We've done

- Matroid Theory
- Task scheduling problem (another matroid example)
- Dijkstra's algorithm (another greedy example)

Now

- Dynamic Programming
  - Matrix Chain Multiplication
  - Longest Common Subsequence

Next

- Dynamic Programming
  - Assembly-line scheduling
  - Optimal Binary Search Trees

Matrix Chain Multiplication (MCM) Problem

Given $A_{10\times 100}$, $B_{100\times 25}$, then calculating AB requires $10 \cdot 100 \cdot 25 = 25,000$ multiplications.

Given $A_{10\times 100}$, $B_{100\times 25}$, $C_{25\times 4}$, then it is true that

$$(AB)C = A(BC) = ABC.$$

- AB requires $25,000$ multiplications
- $(AB)C$ requires $10 \cdot 25 \cdot 4 = 1000$ more multiplications
- totally $26,000$ multiplications

On the other hand

- BC requires $100 \cdot 25 \cdot 4 = 10,000$ multiplications
- $A(BC)$ requires $10 \times 100 \times 4 = 4000$ more multiplications
- totally $14,000$ multiplications

MCM (cont)

If there are 4 matrices A, B, C, D, there are 5 ways to parenthesize the product $ABCD$:

$$(A(B(CD))), (A((BC)D)), ((AB)(CD)),$$
$$((A(BC))D), (((AB)C)D)$$

In general, given n matrices:

$$\begin{array}{ccc} A_1 & \text{of dimension} & p_0 \times p_1 \\ A_2 & \text{of dimension} & p_1 \times p_2 \\ \vdots & \vdots & \vdots \\ A_n & \text{of dimension} & p_{n-1} \times p_n \end{array}$$

There are totally

$$\frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{n!n!} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

ways to parenthesize the product.

Find a parenthesization with the least number of multiplications

Some Observations

- Let's try to find the optimal cost first
- Suppose we split between A_k and A_{k+1}, then the parenthesization of $A_1 \dots A_k$ and $A_{k+1} \dots A_n$ have to also be optimal: optimal substructure.
- Let $c[1,k]$ and $c[k+1,n]$ be the optimal costs for the subproblems, then the cost of splitting at $k, k+1$ is

$$c[1,k] + c[k+1,n] + p_0 p_k p_n$$

because

$$\begin{aligned} A_1 \dots A_k & \quad \text{has dimension} \quad p_0 \times p_k \\ A_{k+1} \dots A_n & \quad \text{has dimension} \quad p_k \times p_n \end{aligned}$$

- The optimal cost $c[1,n]$ is

$$c[1,n] = \min_{1 \le k < n} \left(c[1,k] + c[k+1,n] + p_0 p_k p_n \right)$$

- Hence, in general we need $c[i,j]$ for $i < j$:

$$c[i,j] = \min_{i \le k < j} \left(c[i,k] + c[k+1,j] + p_{i-1} p_k p_j \right)$$

A Recursive Solution

We need the base cases also:

$$c[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \left(c[i,k] + c[k+1,j] + p_{i-1}p_kp_j \right) & \text{if } i < j \end{cases}$$

Opt-MCM(p, i, j)

if $i = j$ **then**
    return 0;
else
    min-so-far $\leftarrow \infty$;
    for $k \leftarrow i$ **to** $j - 1$ **do**
        $c \leftarrow$ Opt-MCM(i, k) + Opt-MCM$(k+1, j)$
        $+p_{i-1}p_kp_j$
        if min-so-far $> c$ **then**
            min-so-far $\leftarrow c$;
        end if
    end for
    return min-so-far;
end if

Running time is exponential for the same reason FibonacciA was exponential. (What's the recurrence?)

