CE100 Algorithms and Programming II

Dynamic Programming

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-5 (Dynamic Programming)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-5-dp.md_doc.pdf), [SLIDE](ce100-week-5-dp.md_slide.pdf), [PPTX](ce100-week-5-dp.md_slide.pptx)

## Quicksort Sort

## Outline

* Convex Hull (Divide & Conquer)
* Dynamic Programming
  + Introduction
  + Divide-and-Conquer (DAC) vs Dynamic Programming (DP)
* Fibonacci Numbers
  + Recursive Solution
  + Bottom-Up Solution
* Optimization Problems
* Development of a DP Algorithms
* Matrix-Chain Multiplication
  + Matrix Multiplication and Row Columns Definitions
  + Cost of Multiplication Operations (pxqxr)
  + Counting the Number of Parenthesizations
* The Structure of Optimal Parenthesization
  + Characterize the structure of an optimal solution
  + A Recursive Solution
    - Direct Recursion Inefficiency.
  + Computing the optimal Cost of Matrix-Chain Multiplication
  + Bottom-up Computation
* Algorithm for Computing the Optimal Costs
  + MATRIX-CHAIN-ORDER
* Construction and Optimal Solution
  + MATRIX-CHAIN-MULTIPLY
* Summary

## Dynamic Programming - **Introduction**

* An algorithm design paradigm like divide-and-conquer
* **Programming:** A tabular method (not writing computer code)
  + Older sense of planning or scheduling, typically by filling in a table
* **Divide-and-Conquer (DAC):** subproblems are independent
* **Dynamic Programming (DP):** subproblems are not independent
* Overlapping subproblems: subproblems share sub-subproblems
  + In solving problems with overlapping subproblems
    - A DAC algorithm **does redundant** work
      * Repeatedly solves common subproblems
    - A DP algorithm solves each problem just once
      * **Saves** its result **in a table**

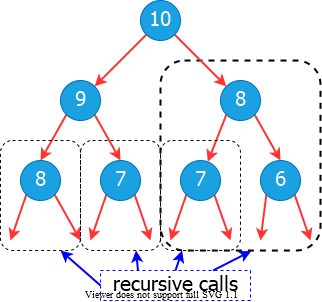
## Problem 1: **Fibonacci Numbers** Recursive Solution

* **Reminder:**

$$
F(0)=0 \text{ and } F(1)=1 \\
F(n)=F(n-1)+F(n-2)
$$

REC-FIBO(n)  
 if n < 2  
 return n  
 else  
 return REC-FIBO(n-1) + REC-FIBO(n-2)

* Overlapping subproblems in different recursive calls. Repeated work!



bg right:40% w:500px

## Problem 1: **Fibonacci Numbers** Recursive Solution

* **Recurrence:**
  + *exponential runtime*
* Recursive algorithm inefficient because it recomputes the same repeatedly in different branches of the recursion tree.

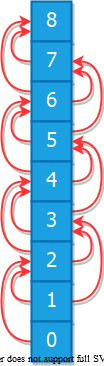
## Problem 1: **Fibonacci Numbers** Bottom-up Computation

* **Reminder:**

$$
F(0)=0 \text{ and } F(1)=1 \\
F(n)=F(n-1)+F(n-2)
$$

* **Runtime**

ITER-FIBO(n)  
 F[0] = 0  
 F[1] = 1  
 for i = 2 to n do  
 F[i] = F[i-1] + F[i-2]  
 return F[n]



bg right:40% w:150px

## Optimization Problems

* **DP** typically applied to optimization problems
* In an optimization problem
  + There are many possible solutions (feasible solutions)
  + Each solution has a value
  + Want to find an optimal solution to the problem
    - *A solution with the optimal value (min or max value)*
  + Wrong to say **the** optimal solution to the problem
    - *There may be several solutions with the same optimal value*

