Longest Common Subsequence

Simon Moura

25 february 2019

University College London
s.moura@ucl.ac.uk
Slides: https://github.com/simmou/ucl_algo

The Longest Common Subsequence (LCS) problem

Subsequences

Let's consider a **sequence** $X = (x_1, x_2, \ldots, x_{n_x})$ of n_x elements, e.g.

- $X = ABC$ is a sequence, where $x_1 = A$, $x_2 = B$ and $x_3 = C$
- $X = 123$ is a sequence, where $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$

Subsequences

Let's consider a **sequence** $X = (x_1, x_2, \dots, x_{n_x})$ of n_x elements, e.g.

- $X = ABC$ is a sequence, where $x_1 = A$, $x_2 = B$ and $x_3 = C$
- $X = 123$ is a sequence, where $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$

A sequence $Z = (z_1, \dots, z_{n_z})$ is called a **subsequence** of X of size n_z if and only if it can be obtained from X by deleting elements from it, e.g.

- $X =$ Hello, world
- $Z =$ Hello is a subsequence of X

Subsequences

Let's consider a **sequence** $X = (x_1, x_2, \ldots, x_{n_x})$ of n_x elements, e.g.

- $X = ABC$ is a sequence, where $x_1 = A$, $x_2 = B$ and $x_3 = C$
- $X = 123$ is a sequence, where $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$

A sequence $Z = (z_1, \ldots, z_{n_z})$ is called a **subsequence** of X of size n_z if and only if it can be obtained from X by deleting elements from it, e.g.

- $X =$ Hello, world
- $Z =$ Hello is a subsequence of X

A subsequence is not necessarily contiguous:

- $Z =$ Hwd is a subsequence of $X =$Hello, world
- $Z =$ elr is also a subsequence of $X =$ Hello, world
- $Z =$ lled is NOT a subsequence of $X =$ Hello, world

Longest Common Subsequence (LCS)

LCS problem: find the longest subsequence common to multiple sequences.

Consider two sequences $X = (x_1, x_2, \ldots, x_{n_x})$ and $Y = (y_1, y_2, \ldots, y_{n_y})$ of size n_x and n_y respectively.

Longest Common Subsequence (LCS)

LCS problem: find the longest subsequence common to multiple sequences.

Consider two sequences $X = (x_1, x_2, \ldots, x_{n_x})$ and $Y = (y_1, y_2, \ldots, y_{n_y})$ of size n_x and n_y respectively.

- $X =$ "Hello, world", where $x_1 = H, \ldots, x_{n_x} = d$
- $Y =$ "Hello", where $y_1 = H, \ldots, y_{n_y} = o$
- LCS(X, Y): Hello

Longest Common Subsequence (LCS)

LCS problem: find the longest subsequence common to multiple sequences.

Consider two sequences $X = (x_1, x_2, \ldots, x_{n_x})$ and $Y = (y_1, y_2, \ldots, y_{n_y})$ of size n_x and n_y respectively.

- X = "Hello, world", where $x_1 = H, \ldots, x_{n_x} = d$
- Y = "Hello", where $y_1 = H, \ldots, y_{n_y} = o$
- LCS(X, Y): Hello

Longest Common Subsequence (LCS)

LCS problem: find the longest subsequence common to multiple sequences.

Consider two sequences $X = (x_1, x_2, \ldots, x_{n_x})$ and $Y = (y_1, y_2, \ldots, y_{n_y})$ of size n_x and n_y respectively.

- $X =$ "Hello, world", where $x_1 = H, \ldots, x_{n_x} = d$
- $Y =$ "hezlospkard", where $y_1 = H, \ldots, y_{n_y} = o$

Longest Common Subsequence (LCS)

LCS problem: find the longest subsequence common to multiple sequences.

Consider two sequences $X = (x_1, x_2, \ldots, x_{n_x})$ and $Y = (y_1, y_2, \ldots, y_{n_y})$ of size n_x and n_y respectively.

