Dynamic Programming
part 2

Week 7 Objectives

- More dynamic programming examples
  - Matrix Multiplication Parenthesis
  - Longest Common Subsequence
- Subproblem Optimal structure
- Defining the dynamic recurrence
- Bottom up computation
- Tracing the solution

Subproblem Optimal Structure

- Divide and conquer - optimal subproblems
- divide PROBLEM into SUBPROBLEMS, solve SUBPROBLEMS
- combine results (conquer)
- critical/optimal structure: solution to the PROBLEM must include solutions to subproblems (or subproblem solutions must be combinable into the overall solution)
- PROBLEM = {DECISION/MERGING + SUBPROBLEMS}

Optimal Structure - NON GREEDY

- Cannot make a choice decision/CHOICE without solving subproblems first
- Might have to solve many subproblems before deciding which results to merge.

Matrix Multiplication (Parenthesis)

- Task: multiply matrices $A_1*A_2*...*A_n$
- Ai matrix has p_{i-1} rows and p_i columns (size p_{i-1} X p_i)
  - #rows of matrix A_{i+1} has to be the same as #columns of A_i
- Minimize the number of scalar multiplications
- Note that matrices can be multiplied in any order:
  - $A_1*(A_2*A_3)*A_4$; $(A_1*A_2)*(A_3*A_4)$; $A_1*(A_2*A_3*A_4)$
  - A_1(size $p_0 x p_1$) * A_2(size $p_1 x p_2$) takes $p_0*p_1*p_2$ scalar multiplications
  - order matters, example: $A_1(10x100)$, $A_2(100x5)$; $A_3(5x50)$ (p_0= 10; p_1=100; p_2=5; p_3=50)
    - then $A_1*(A_2*A_3)$ takes 75000 scalar multiplications
    - while $(A_1*A_2)*A_3$ takes 7500 scalar multip., 10 times less.

Matrix Multiplication (Parenthesis)

- NAIVE SOLUTION: try all ways to put parenthesis to see which one is best/minimum
  - $A_1*((A_2*A_3)*A4)$; $(A_1*A_2)*(A_3*A_4)$; $A_1*(A_2*(A_3*A4))$
  - $((A_1*A_2)*A_3)*A_4$; $(A_1*(A_2*A_3))*A_4$
- P(n) = number of ways to parenthesize n matrices
- recursion on n

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- why? proof this recursion
- show that this P(n) is exponential in n

Matrix Multiplication (Parenthesis)

- 1) characterize optimal solution structure
- optimal solution SOL parenthesis has a "main split", or "last product" - that is the last matrix multiplication
  - say it is between matrices A_k and A_{k+1}

$$\overbrace{((A_i A_{i+1} \ldots A_k)}^{\text{prefix subchain}}\overbrace{(A_{k+1} A_{k+2} \ldots A_j))}^{\text{suffix subchain}}$$

- then SOL parenthesis on the left side (A_i*...*A_k) must be optimal
- same for right side: parenthesis on (A_{k+1}*...*A_j) must be optimal
  - why? use an exchange argument

Matrix Multiplication (Parenthesis)

- 2) dynamic programming recursion
- $C[i,j]$ = min scalar multip. to multiply $A_i * A_{i+1} * \ldots * A_j$
  - $C[i,i]=0$; $C[i,i+1] = p_{i-1} * p_i * p_{i+1}$
- $A_i * A_{i+1} * \ldots * A_j$ can be computed by first deciding the main split at some k, $1<k<j$
  - for that split C[i,j] = C[i,k] + C[k+1,j] + pi-1*pk*pj

$C[i,k]$ $\quad p_{i-1} * p_k * p_j \quad$ $C[k+1,j]$

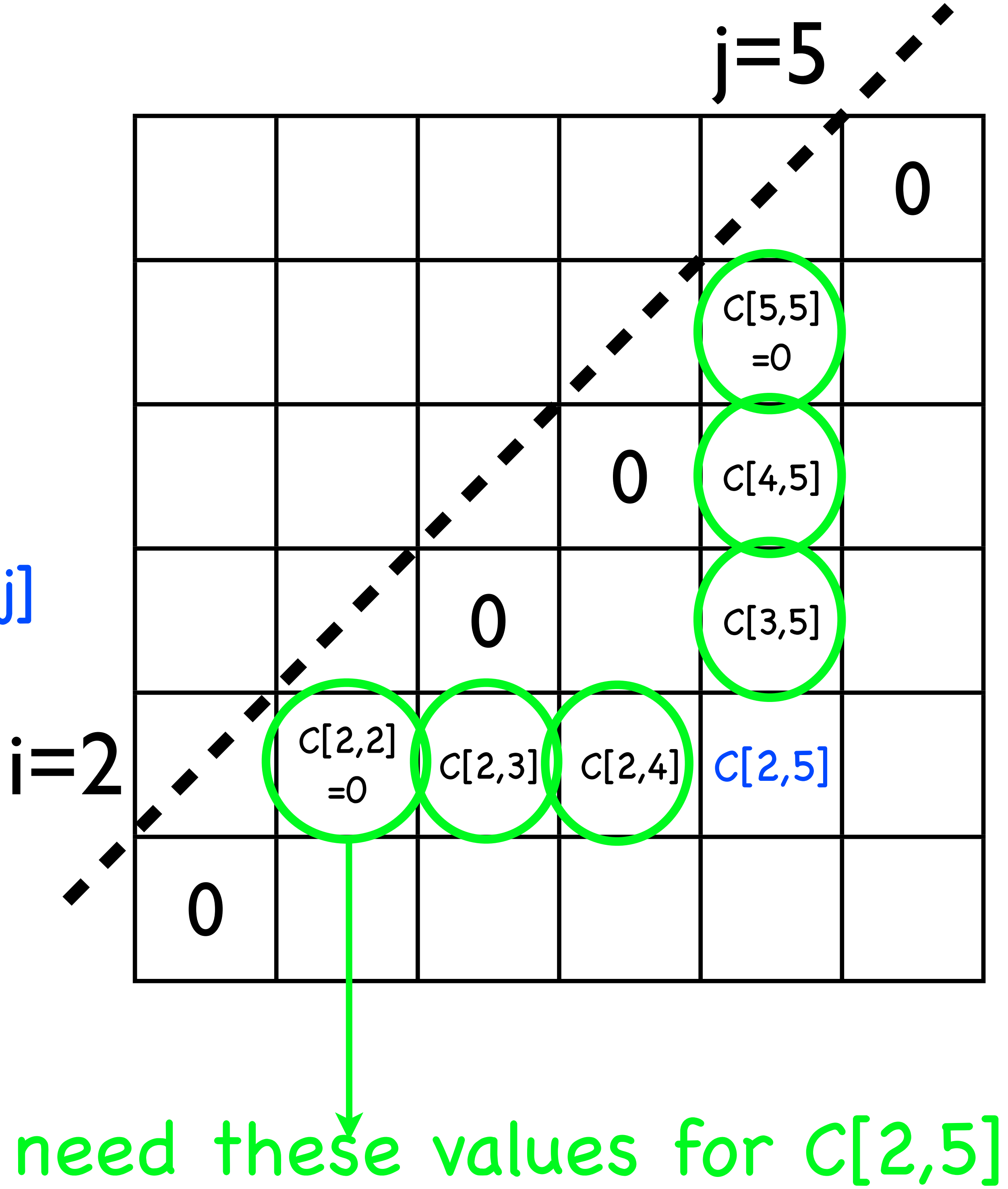
$$((A_i A_{i+1} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j))$$

