CE100 Algorithms and Programming II

Matrix Chain Order / LCS

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-6 (Matrix Chain Order / LCS)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-6-lcs.md_doc.pdf), [SLIDE](ce100-week-6-lcs.md_slide.pdf), [PPTX](ce100-week-6-lcs.md_slide.pptx)

## Matrix Chain Order / Longest Common Subsequence

## Outline

* Elements of Dynamic Programming
  + Optimal Substructure
  + Overlapping Subproblems
* Recursive Matrix Chain Order Memoization
  + Top-Down Approach
  + RMC
  + MemoizedMatrixChain
    - LookupC
  + Dynamic Programming vs Memoization Summary
* Dynamic Programming
  + Problem-2 : Longest Common Subsequence
    - Definitions
    - LCS Problem
    - Notations
    - Optimal Substructure of LCS
      * Proof Case-1
      * Proof Case-2
      * Proof Case-3
* A recursive solution to subproblems (inefficient)
* Computing the length of and LCS
  + LCS Data Structure for DP
  + Bottom-Up Computation
* Constructing and LCS
  + PRINT-LCS
  + Back-pointer space optimization for LCS length
* Most Common Dynamic Programming Interview Questions

## Elements of Dynamic Programming

* When should we look for a DP solution to an optimization problem?
* Two key ingredients for the problem
  + Optimal substructure
  + Overlapping subproblems

## DP Hallmark #1

* **Optimal Substructure**
  + A problem exhibits optimal substructure
    - if an optimal solution to a problem contains within it optimal solutions to subproblems
  + **Example:** *matrix-chain-multiplication*
    - Optimal parenthesization of that splits the product between and , contains within it **optimal soln’s** to the problems of parenthesizing and

## Optimal Substructure

* Finding a suitable space of subproblems
  + Iterate on subproblem instances
  + **Example:** *matrix-chain-multiplication*
    - Iterate and look at the structure of optimal soln’s to subproblems, sub-subproblems, and so forth
    - Discover that all subproblems consists of subchains of
    - Thus, the set of chains of the form for
    - Makes a natural and reasonable space of subproblems

## DP Hallmark #2

* **Overlapping Subproblems**
  + Total number of distinct subproblems should be **polynomial** in the input size
  + When a **recursive** algorithm revisits the same problem **over and over again**,
    - We say that the optimization problem has **overlapping subproblems**

## Overlapping Subproblems

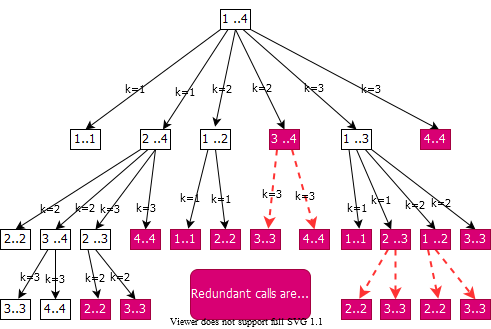
* **DP** algorithms typically take advantage of overlapping subproblems
  + by solving each problem once
  + then storing the solutions in a table
    - where it can be looked up when needed
  + using constant time per lookup

## Overlapping Subproblems

* Recursive matrix-chain order

## Direct Recursion: **Inefficient!**

* Recursion tree for
* Nodes are labeled with and values



bg right:60% w:650px

## Running Time of RMC

- For each term appears twice - Once as , and once as - Collect ’s in the summation together with the front

* Prove that using the **substitution method**

## Running Time of RMC: **Prove that**

* Try to show that (**by substitution**)
* **Base case:** for
* **Ind. Hyp.:**

## Running Time of RMC:

* **Whenever**
  + a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly
  + the total number of different subproblems is small
    - it is a good idea to see if can be applied

## Memoization

* Offers the efficiency of the usual approach while maintaining **top-down** strategy
* Idea is to **memoize** the natural, but inefficient, **recursive algorithm**

## Memoized Recursive Algorithm

* Maintains an **entry** in a **table** for the soln to each subproblem
* Each table entry contains **a special value** to indicate that the entry has yet to be filled in
* When the subproblem is **first encountered** its solution is **computed** and then **stored** in the table
* Each **subsequent** time that the subproblem encountered the value stored in the table is simply **looked up** and **returned**

## Memoized Recursive Matrix-chain Order

* Shaded subtrees are looked-up rather than recomputing

## Memoized Recursive Algorithm

* The approach assumes that
  + The set of **all possible subproblem parameters** are known
  + The relation between the **table positions** and **subproblems** is established
* Another approach is to memoize
  + by using **hashing** with subproblem parameters as key

## Dynamic Programming **vs** Memoization Summary (1)

* Matrix-chain multiplication can be solved in time
  + by either a top-down memoized recursive algorithm
  + or a bottom-up dynamic programming algorithm
* Both methods exploit the **overlapping subproblems** property
  + There are only different subproblems in total
  + Both methods **compute** the soln to **each problem once**
* **Without memoization** the natural **recursive** algorithm runs in **exponential time** since subproblems are solved repeatedly

## Dynamic Programming **vs** Memoization Summary (2)

* **In general practice**
  + If all subproblems must be solved at once
    - a bottom-up **DP algorithm always outperforms** a top-down memoized algorithm by a constant factor
  + because, bottom-up **DP** algorithm
    - Has no overhead for recursion
    - Less overhead for maintaining the table
  + **DP:** **Regular** pattern of **table accesses** can be exploited to reduce the time and/or space requirements even further
  + **Memoized:** If some problems need not be solved at all, it has the advantage of avoiding solutions to those subproblems