- X = "Hello, world", where $x_1 = H, \ldots, x_{n_x} = d$
- Y = "hezlospkard", where $y_1 = H, \ldots, y_{n_y} = o$
- LCS(X, Y): elord

Real life application

Genomic research: find mutations in DNA (e.g. to discover mutations specific to sick patients)

Biologist encode strands of DNA with 4 characters "A", "T", "C" and "G" which represent the base molecules:

- Adenine
- Thymine
- Cytosine
- Guanine

- X = GACT
- Y = TTAT

Real life application

Genomic research: find mutations in DNA (e.g. to discover mutations specific to sick patients)

Biologist encode strands of DNA with 4 characters "A", "T", "C" and "G" which represent the base molecules:

- Adenine
- Thymine
- Cytosine
- Guanine

- $X = \text{GACT}$
- $Y = \text{TTAT}$

$\text{LCS}(X, Y) = \text{AT}$

LCS of long sequences?

What is the LCS of the following two sequences?

- $X =$ ACGGTGTCGTGCTATGCTGATGCTGACTTATATGCTA
- $Y =$ CGTTCGGCTATCGTACGTTCTATTCTATGATTTCTAA

LCS of long sequences?

What is the LCS of the following two sequences?

- $X = \text{ACGGTGTCGTGCTATGCTGATGCTGACTTATATGCTA}$
- $Y = \text{CGTTCGGCTATCGTACGTTCTATTCTATGATTTCTAA}$

Answer: $\text{LCS}(X, Y) = \text{CGTTCGGCTATGCTTCTACTTATTCTA}$

Even knowing the answer, it would take time to verify the solution without a proper algorithm!

A naive LCS algorithm

Could you propose a **simple algorithm** to find the longest subsequence of the two following sequences?

- $X = \text{CTGA}$
- $Y = \text{CGA}$

A naive LCS algorithm

Could you propose a **simple algorithm** to find the longest subsequence of the two following sequences?

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Naive algorithm

- **Step 1:** Enumerate all subsets for sequence X:
  - **Subsets for** X: {}, {C}, {G}, {T}, {A}, {C, T}, {C, G}, {T, G}, {C, A}, {T, A}, {C, T, G}, {C, T, A}, {G, A}, {C, G, A}, {T, G, A}, {C, T, G, A}

A naive LCS algorithm

Could you propose a **simple algorithm** to find the longest subsequence of the two following sequences?

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Naive algorithm

- **Step 1:** Enumerate all subsets for sequence X:
  - **Subsets for** X: {}, {C}, {G}, {T}, {A}, {C, T}, {C, G}, {T, G}, {C, A}, {T, A}, {C, T, G}, {C, T, A}, {G, A}, **{C, G, A}**, {T, G, A}, {C, T, G, A}
- **Step 2:** Find the larger subset of X that is also a subset of Y.

Naive algorithm complexity

How many subsets do we have for X?

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Naive algorithm complexity

How many subsets do we have for X? 2^4

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Complexity (worst case scenario): $\mathcal{O}(n_y * 2^{n_x})$, where n_y and n_y represent respectively the length of the sequences X and Y.

Naive algorithm complexity

How many subsets do we have for X? 2^4

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Complexity (worst case scenario): $\mathcal{O}(n_y * 2^{n_x})$, where n_y and n_y represent respectively the length of the sequences X and Y.

What would be the complexity of the naive algorithm for the following sequences?

- $X = \text{ACGGTGTCGTGCTATGCTGATGCTGACTTATATGCTA}$
- $Y = \text{CGTTCGGCTATCGTACGTTCTATTCTATGATTTCTAA}$

Naive algorithm complexity

How many subsets do we have for X? 2^4

- $X = \text{CTGA}$
- $Y = \text{CGA}$

Complexity (worst case scenario): $\mathcal{O}(n_y * 2^{n_x})$, where n_y and n_y represent respectively the length of the sequences X and Y.

What would be the complexity of the naive algorithm for the following sequences?

- $X = \text{ACGGTGTCGTGCTATGCTGATGCTGACTTATATGCTA}$
- $Y = \text{CGTTCGGCTATCGTACGTTCTATTCTATGATTTCTAA}$

Complexity (worst case scenario): $\mathcal{O}(37 * 2^{37}) \approx 5 * 10^{12}$

Could we propose a better solution to this problem?

