Dynamic Programming II: Edit Distance & LIS

Admin

- **Midterm** Friday it goes out
    - Will be released 4pm Friday; can be taken in any 24 hour period starting **4pm Friday and ending 10pm Wednesday**
    - **No class on Monday**
        - Normal office hour schedule, plus I'll be available during what would have been class.
    - Midterm will be like a "homework's greatest hits"
      Questions should be:
        - Short and sweet
        - Straightforward (which is different from easy!)

Today’s Outline

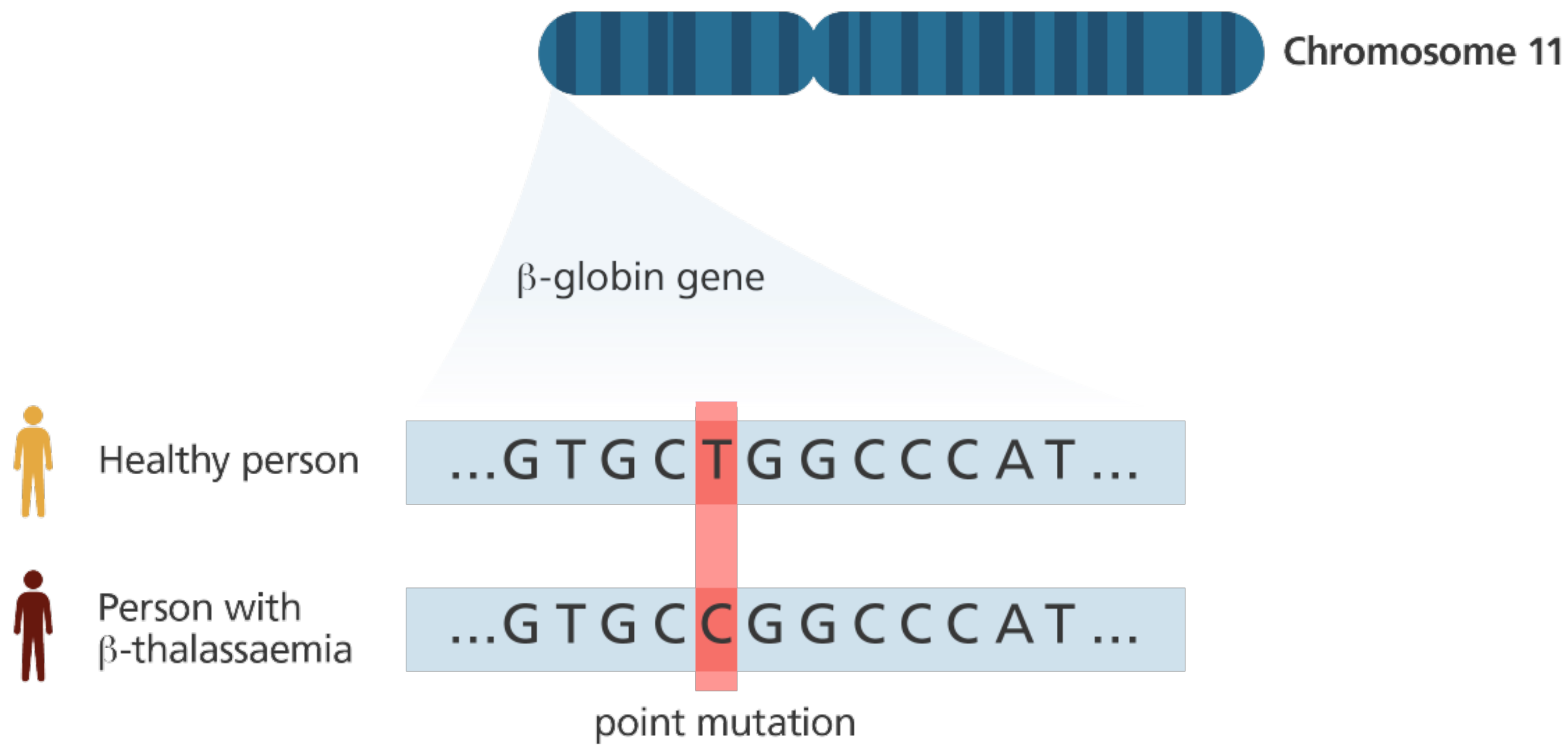
- **Edit distance**
    - Classic problem with many applications
    - Requires a 2D memoization structure
- **Longest Increasing Subsequence**
    - More DP practice

Edit Distance

Further Reading: Chapter 3.7, Erickson

Motivation

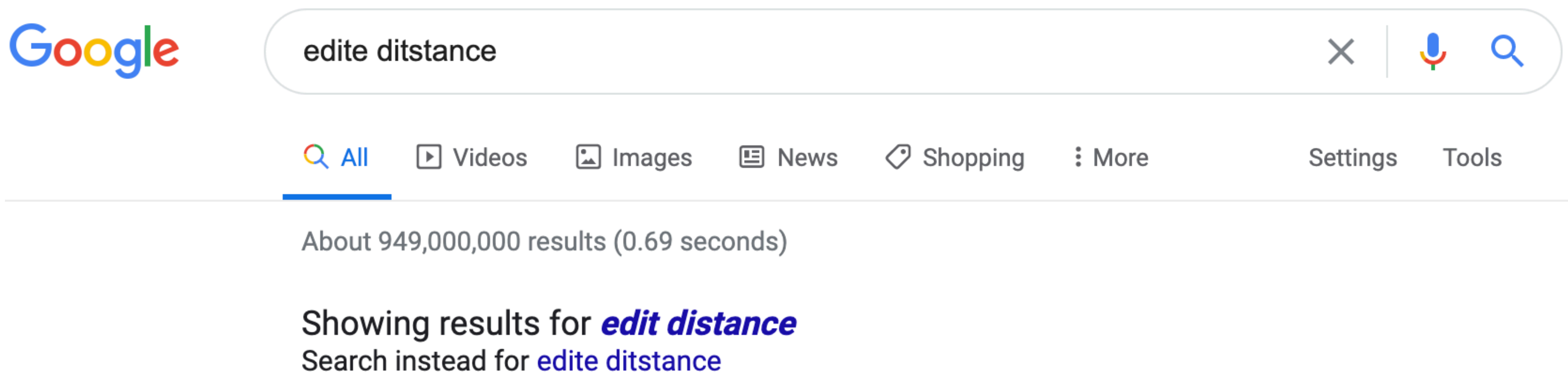
- **Edit distance**: is a **metric** that captures the similarity between two strings



DNA sequencing: finding similarities between two genome sequences

Motivation

- **Edit distance**: is a **metric** that captures the similarity between two strings



Text processing: finding similar strings and NLP

Problem Defintion

Problem. Given two strings $A = a_1 \cdot a_2 \cdots a_n$ and $B = b_1 \cdot b_2 \cdots b_m$, find the **edit distance** between them.