  - but we dont know what k is best, so we have to try all of them

$$C[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{C[i,k] + C[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

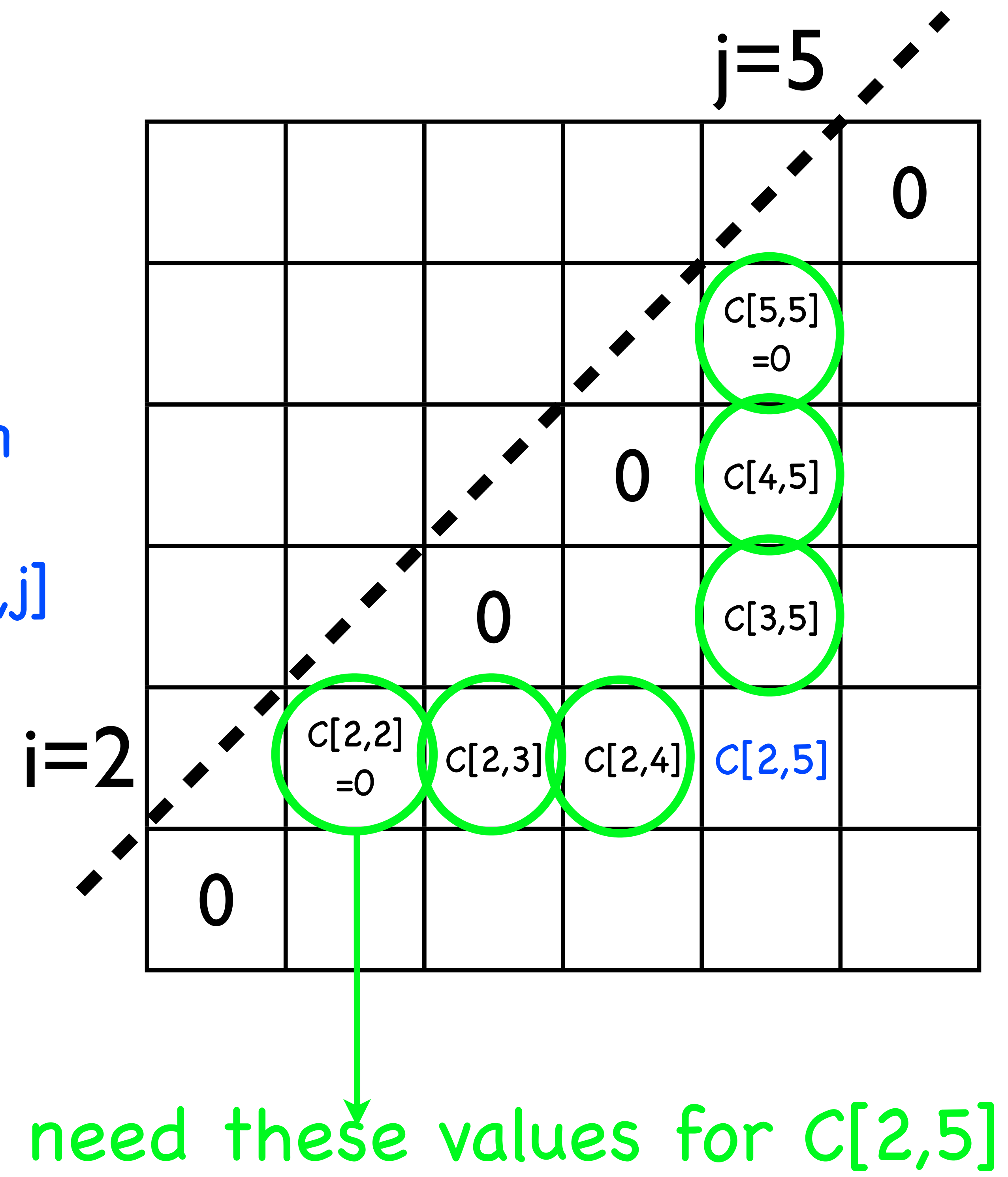
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]



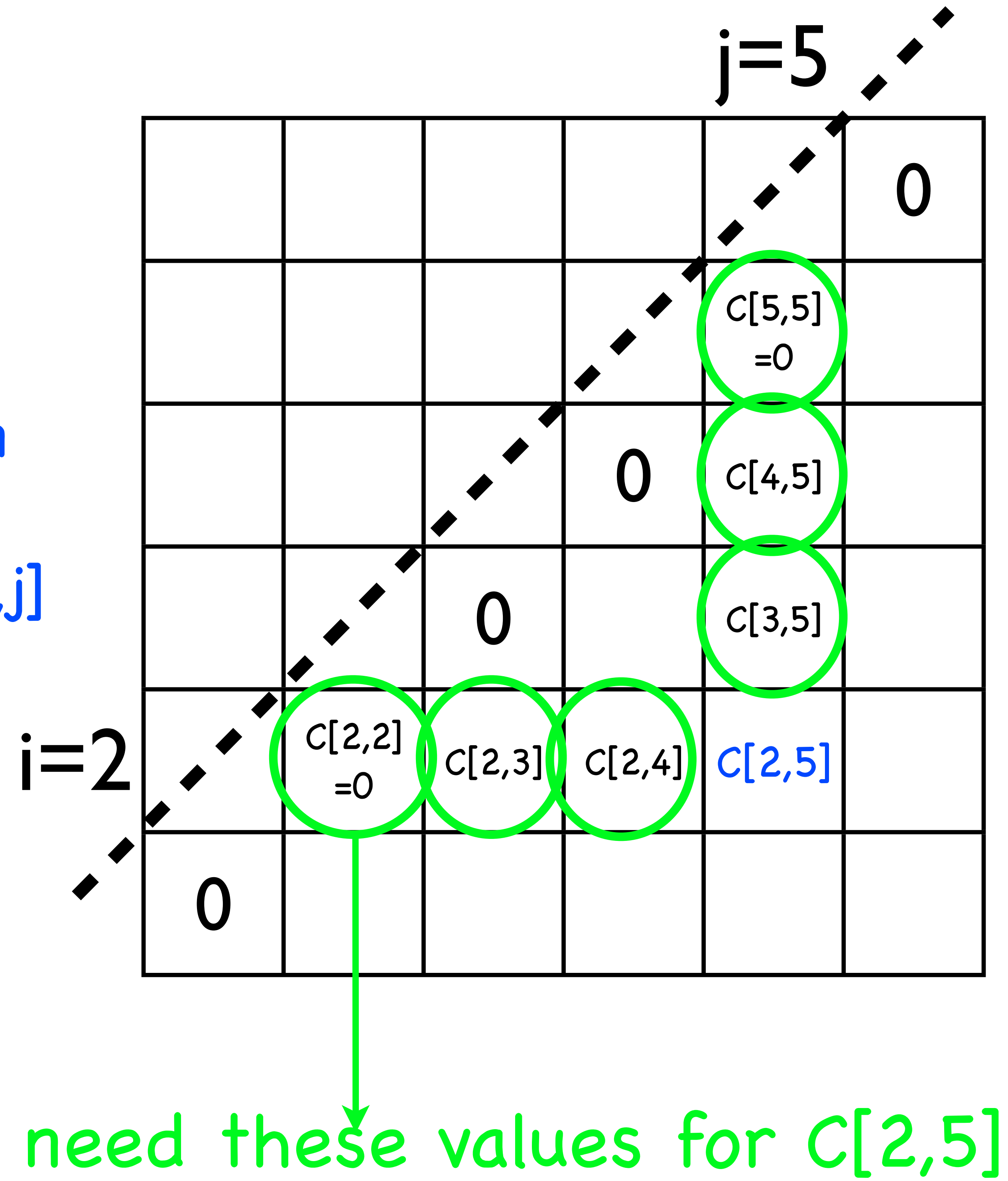
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k