Interlude: recursive algorithms

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Iterative algorithm

```
Factorial(n):
    fact = 1
    for i in [1, n] do
        fact = fact * i
    end for
    return fact
```

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Iterative algorithm

```
Factorial(n):
    fact = 1
    for i in [1, n] do
        fact = fact * i
    end for
    return fact
```

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	
Factorial(3)	
Factorial(2)	
Factorial(1)	
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	
Factorial(3)	
Factorial(2)	
Factorial(1)	
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	4 * factorial(3)
Factorial(3)	
Factorial(2)	
Factorial(1)	
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	4 * factorial(3)
Factorial(3)	3 * factorial(2)
Factorial(2)	
Factorial(1)	
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	4 * factorial(3)
Factorial(3)	3 * factorial(2)
Factorial(2)	2 * factorial(1)
Factorial(1)	
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	4 * factorial(3)
Factorial(3)	3 * factorial(2)
Factorial(2)	2 * factorial(1)
Factorial(1)	1 * factorial(0)
Factorial(0)	

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5 * factorial(4)
Factorial(4)	4 * factorial(3)
Factorial(3)	3 * factorial(2)
Factorial(2)	2 * factorial(1)
Factorial(1)	1 * factorial(0)
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	
Factorial(3)	
Factorial(2)	
Factorial(1)	
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	
Factorial(3)	
Factorial(2)	
Factorial(1)	1*1
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	
Factorial(3)	
Factorial(2)	2*1
Factorial(1)	1*1
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	
Factorial(3)	3*2
Factorial(2)	2*1
Factorial(1)	1*1
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	
Factorial(4)	4*6
Factorial(3)	3*2
Factorial(2)	2*1
Factorial(1)	1*1
Factorial(0)	1

Recursive algorithms: Computing a factorial (example 1)

In short, a recursive algorithm is an algorithm that calls itself on a smaller subproblem to solve the general problem.

Computing a factorial: n! = n(n-1)(n-2)...3.2.1

Recursive algorithm

```
Factorial(n):
    if i == 0 then
        return 1
    else
        return n * Factorial(n-1)
    end if
```

Execution stack for Factorial(5)

Function call	Returns
Factorial(5)	5*24
Factorial(4)	4*6
Factorial(3)	3*2
Factorial(2)	2*1
Factorial(1)	1*1
Factorial(0)	1

Recursive algorithms: Number of "A" in an array (example 2)

Compute the number of "A" in an array:

Iterative algorithm

```
NumberOfA(ar):
    nbA = 0
    for i in [1, n]  do
        if ar[i] == "A" then
            nbA = nbA + 1
        end if
    end for
    return nbA
```

Recursive algorithms: Number of ”A“ in an array (example 2)

Compute the number of ”A“ in an array:

Iterative algorithm

```
NumberOfA(ar):
    nbA = 0
    for i in [1, n]  do
        if ar[i] == ”A“ then
            nbA = nbA + 1
        end if
    end for
    return nbA
```

Recursive algorithm

```
NumberOfA(ar, i):
    if i == 0 then
        return 0
    else if ar[i] == ”A“ then
        return 1 + NumberOfA(ar, i-1)
    else
        return NumberofA(ar, i-1)
    end if
```

LCS with Dynamic Programming

Steps toward an O($n_x * n_y$) algorithm

In order reduce the complexity of the last algorithm, we divide the problem in two steps:

1. **Compute the length of the LCS.**
2. "Traceback" the process of step 1. to find the actual LCS.

Optimal Substructure of the LCS problem

Optimal Substructure: an optimal solution to a problem contains optimal solutions to its subproblems.

In other words: The LCS problem can be broken to smaller sub-problems until we have a trivial problem to solve.

Optimal Substructure of the LCS problem

Optimal Substructure: an optimal solution to a problem contains optimal solutions to its subproblems.

In other words: The LCS problem can be broken to smaller sub-problems until we have a trivial problem to solve.

Any ideas how to use that principle on the following sequences?

- $X = \text{GACT}$
- $Y = \text{TTAT}$

Optimal Substructure of the LCS problem

Sequences:

- $X = \text{GACT}$
- $Y = \text{TTAT}$

First case:

- Is the last element the same? If yes, we can remove it and consider the truncated problem.

