Dynamic Programming

- Matrix chain products
- Dynamic programming
- Longest common subsequence

- Matrix chain products
- Dynamic programming
- Longest common subsequence

Matrix multiplication

Dynamic Programming is a general algorithm design paradigm.

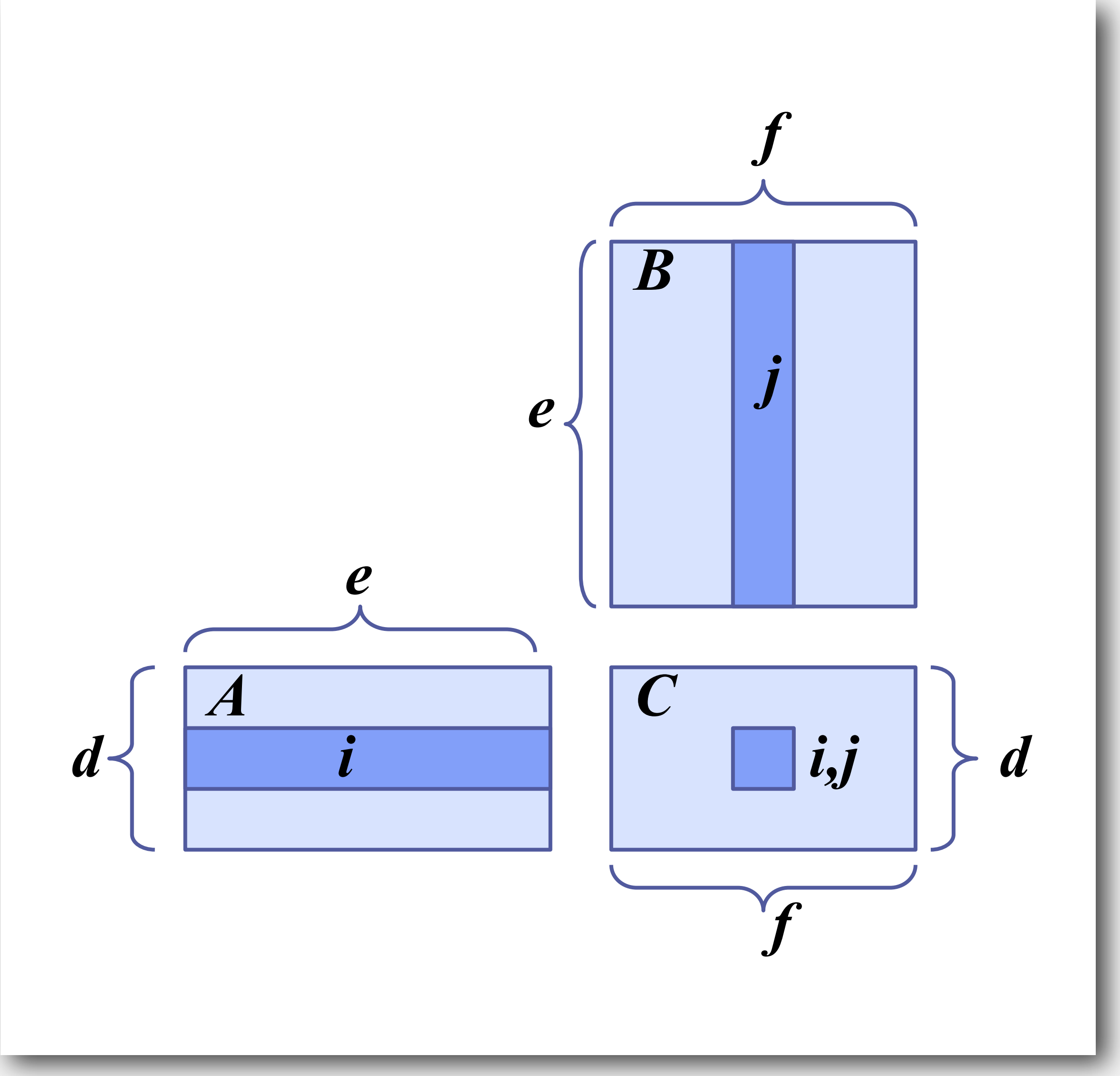
- Rather than start with the general structure, we first consider a motivating example: matrix-chain products

Review: Matrix Multiplication.

- C = A*B
- A is d × e and B is e × f

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

- O(def) time



Matrix-chain products

Compute $A = A_0 * A_1 * \ldots * A_{n-1}$.

