Dynamic Programming

Dynamic Programming

- It is a general algorithm design paradigm (different than divide & conquer)
  - Rather than give the general structure, let us first give a motivating example:

Fibonacci Sequence

Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Thanks to David Luebke…

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item (solved previously with *greedy optimization*)

0-1 Knapsack problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i and *W* are integer values)
- Problem: How to pack knapsack to achieve max total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

  - The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: dynamic programming

- We need to carefully identify the subproblems

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack knapsack to achieve max total value of packed items?

Defining a Subproblem

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

S_k = *{items labeled 1, 2, .. k}*

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that.

Defining a Subproblem

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$

?

Max weight: W = 20

For S_4:

Total weight: 14

Maximum benefit: 20

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4

S_5

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_5 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_5 = 10$

For S_5:

Total weight: 20

Maximum benefit: 26

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add another parameter: *w*, which will represent the maximum weight for each subset of items

- The subproblem then will be to compute *V[k,w], i.e.,* to find an optimal solution for *S_k = {items labeled 1, 2, .. k} in a knapsack of size w*

Recursive Formula for subproblems

- The subproblem will then be to compute *V[k,w], i.e.,* to find an optimal solution for *S_k = {items labeled 1, 2, .. k} in a knapsack of size w*

- Assuming knowing V[i, j], where i=0,1, 2, … k-1, j=0,1,2, …w, how to derive V[k,w]?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight $\leq w$, **or**

2) the best subset of S_{k-1} that has total weight $\leq w\text{-}w_k$ plus the item k

Recursive Formula

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]
            else
                V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

V[0,w] = 0

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 1 to n
    V[i,0] = 0

Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=1 4: (5,6)

b_i=3

w_i=2

w=1

w-w_i =-1

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else **V[i,w] = V[i-1,w]** // w_i > w

Example (5)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=1 4: (5,6)

b_i=3

w_i=2

w=2

$w-w_i$ =0

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	**3**		
2	0					
3	0					
4	0					

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=1 4: (5,6)

b_i=3

w_i=2

w=3

w-w_i =1

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=1 4: (5,6)

b_i=3

w_i=2

w=4

$w-w_i$ =2

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	**3**	
2	0					
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (8)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=1 4: (5,6)

b_i=3

w_i=2

w=5

w-w_i =3

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0					
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (9)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=2 4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	**0**				
3	0					
4	0					

b_i=4

w_i=3

w=1

w-w_i =-2

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else **V[i,w] = V[i-1,w]** // w_i > w

Example (10)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=2 4: (5,6)

b_i=4

w_i=3

w=2

w-w_i =-1

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	**3**			
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else **V[i,w] = V[i-1,w]** // w_i > w

Example (11)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=2 4: (5,6)

b_i=4

w_i=3

w=3

w-w_i =0

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	**4**		
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (12)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=2 4: (5,6)

b_i=4

w_i=3

w=4

$w-w_i$ =1

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	**4**	
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (13)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

i=2 4: (5,6)

b_i=4

w_i=3

w=5

$w-w_i$ =2

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	**7**
3	0					
4	0					

if w_i <= w // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
i=3 4: (5,6)

b_i=5

w_i=4

w= 1..3

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	**0**	**3**	**4**		
4	0					

if w_i <= w // item i can be part of the solution
if b_i + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else **V[i,w] = V[i-1,w]** // w_i > w

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
i=3 4: (5,6)

b_i=5

w_i=4

w= 4

w- w_i=0

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	**5**	
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
i=3 4: (5,6)

b_i=5

w_i=4

w= 5

w- w_i=1

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	**7**
4	0					

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
i=4 4: (5,6)

b_i=6

w_i=5

w= 1..4

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	**0**	**3**	**4**	**5**	

if w_i <= w // item i can be part of the solution
if b_i + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else **V[i,w] = V[i-1,w]** // w_i > w

Example (18)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
i=4 4: (5,6)