## Problem 3: **Longest Common Subsequence**

**Definitions**

* A **subsequence** of a given sequence is just the **given sequence** with **some elements** (possibly none) **left out**
* **Example:**
  + - is a subsequence of

## Problem 3: **Longest Common Subsequence**

**Definitions**

* **Formal definition:** Given a sequence , sequence is a subsequence of
  + if a **strictly increasing sequence** of indices of such that for all , where
* **Example:** is a subsequence of with the **index sequence**

## Problem 3: **Longest Common Subsequence**

**Definitions**

* If is a subsequence of both and , we denote as a **common subsequence** of and .
* **Example:**
* is a common subsequence (**of length 3**) of and .
* **Two longest common subsequence (LCSs)** of and ?
  + of length
  + of length
    - *The optimal solution value = 4*

## Longest Common Subsequence (LCS) Problem

* **LCS problem:** Given two sequences
  + and
  + , find the **LCS** of
* **Brute force approach:**
  + Enumerate all subsequences of
  + Check if each subsequence is also a subsequence of
  + Keep track of the **LCS**
  + What is the complexity?
  + There are subsequences of
    - **Exponential runtime**

## Notation

* **Notation:** Let denote the prefix of
  + i.e.
* **Example:**

## Optimal Substructure of an **LCS**

* Let and are given
* Let be an **LCS** of and



center h:250px

* **Question 1:** If , how to define the optimal substructure?
  + We must have and

## Optimal Substructure of an **LCS**

* Let and are given
* Let be an **LCS** of and



center h:250px

* **Question 2:** If , how to define the optimal substructure?
  + We must have

## Optimal Substructure of an **LCS**

* Let and are given
* Let be an **LCS** of and



center h:250px

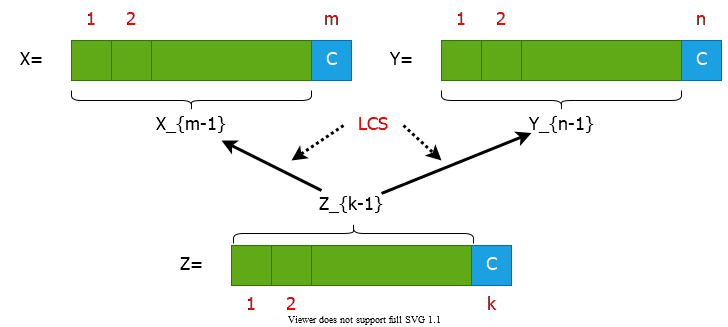
* **Question 3:** If , how to define the optimal substructure?
  + We must have

## Theorem: Optimal Substructure of an **LCS**

* Let and Y = <y1, y2, …, yn> are given
* Let be an **LCS** of and
* **Theorem:** Optimal substructure of an LCS:
  + If
    - then and is an **LCS** of and
  + If and
    - then is an **LCS** of and
  + If and
    - then is an **LCS** of and

## Optimal Substructure Theorem **(case 1)**

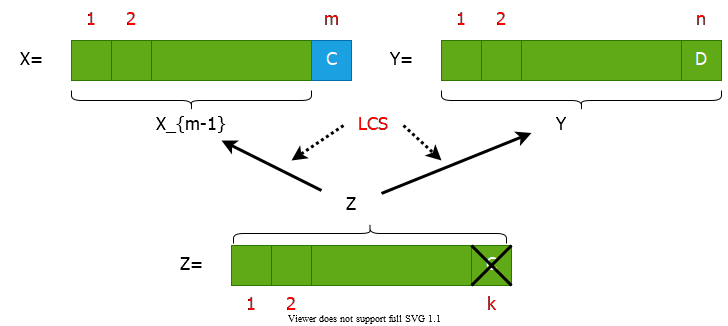
* If then and is an **LCS** of and



center h:350px

## Optimal Substructure Theorem (case 2)

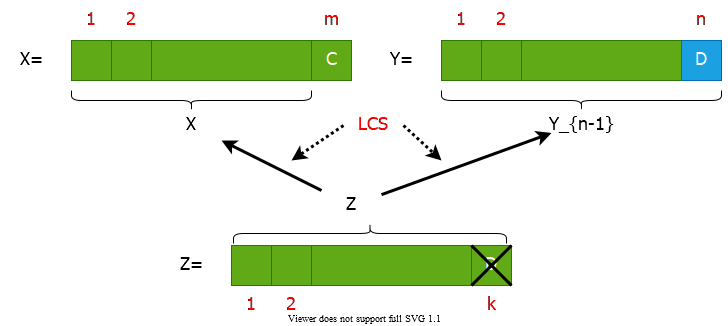
* If and then is an **LCS** of and



center h:350px

## Optimal Substructure Theorem (case 3)

* If and then is an **LCS** of and



center h:350px

## Proof of Optimal Substructure Theorem (case 1)

* If then and is an **LCS** of and
* **Proof:** If then
  + we can append to to obtain a common subsequence of length **contradiction**
  + Thus, we must have
  + Hence, the prefix is a **length-() CS** of and
* **We have to show that** is in fact an **LCS** of and
* **Proof by contradiction:**
  + **Assume that** a CS of and with
  + Then appending to produces a **CS** of length

