	I	S	T	A	N	C	E
D	I	-	T	I	N	-	G

Example

		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



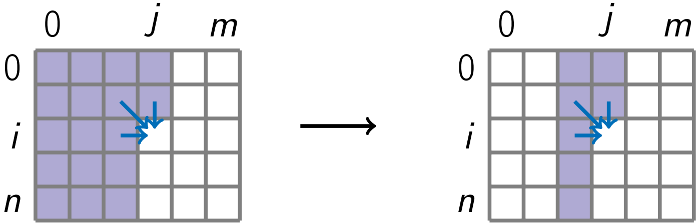
-	D	I	S	T	A	N	C	E
E	D	I	-	T	I	N	-	G

Technical Slide

Module 5: Dynamic Programming

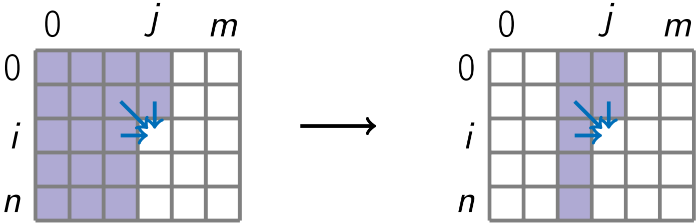
Saving Space

- When filling in the matrix it is enough to keep only the current column and the previous column:



Saving Space

- When filling in the matrix it is enough to keep only the current column and the previous column:



- Thus, one can compute the edit distance of two given strings $A[1 \ldots n]$ and $B[1 \ldots m]$ in time $O(nm)$ and space $O(\min\{n, m\})$.

Reconstructing a Solution

- However we need the whole table to find an actual alignment (we trace an alignment from the bottom right corner to the top left corner)

Reconstructing a Solution

- However we need the whole table to find an actual alignment (we trace an alignment from the bottom right corner to the top left corner)
- There exists an algorithm constructing an optimal alignment in time $O(nm)$ and space $O(n+m)$ (Hirschberg's algorithm)

Weighted Edit Distance

- The cost of insertions, deletions, and substitutions is not necessarily identical
- Spell checking: some substitutions are more likely than others
- Biology: some mutations are more likely than others

Generalized Recurrence Relation

$$\min \begin{cases} ED(i, j-1) + \text{inscost}(B[j]), \\ ED(i-1, j) + \text{delcost}(A[i]), \\ ED(i-1, j-1) + \text{substcost}(A[i], B[j]) \end{cases}$$