- A_i is $d_i \times d_{i+1}$
- Problem: How to order operations (i.e., parenthesize)?

Example

- B is 3 × 100
- C is 100 × 5
- D is 5 × 5
- (B*C)*D takes 1500 + 75 = 1575 ops
- B*(C*D) takes 1500 + 2500 = 4000 ops

An enumeration approach

Matrix Chain-Product Algorithm

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of operations for each one
- Pick the one that is best

Running time:

- The number of parenthesizations is equal to the number of binary trees with n nodes
- This is exponential! It is called the Catalan number, and it is almost 4^n.
- ...terrible choice of algorithm

A greedy approach

Idea #1: repeatedly select the product that uses (up) the most operations.

Counter-example:

- A is 10×5
- B is 5×10
- C is 10×5
- D is 5×10

Greedy idea #1 selects (A*B)*(C*D)... which takes 500+1000+500 = 2000 ops

But A*((B*C)*D) takes 500+250+250 = 1000 ops

Another greedy approach

Idea #2: repeatedly select the product that uses the fewest operations.

Counter-example:

- A is 101 × 11
- B is 11 × 9
- C is 9 × 100
- D is 100 × 99

Greedy idea #2 selects A*((B*C)*D))...

- which takes 109989+9900+108900=228789 ops

But (A*B)*(C*D) takes 9999+89991+89100=189090 ops

...The greedy approach is not giving us the optimal value.

- Matrix chain products
- **Dynamic programming**
- Longest common subsequence

A recursive approach

Define subproblems:

- Find the best parenthesization of $A_i * A_{i+1} * \ldots * A_j$.
- Let $N_{i,j}$ denote the number of operations done by this subproblem.
- The optimal solution for the whole problem is $N_{0,n-1}$.

Subproblem optimality: The optimal solution can be defined in terms of optimal subproblems

- There has to be a final multiplication (root of the expression tree) for the optimal solution.
- Say, the final multiply is at index i: $(A_0 * \ldots * A_i) * (A_{i+1} * \ldots * A_{n-1})$.
- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
- If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution

Characterizing equation

The global optimal has to be defined in terms of optimal subproblems, depending on the location of the final multiply.

Let us consider all possible places for that final multiply:

Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.

So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

Note that subproblems are not independent--the subproblems overlap.

A dynamic programming algorithm

Since subproblems overlap, we won't use recursion.

Instead, we construct optimal subproblems "bottom-up."

- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.

The running time is $O(n^3)$

Algorithm *matrixChain*(*S*):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

```
for i ← 1 to n-1 do
    N_{i,i} ← 0
for b ← 1 to n-1 do
    for i ← 0 to n-b-1 do
        j ← i+b
        N_{i,j} ← +infinity
        for k ← i to j-1 do
            N_{i,j} ← min{N_{i,j} , N_{i,k} +N_{k+1,j} +d_i d_{k+1} d_{j+1}}
```