b_i=6

w_i=5

w= 5

w- w_i=0

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

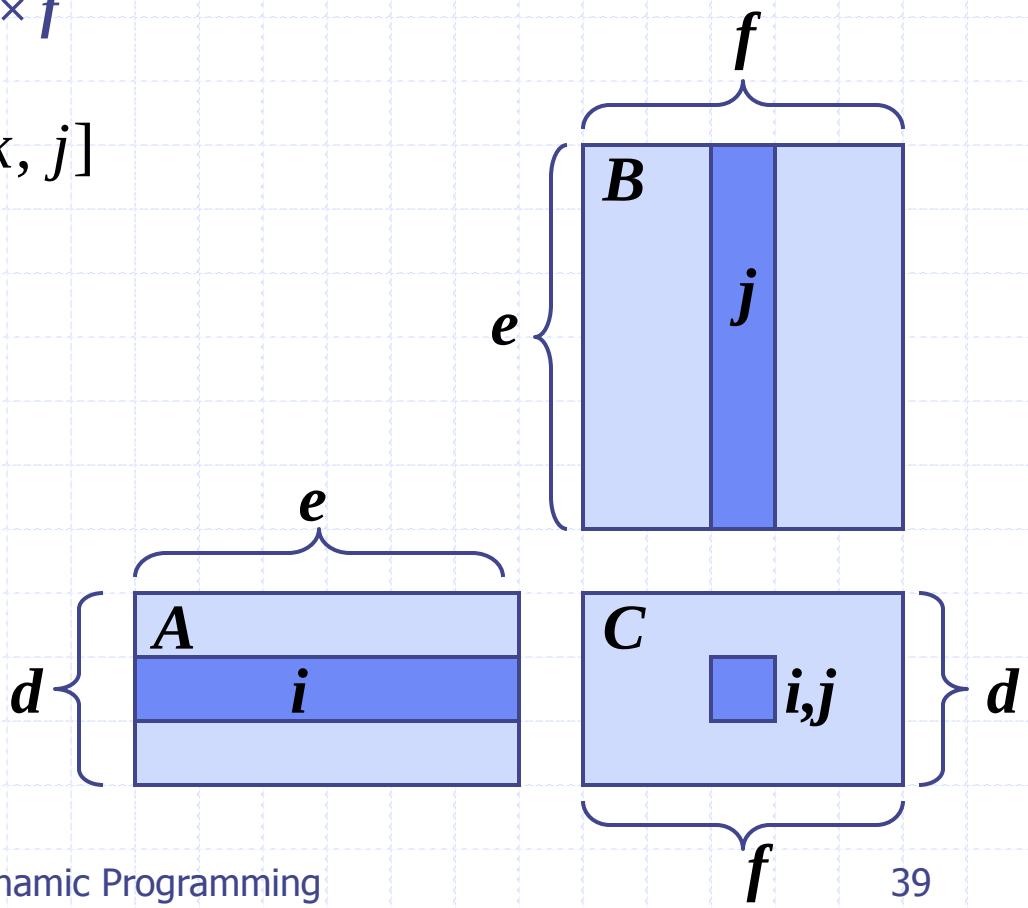
else V[i,w] = V[i-1,w] // w_i > w

Dynamic Programming

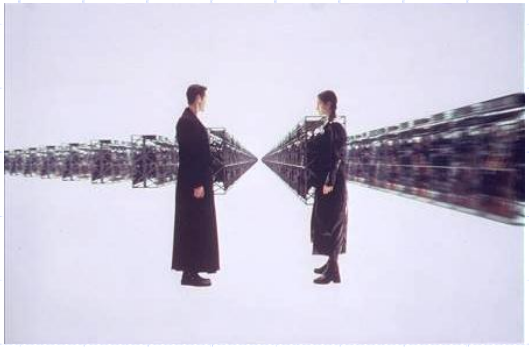
- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

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

  - $O(def)$ time



Matrix Chain-Products



- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to 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

- **Algorithm:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops 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.
  - **This is a terrible 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 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

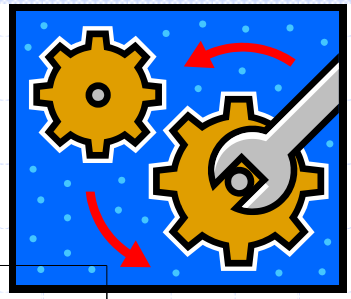
Another Greedy Approach

- 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 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

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.

A Characterizing Equation

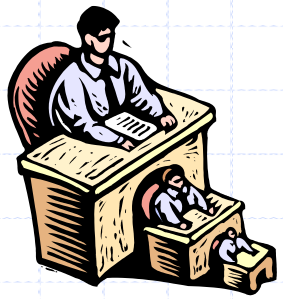


- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the 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 don'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 paranethization of *S*