## Proof of Optimal Substructure Theorem (case 2)

* If and then is an **LCS** of and
* **Proof :** If then is a CS of and
  + **We have to show that** is in fact an **LCS** of and
* **(Proof by contradiction)**
  + Assume that a CS of and with
  + Then would also be a CS of and
  + Contradiction to the assumption that
    - is an LCS of and with
* **Case 3:** Dual of the proof for (case 2)

## A Recursive Solution to Subproblems

* Theorem implies that there are one or two subproblems to examine
* **if** **then**
  + we must solve the subproblem of finding an **LCS** of
  + appending to this **LCS** yields an **LCS** of
* **else**
  + we must solve **two subproblems**
    - finding an **LCS** of
    - finding an **LCS** of
  + longer of these two **LCS** s is an **LCS** of
* **endif**

## Recursive Algorithm **(Inefficient)**

## A Recursive Solution

* length of an **LCS** of and

## Computing the Length of an **LCS**

* We can easily write an **exponential-time recursive algorithm** based on the given recurrence. **Inefficient!**
* How many distinct subproblems to solve?
* **Overlapping subproblems property:** Many subproblems share the same sub-subproblems.
  + **e.g.** Finding an **LCS** to and an **LCS** to
  + has the sub-subproblem of finding an **LCS** to
* Therefore, we can use **dynamic programming**.

## Data Structures

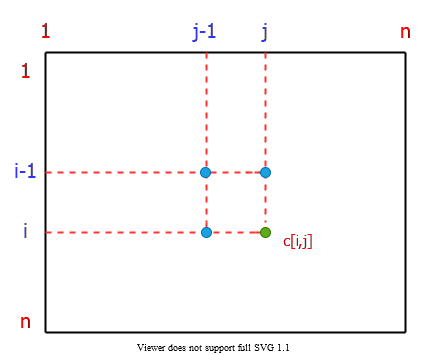
* Let:
  + length of an **LCS** of and
  + direction towards the table entry corresponding to the optimal subproblem solution chosen when computing .
  + Used to simplify the construction of an optimal solution at the end.
* Maintain the following tables:

## Bottom-up Computation

* **Reminder:**
* How to choose the order in which we process values?
* The values for , , and must be computed before computing .

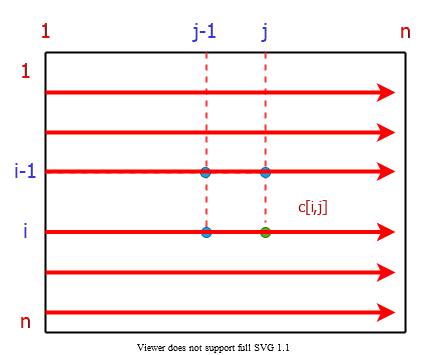
## Bottom-up Computation

**Need to process:** **after computing:** , ,



bg right:40% h:350px

## Bottom-up Computation



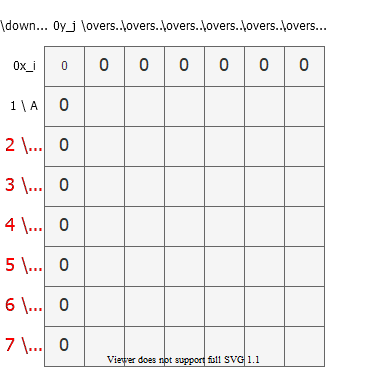
bg right:40% h:350px

## Computing the Length of an **LCS**

$$
\begin{align\*}
\frac{\text{Total Runtime} = \Theta(mn)}{\text{Total Space} = \Theta(mn)}
\begin{cases}
& LCS-LENGTH(X,Y) \\
& \quad m \leftarrow length[X]; n \leftarrow length[Y] \\
& \quad \text{for} \ i \leftarrow 0 \ \text{to} \ m \ \text{do} \ c[i, 0] \leftarrow 0 \\
& \quad \text{for} \ j \leftarrow 0 \ \text{to} \ n \ \text{do} \ c[0, j] \leftarrow 0 \\
& \quad \text{for} \ i \leftarrow 1 \ \text{to} \ m \ \text{do} \\
& \qquad \text{for} \ j \leftarrow 1 \ \text{to} \ n \ \text{do} \\
& \qquad \quad \text{if} \ x\_i = y\_j \ \text{then} \\
& \qquad \quad \quad c[i, j] \leftarrow c[i-1, j-1]+1 \\
& \qquad \quad \quad b[i, j] \leftarrow " \nwarrow " \\
& \qquad \quad \text{else if} \ c[i - 1, j] \geq c[i, j-1] \\
& \qquad \quad \quad c[i, j] \leftarrow c[i-1, j] \\
& \qquad \quad \quad b[i, j] \leftarrow "\uparrow " \\
& \qquad \quad \text{else} \\
& \qquad \quad \quad c[i, j] \leftarrow c[i, j-1] \\
& \qquad \quad \quad b[i, j] \leftarrow " \leftarrow " \\
\end{cases}
\end{align\*}
$$