## Development of a DP Algorithm

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

## Problem 2: **Matric Chain Multiplication**

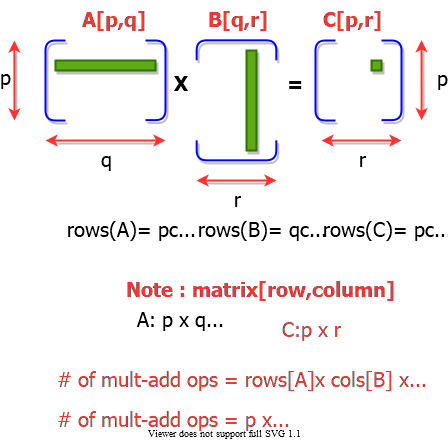
* **Input:** a sequence (chain) of matrices
* **Aim:** compute the product
* A **product of matrices** is **fully parenthesized** if
  + It is either a **single matrix**
  + Or, the **product** of **two fully parenthesized matrix products** surrounded by a pair of parentheses. for
* All parenthesizations yield the same product; matrix product is associative

## Matrix-chain Multiplication: **An Example Parenthesization**

* **Input:** ( distinct ways of full parenthesization)
* The way we parenthesize a chain of matrices can have a dramatic effect on the cost of computing the product

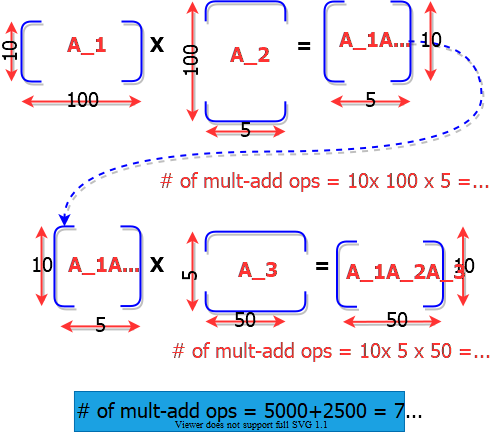
## Matrix-chain Multiplication: **Reminder**

MATRIX-MULTIPLY(A, B)  
 if cols[A]!=rows[B] then   
 error(“incompatible dimensions”)  
 for i=1 to rows[A] do  
 for j=1 to cols[B] do   
 C[i,j]=0  
 for k=1 to cols[A] do   
 C[i,j]=C[i,j]+A[i,k]·B[k,j]  
 return C

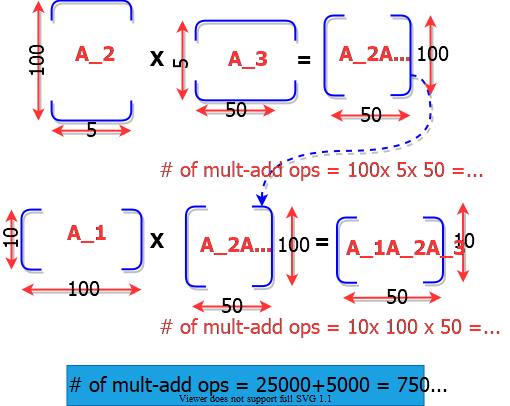


bg right:50% w:500px

## Matrix Chain Multiplication: **Example**

* , ,
  + Which paranthesization is better? or ?
* 
* bg right:50% w:650px

## Matrix Chain Multiplication: **Example**

* , ,
  + Which paranthesization is better? or ?
* 
* bg right:50% w:650px

## Matrix Chain Multiplication: **Example**

* , ,
  + Which paranthesization is better? or ?

**In summary:**

* = of multiply-add ops:
* = of multiple-add ops:

First parenthesization yields **10x faster** computation

## Matrix-chain Multiplication Problem

* **Input:** A chain of matrices,
  + where is a matrix
* **Objective:** Fully parenthesize the product
  + - such that the number of **scalar mult-adds** is minimized.