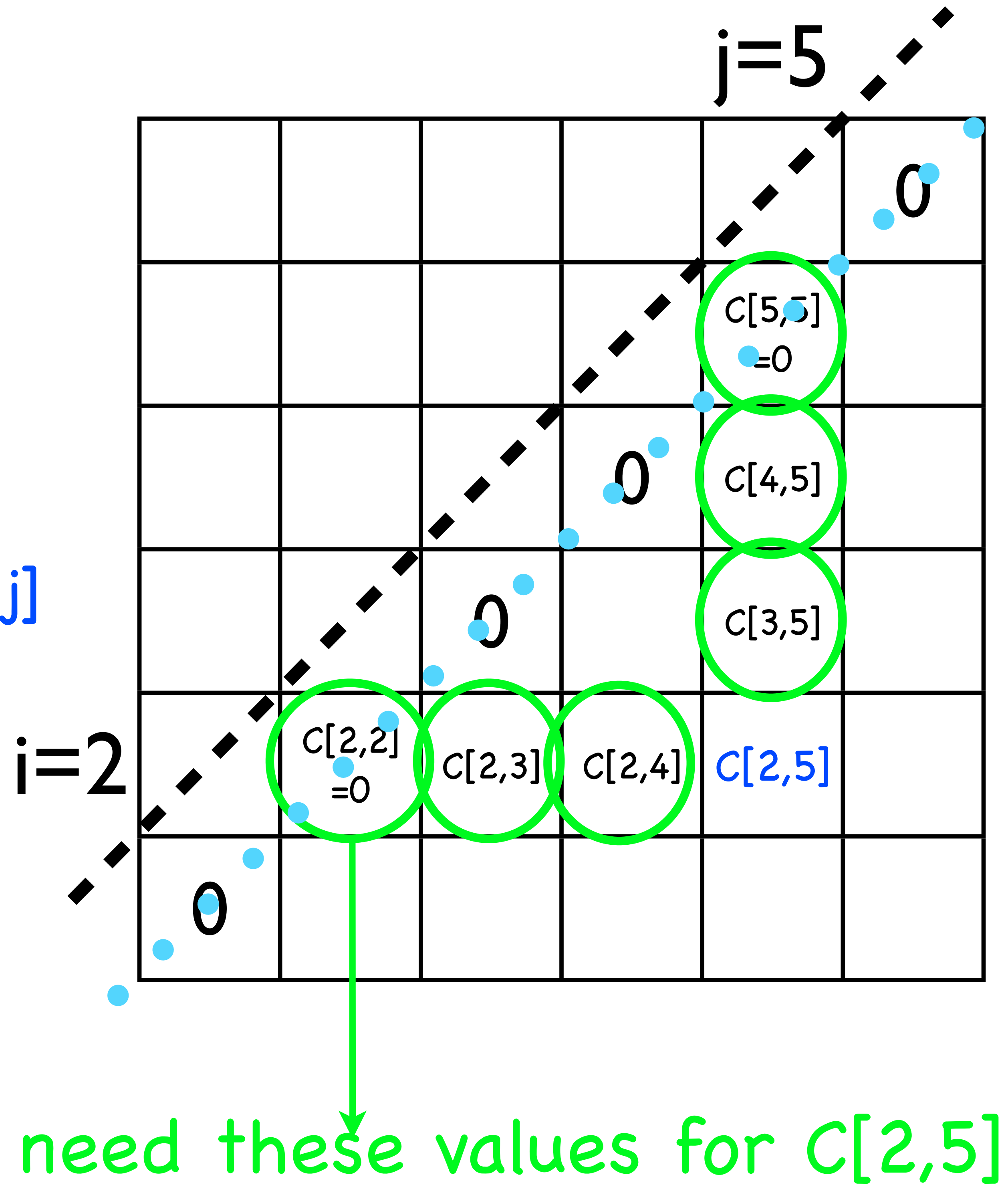
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