## Computing the Length of an LCS-1

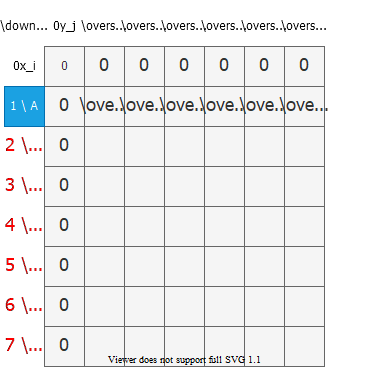
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-2

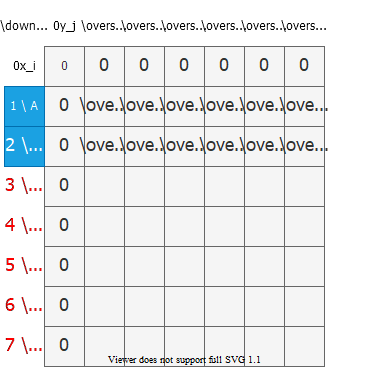
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-3

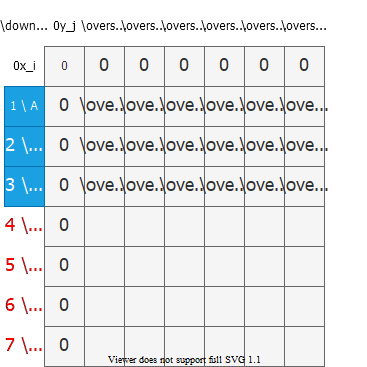
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-4

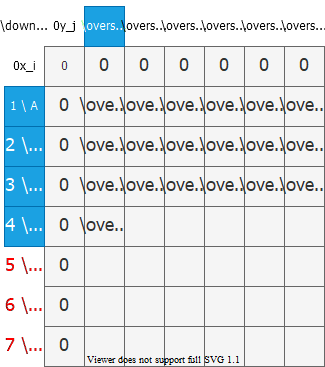
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-5

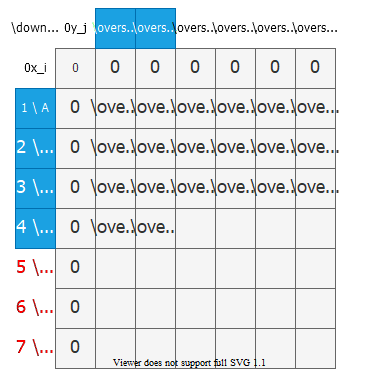
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-6

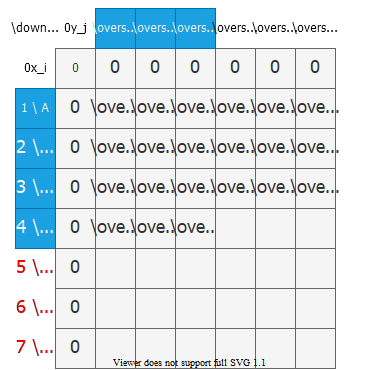
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-7

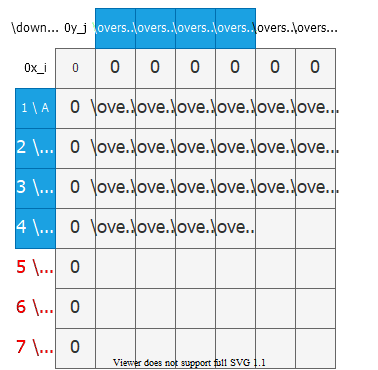
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-8

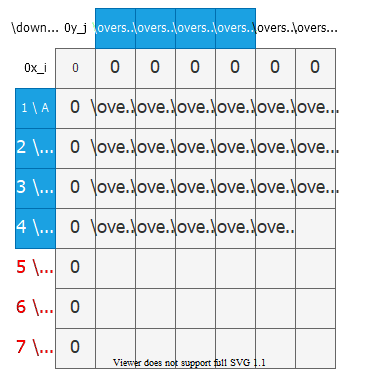
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-9

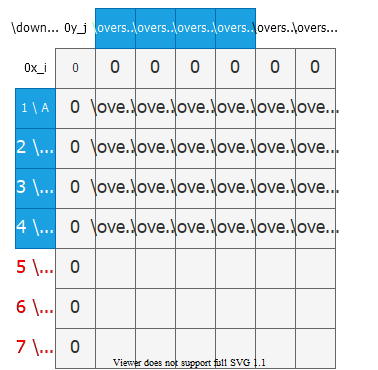
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-10

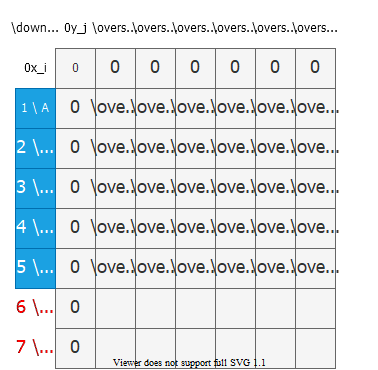
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-11