## Counting the Number of Parenthesizations

* **Brute force approach:** exhaustively check all parenthesizations
* : of parenthesizations of a sequence of n matrices
* We can split sequence between and matrices for any , then parenthesize the two resulting sequences independently, i.e.,
* We obtain the recurrence

## Number of Parenthesizations:

* and
* The recurrence generates the sequence of **Catalan Numbers** Solution is where
* The number of solutions is **exponential** in
* Therefore, brute force approach is a poor strategy

## The Structure of Optimal Parenthesization

* **Notation:** : The matrix that results from evaluation of the product:
* **Observation:** Consider the last multiplication operation in any parenthesization:
  + There is a value such that:
    - First, the product is computed
    - Then, the product is computed
    - Finally, the matrices and are multiplied

## **Step 1:** Characterize the Structure of an Optimal Solution

* An optimal parenthesization of product will be: for some value
* The **cost of this optimal parenthesization** will be: Cost of computing Cost of computing Cost of multiplying

## **Step 1:** Characterize the Structure of an Optimal Solution

* **Key observation:** Given optimal parenthesization
* Parenthesization of the subchain
* Parenthesization of the subchain

should both be optimal

* Thus, optimal solution to an instance of the problem contains optimal solutions to subproblem instances
  + **i.e.**, optimal substructure within an optimal solution exists.

## **Step 2:** A Recursive Solution

* **Step 2:** Define the value of an optimal solution recursively in terms of optimal solutions to the subproblems
* Assume we are trying to determine the min cost of computing
* : min of scalar multiply-add opns needed to compute
  + **Note:** *The optimal cost of the original problem:*
* How to compute recursively?

## **Step 2:** A Recursive Solution

* Base case: (single matrix, no multiplication)
* Let the size of matrix be
* Consider an optimal parenthesization of chain
* The optimal cost:
* **where:**
  + : Optimal cost of computing
  + : Optimal cost of computing
  + : Cost of multiplying and

## **Step 2:** A Recursive Solution

* In an optimal parenthesization: must be chosen to minimize
* The recursive formulation for :

## **Step 2:** A Recursive Solution

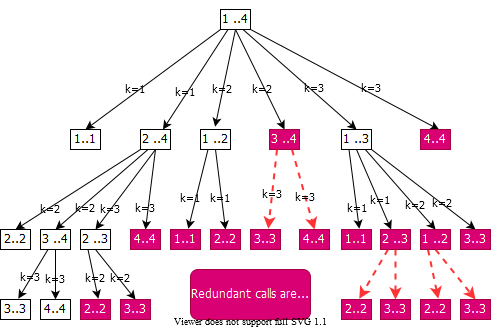
* The values give the **costs of optimal solutions** to subproblems
* In order to keep track of how to construct an optimal solution
  + Define to be the value of which yields the optimal split of the subchain
    - That is, such that
      * holds

## Direct Recursion: **Inefficient!**

* Recursive Matrix-Chain (**RMC**) Order

RMC(p,i,j)  
   
 if (i == j) then   
 return 0  
   
 m[i, j] = INF   
   
 for k=i to j-1 do  
   
 q = RMC(p, i, k) + RMC(p, k+1, j) + p\_{i-1} p\_k p\_j  
   
 if q < m[i, j] then  
 m[i, j] = q  
   
 endfor  
  
 return m[i, j]

## Direct Recursion: **Inefficient!**

* Recursion tree for
* Nodes are labeled with and values
* 
* bg right:60% w:650px

## Computing the Optimal Cost (**Matrix-Chain Multiplication**)

**An important observation:** - We have **relatively few subproblems** - one problem for each choice of and satisfying - total subproblems - We can write a **recursive** algorithm based on recurrence. - However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree - This property, **overlapping subproblems**, is the **second important feature** for applicability of **dynamic programming**