A Bottom Up Solution

- We use a table to store $c[i, j], i \leq j$.
- For each $l = 1$ to $n - 1$, recursively calculate the entries $c[i, i + l]$

MCM-Order(p, n)

```
for i = 1 to n do
    c[i, i] ← 0 // base cases
end for
for l = 1 to n − 1 do
    for i ← 1 to n − l do
        j ← i + l; // not really needed, just to be clearer
        c[i, j] ← ∞;
        for k ← i to j − 1 do
            t ← c[i, k] + c[k + 1, j] + p_{i−1} p_k p_j;
            if c[i, j] > t then
                c[i, j] ← t;
            end if
        end for
    end for
end for
return c[1, n];
```

Also Record the Splitting Points

Use $s[i, j]$ to store the optimal splitting point k:

MCM-Order(p, n)

for $i = 1$ **to** n **do**
  $c[i, i] \leftarrow 0$ // base cases
end for
for $l = 1$ **to** $n - 1$ **do**
  for $i \leftarrow 1$ **to** $n - l$ **do**
    $j \leftarrow i + l$; // not really needed, just to be clearer
    $c[i, j] \leftarrow \infty$;
    for $k \leftarrow i$ **to** $j - 1$ **do**
      $t \leftarrow c[i, k] + c[k + 1, j] + p_{i-1}p_kp_j$;
      if $c[i, j] > t$ **then**
        $c[i, j] \leftarrow t$;
        $s[i, j] \leftarrow k$;
      end if
    end for
  end for
end for
return c;

The Actual MCM

Knowing the splitting points, it is now easy:

Matrix-Chain-Multiply(A, i, j, s)

if $j > i$ **then**
    $k \leftarrow s[i, j]$;
    $X \leftarrow$ Matrix-Chain-Multiply(A, i, k, s);
    $Y \leftarrow$ Matrix-Chain-Multiply$(A, k + 1, j, s)$;
    return XY;
else
    return A_i; // $i = j$ in this case
end if

Analysis of MCM's Algorithm

- We also are concerned about space, not only time
- Space needed is $O(n^2)$ for the tables c and s
- Suppose the inner-most loop takes about 1 time unit, then the running time is

$$\begin{aligned}
\sum_{l=1}^{n-1}\sum_{i=1}^{n-l} l &= \sum_{l=1}^{n-1} l(n-l) \\
&= n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 \\
&= n\frac{n(n-1)}{2} - \frac{(n-1)n(2(n-1)+6)}{6} \\
&= \Theta(n^3)
\end{aligned}$$

Memoization

Memoized-MCM-Order(p, n)

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

    $c[i, j] \leftarrow \infty$;

end for

Lookup$(p, 1, n)$;

Lookup(p, i, j)

if $c[i, j] < \infty$ **then**

    return $c[i, j]$; // it's calculated!! Time saved right here

end if

if $i = j$ **then**

    $c[i, i] \leftarrow 0$;

else

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

        $t \leftarrow$ Lookup$(p, i, k)+$ Lookup$(p, k + 1, n)+$ $p_{i-1}p_kp_j$;

        if $t < c[i, j]$ **then**

            $c[i, j] \leftarrow t$; $s[i, j] \leftarrow k$;

        end if

    end for

end if

return $c[i, j]$;

Longest Common Subsequence (LCS) Problem

X	=	t	h	i	s	i	s	c	r	a	z	y
Z	=		h	i				c		a	z	y

Z is a subsequence of X.

X	=	**t**	h	**i**	**s**	**i**	s	c	r	a	z	y				
Y	=	b	u	**t**	**i**	n	t	e	r	e	**s**	t	**i**	n	g	

So, $Z = [t, i, s, i]$ is a common subsequence of X and Y

Given 2 sequences X and Y of lengths m and n, respectively

Find a common subsequence Z of longest length

Analyzing the LCS Problem

- Somehow, find a recursive formula for the objective function
- Suppose $X = [x_1, \dots, x_m]$, $Y = [y_1, \dots, y_n]$

Key observation: optimal substructure

Theorem 1. *Let LCS(X, Y) be the length of a LCS of X and Y*

- *If $x_m = y_n$, then*

$$LCS(X, Y) = 1 + LCS([x_1, \dots, x_{m-1}], [y_1, \dots, y_{n-1}])$$

- *If $x_m \neq y_n$, then either*

$$LCS(X, Y) = LCS([x_1, \dots, x_m], [y_1, \dots, y_{n-1}])$$

or

$$LCS(X, Y) = LCS([x_1, \dots, x_{m-1}], [y_1, \dots, y_n])$$

In other words, LCS(X, Y) is the max of the two in this case.