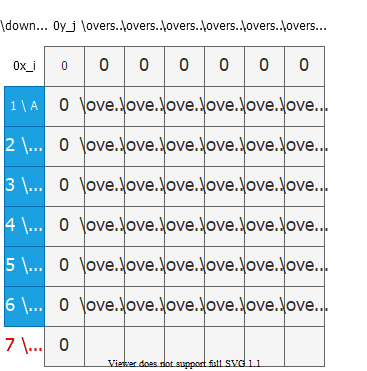
Operation of LCS-LENGTH on the sequences



bg right:60% h:650px

## Computing the Length of an LCS-12

Operation of LCS-LENGTH on the sequences

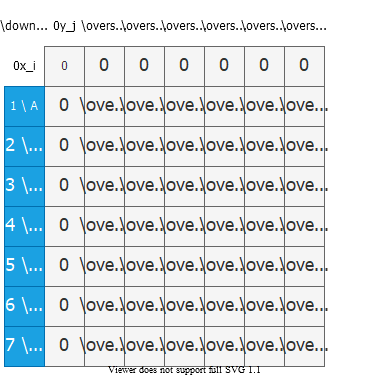


bg right:60% h:650px

## Computing the Length of an LCS-13

Operation of LCS-LENGTH on the sequences

* Running-time = since each table entry takes time to compute

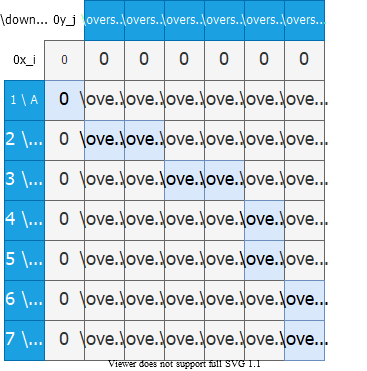


bg right:60% h:650px

## Computing the Length of an LCS-14

Operation of LCS-LENGTH on the sequences

* Running-time = since each table entry takes time to compute
* **LCS** of



bg right:60% h:650px

## Constructing an **LCS**

* The table returned by **LCS-LENGTH** can be used to quickly construct an **LCS** of
* Begin at and trace through the table following arrows
* Whenever you encounter a “” in entry it implies that is an element of **LCS**
* The elements of **LCS** are encountered in **reverse order**

## Constructing an **LCS**

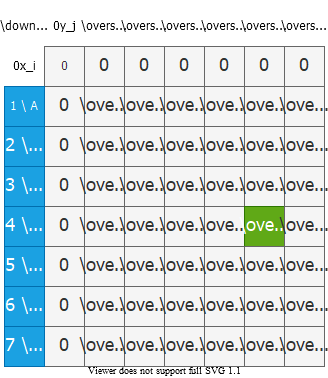
* The recursive procedure prints out in proper order
* This procedure takes time since at least one of and is decremented in each stage of the recursion

$$
\begin{align\*}
& \text{PRINT-LCS}(b, X, i, j) \\
& \quad \text{if} \ i = 0 \ \text{or} j = 0 \ \text{then} \\
& \quad \text{return} \\
& \quad \text{if} \ b[i, j] = " \nwarrow " \ \text{then} \\
& \qquad \text{PRINT-LCS}(b, X, i-1, j-1) \\
& \qquad \text{print} \ x\_i \\
& \quad \text{else if} \ b[i, j] = " \uparrow " \ \text{then} \\
& \qquad \text{PRINT-LCS}(b, X, i-1, j) \\
& \quad \text{else} \\
& \qquad \text{PRINT-LCS}(b, X, i, j-1)
\end{align\*}
$$

* **The initial invocation:**

## Do we really need the b table (back-pointers)?

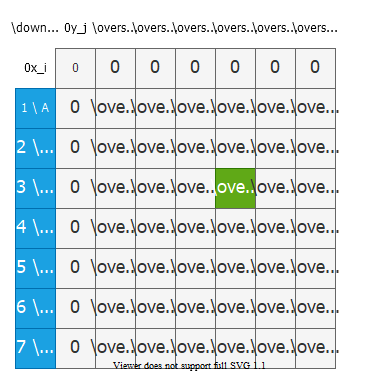
* **Question:** From which neighbor did we expand to the highlighted cell?
* **Answer:** Upper-left neighbor,because .



bg right:50% h:650px

## Do we really need the b table (back-pointers)?

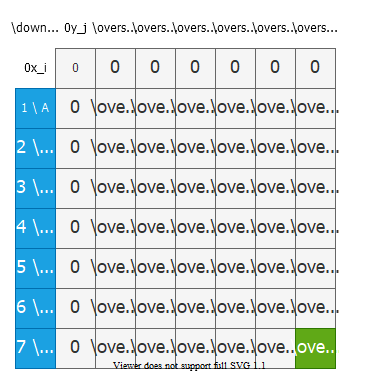
* **Question:** From which neighbor did we expand to the highlighted cell?
* **Answer:** Left neighbor, because and .



bg right:50% h:650px

## Do we really need the b table (back-pointers)?

* **Question:** From which neighbor did we expand to the highlighted cell?
* **Answer:** Upper neighbor,because and . *(See pseudo-code to see how ties are handled.)*



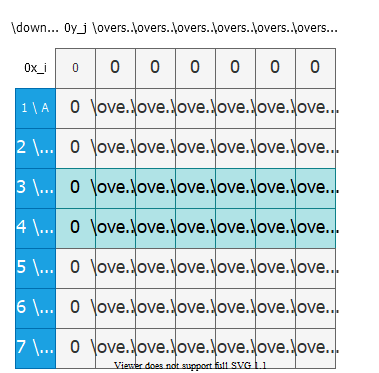
bg right:50% h:650px

## Improving the Space Requirements

* We can eliminate the b table altogether
  + each entry depends only on other table entries: , and
* Given the value of :
  + We can determine in time which of these values was used to compute without inspecting table
  + We save space by this method
  + However, space requirement is still since we need space for the table anyway