Algorithm visualization

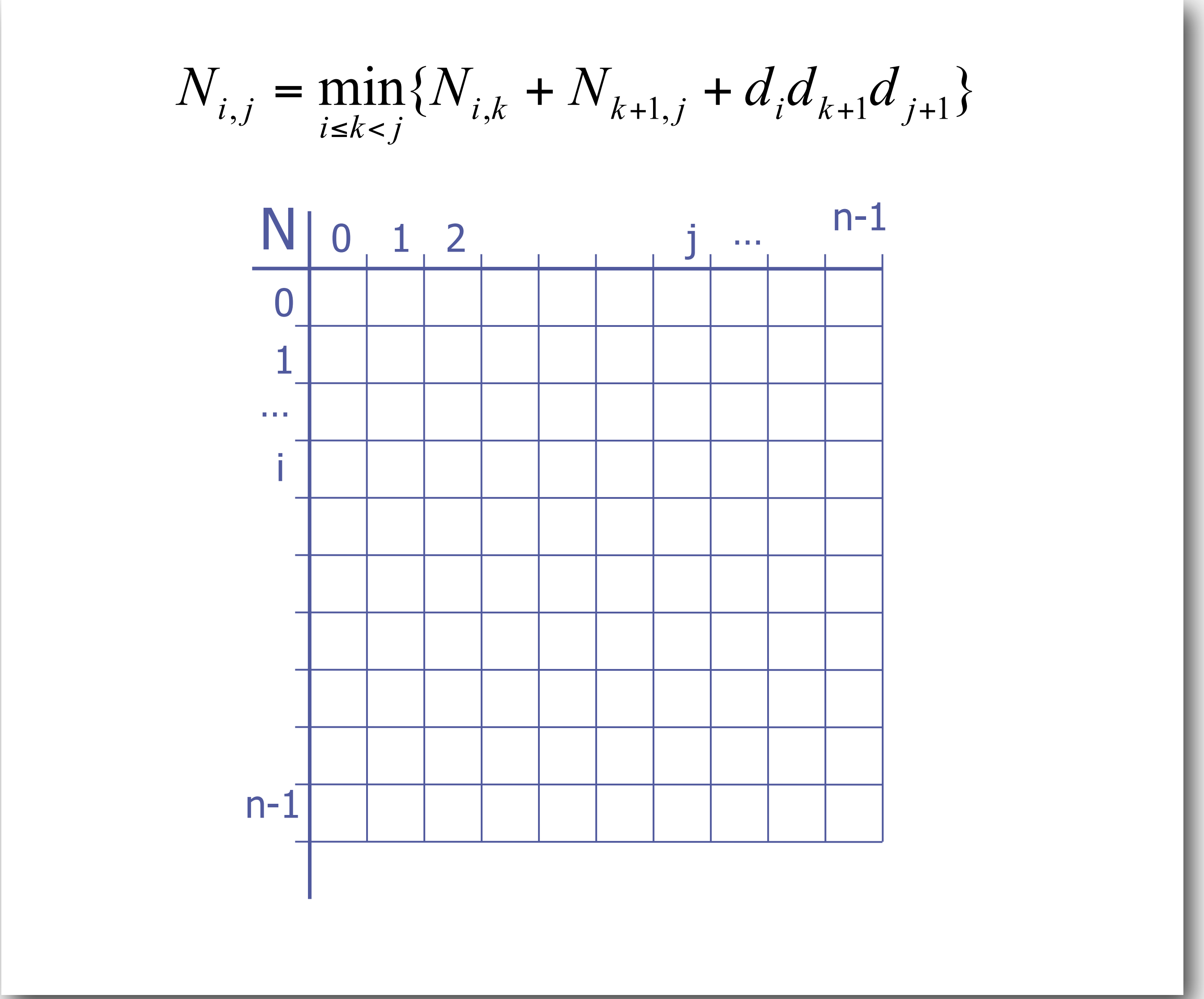
The bottom-up construction fills in the N array by diagonals

$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering "k" for each N entry



Algorithm visualization

The bottom-up construction fills in the N array by diagonals

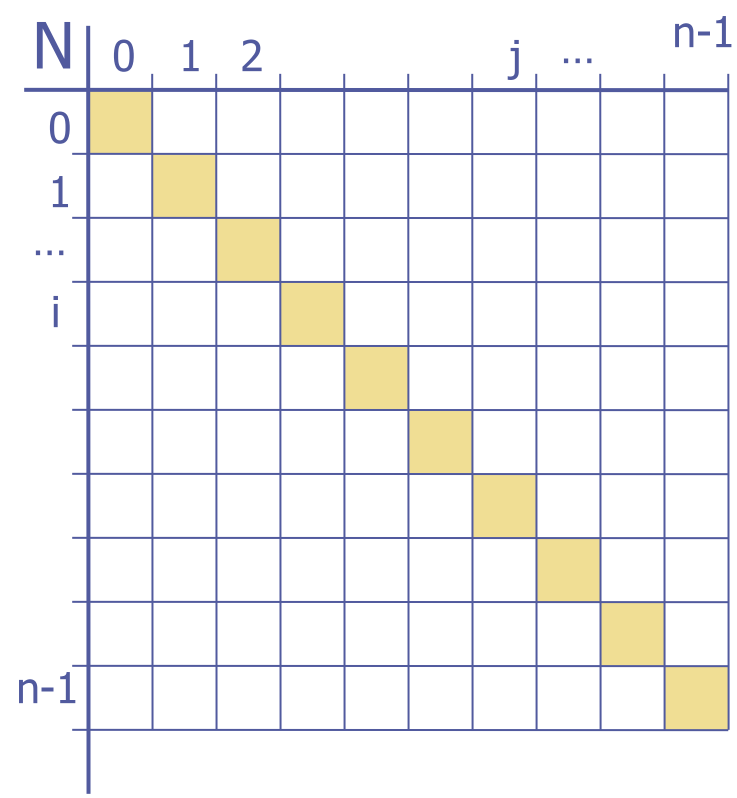
$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