Conclusions From the Theorem

- For $0 \le i \le m$, $0 \le j \le n$, let

$$\begin{aligned} X_i &= [x_1, \dots, x_i] \\ Y_j &= [y_1, \dots, y_j] \end{aligned}$$

- If $x_m = y_n = z$, then a LCS Z of X and Y can be found by computing a LCS Z' of X_{m-1} and Y_{n-1}, and append z at the end, i.e. $Z = [Z', z]$.

- If $x_m \neq y_n$, then let Z_1 be a LCS of X_{m-1} and Y_n, Z_2 be a LCS of X_m and Y_{n-1}.
  Z is then either Z_1 or Z_2, whichever is longer.

- Let $c[i, j] = LCS[X_i, Y_j]$, then

$$c[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } 0 \\ 1 + c[i-1, j-1] & \text{if } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

  Hence, $c[i, j]$ in general depends on one of three entries: the North entry $c[i-1, j]$, the West entry $c[i, j-1]$, and the NorthWest entry $c[i-1, j-1]$.

Computing LCS length

We maintain a cost table $c[0..m, 0..n]$ of optimal lengths, and a "direction" table $d[1..m, 1..n]$ of $\{N, W, NW\}$ recording where $c[i, j]$ comes from.

LCS-Length(X, Y, m, n)

```
c[i, 0] ← 0 for each i = 0, ..., m;
c[0, j] ← 0 for each j = 0, ..., n;
for i ← 1 to m do
    for j ← 1 to n do
        if x_i = y_j then
            c[i, j] ← 1 + c[i − 1, j − 1];
            d[i, j] ← NW;
        else
            if c[i − 1, j] > c[i, j − 1] then
                c[i, j] ← c[i − 1, j];
                d[i, j] ← N;
            else
                c[i, j] ← c[i, j − 1];
                d[i, j] ← W;
            end if
        end if
    end for
end for
```

Constructing an LCS

Suppose Z is a global array.

(The first call is Construct-LCS(Z, m, n).)

Construct-LCS(Z, i, j)

```
if i = 0 or j = 0 then
    return Z;
else
    k ← c[i, j];
    if d[i, j] = NW then
        Z[k] ← x_i; // which is the same as Y[j]
        Construct-LCS(Z, i − 1, j − 1);
    end if
    if d[i, j] = N then
        Construct-LCS(Z, i − 1, j);
    end if
    if d[i, j] = W then
        Construct-LCS(Z, i, j − 1);
    end if
end if
```

Space and Time Analysis

- Filling out the c and d tables take $\Theta(mn)$-time, which is also the running time of LCS-Length
- The space requirement is also $\Theta(mn)$-time
- Construct-LCS takes $O(m+n)$ (why?)

Note:

- We don't really need the direction table (why?)
- Memoizing this is quite simple too (homework)

A General Look at Dynamic Programming

Step 1

- Identify the sub-problems
- The sub-problems of sub-problems are overlapping
- The total number of sub-problems is a polynomial in input size (why do we need this?)

Step 2

- Write a recurrence for the objective function
- Carefully identify the base cases

Step 3

- Investigate the recurrence to see how to fill out the cost table in a “bottom-up” fashion
- Design appropriate data structure(s) for constructing an optimal solution later on

Step 4 Pseudo Code

Step 5 Analysis of time and space