for $i \leftarrow 1$ **to** $n-1$ **do**

  $N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n-1$ **do**

  for $i \leftarrow 0$ **to** $n-b-1$ **do**

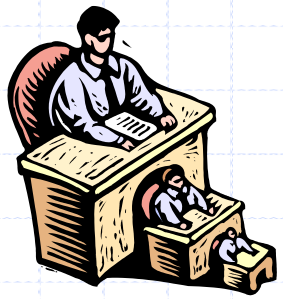
    $j \leftarrow i+b$

    $N_{i,j} \leftarrow +\infty$

    for $k \leftarrow i$ **to** $j-1$ **do**

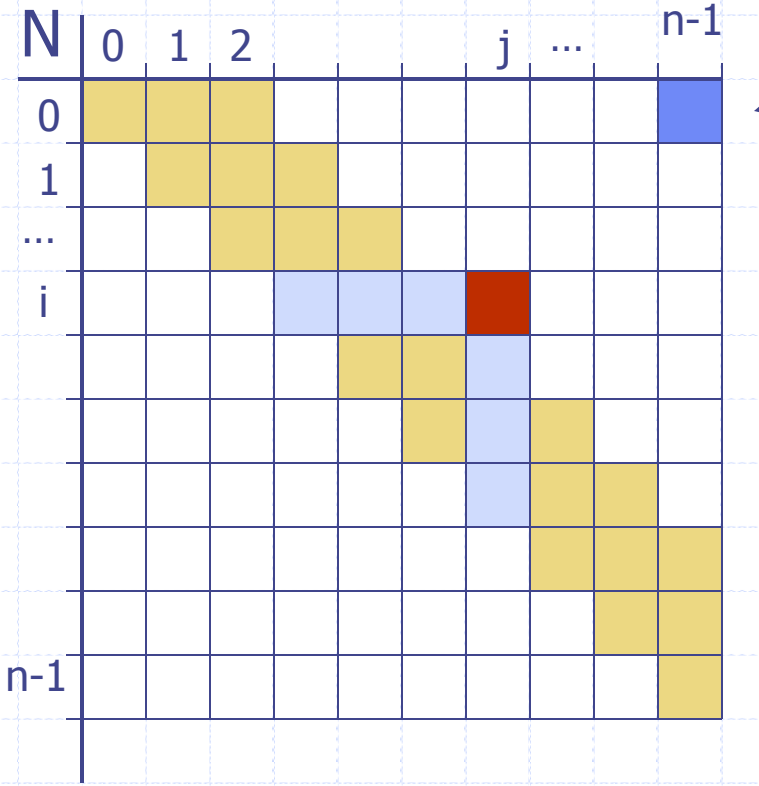
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

A Dynamic Programming Algorithm Visualization

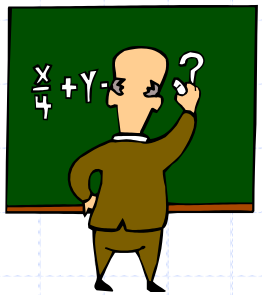


$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- 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: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry



The 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).

The Longest Common Subsequence (LCS) 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

Subsequences

- A ***subsequence*** of a character string $x_0x_1x_2...x_{n-1}$ is a string of the form $x_{j1}x_{j2}...x_{jk}$, where $i_j < i_{j+1}$.
- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGIJK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

The Longest Common Subsequence (LCS) 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
- Has applications to DNA similarity testing (alphabet is {A,C,G,T})
- Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

A Poor Approach to the LCS Problem

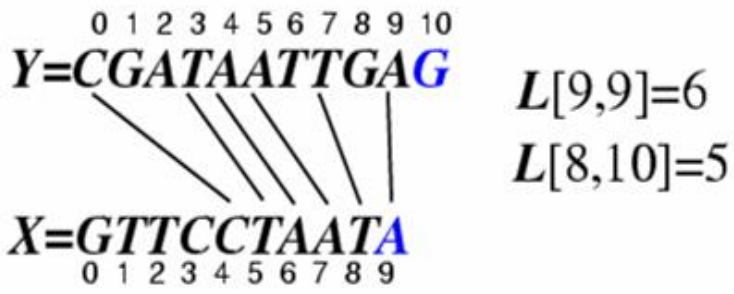
- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

A Dynamic-Programming Approach to the LCS Problem

- Define L[i,j] to be the length of the longest common subsequence of 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:
  1. If xi=yj, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If xi≠yj, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

Case 1:

Case 2:



An LCS Algorithm

Algorithm LCS(X,Y):

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

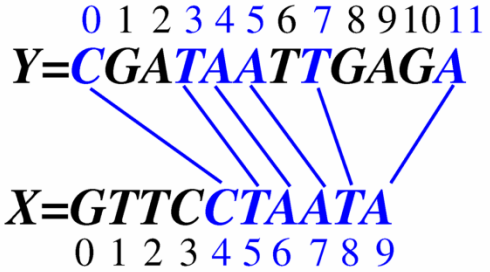
Output: For i = 0,…,n-1, j = 0,…,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = $x_0x_1x_2…x_i$ and the string Y [0.. j] = $y_0y_1y_2…y_j$

```
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 x_i = y_j  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**

https://www.cs.usfca.edu/~galles/visualization/DPLCS.html



Analysis of 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).