- Edit distance between A and B is the smallest number of the following operations that are needed to transform A into B
  - Replace a character (substitution)
  - Delete a character
  - Insert a character

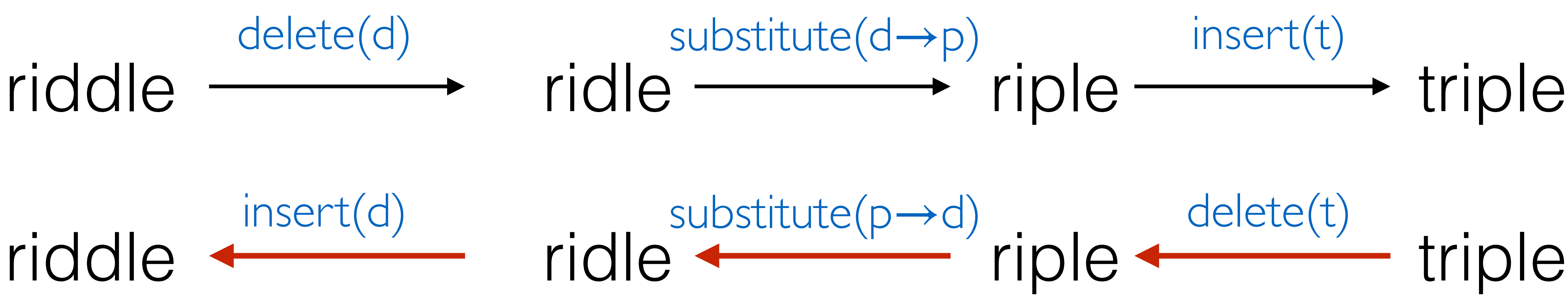
riddle $\xrightarrow{\text{delete(d)}}$ ridle $\xrightarrow{\text{substitute(d}\rightarrow\text{p)}}$ riple $\xrightarrow{\text{insert(t)}}$ triple

Edit distance(riddle, triple): 3

Structure of the Problem

Problem. Given two strings $A = a_1 \cdot a_2 \cdots a_n$ and $B = b_1 \cdot b_2 \cdots b_m$, find the **edit distance** between them.

- Notice that the process of getting from string A to string B by doing substitutions, inserts and deletes is **reversible**
- **Inserts** in one string correspond to **deletes** in another

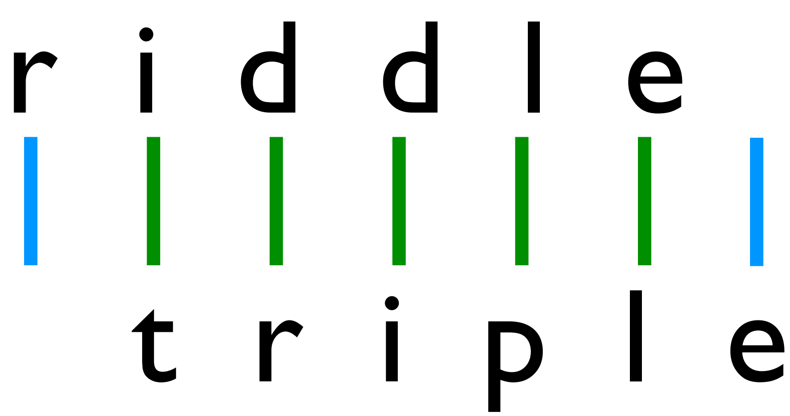


Edit distance(riddle, triple): 3

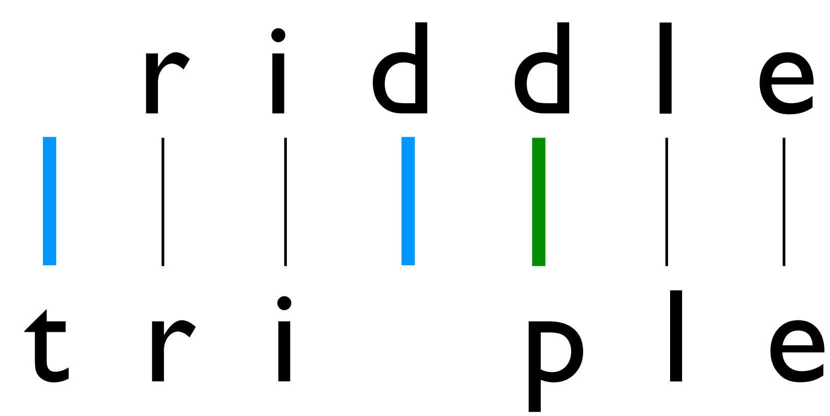
Sequence Alignment

We can visualize the problem of finding the edit distance as an the problem of finding the best **alignment** between two strings

- **Gaps** in alignment represent inserts to top/deletes to bottom
- **Mismatches** in alignment represent substitutes
  - Cost of an alignment = number of **gaps + mismatches**
- **Edit distance**: minimum cost alignment



cost = 7



cost = 3

Sequence Alignment

```
prin-ciple
|||| |||xx
prinncipal
(1 gap, 2 mm)
```

```
prin-cip-le
|||| ||| |
prinncipal-
(3 gaps, 0 mm)
```

```
misspell
||| ||||
mis-pell
(1 gap)
```

```
prehistoric
   ||||||||
---historic
(3 gaps)
```

```
aa-bb-ccaabb
|x || | | |
ababbbc-a-b-
(5 gaps, 1 mm)
```

```
al-go-rithm-
|| xx ||x |
alKhwariz-mi
(4 gaps, 3 mm)
```