Algorithm visualization

The bottom-up construction fills in the N array by diagonals

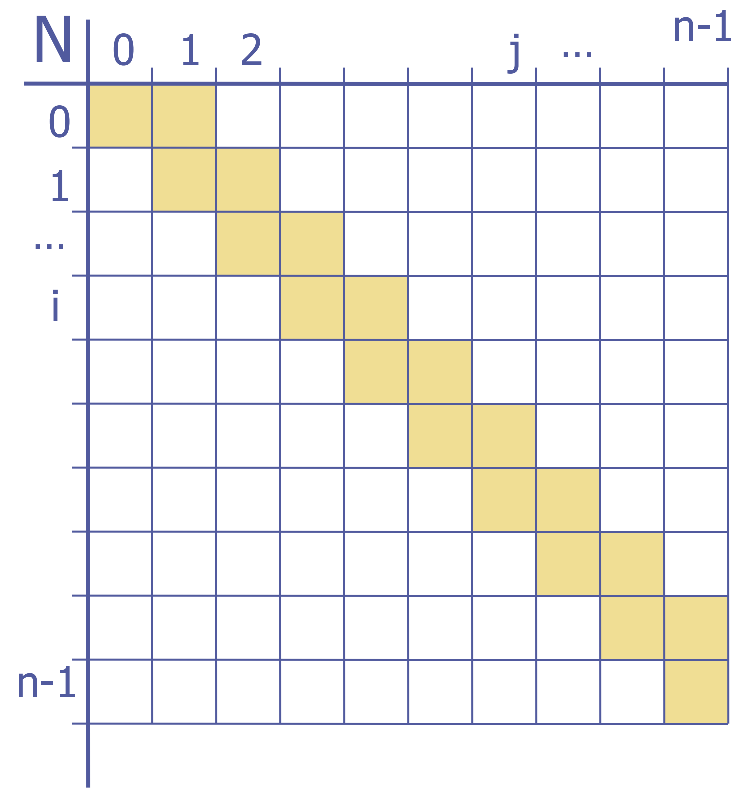
$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering “k” for each N entry

$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



Algorithm visualization

The bottom-up construction fills in the N array by diagonals

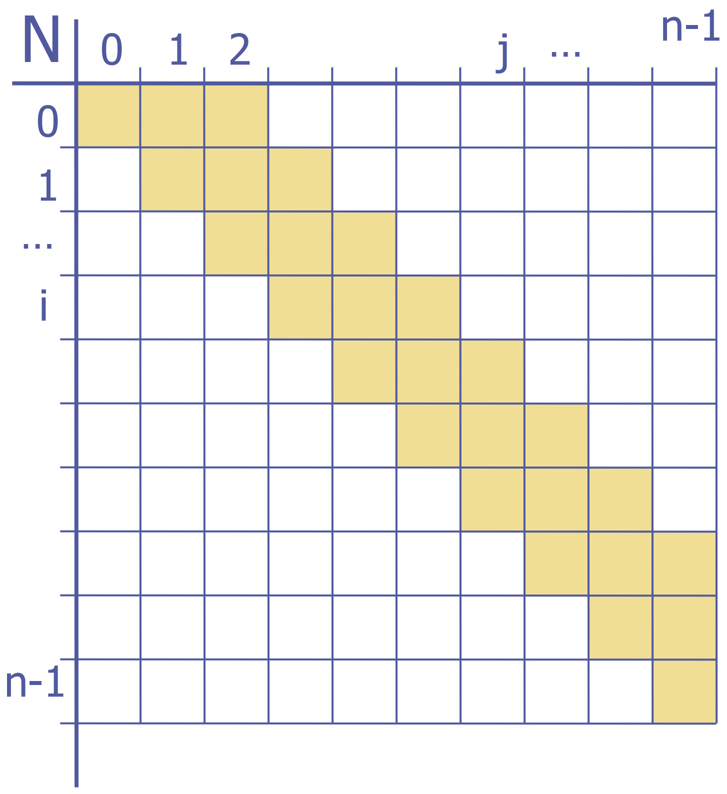
$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