## What if we store the last 2 rows only?

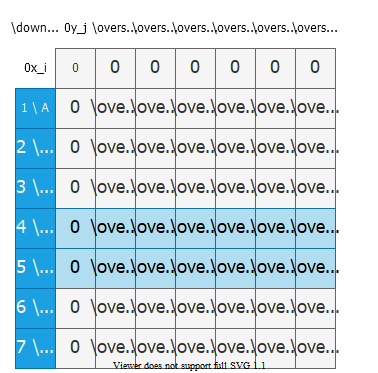
* To compute , we only need , ,and
* So, we can store only the last two rows.



bg right:50% h:650px

## What if we store the last 2 rows only?

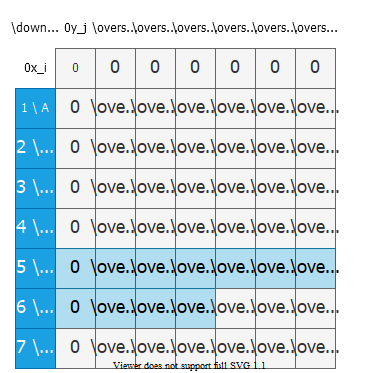
* To compute , we only need , , and
* So, we can store only the last two rows.



bg right:50% h:650px

## What if we store the last 2 rows only?

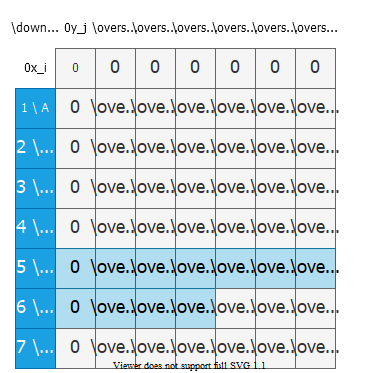
* To compute , we only need , , and
* So, we can store only the last two rows.
* This reduces space complexity from to .
* Is there a problem with this approach?



bg right:50% h:650px

## What if we store the last 2 rows only?

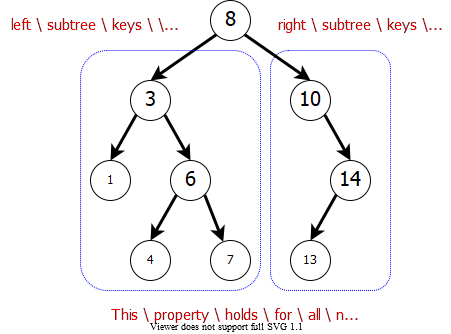
* **Is there a problem with this approach?**
  + We cannot construct the optimal solution because we cannot backtrace anymore.
  + This approach works if we only need the length of an LCS, not the actual LCS.



bg right:50% h:650px

## Problem 4 **Optimal Binary Search Tree**

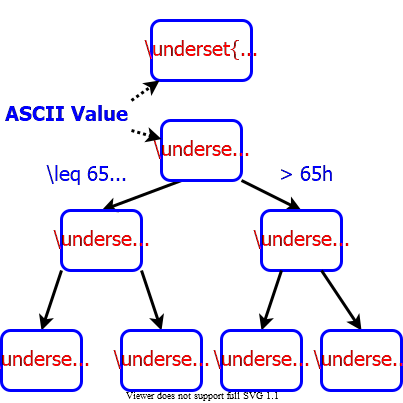
## **Reminder:** Binary Search Tree (BST)