https://www.cs.cmu.edu/~ckingsf/class/02713-s13

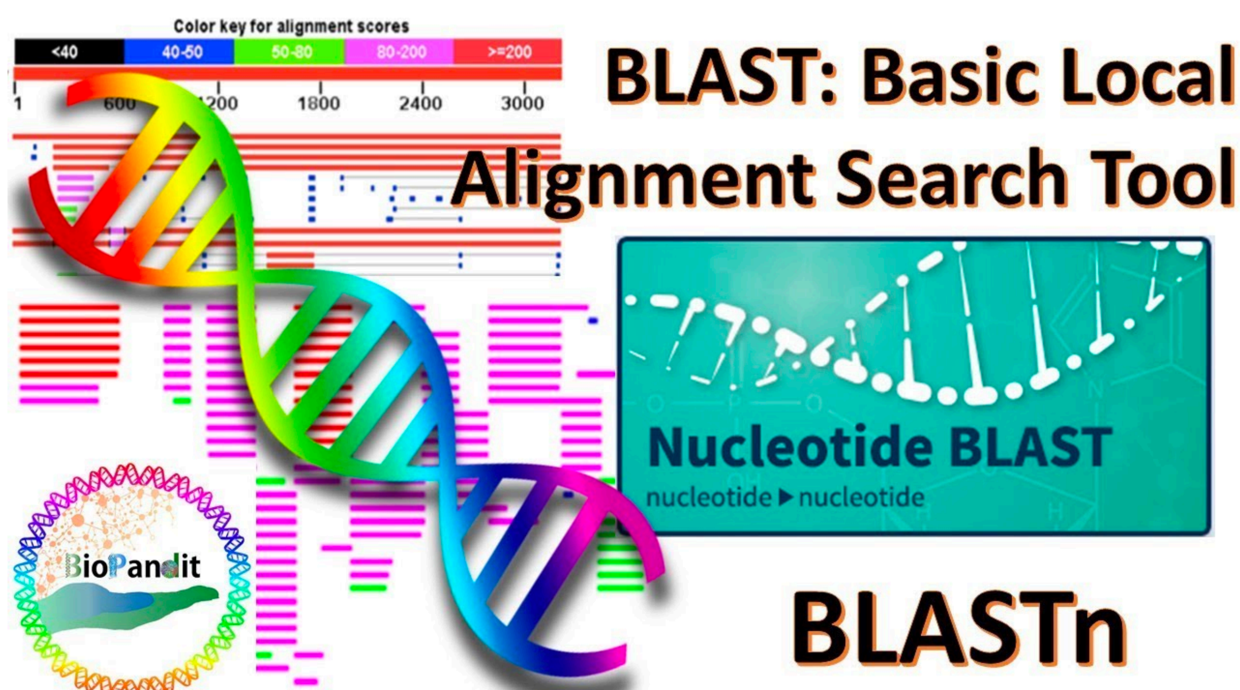
Sequence Alignment

```
>gb|AC115706.7| Mus musculus chromosome 8, clone RP23-382B3, complete sequence

Query  1650   gtgtgtgtgggtgcacatttgtgtgtgtgcgcctgtgtgtgtgggtgcctgtgtgtgt  1709
              ||||||||||  |    ||  | ||||||||| | ||||||||    ||| ||  |||||
Sbjct  56838  GTGTGTGTGGAAGTGAGTTCATCTGTGTGTGCACATGTGTGTGCA--TGCATGCATGTGT  56895

Query  1710   gtg-gggcacatttgtgtgtgtgtgtgcctgtgtgtgggtgcacatttgtgtgtgtgc  1768
              ||  |||||      ||  ||| ||||||| |||||||| |||  |||  ||||| || |
Sbjct  56896  GTCCGGGCA------TGCATGTCTGTGTGCATGTGTGTGTGTGCAT--GTGTGAGTAC  56947

Query  1769   ctgtgtgtgtgtgcctgtgtgtgggggtgcacatttgtgtgtgtgtgtgcctgtgtgtgg  1828
              |||||||||| ||| ||| |||| | |||  ||| |||||  ||||||   |||||  |
Sbjct  56948  CTGTGTGTGTATGCTTGTATGTGTGTGTGCATGTGTGTAGGTGTGTATATGTGTAAGT  57007
```



Sequence Alignment Problem

Problem: Find an alignment of the two strings A, B where

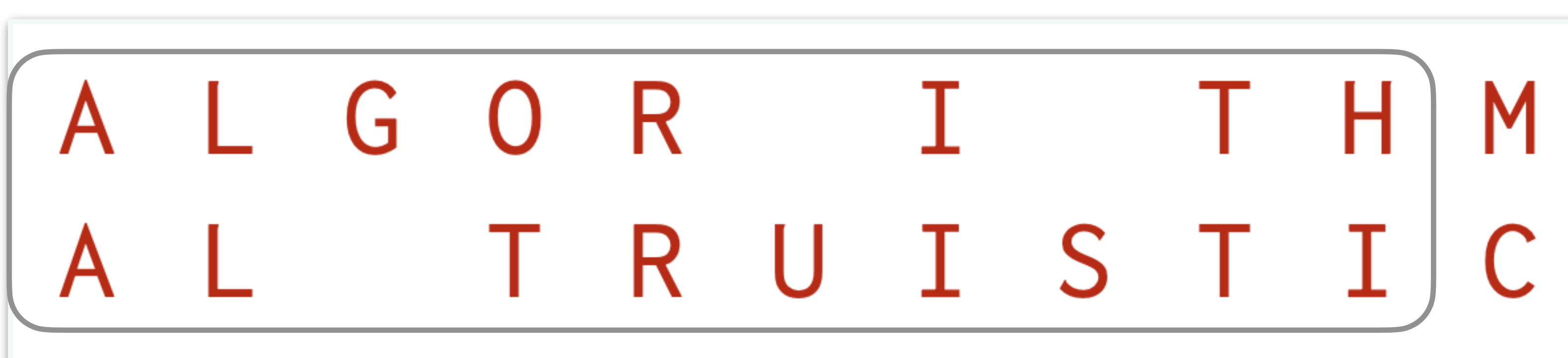
- each character a_i in A is matched to a string b_j in B or unmatched
- each character b_j in B is matched to a string a_i in A or unmatched

- cost$(a_i, b_j) = 0$ if $a_i = b_j$, else cost$(a_i, b_j) = 1$
- cost of an unmatched letter (gap) $= 1$
- Total cost = # unmatched (gaps) $+ \sum_{a_i, b_j} \text{cost}(a_i, b_j)$
- **Goal**. Compute **edit distance** by finding an **alignment** of the minimum total cost

Recursive Structure

Before we develop a dynamic program, we need to figure out the recursive structure of the problem

- Our alignment representation has an optimal substructure:
  - Suppose we have the mismatch/gap representation of the shortest edit sequence of two strings
  - If we remove the last column, the remaining columns must represent the shortest edit sequence of the remaining prefixes!



Subproblem

- **Subproblem**

Edit(i, j): edit distance between the strings $a_1 \cdot a_2 \cdots a_i$ and $b_1 \cdot b_2 \cdots b_j$, where $0 \leq i \leq n$ and $0 \leq j \leq m$

- **Final answer**

Edit(n, m)

Base Cases

We have to fill out a **two-dimensional array** to memoize our recursive dynamic program.