## Computing the Optimal Cost (**Matrix-Chain Multiplication**)

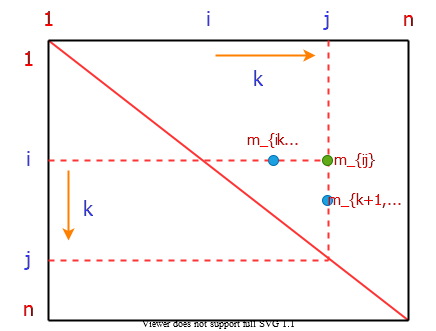
* Compute the value of an optimal solution in a **bottom-up** fashion
  + matrix has dimensions for
  + the input is a sequence where
* Procedure uses the following auxiliary tables:
  + : for storing the costs
  + : records which index of achieved the optimal cost in computing

## **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

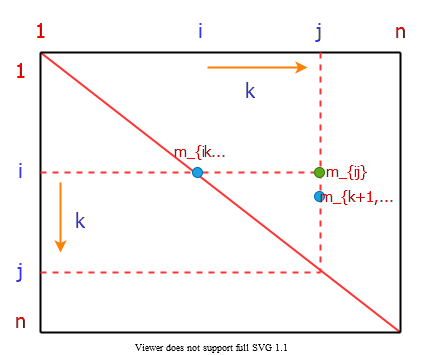
* must be processed after and
* **Reminder:** computed only for



bg right:50% h:500px

## **Bottom-Up** Computation

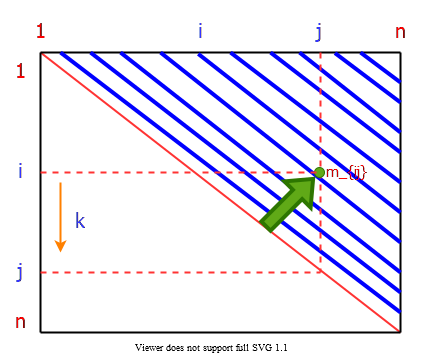
* must be processed after and
* How to set up the iterations over and to compute ?



bg right:50% h:500px

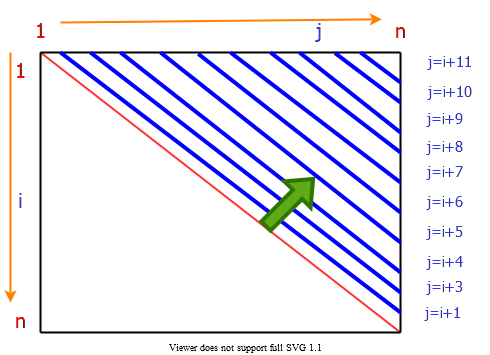
## **Bottom-Up** Computation

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



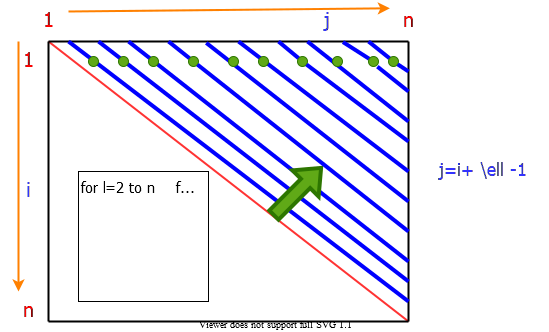
bg right:50% h:500px

## **Bottom-Up** Computation



bg right:50% h:450px

## **Bottom-Up** Computation



bg right:65% h:500px

## Algorithm for Computing the Optimal Costs

* *Note*: l and p\_{i-1} p\_k p\_j

MATRIX-CHAIN-ORDER(p)  
 n = length[p]-1  
 for i=1 to n do  
 m[i, i]=0  
 endfor  
 for l=2 to n do  
 for i=1 to n n-l+1 do  
 j=i+l-1  
 m[i, j]=INF  
 for k=i to j-1 do  
 q=m[i,k]+m[k+1, j]+p\_{i-1} p\_k p\_j  
 if q < m[i,j] then  
 m[i,j]=q  
 s[i,j]=k  
 endfor  
 endfor  
 endfor  
 return m and s