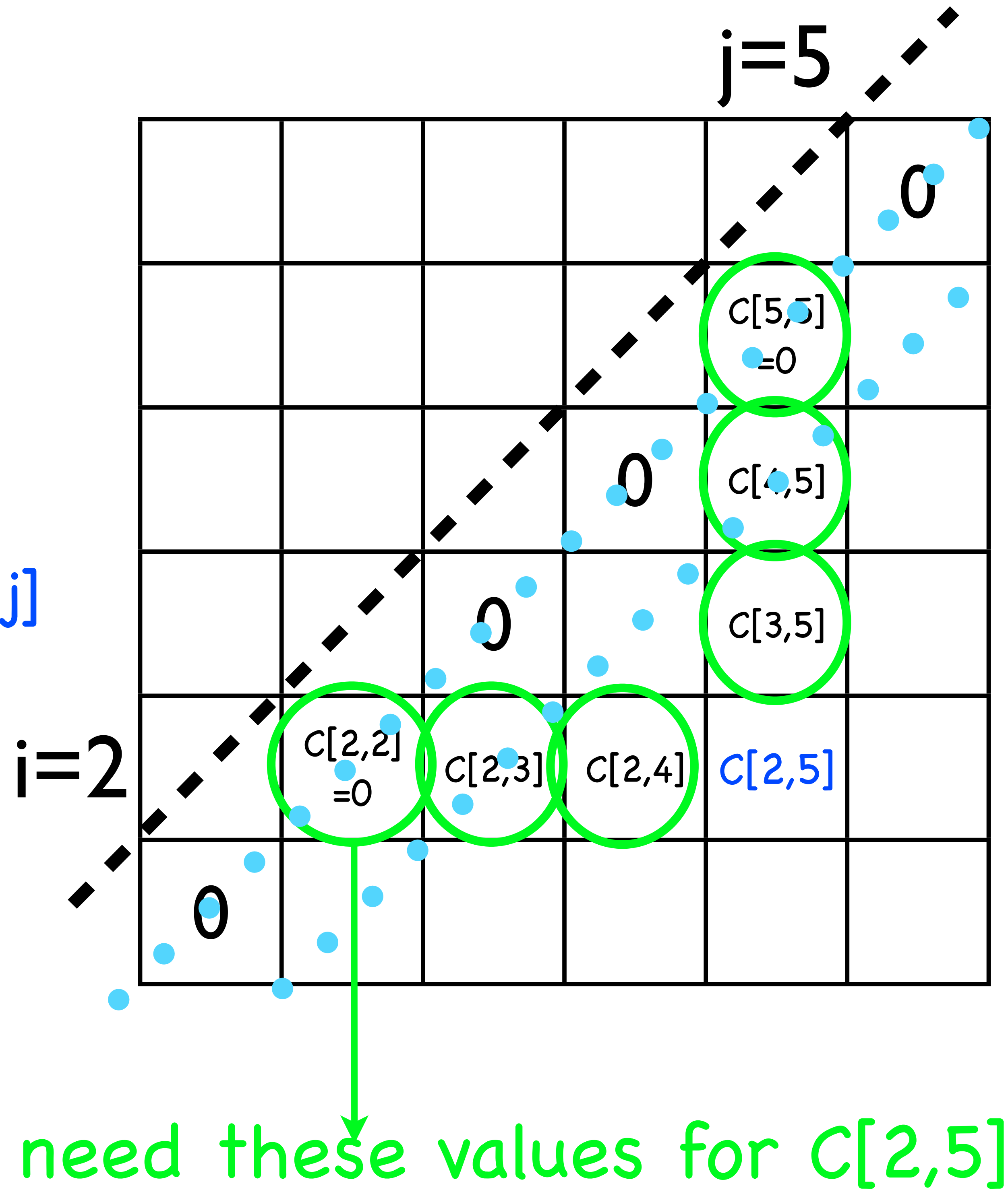
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



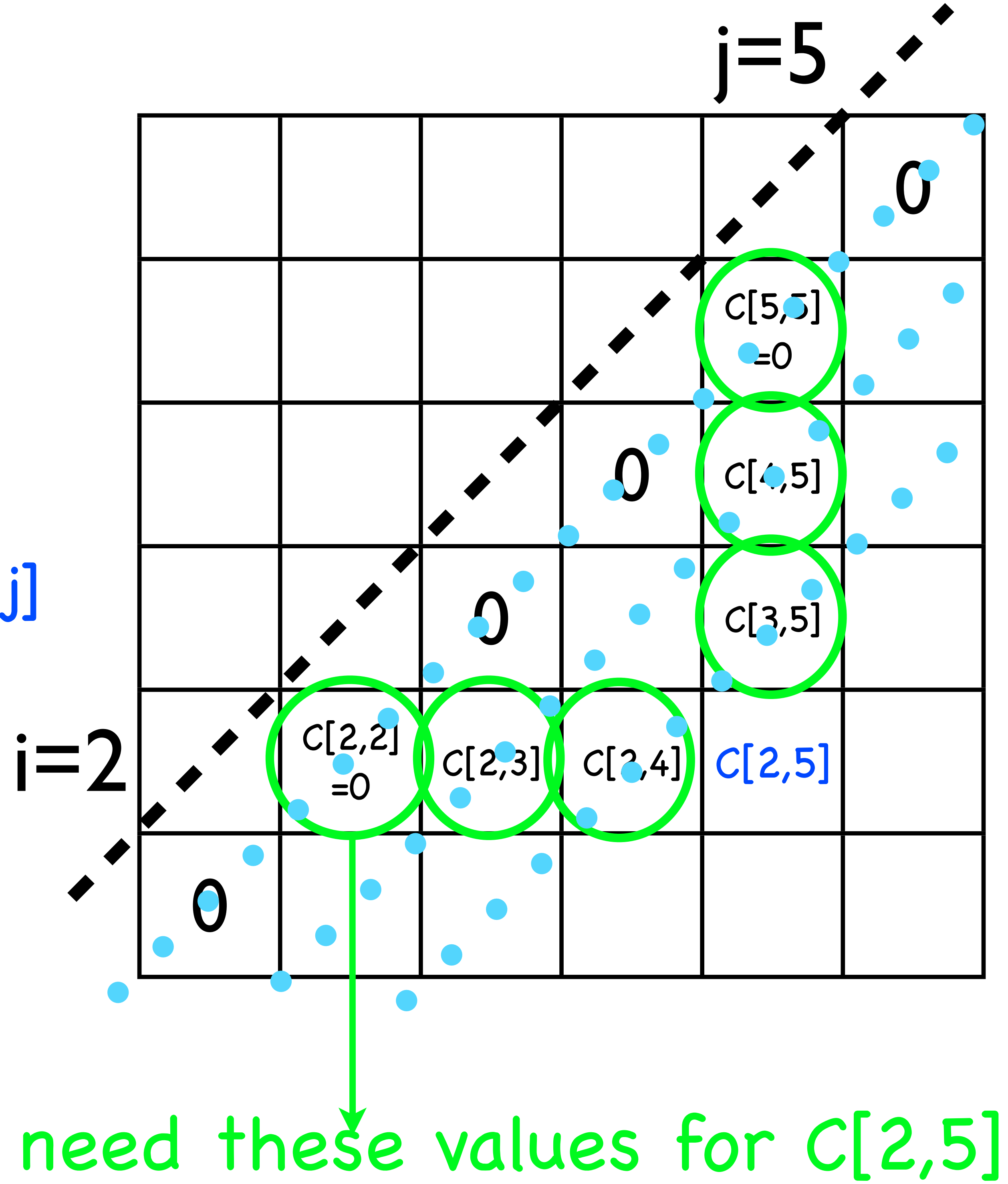
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