- Which rows/columns can we fill immediately?
- Edit$(i,0)$: Min number of edits to transform a string of length i to an empty string

$$\text{Edit}(i, 0) = i \text{ for } 0 \le i \le n$$

$$\text{Edit}(0, j) = j \text{ for } 0 \le j \le m$$

Recurrence

Imagine the optimal alignment between the two strings

- What are the possibilities for the last column?
  - It could be that both letters match: cost 0
  - It could be that both letters do not match: cost 1
  - It could be that there an unmatched character (gap): either from A or from B: cost 1

Recurrence

Break the possibilities down for the last column in the optimal alignment of $a_1 \cdot a_2 \cdots a_i$ and $b_1 \cdot b_2 \cdots b_j$:

- **Case 1.** Only one row has a character:
  - Case 1a. Letter a_i is unmatched
    $\text{Edit}(i, j) = \text{Edit}(i-1, j) + 1$

  - Case 1b. Letter b_j is unmatched
    $\text{Edit}(i, j) = \text{Edit}(i, j-1) + 1$

- **Case 2**: Both rows have characters:
  - Case 2a. Same characters:
    $\text{Edit}(i, j) = \text{Edit}(i-1, j-1)$

  - Case 2b. Different characters:
    $\text{Edit}(i, j) = \text{Edit}(i-1, j-1) + 1$

Final Recurrence

For $1 \leq i \leq n$ and $1 \leq j \leq m$, we have:

$$\text{Edit}(i, j) = \min \begin{cases} \text{Edit}(i, j-1) + 1 \\ \text{Edit}(i-1, j) + 1 \\ \text{Edit}(i-1, j-1) + (a_i \neq b_j) \end{cases}$$

Uses the shorthand: $(a_i \neq b_j)$ which is 1 if it is true (i.e., they mismatch), and zero otherwise

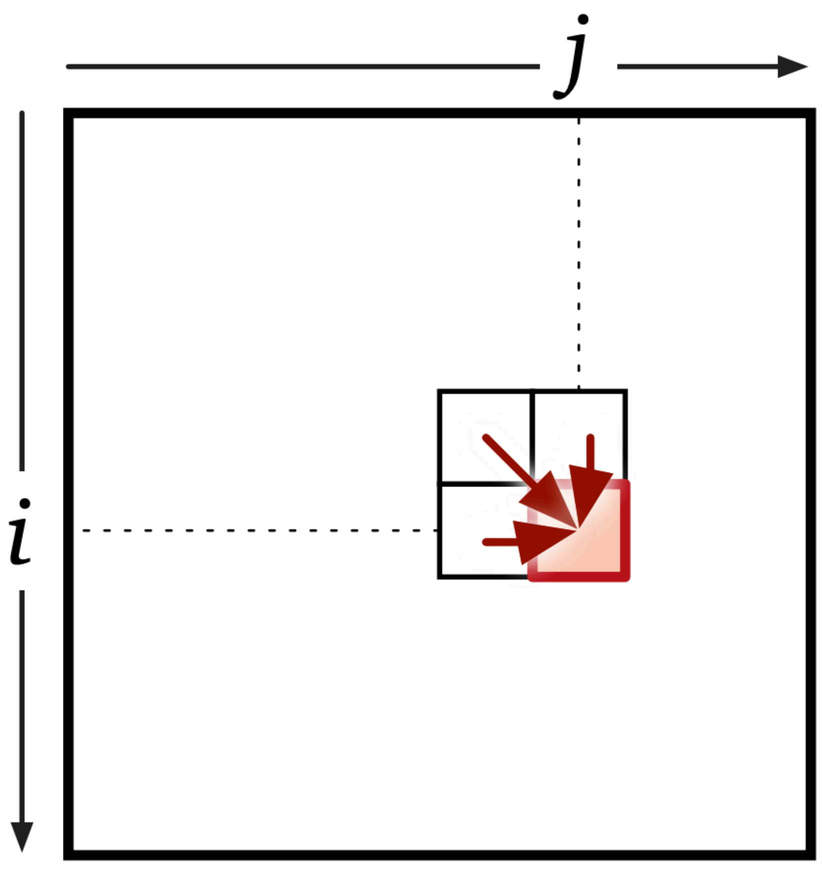
From Recurrence to DP

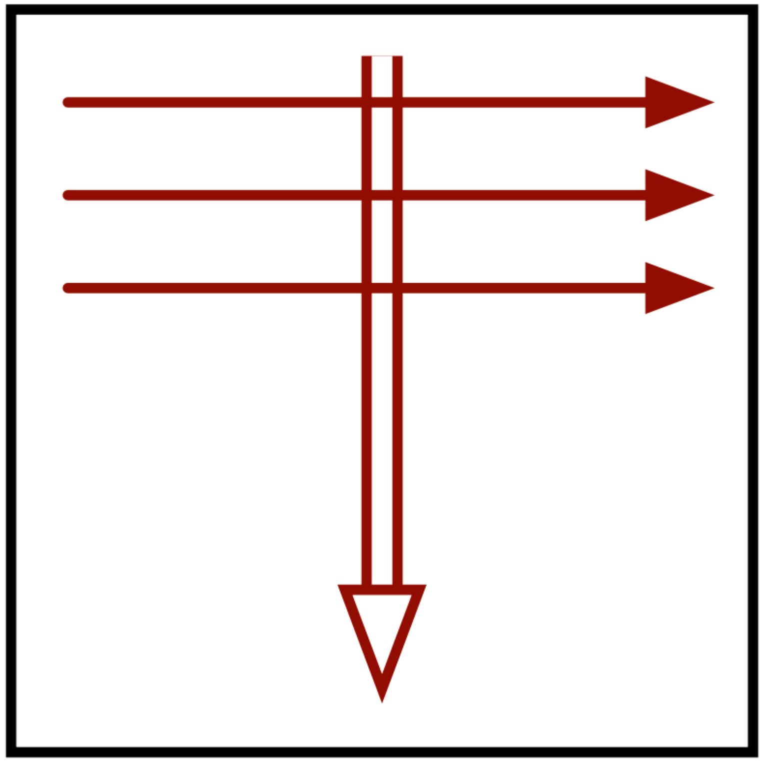
- We can now transform our recurrence into a dynamic program
- **Memoization Structure**: We can memoize all possible values of Edit(i, j) in a table/ two-dimensional array of size $O(nm)$:
    - Store Edit$[i, j]$ in a 2D array; $0 \leq i \leq n$ and $0 \leq j \leq m$
- **Evaluation order**:
    - Is interesting for a 2D problem
    - Based on dependencies between subproblems
    - We want referenced values to be already computed