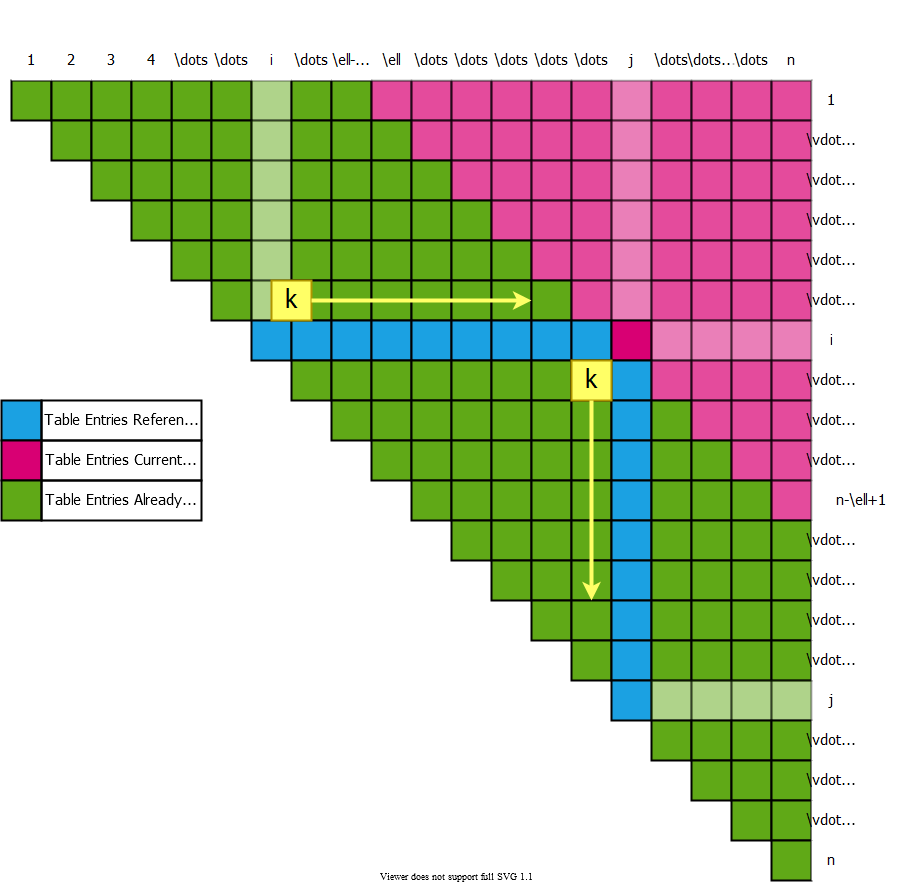
## Algorithm for Computing the Optimal Costs

* The algorithm first computes
  + for min costs for all chains of length 1
* **Then**, for computes
  + for min costs for all chains of length
* For each value of ,
  + depends only on table entries for , which are already computed