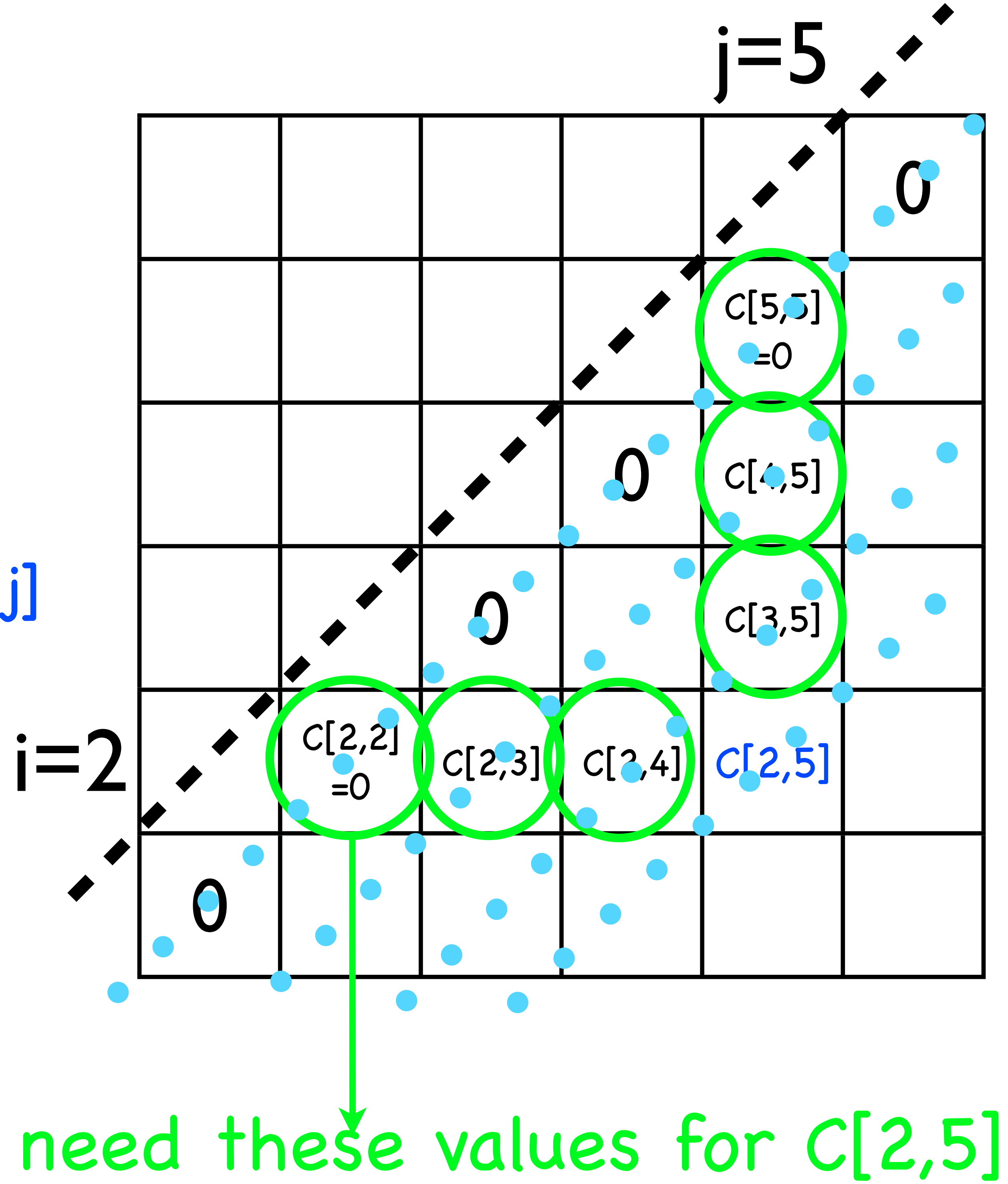
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



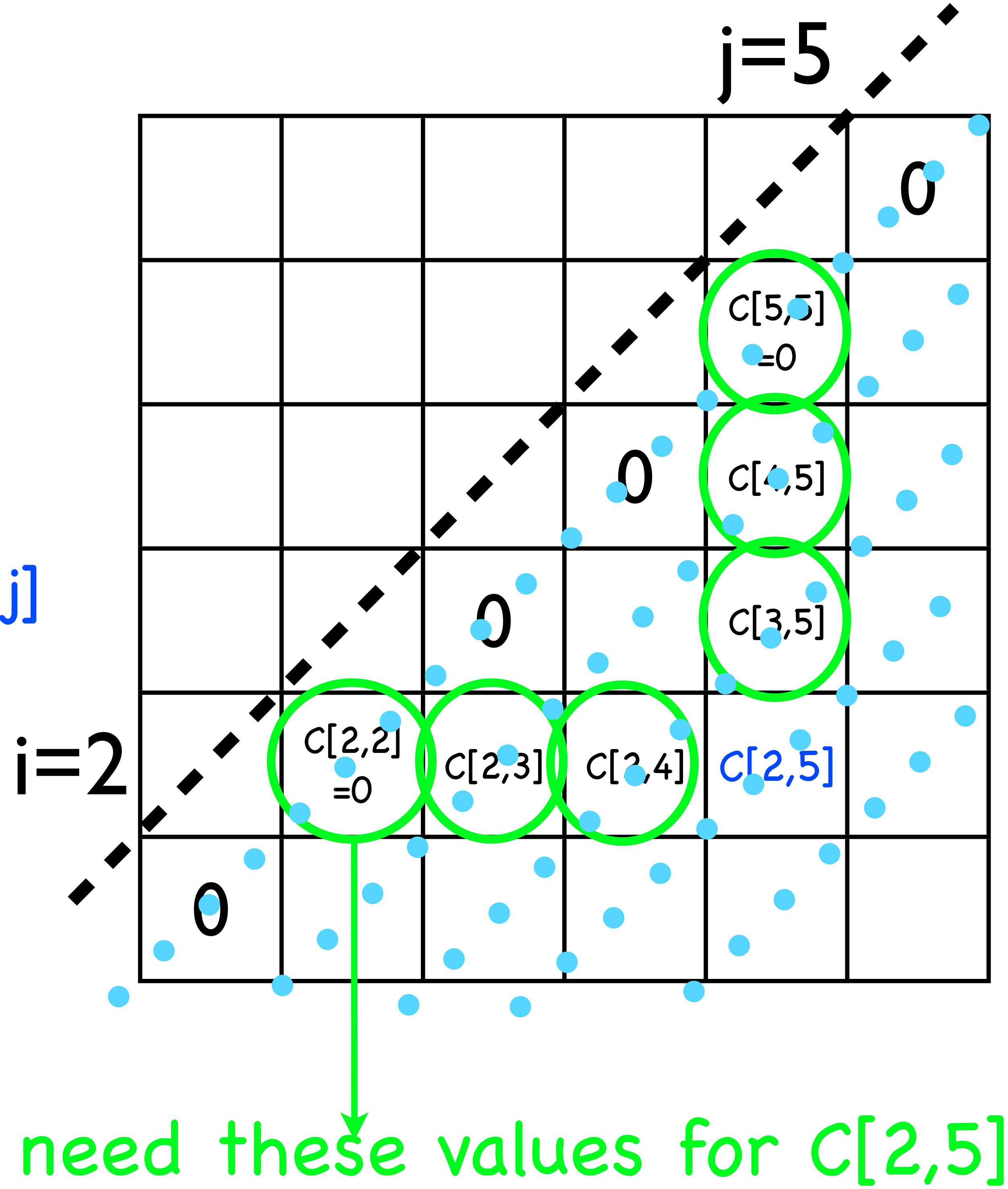
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