center h:470px

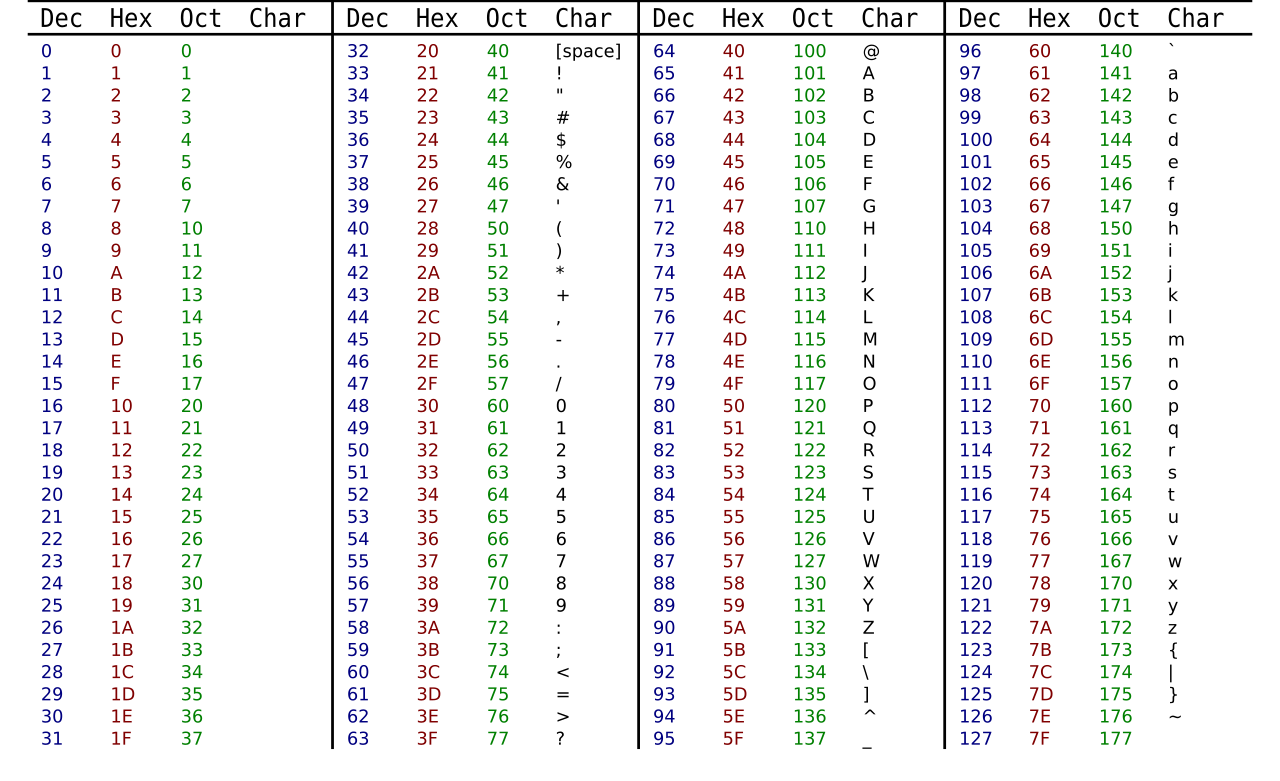
## Binary Search Tree Example

* **Example:** English-to-French translation
  + Organize (English, French) word pairs in a BST
    - **Keyword:** English word
    - **Satellite Data:** French word
* We can search for an English word (node key) efficiently, and return the corresponding French word (satellite data).



bg right:40% h:460px

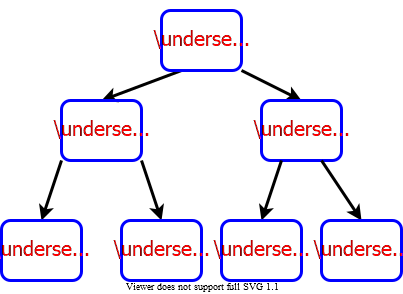
## **ASCII** Table



center h:550

## Binary Search Tree Example

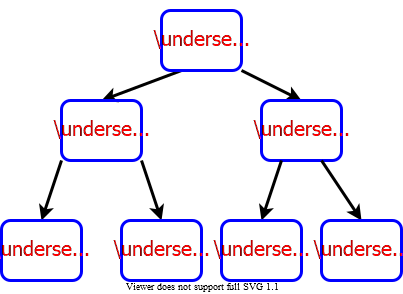
Suppose we know the frequency of each keyword in texts:



bg right:50% h:400px

## Cost of a Binary Search Tree

**Example:** If we search for keyword **“while”**, we need to access nodes. So, of the queries will have cost of .

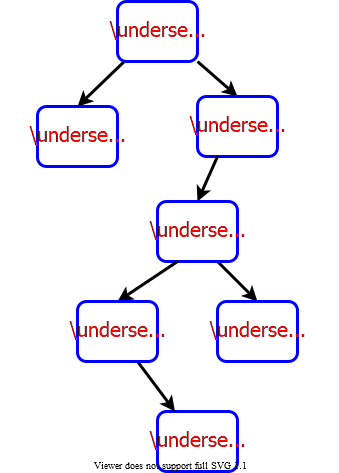


bg right:50% h:400px

## Cost of a Binary Search Tree

**Example:** If we search for keyword **“while”**, we need to access nodes. So, of the queries will have cost of .

* This is in fact an optimal BST.



bg right:40% h:650px

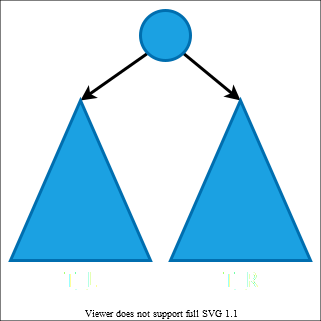
## Optimal Binary Search Tree Problem

* **Given:**
  + A collection of keys to be stored in a **BST**.
  + The corresponding values for
    - : probability of searching for key
* **Find:**
  + An **optimal BST** with minimum total cost:
* **Note:** The BST will be static. Only search operations will be performed. No insert, no delete, etc.

## Cost of a **Binary Search Tree**

* **Lemma 1**: Let be a BST containing keys . Let and be the left and right subtrees of . Then we have:

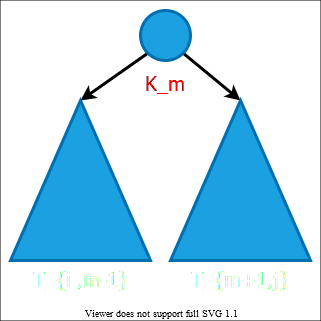
**Intuition:** When we add the root node, the depth of each node in and increases by . So, the cost of node increases by . In addition, the cost of root node is . That’s why, we have the last term at the end of the formula above.