From Recurrence to DP

Evaluation order

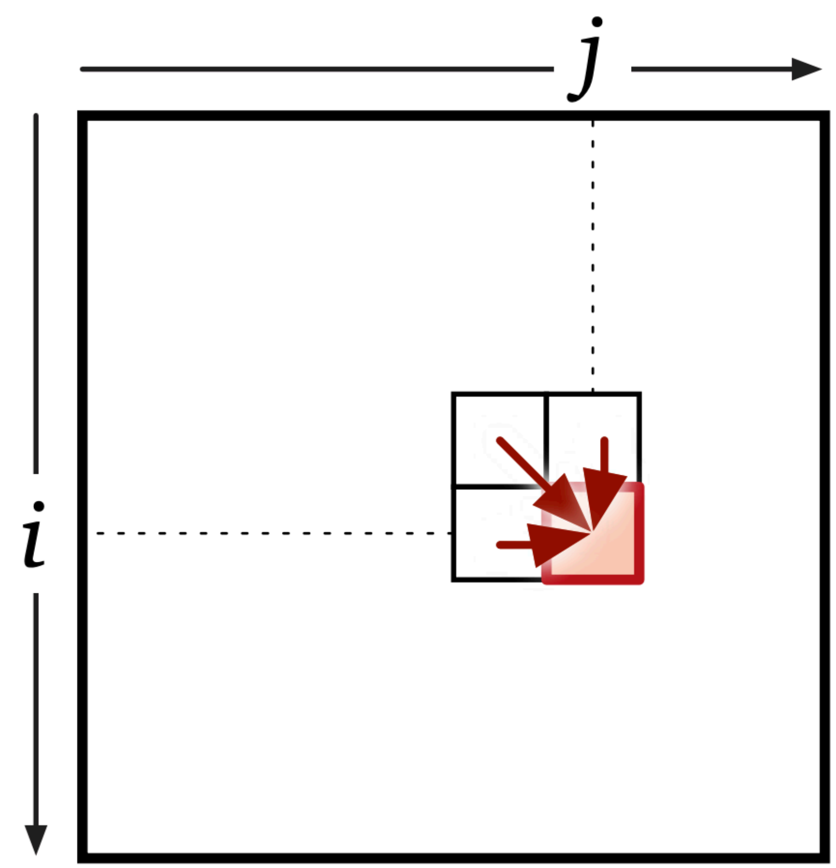
- We can fill in **row major order**, which is row by row from top down, each row from left to right
    - With this order, when we reach an entry in the table, our recurrence references only filled-in entries

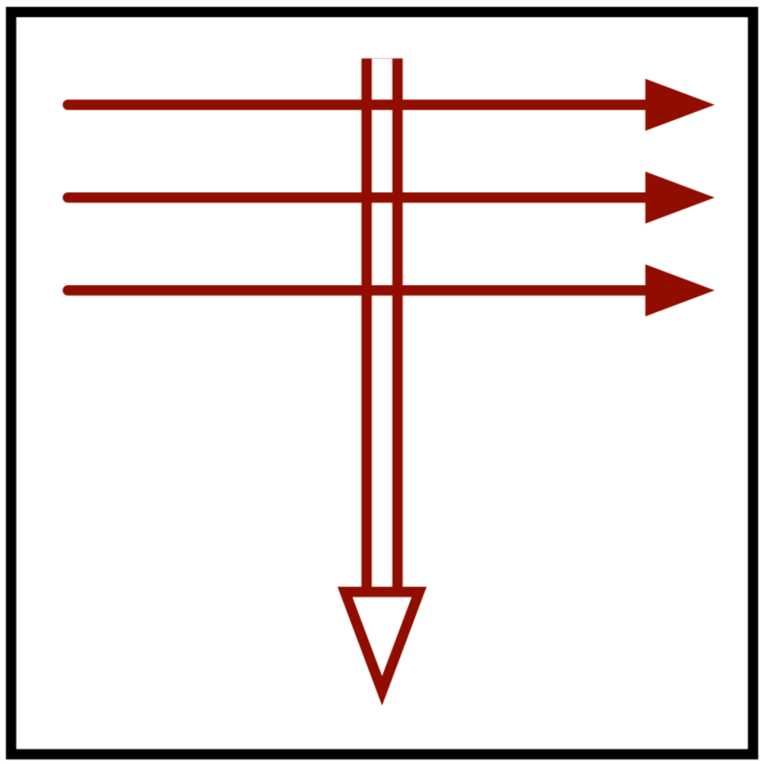




Space and Time

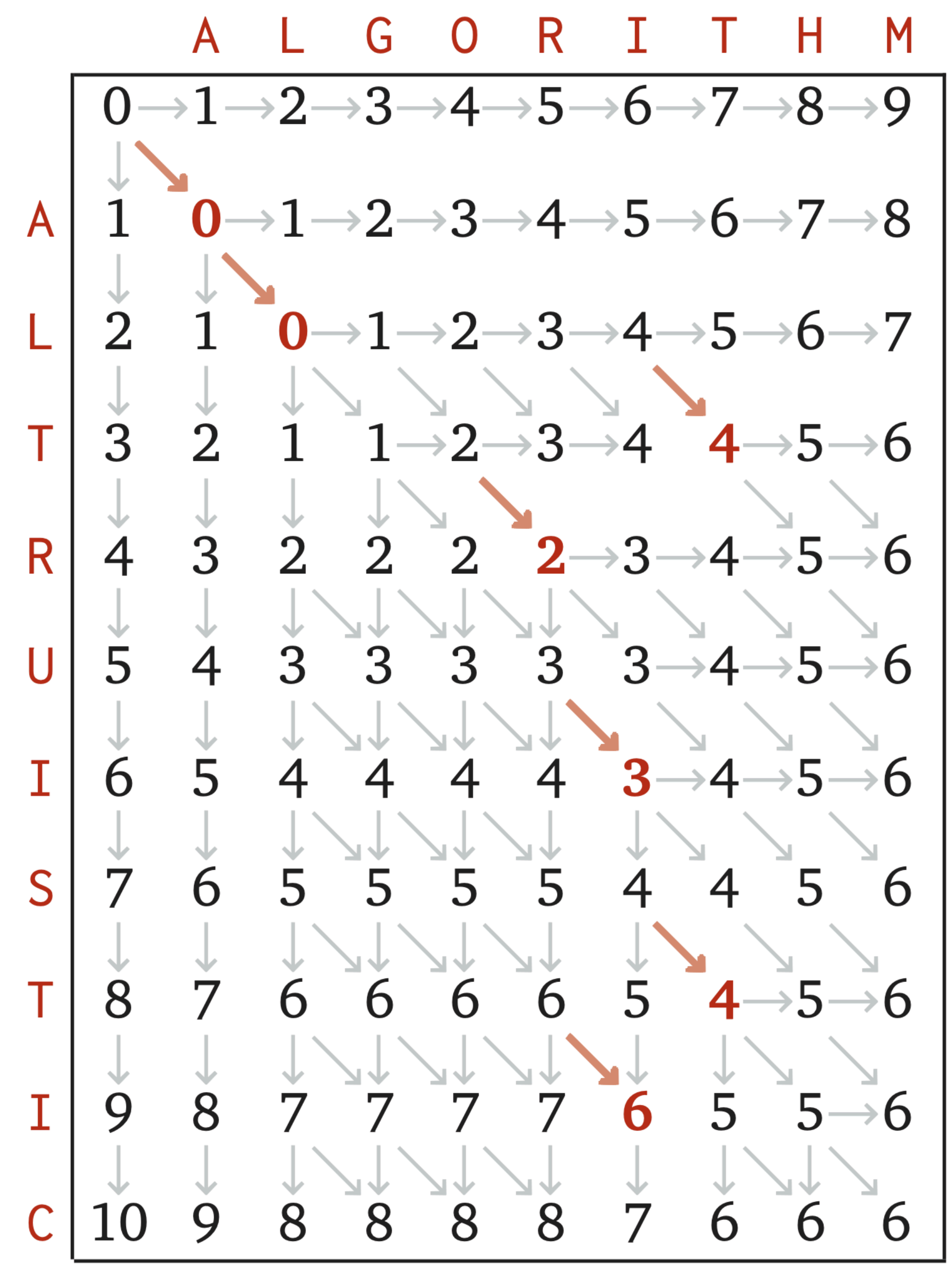
- The memoization uses $O(nm)$ space
- We can compute each Edit$[i, j]$in $O(1)$ time
- Overall running time: $O(nm)$