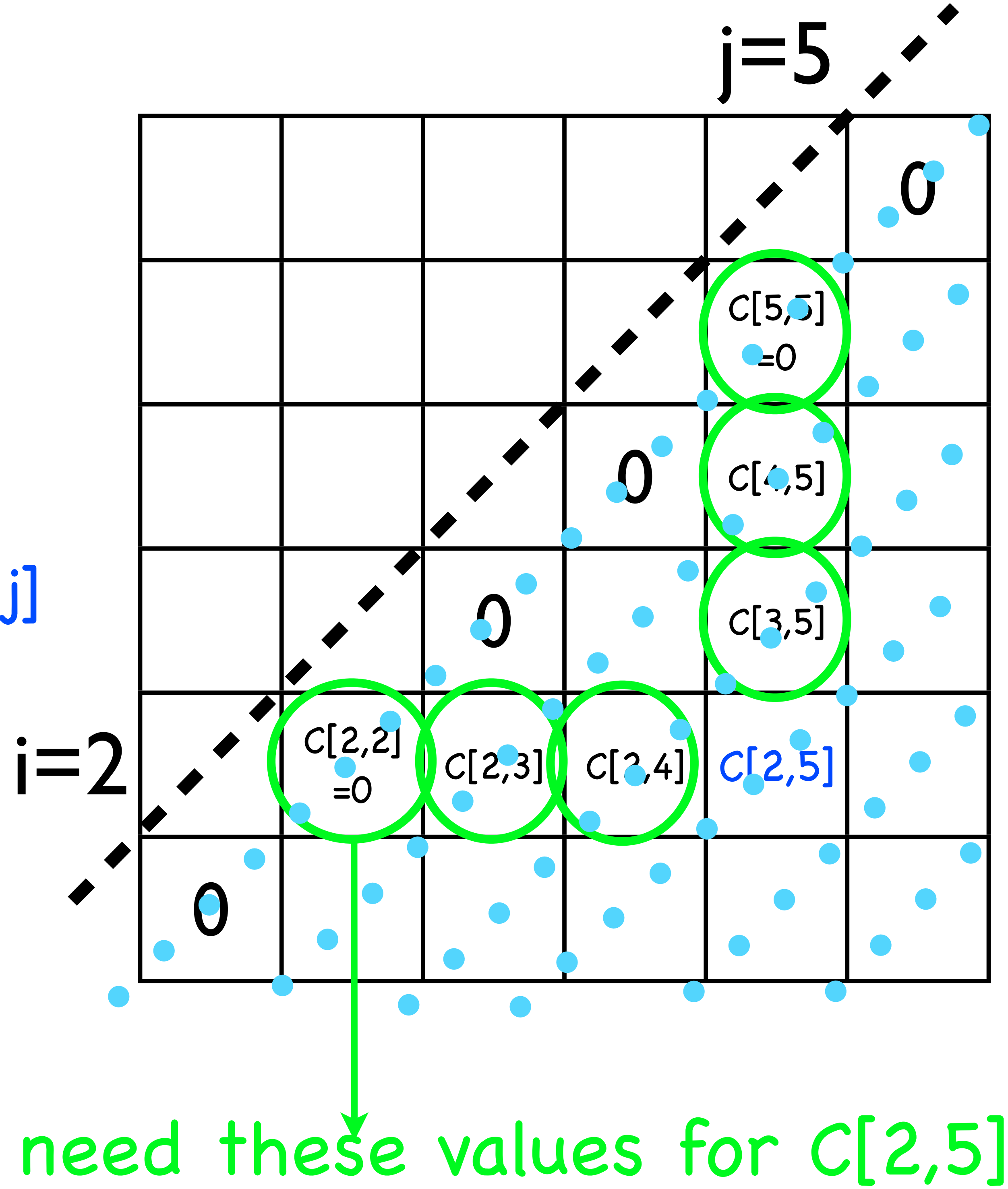
Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



Matrix Multiplication (Parenthesis)

- 3) bottom up computation of table C[]
  - what is the right order to fill the table?
  - guarantee that values needed for recursion are already computed when we compute C[i,j]
  - might need any value C[i,k] and C[k+1,j]
- note length(i,j)=j-i
  - when computing C[i,j], length=j-i
  - values needed C[i,k] and C[k+1,j] have smaller lengths for any k
- fill table C[] by length
  - from cells with small length (main diagonal) to cells of high lengths (corners)



Matrix Multiplication (Parenthesis)

- 3) Bottom-up computation of C[]
  - by diagonal from short length, to long length
- keep track of split at k, for sequence [i...j]: S[i,j]=k
  - $A_i*A_2*...A_j$ multiplied best as $(A_i*A_{i+1}*...*A_k)(A_{k+1}*...*A_j)$

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let C[1..n, 1..n] and S[1..n − 1, 2..n] be new tables
for i = 1 to n
    C[i, i] = 0
for l = 2 to n //l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        C[i, j] = 0
        for k = i to j − 1
            q = C[i, k] + C[k + 1, j] + p_{i−1} p_k p_j
            if q < C[i, j]
                C[i, j] = q
                S[i, j] = k
return C and S
```

Matrix Multiplication (Parenthesis)

- 4) Trace the solution - Exercise
  - use S[i,j] to determine the main split
  - run recursion on both sides of the split
- also calculate the running time of the trace

Matrix Multiplication (Parenthesis)

- Running time
  - C[] table fills about 1/2 * n * n cells – $\Theta(n^2)$ cells
  - each cell C[i,j] tries all k ; $1 \le k < j$ – $\Theta(n)$ steps
- Total $\Theta(n^3)$ time for bottom up computation
- Trace solution: certainly lower than $\Theta(n^3)$, so it doesnt add to the running time asymptote.

Top-down computation instead of bottom up

- Suppose we want to do the computation top down
- Recursively follow the recursion