Algorithm visualization

The bottom-up construction fills in the N array by diagonals

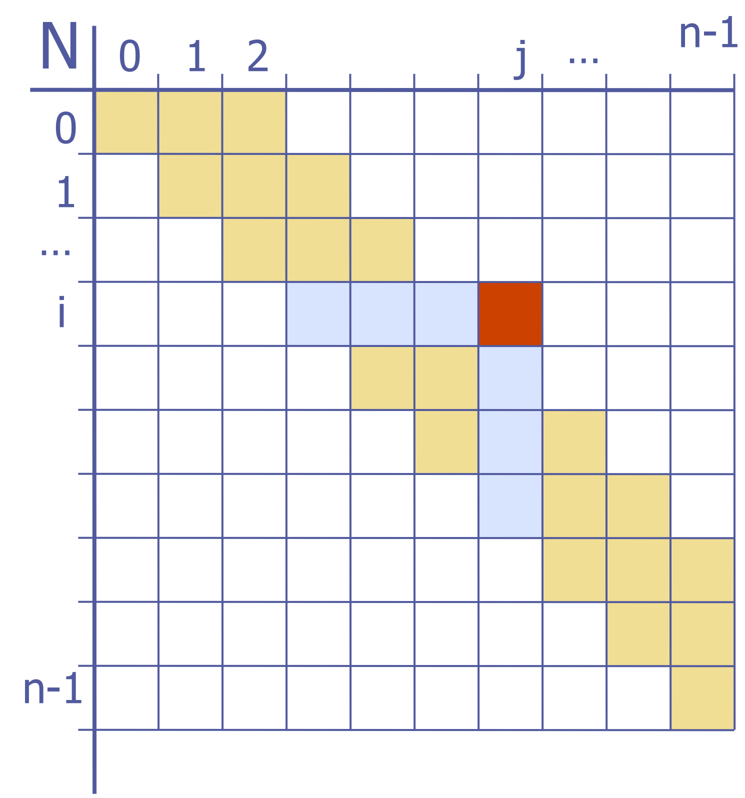
$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering “k” for each N entry

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



Algorithm visualization

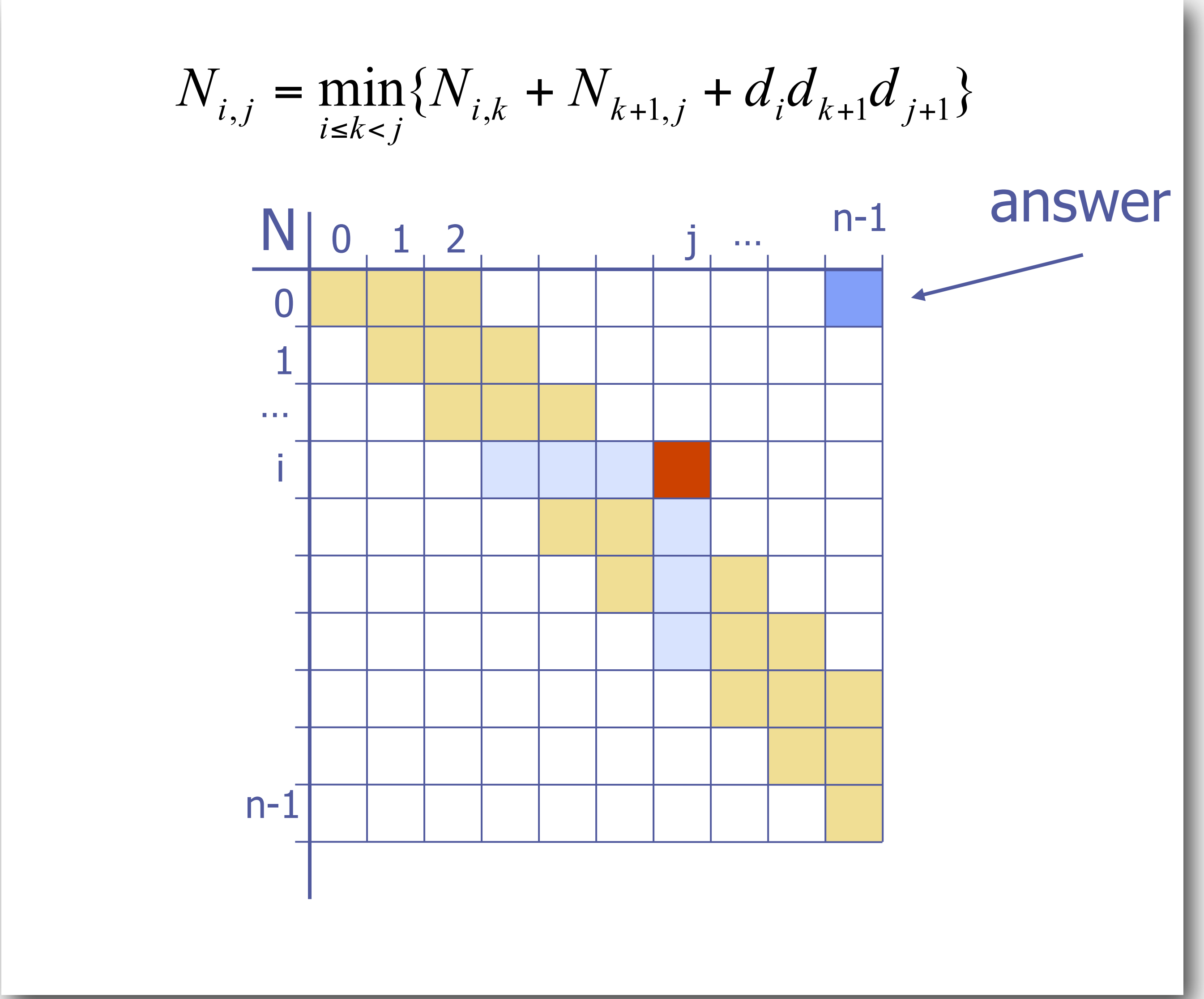
The bottom-up construction fills in the N array by diagonals