Memoization Table: Example

- Memoization table for **ALGORITHM** and **ALTRUISTIC**
- Bold numbers indicate where characters are same
- Horizontal arrow: deletion in A
- Vertical arrow: insertion in A
- Diagonal: substitution
- Bold red: free substitution
- Only draw an arrow if used in DP
- Any directed path of arrows from top left to bottom right represents an optimal edit distance sequence



Reconstructing the Edits

- We don't need to store the arrow!
- An arrow can be reconstructed on the fly in $O(1)$ time using the numerical values
- Once the table is built, we can construct the shortest edit distance sequence in $O(n+m)$ time
- Does this remind you of any other dynamic programs we've seen?

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Longest Increasing Subsequence

Further Reading: Chapter 3.7, Erickson

Longest Increasing Subsequence

Problem: Given a sequence of integers as an array $A[1,\ldots n]$, find the longest subsequence whose elements are in increasing order

An increasing subsequence with length 4

1	**2**	**10**	3	7	6	4	8	**11**

Longest Increasing Subsequence: length 6

1	**2**	10	**3**	7	6	**4**	**8**	**11**

(Stated more formally…) Find the longest possible sequence of indices $1 \leq i_1 < i_2 < \ldots < i_\ell \leq n$ such that $A[i_k] < A[i_{k+1}]$

To simplify, we will only compute length of the LIS

Formalize the Subproblem

$L[i]$: length of the longest increasing subsequence in $A[1,\ldots,i]$ that ends at (and includes) $A[i]$

Identify the Base Case

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = ?$

Identify the Final Answer

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = 1$

Final answer. $?$

Base Case & Final Answer

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

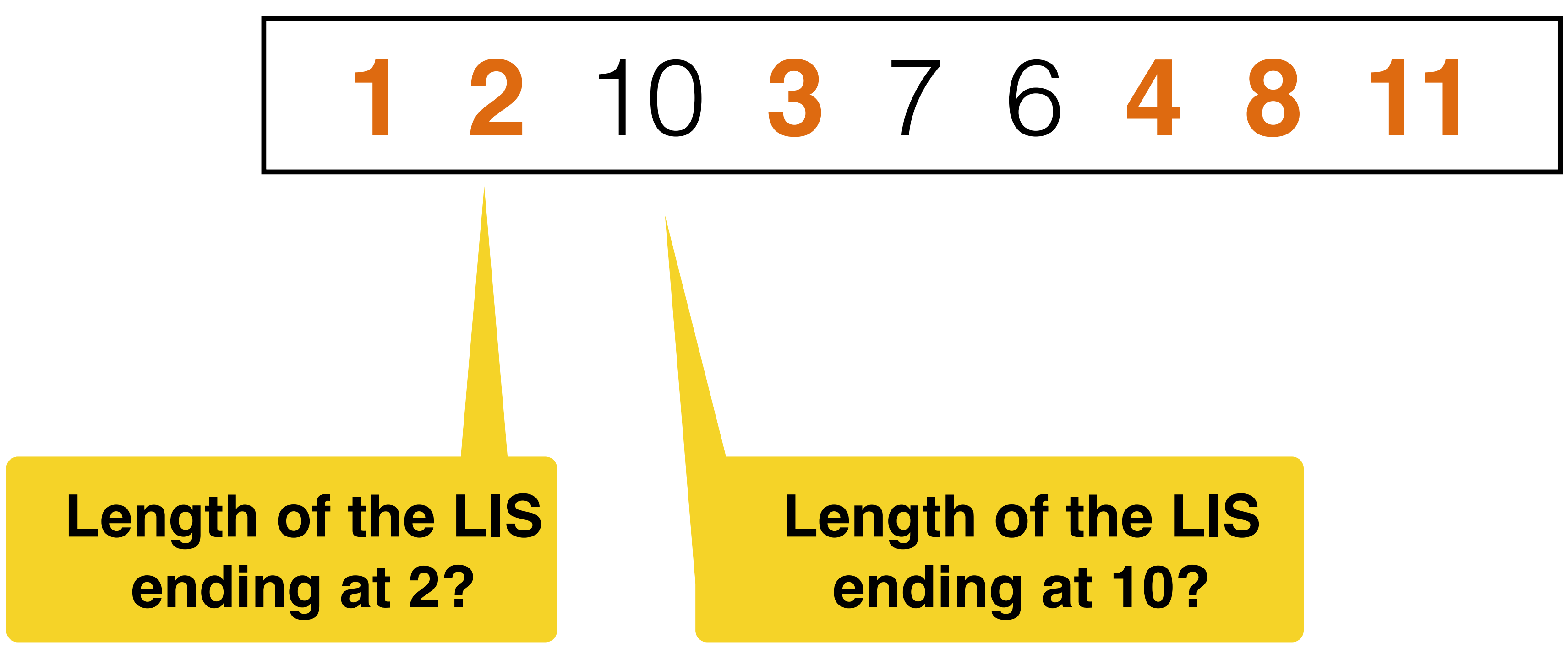
Base Case. $L[1] = 1$

Final answer. $\max_{1 \leq i \leq n} L[i]$

Recurrence

How do we go from one subproblem to the next?