```
Rec-Matrix-Chain(p,i,j)//bad running time
    if(i==j) return 0;
    m[i,j]=∞
    for k=i:j-1
        q=Rec-Matrix-Chain(p,i,k) + Rec-Matrix-Chain(p,k+1,j) + p_{i-1}p_kp_j;
        if (q<m[i,j]) m[i,j]=q;
    return m[i,j]
```

- Exponential number of calls VS bottom up which is only $\Theta(n^2)$ for this section of the code

Top-down with memoization

- memoization: "store, dont recompute" the computed results; each actual computation only happen once
- init all m[i,j]=∞; call MEMOIZATION-top-down(p,1,n)

```
MEMOIZATION-top-down(p,i,j)
    if (m[i,j]<∞) return m[i,j] // look up previous computed values
    if(i==j) m[i,j] = 0;
    else for k=i:j-1
        q=Rec-Matrix-Chain(p,i,k) + Rec-Matrix-Chain(p,k+1,j) + p_{i-1}p_k p_j;
        if (q<m[i,j]) m[i,j]=q; //store value for future look up
    return m[i,j]
```

Memoization

- now same running time as bottom-up : $\Theta(n^3)$ overall
- bottom-up (DP) VS top-down (Memoization):
  - DP advantage: no overhead (stack of calls, recursion), efficient when the whole table has to be computed anyway
  - DP requires a certain fill-order of the table
  - Memoization: better when not all values must be computed
  - Memoization follow literally the recursionl; easier to implement

Longest Common Subsequence
(LCS)

Longest Common Subsequence

- Given two $X=(x_1, x_2, \ldots, x_m)$ and $Y=(y_1, y_2, \ldots, y_n)$ find the longest common subsequence
  - it doesnt have to be continuos in either X or Y
  - not unique: possible that several common sequences have maximum length
- example
  - X=(absscddegt) Y=(xasbsdcggg)
  - LCS=Z=(absdg)

Longest Common Subsequence

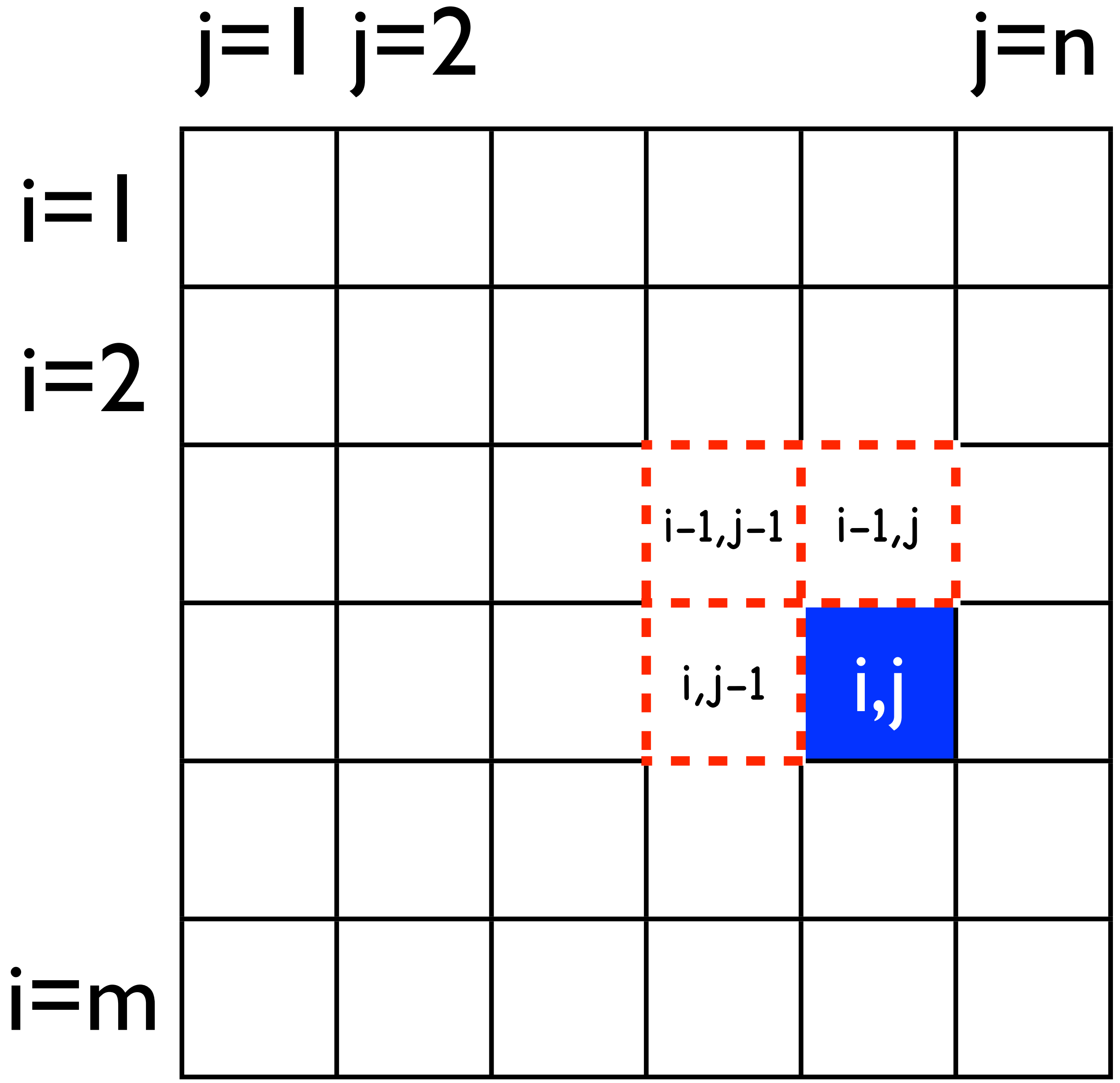
- 1) Characterize optimal solution structure - (add general army- needs more cannons story)
  - notation: $X_{m-1} = (x_1, x_2, \ldots, x_{m-1})$; $Y_{n-1} = (y_1, y_2, \ldots, y_{n-1})$ etc
- if $X=(x_1, x_2, \ldots, x_m)$ and $Y=(y_1, y_2, \ldots, y_n)$ have an LCS $Z=(z_1, z_2, \ldots, z_k)$ then
  - if $x_m=y_n$; then $z_k=x_m=y_n$ and $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$
  - if $x_m \neq y_n$ and $z_k \neq x_m$ then $Z=LCS(X_{m-1}, Y)$
  - if $x_m \neq y_n$ and $z_k \neq y_n$ then $Z=LCS(X_m, Y_{n-1})$

Longest Common Subsequence

- 2) dynamic recursion
- $C[i,j]$ = LCS (X_i, Y_j) where $X_i=(x_1,x_2,\ldots x_i)$ $Y_j=(y_1,y_2,\ldots y_j)$
- $C[i,j]$ is
  - 0 ; for base case i=0 or j=0