Optimal Substructure of the LCS problem

Sequences:

- X = GACT
- Y = TTAT

First case:

- Is the last element the same? If yes, we can remove it and consider the truncated problem.
- In our example, as T is a common last element of X and Y we can "remove" it and consider the following LCS sub-problem:
  - LCS(GAC~~T~~, TTA~~T~~) = LCS(GAC, TTA) + T

Optimal Substructure of the LCS problem

Sequences:

- X = GACT
- Y = TTAT

First case:

- Is the last element the same? If yes, we can remove it and consider the truncated problem.
- In our example, as T is a common last element of X and Y we can "remove" it and consider the following LCS sub-problem:
  - LCS(GAC~~T~~, TTA~~T~~) = LCS(GAC, TTA) + T
- Formally:
  $\text{LCS}(X[1, n_x],\ Y[1, n_y]) = \text{LCS}(X[1, n_x - 1],\ Y[1, n_y - 1]) + X[n_x]$,
  where
  - $X[1, n_x - 1] = \text{GAC}$, and $Y[1, n_y - 1] = \text{TTA}$
  - $X[n_x] = Y[n_y] = T$

  Where $S[a, b]$ means we consider the sequence from character a up to character b in sequence S. E.g. $S = Hello$, $S[2, 4] = ell$.

Optimal Substructure of the LCS problem

Evaluating LCS(GAC, TTA) $=$ LCS($X[1, n_x - 1]$, $Y[1, n_y - 1]$)

Sequences considered:

- $X[1, n_x - 1] = \mathrm{GAC}$
- $Y[1, n_y - 1] = \mathrm{TTA}$

Optimal Substructure of the LCS problem

Evaluating $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(X[1, n_x - 1],\ Y[1, n_y - 1])$

Sequences considered:

- $X[1, n_x - 1] = \text{GAC}$
- $Y[1, n_y - 1] = \text{TTA}$

Second case: if the last element of the sequences is different, we consider two cases:

1. Either the LCS finishes with a C, then:
   $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(\text{GAC, TT}) = \text{LCS}(X[1, n_x - 1], Y[1, n_y - 2])$
2. Or the LCS finishes with an A, then:
   $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(\text{GA, TTA}) = \text{LCS}(X[1, n_x - 2], Y[1, n_y - 1])$

Optimal Substructure of the LCS problem

Evaluating $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(X[1, n_x - 1],\ Y[1, n_y - 1])$

Sequences considered:

- $X[1, n_x - 1] = \text{GAC}$
- $Y[1, n_y - 1] = \text{TTA}$

Second case: if the last element of the sequences is different, we consider two cases:

1. Either the LCS finishes with a C, then:
   $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(\text{GAC, TT}) = \text{LCS}(X[1, n_x - 1], Y[1, n_y - 2])$
2. Or the LCS finishes with an A, then:
   $\text{LCS}(\text{GAC, TTA}) = \text{LCS}(\text{GA, TTA}) = \text{LCS}(X[1, n_x - 2], Y[1, n_y - 1])$

Thus LCS(GAC, TTA) = max(LCS(GAC, TT), LCS(GA, TTA))

Optimal Substructure of the LCS problem

Evaluating $\text{LCS(GAC, TT)} = \text{LCS}(X[1, n_x - 1], Y[1, n_y - 2])$

Sequences considered:

- $X[1, n_x - 1] = \text{GAC}$
- $Y[1, n_y - 2] = \text{TT}$

Second case: if the last element of the sequences is different, we consider two cases:

1. Either the LCS finishes with a C, then:
   $\text{LCS(GAC, TT)} = \text{LCS(GAC, T)} = \text{LCS}(X[1, n_x - 1], Y[1, n_y - 3])$
2. Or the LCS finishes with a T, then:
   $\text{LCS(GAC, TT)} = \text{LCS(GA, TT)} = \text{LCS}(X[1, n_x - 2], Y[1, n_y - 2])$

Optimal Substructure of the LCS problem

Evaluating LCS(GAC, TT) $=$ LCS($X[1, n_x - 1]$, $Y[1, n_y - 2]$)

Sequences considered:

- $X[1, n_x - 1] = \text{GAC}$
- $Y[1, n_y - 2] = \text{TT}$

Second case: if the last element of the sequences is different, we consider two cases:

1. Either the LCS finishes with a C, then:
   LCS(GAC, TT) $=$ LCS(GAC, T) $=$ LCS($X[1, n_x - 1], Y[1, n_y - 3]$)
2. Or the LCS finishes with a T, then:
   LCS(GAC, TT) $=$ LCS(GA, TT) $=$ LCS($X[1, n_x - 2], Y[1, n_y - 2]$)

Thus LCS(GAC, TT) = max(LCS(GAC, T), LCS(GA, TT))

Optimal Substructure of the LCS problem

Evaluating LCS(GA, TTA) $= \mathrm{LCS}(X[1, n_x - 2],\ Y[1, n_y - 1])$

Sequences considered:

- $X[1, n_x - 2] = \mathrm{GA}$
- $Y[1, n_y - 1] = \mathrm{TTA}$

Optimal Substructure of the LCS problem

Evaluating LCS(GA, TTA) $=$ LCS($X[1, n_x - 2]$, $Y[1, n_y - 1]$)

Sequences considered:

- $X[1, n_x - 2] =$ GA
- $Y[1, n_y - 1] =$ TTA

The two **last elements are identical**, thus, from 1st case, we can remove the last A:

- LCS(GA, TTA) $=$ A $+$ LCS(G, TT)

LCS optimal substructure: summary of properties

We can rewrite the LCS properties as

$$LCS(X[1,i], Y[1,j]) = \begin{cases} 0 & \text{if } i == 0 \text{ or} j == 0 \\ & \\ & \end{cases}$$

LCS optimal substructure: summary of properties

We can rewrite the LCS properties as

$$LCS(X[1,i],Y[1,j]) = \begin{cases} 0 & \text{if } i == 0 \text{ or} j == 0 \\ 1 + LCS(X[1,i-1],Y[1,j-1]) & \text{if } X[i] == Y[j] \\ & \end{cases}$$

LCS optimal substructure: summary of properties

We can rewrite the LCS properties as

$$LCS(X[1,i],Y[1,j]) = \begin{cases} 0 & \text{if } i == 0 \text{ or} j == 0 \\ 1 + LCS(X[1,i-1],Y[1,j-1]) & \text{if } X[i] == Y[j] \\ max(LCS(X[1,i-1],Y[1,j]), & \\ \qquad LCS(X[1,i],Y[1,j-1])) & \text{if } X[i] \text{ differs from } Y[j] \end{cases}$$

A recursive algorithm to compute the length of an LCS

```
LCS-length(X, i, Y, j):
    if i = 0 OR j = 0 then
        return 0
    else if X[i] = Y[j] then
        result = 1 + LCS(X, i − 1, Y, j − 1)
    else
        result = max(LCS(X, i − 1, Y, j), LCS(X, i, Y, j − 1))
    end if
    return result
```

A recursive algorithm to compute the length of an LCS

LCS-length(X, i, Y, j):
    if $i = 0$ OR $j = 0$ **then**
        return 0
    else if $X[i] = Y[j]$ **then**
        result $= 1 +$ LCS$(X, i-1, Y, j-1)$
    else
        result $=$ max(LCS$(X, i-1, Y, j)$, LCS$(X, i, Y, j-1)$)
    end if
    return result

A recursive algorithm to compute the length of an LCS

LCS-length(X, i, Y, j):
    if $i = 0$ OR $j = 0$ **then**
        return 0
    else if $X[i] = Y[j]$ **then**
        result $= 1 +$ LCS$(X, i-1, Y, j-1)$
    else
        result $=$ max(LCS$(X, i-1, Y, j)$, LCS$(X, i, Y, j-1)$)
    end if
    return result

A recursive algorithm to compute the length of an LCS

```
LCS-length(X, i, Y, j):
    if i = 0 OR j = 0 then
        return 0
    else if X[i] = Y[j] then
        result = 1 + LCS(X, i − 1, Y, j − 1)
    else
        result = max(LCS(X, i − 1, Y, j), LCS(X, i, Y, j − 1))
    end if
    return result
```

Overlapping sub-problems

LCS(GACT, TTAT)

Overlapping sub-problems

LCS(GACT, TTAT)

LCS(GAC, TTA)