- That is, how do we compute $L[i]$ assuming I know the values of $L[1],\ \ldots,\ L[i-1]$



Recurrence

- Let's say we know the length of the longest subsequence ending at $A[1], A[2], \ldots A[i-1]$
- What is the longest subsequence ending at $A[i]$? Either:
- $A[i]$ could potentially extend an earlier subsequence:
    - Can extend a longest subsequence ending at some $A[k]$, with $A[k] < A[i]$, but which k?
        - We could try all k to get the answer
- Or $A[i]$ could start a new subsequence (i.e., it doesn't extend any earlier increasing subsequence)

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A	1	2	10	3	7	6	4	8	11

L	1							

Example: Building a Recurrence

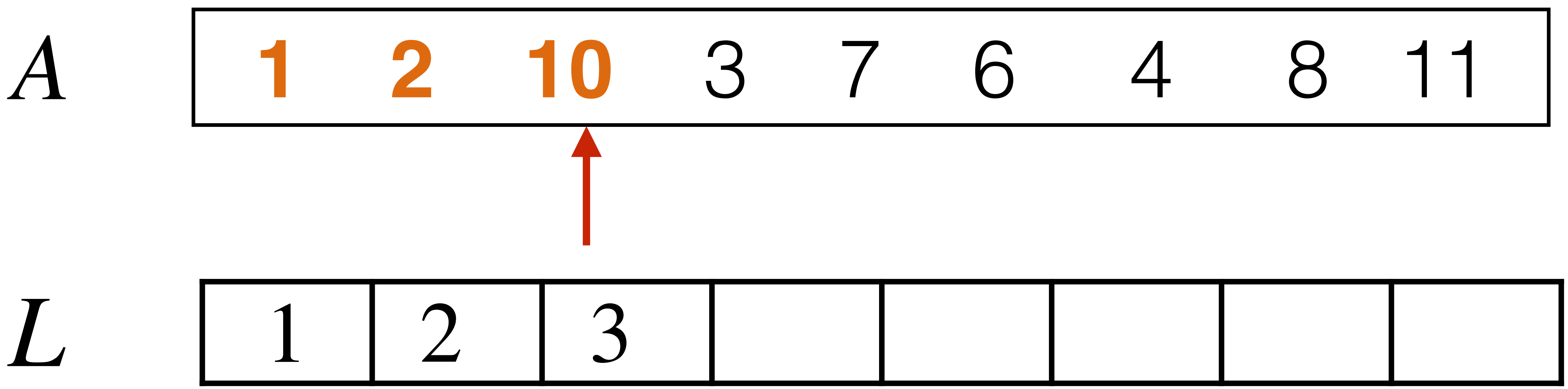
$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A	**1**	**2**	10	3	7	6	4	8	11

L	1	2						

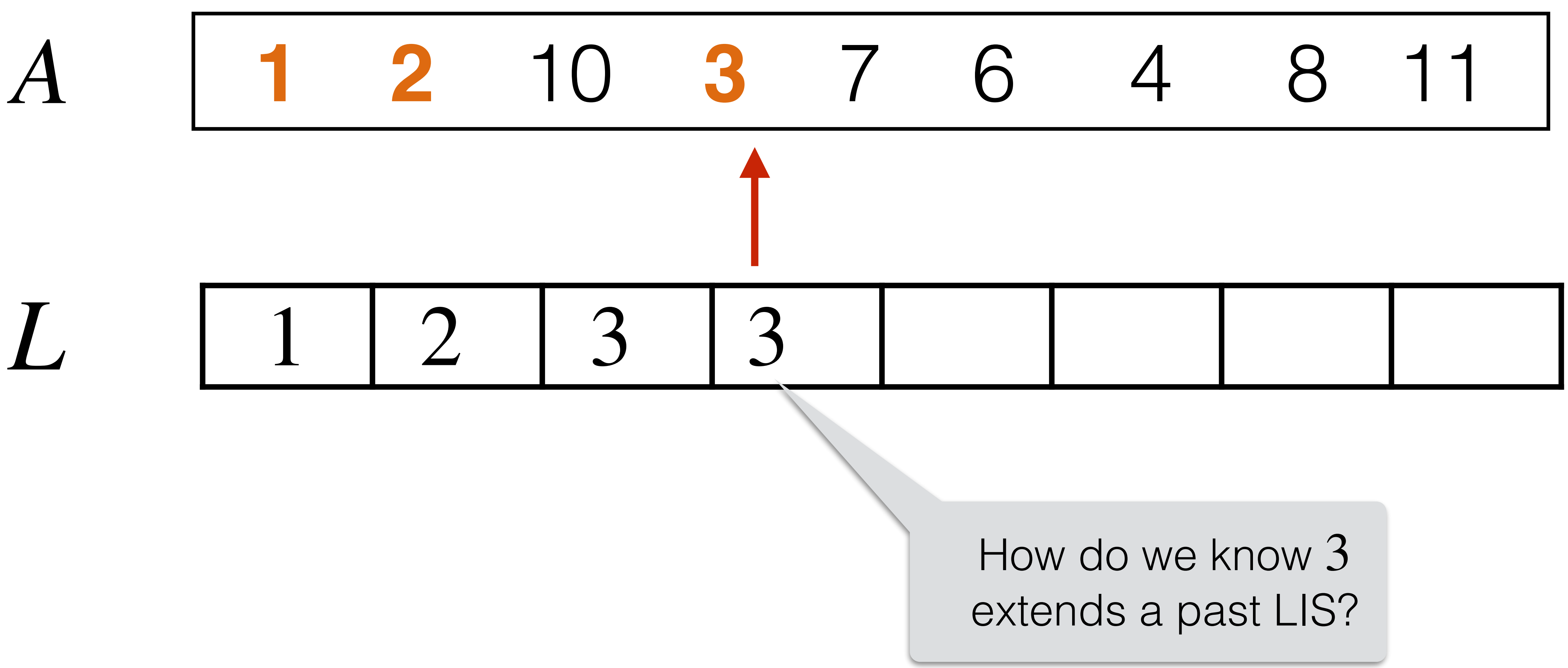
Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$