  - $C[i-1,j-1]+1$; for i,j>0 and xi=yj
  - max $\{C[i-1,j], C[i,j-1]\}$; for i,j>0 and xi≠yj

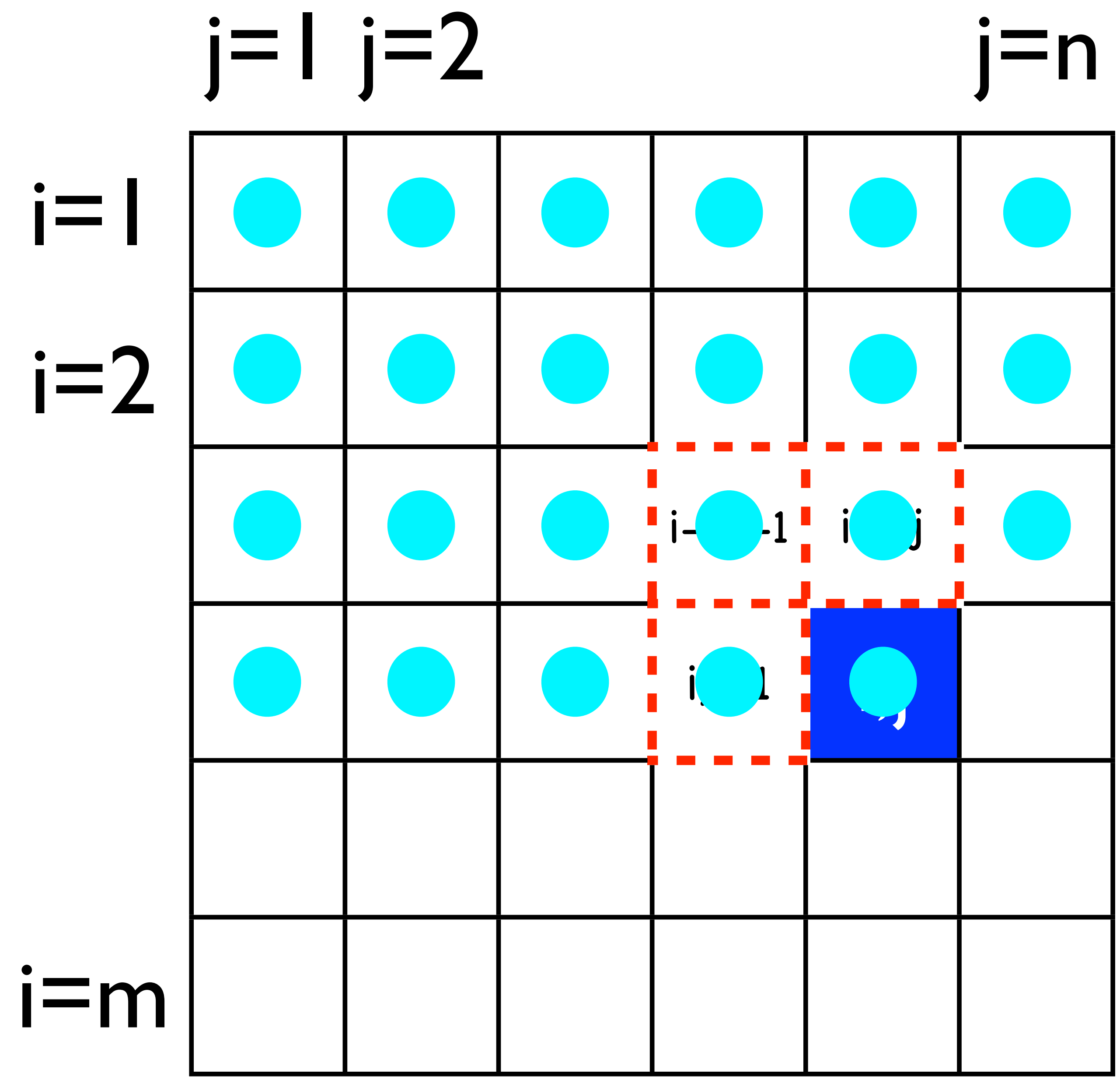
Longest Common Subsequence

- 3) bottom up computation
- in order to compute C[i,j] we need to have already computed the following three values:
  - C[i−1,j−1]
  - C[i,j−1]
  - C[i−1,j]



Longest Common Subsequence

- 3) bottom up computation
- in order to compute C[i,j] we need to have already computed the following three values:
  - C[i-1,j-1]
  - C[i,j-1]
  - C[i-1,j]
- fill row by row, each row from left to right



Longest Common Subsequence

- 3) bottom up computation
- keep track of the solution: S[i,j] remembers which one of the three possibilities we used:
  - C[i-1,j-1] + 1 ; S[i,j] ="↖"
  - C[i,j-1] ; S[i,j] =" ↑ " ;
  - C[i-1,j] ; S[i,j]="←"

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let S[1..m, 1..n] and C[0..m, 0..n] be
for i = 1 to m
    C[i, 0] = 0
for j = 0 to n
    C[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            C[i, j] = C[i − 1, j − 1] + 1
            S[i, j] = "↖"
        elseif C[i − 1, j] ≥ C[i, j − 1]
            C[i, j] = C[i − 1, j]
            S[i, j] = "↑"
        else C[i, j] = C[i, j − 1]
            S[i, j] = "←"
return C and S
```

Longest Common Subsequence

- 3) bottom up computation
  - illustrated are C[] and S[] tables on the same grid
  - C[i,j] is the size of LCS(X_i,Y_j)
- S[i,j] is the arrow pointing to the subproblem
  - "↖" indicates a common item, part of LCS; subproblem decreases both i and j
  - " ↑ " indicates discarding last vale of X_i; decrease i
  - "←" indicates discarding last value of Y_j; decrease j

j		0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest Common Subsequence

- 4) trace solution
- start at S[m,n], follow arrows:
- every "↖" r means a common item is found by LCS

j		0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
PRINT-LCS(S, X, i, j)
  if i == 0 or j == 0
      return
  if S[i, j] == "↖"
      PRINT-LCS(S, X, i − 1, j − 1)
      print x_i
  elseif S[i, j] == "↑"
      PRINT-LCS(S, X, i − 1, j)
  else PRINT-LCS(S, X, i, j − 1)
```

Longest Common Subsequence

- Running time
  - bottom up computation fills a table of m x n cells
  - each cell takes constant time
- overall $\Theta(mn)$
- Trace solution $O(m+n)$
  - we "walk" on the table towards the [0,0] cell either vertical or horizontal or diagonal.