## Algorithm for Computing the Optimal Costs

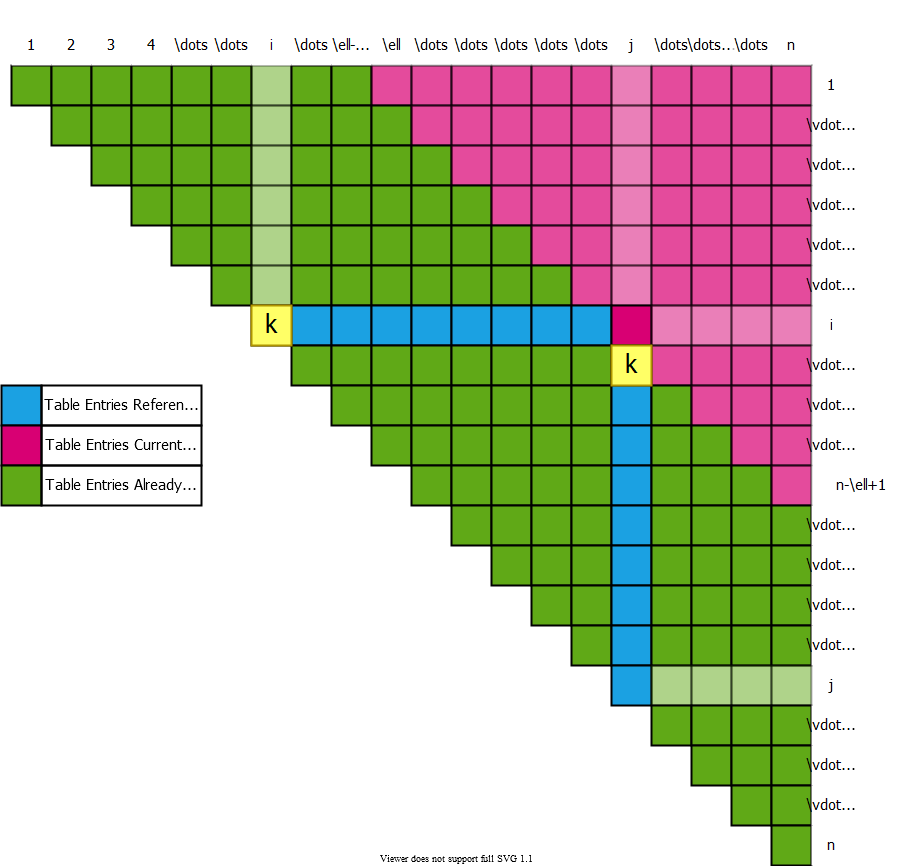
$$
\begin{align\*}
{
\begin{align\*}
\text{compute } m[i,i+1] \\
\underbrace{ \{ m[1,2],m[2,3], \dots ,m[n-1,n]\} }\_{(n-1) \text{ values}}
\end{align\*}
}
& \begin{cases}
& \ell=2 \\
& \text{for } i=1 \text{ to } n-1 \text{ do } \\
& \quad m[i,i+1]=\infty \\
& \quad \quad \text{for } k=i \text{ to } i \text{ do } \\
& \quad \quad \quad \vdots
\end{cases}
\end{align\*}
\\
\begin{align\*}
{
\begin{align\*}
\text{compute } m[i,i+2] \\
\underbrace{ \{ m[1,3],m[2,4], \dots ,m[n-2,n]\} }\_{(n-2) \text{ values}}
\end{align\*}
}
& \begin{cases}
& \ell=3 \\
& \text{for } i=1 \text{ to } n-2 \text{ do } \\
& \quad m[i,i+2]=\infty \\
& \quad \quad \text{for } k=i \text{ to } i+1 \text{ do } \\
& \quad \quad \quad \vdots
\end{cases}
\end{align\*}
\\
\begin{align\*}
{
\begin{align\*}
\text{compute } m[i,i+3] \\
\underbrace{ \{ m[1,4],m[2,5], \dots ,m[n-3,n]\} }\_{(n-3) \text{ values}}
\end{align\*}
}
& \begin{cases}
& \ell=4 \\
& \text{for } i=1 \text{ to } n-3 \text{ do } \\
& \quad m[i,i+3]=\infty \\
& \quad \quad \text{for } k=i \text{ to } i+2 \text{ do } \\
& \quad \quad \quad \vdots
\end{cases}
\end{align\*}
$$