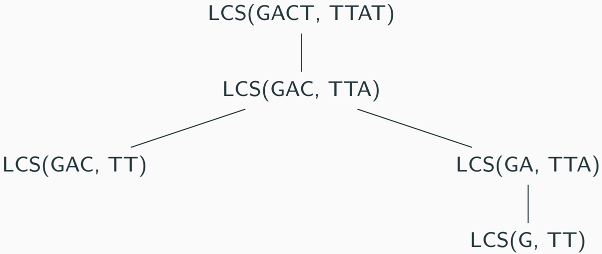
Overlapping sub-problems

LCS(GACT, TTAT)

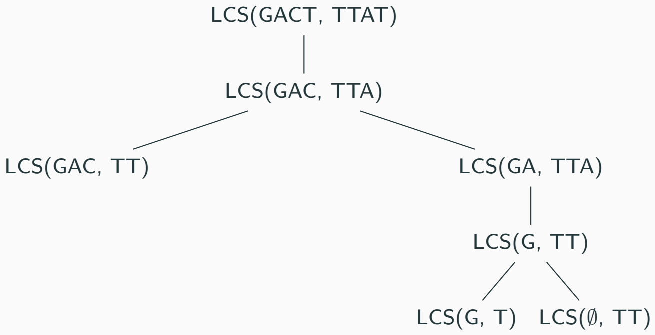
LCS(GAC, TTA)

LCS(GAC, TT) LCS(GA, TTA)

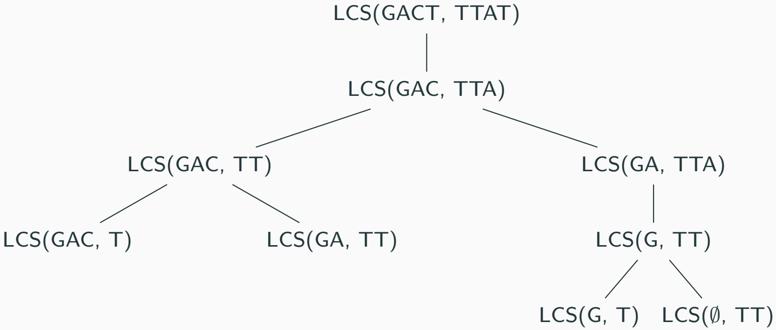
Overlapping sub-problems



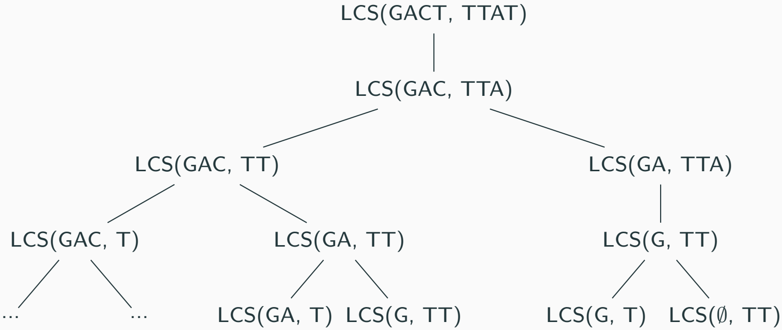
Overlapping sub-problems



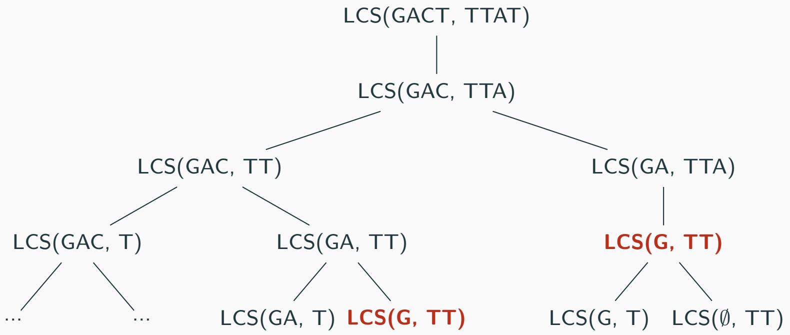
Overlapping sub-problems



Overlapping sub-problems



Overlapping sub-problems



We solve two times the same sub-problem LCS(G, TT)!