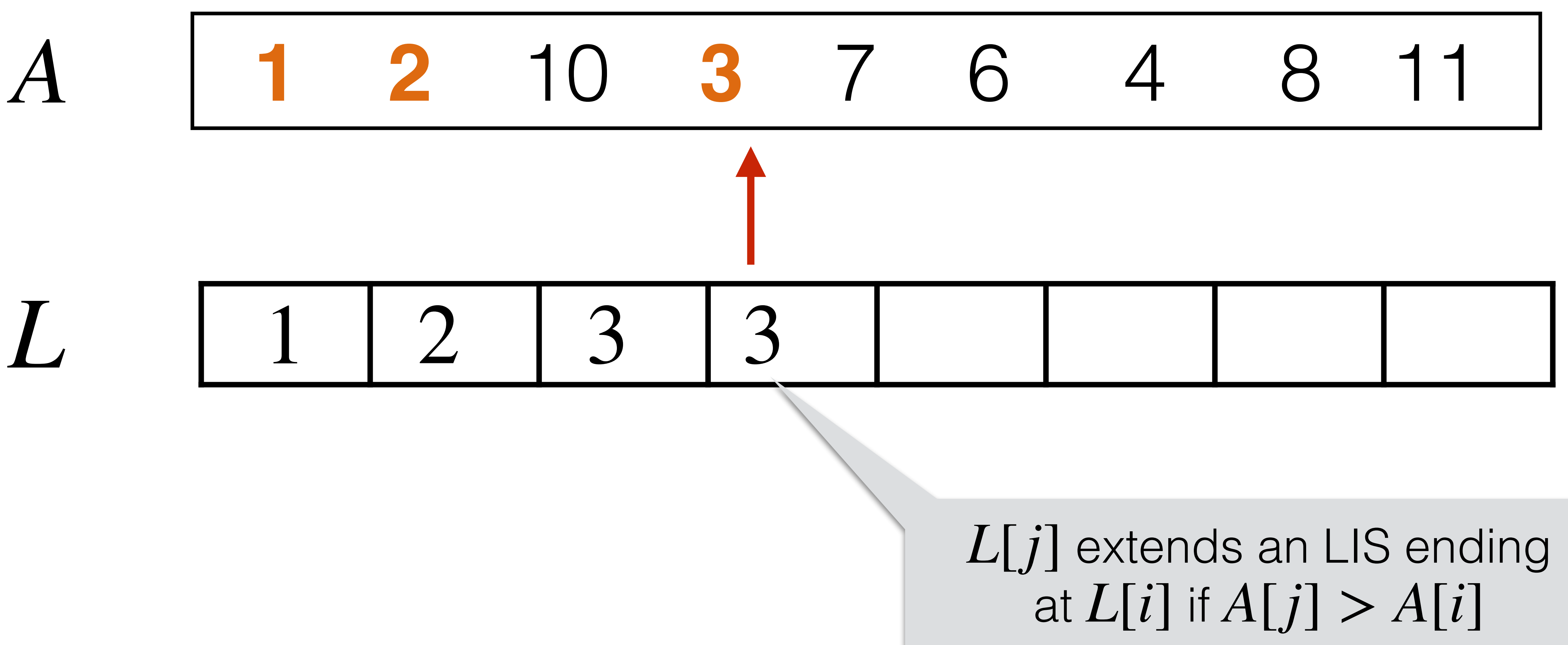
Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$



Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$



Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

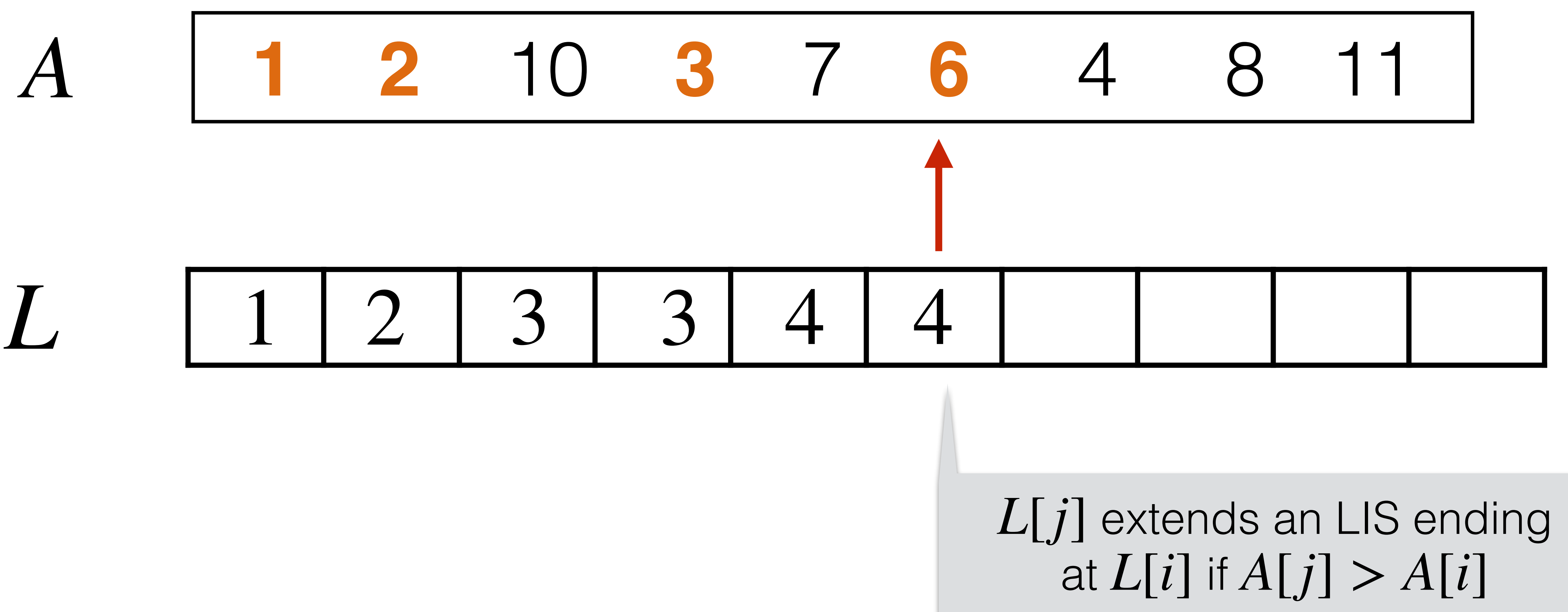
A	**1**	**2**	10	**3**	**7**	6	4	8	11

L	1	2	3	3	4					

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$



Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A	**1**	**2**	10	**3**	7	6	**4**	8	11

L	1	2	3	3	4	4	4			

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

LIS: Recurrence

$$L[j] = 1 + \max\{L[i] \mid i < j \text{ and } A[i] < A[j]\}$$

Assuming $\max \varnothing = 0$

Recursion → DP

- If we used recursion (without memoization) we'll be inefficient—we'll do a lot of repeated work
- Once you have your recurrence, the remaining pieces of the dynamic programming algorithm are
  - **Evaluation order.** In what order should I evaluate my subproblems so that everything I need is available to evaluate a new subproblem?
    - For LIS we just left-to-right on array indices
  - **Memoization structure.** Need a table (array or multi-dimensional array) to store computed values
    - For LIS, we just need a one dimensional array
    - For others, we may need a table (two-dimensional array)

LIS Analysis

- Correctness
    - Follows from the recurrence using induction
- Running time?
    - Solve $O(n)$ subproblems
    - Each one requires $O(n)$ time to take the min
    - $O(n^2)$
- Space?
    - $O(n)$ to store array $L[]$

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it!
- **State the final answer.** (In terms of the subproblem)
- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)
- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order
- **Analyze space and running time.** As always!

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)
  - Shikha Singh