## **Table access pattern** in computing s for



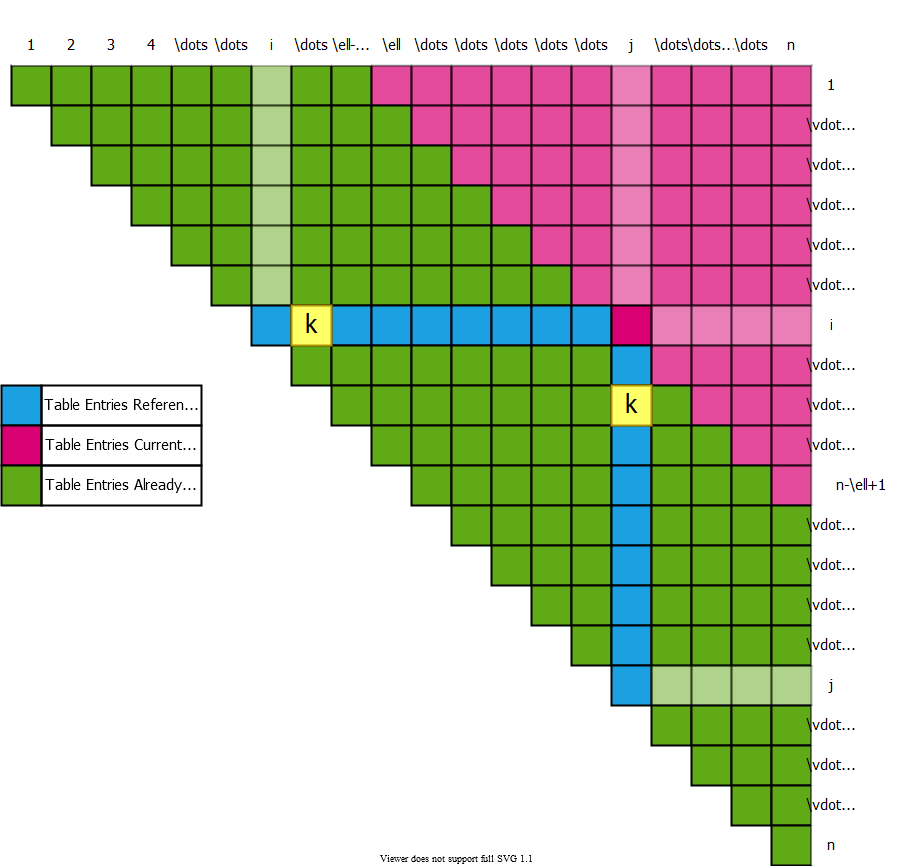
bg right:50% h:600px

## **Table access pattern** in computing s for



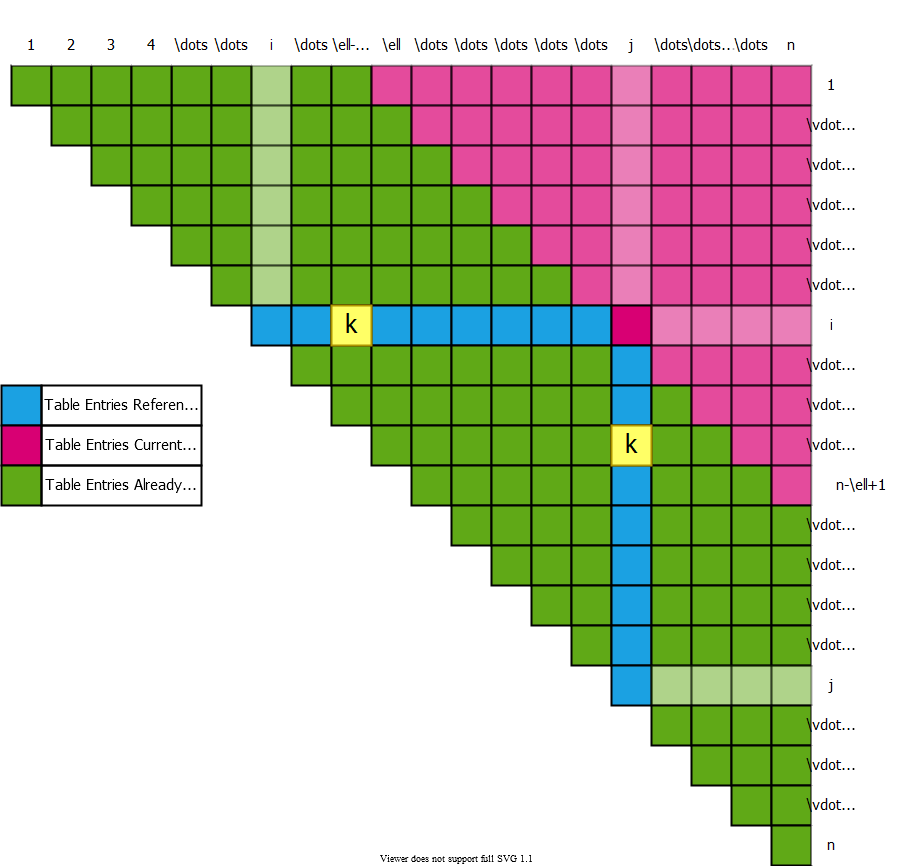
bg right:50% h:600px