$N_{i,j}$ gets values from previous entries in i-th row and j-th column

Filling in each entry in the N table takes O(n) time.

Total run time: O(n3)

Determining actual parenthesization can be done by remembering “k” for each N entry



General dynamic programming technique

Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:

- Simple subproblems: the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
- Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
- Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

- Matrix chain products
- Dynamic programming
- **Longest common subsequence**

Longest common subsequence problem

Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y

- Note that subsequence is different than substring
- Subsequence do not need to be consecutive parts of a string

Has applications to DNA similarity testing (alphabet is {A,C,G,T})

Example:

- ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

DP approach to longest common subsequence (LCS)

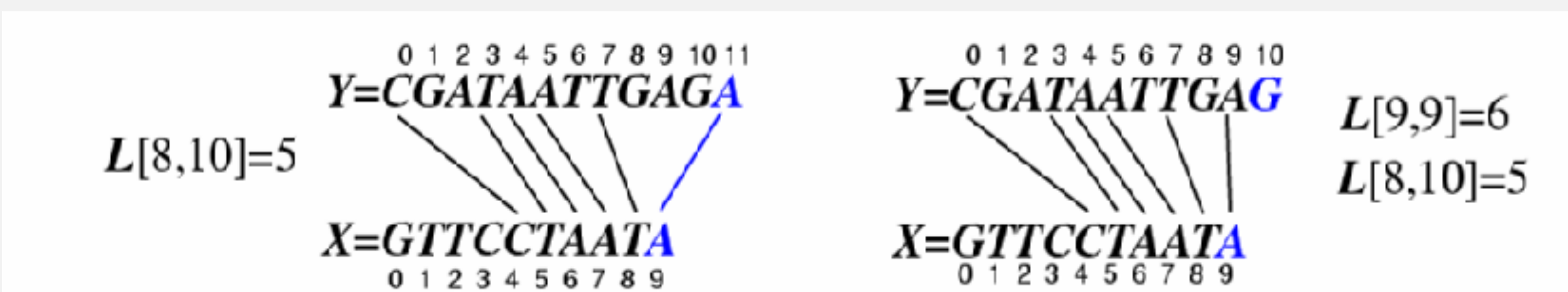
Define L[i,j] to be the length of the longest common subsequence of two strings X[0..i] and Y[0..j].

Allow for -1 as an index, so L[-1,k] = 0 and L[k,-1]=0, to indicate that the null part of X or Y has no match with the other.

Then we can define L[i,j] in the general case as follows:

- If $x_i = y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
- If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

Case 1: Case 2:



LCS algorithm

Algorithm LCS(X,Y)

Input: Strings X and Y with n and m elements, respectively

Output: For i = 0,…,n-1, j = 0,…,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = $x_0x_1x_2…x_i$ and the string Y [0.. j] = $y_0y_1y_2…y_j$

```
for i =1 to n-1 do
        L[i,-1] = 0
for j =0 to m-1 do
        L[-1,j] = 0
for i =0 to n-1 do
        for j =0 to m-1 do
                if xi = yj then
                        L[i, j] = L[i-1, j-1] + 1
        else
                        L[i, j] = max{L[i-1, j] , L[i, j-1]}
return array L
```

Visualizing the LCS algorithm

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

Analysis of the LCS algorithm

We have two nested loops:

- The outer one iterates n times
- The inner one iterates m times
- A constant amount of work is done inside each iteration of the inner loop

Thus, the total running time is $O(nm)$

Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).