bg right:45% h:500px

## Optimal Substructure Property

* **Lemma 2:** Optimal substructure property
  + Consider an optimal **BST** for keys
  + Let be the key at the root of
* **Then:**
  + is an **optimal BST** for subproblem containing keys:
  + is an **optimal BST** for subproblem containing keys:



bg right:30% h:390px

## Recursive Formulation

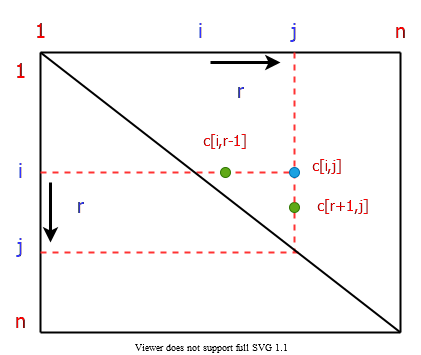
* **Note:** *We don’t know which root vertex leads to the minimum total cost. So, we need to try each vertex , and choose the one with minimum total cost.*
* : cost of an optimal BST for the subproblem

## Bottom-up computation

* How to choose the order in which we process values?
* Before computing , we have to make sure that the values for and have been computed for all .

## Bottom-up computation

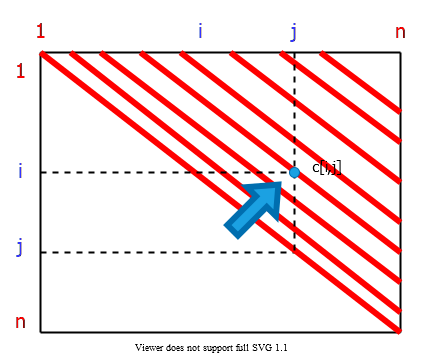
* must be processed after and



bg right:40% h:430px

## Bottom-up computation

* If the entries are computed in the shown order, then and values are guaranteed to be computed before .



bg right:40% h:430px

## Computing the Optimal BST Cost

## Note on Prefix Sum

* We need values for each , where:
* If we compute the summation directly for every pair, the runtime would be .
* Instead, we spend time in preprocessing to compute the prefix sum array **PS**. Then we can compute each in time using **PS**.

## Note on Prefix Sum

* In preprocessing, compute for each :
  + : the sum of values for
* Then, we can compute in time as follows:
* **Example:**

## Most Common Dynamic Programming **Interview** Questions

* Problem-1: Longest Increasing Subsequence
  + https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/
  + https://en.wikipedia.org/wiki/Longest\_increasing\_subsequence#:~:text=In%20computer%20science%2C%20the%20longest,not%20necessarily%20contiguous%2C%20or%20unique.
  + https://www.youtube.com/watch?v=22s1xxRvy28&ab\_channel=StableSort
* Problem-2: Edit Distance
  + https://www.geeksforgeeks.org/edit-distance-dp-5/
  + https://www.youtube.com/watch?v=tU2f2JwHmfQ&feature=youtu.be&ab\_channel=PrepForTech
  + Recursive
    - https://www.youtube.com/watch?v=8Q2IEIY2pDU&ab\_channel=BenLangmead
  + DP
    - https://www.youtube.com/watch?v=0KzWq118UNI&ab\_channel=BenLangmead
    - https://www.youtube.com/watch?v=eAVGRWSryGo&ab\_channel=BenLangmead
* Problem-3: Partition a set into two subsets such that the difference of subset sums is minimum
  + https://www.geeksforgeeks.org/partition-a-set-into-two-subsets-such-that-the-difference-of-subset-sums-is-minimum/
* Problem-4: Count number of ways to cover a distance
  + https://www.geeksforgeeks.org/count-number-of-ways-to-cover-a-distance/
* Problem-5: Find the longest path in a matrix with given constraints
  + https://www.geeksforgeeks.org/find-the-longest-path-in-a-matrix-with-given-constraints/
* Problem-6: Subset Sum Problem
  + https://www.geeksforgeeks.org/subset-sum-problem-dp-25/
* Problem-7: Optimal Strategy for a Game
  + https://www.geeksforgeeks.org/optimal-strategy-for-a-game-dp-31/
* Problem-8: 0-1 Knapsack Problem
  + https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
* Problem-9: Boolean Parenthesization Problem
  + https://www.geeksforgeeks.org/boolean-parenthesization-problem-dp-37/
* Problem-10: Shortest Common Supersequence
  + https://www.geeksforgeeks.org/shortest-common-supersequence/
  + https://en.wikipedia.org/wiki/Shortest\_common\_supersequence\_problem
* Problem-11: Partition Problem
  + https://www.geeksforgeeks.org/partition-problem-dp-18/
* Problem-12: Cutting a Rod
  + https://www.geeksforgeeks.org/cutting-a-rod-dp-13/
* [Problem-13: Coin Change](https://www.geeksforgeeks.org/coin-change-dp-7)
* [Problem-14: Word Break Problem](https://www.geeksforgeeks.org/word-break-problem-dp-32/)
* [Problem-15: Maximum Product Cutting](https://www.geeksforgeeks.org/maximum-product-cutting-dp-36/)
* Problem-16: Dice Throw
  + https://www.geeksforgeeks.org/dice-throw-dp-30/
* Problem-17: Box Stacking Problem
  + https://www.geeksforgeeks.org/box-stacking-problem-dp-22/
* Problem-18: Egg Dropping Puzzle
  + https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)