## **Table access pattern** in computing s for



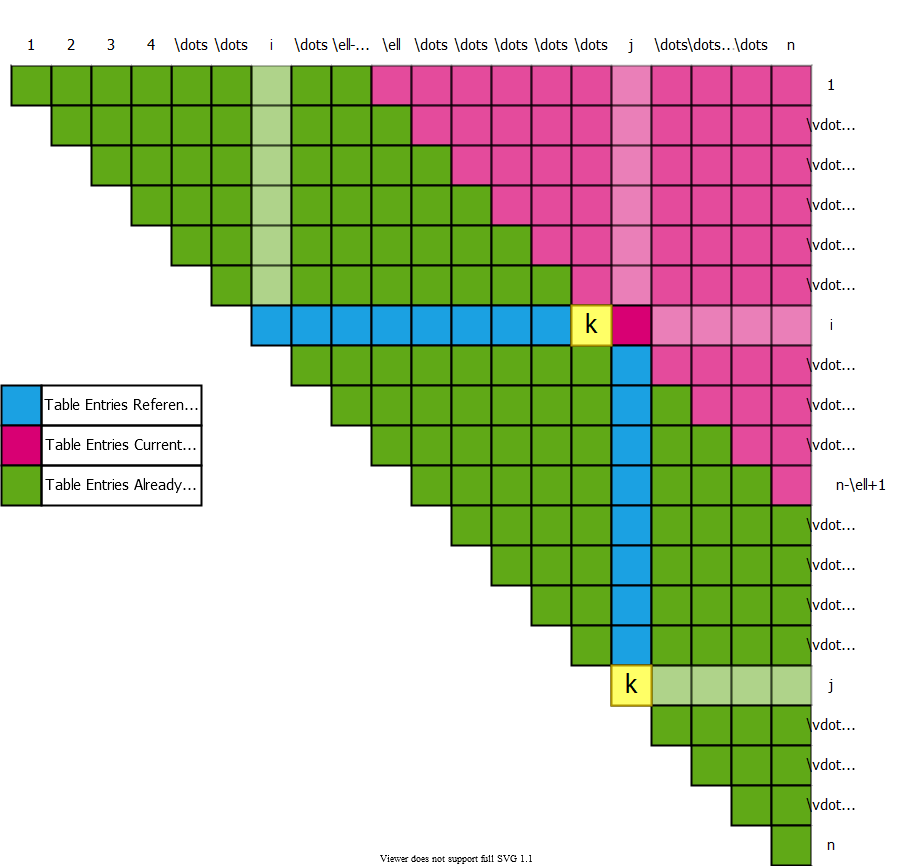
bg right:50% h:600px

## **Table access pattern** in computing s for



bg right:50% h:600px

## **Table access pattern** in computing s for



bg right:50% h:600px

## References