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$				
2	A	$\emptyset$				
3	C	$\emptyset$				
4	T	$\emptyset$				

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	A	$\emptyset$				
3	C	$\emptyset$				
4	T	$\emptyset$				

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then
   $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then
   $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	A	$\emptyset$	$\emptyset$	$\emptyset$	1	1
3	C	$\emptyset$				
4	T	$\emptyset$				

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	A	$\emptyset$	$\emptyset$	$\emptyset$	1	1
3	C	$\emptyset$	$\emptyset$	$\emptyset$	1	1
4	T	$\emptyset$				

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	A	$\emptyset$	$\emptyset$	$\emptyset$	1	1
3	C	$\emptyset$	$\emptyset$	$\emptyset$	1	1
4	T	$\emptyset$	1	1	1	2

Computing the length of the LCS: filling the length table

We build an array L, row by row, that iteratively compute the LCS length of sequences X and Y using two rules:

1. If $X[i] == Y[j]$, then $L[i,j] = L[i-1, j-1] + 1$
2. If $X[i]$ differs from $Y[j]$, then $L[i,j] = max(L(i, j-1), L[i-1, j])$

$L[i,j]$		$\emptyset$	1	2	3	4
		$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	A	$\emptyset$	$\emptyset$	$\emptyset$	1	1
3	C	$\emptyset$	$\emptyset$	$\emptyset$	1	1
4	T	$\emptyset$	1	1	1	2

Constructing the L array only cost $\mathcal{O}(n_x \times n_y)$

Computing the length of the LCS: algorithm

```
COMPUTE-LCS-LENGTH(X, Y):
    for i in [0, n_x] do
        L[i, 0] = 0
    end for
    for j in [0, n_y] do
        L[0, j] = 0
    end for
    for i in [1, n_x] do
        for j in [1, n_y] do
            if X[i] == Y[j] then
                L[i, j] = L[i − 1, j − 1] + 1
            else
                L[i, j] = max(L[i, j]L[i, j])
            end if
        end for
    end for
    return L
```

Computing the length of the LCS: algorithm

COMPUTE-LCS-LENGTH(X, Y):
    for i in $[0, n_x]$ **do**
        $L[i, 0] = 0$
    end for
    for j in $[0, n_y]$ **do**
        $L[0, j] = 0$
    end for
    for i in $[1, n_x]$ **do**
        for j in $[1, n_y]$ **do**
            if $X[i] == Y[j]$ **then**
                $L[i, j] = L[i-1, j-1] + 1$
            else
                $L[i, j] = \max(L[i, j] L[i, j])$
            end if
        end for
    end for
    return L

Computing the length of the LCS: algorithm

```
COMPUTE-LCS-LENGTH(X, Y):
    for i in [0, n_x] do
        L[i, 0] = 0
    end for
    for j in [0, n_y] do
        L[0, j] = 0
    end for
    for i in [1, n_x] do
        for j in [1, n_y] do
            if X[i] == Y[j] then
                L[i, j] = L[i − 1, j − 1] + 1
            else
                L[i, j] = max(L[i, j]L[i, j])
            end if
        end for
    end for
    return L
```

Steps toward an O($n_x * n_y$) algorithm

In order reduce the complexity of the last algorithm, we divide the problem in two steps:

1. Compute the length of the LCS.
2. **"Traceback" the process of step 1. to find the actual LCS.**

Reconstructing the LCS

```
BUILD-LCS(X, Y, i, j, L):
    if L[i,j] == ∅ then
        return ∅
    else if X[i] == Y[j] then
        return BUILD-LCS(X, Y, i − 1, j − 1, L) + X[i]
    else if L[i, j − 1] > L[i − 1, j] then
        return BUILD-LCS(X, Y, i, j − 1, L)
    else
        return BUILD-LCS(X, Y, i − 1, j, L)
    end if
```

$L[i,j]$	$\emptyset$	T	T	A	T
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
A	$\emptyset$	$\emptyset$	$\emptyset$	**1**	1
C	$\emptyset$	$\emptyset$	$\emptyset$	**1**	1
T	$\emptyset$	1	1	1	**2**

Summary about dynamic programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps (CLRS):

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.