Dynamic programming, careful table book-keeping

MATRIX-CHAIN-ORDER $(n_1, n_2, \ldots, n_{N+1}, N)$

// First, fill all the elements in the diagonal with zero

for $i \leftarrow 1$ **to** N $\quad m[i,i] \leftarrow 0$

// Next, fill all elements at distance l from the diagonal

for $l \leftarrow 1$ **to** $N-1$

for $i \leftarrow 1$ **to** $N-l$

$j \leftarrow i+l$

define $m[i,j]$ as the minimum of

$m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1}$

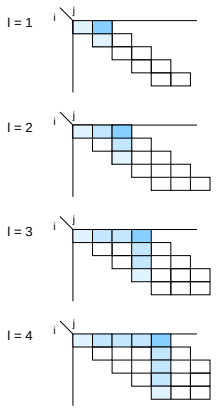
for $i \leq k < j$.

Dynamic programming, more intuition 1

Now, focus on $\quad m[i,j] = \min_k m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1}$.

The Figure on the right shows how the entries of $m[i,j]$ are filled. For each fixed value of l the process fills the entries that are l places to the right of the main diagonal. Notice that all that is needed to compute $m[i,j]$ are the entries $m[i,k]$ and $m[k+1,j]$ for $i \leq k < j$.

For example, $m[1,5]$ will need $m[1,1]$ and $m[2,5]$, or $m[1,2]$ and $m[3,5]$, and so on.



Problems

1. Run the algorithm on the following sequence: 12, 5, 2, 5, 8, 4, 7 ($N = 6$).
2. Estimate the complexity of the divide & conquer algorithm described in the previous lecture.
3. Give some evidence that the complexity of MATRIX-CHAIN-ORDER is $O(n^3)$.
4. Implement in Java the dynamic programming algorithm described last time.

Input instance has $N = 6$ and

i	1	2	3	4	5	6	7
$n[i]$	12	5	2	5	8	4	7

```
0   12*2*5  0     0     0     0
      0   5*5*2   0     0     0
            0    2*8*5  0     0
                  0    5*4*8  0
                        0    8*7*4
                              0
```

$l = 1$
$i \in \{1,2,3,4,5\}$,
$j \leftarrow i+1$ (so we compute $m[i, i+1]$).
$m[i, i+1] \leftarrow n_i n_{i+1} n_{i+2}$

```
0     120   240   0     0     0
      0     50    160   0     0
            0     80    144   0
                  0     160   300
                        0     224
                              0
```

$l = 2$
$i \in \{1,2,3,4\}$,
$j \leftarrow i+2$ (so we compute $m[i, i+2]$).
Define $m[i, i+2]$ as the minimum between
$m[i+1, i+2] + n_i n_{i+1} n_{i+3}$, and
$m[i, i+1] + n_i n_{i+2} n_{i+3}$.

```
0     120   240   392   0     0
      0     50    160   184   0
            0     80    144   200
                  0     160   300
                        0     224
                              0
```

$l = 3$
$i \in \{1, 2, 3\}$,
$j \leftarrow i + 3$
Define $m[i, i + 3]$ as the minimum between
$m[i + 1, i + 3] + n_i n_{i+1} n_{i+4}$,
$m[i, i+1] + m[i+2, i+3] + n_i n_{i+2} n_{i+4}$
$m[i, i + 2] + n_i n_{i+3} n_{i+4}$

```
0     120   240   392   360   0
      0     50    160   184   270
            0     80    144   200
                  0     160   300
                        0     224
                              0
```

$l = 4$
$i \in \{1, 2\}$,
$j \leftarrow i + 4$
Define $m[i, i + 4]$ as the minimum between
$m[i + 1, i + 4] + n_i n_{i+1} n_{i+5}$,
$m[i, i+1] + m[i+2, i+4] + n_i n_{i+2} n_{i+5}$
$m[i, i+2] + m[i+3, i+4] + n_i n_{i+3} n_{i+5}$
$m[i, i + 3] + n_i n_{i+4} n_{i+5}$

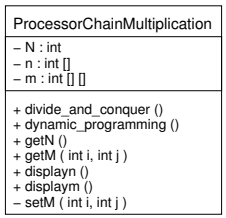
```
0     120   240   392   360   488
      0     50    160   184   270
            0     80    144   200
                  0     160   300
                        0     224
                              0
```

$l = 5$
$i \leftarrow 1, j \leftarrow 6$
Define $m[1, 6]$ (the final answer) as the minimum between
$m[i + 1, i + 5] + n_i n_{i+1} n_{i+6}$,
$m[i, i + 1] + m[i + 2, i + 5] + n_i n_{i+2} n_{i+6}$
$m[i, i + 2] + m[i + 3, i + 5] + n_i n_{i+3} n_{i+6}$
$m[i, i + 3] + m[i + 4, i + 5] + n_i n_{i+4} n_{i+6}$
$m[i, i + 4] + n_i n_{i+5} n_{i+6}$

Divide&Conquer

```
private int setM( int i, int j ) {
    int k;
    int res = 0;
    int opt = 0;

    if ( i < j )
        for ( k = i ; k < j ; k++ ) {
            // compute with k
            res = setM(i,k) + setM(k+1,j) + n[k]*n[i-1]*n[j];

            // keep the smallest do far
            if (( k == i ) || ( res < opt ))
                opt = res;
        }
    return opt;
}


public int divide_and_conquer () {

    return setM(1,N);
}
```


Java Implementation




```
class DP {
  public static void main (String[] args)
      throws IOException {

    ProcessorChainMultiplication P =
      new ProcessorChainMultiplication ();

    P.dynamic_programming();
    System.out.println( P.getM(0,P.getN()-1) );
    // System.out.println( P.divide_and_conquer() );
  }
}
```


Recursive algorithm complexity

Let $T(N)$ be the time to compute `setM(1,N)`. Under some obvious simplifications

$$\begin{aligned} T(1) &\geq 1 \\ T(N) &\geq 1 + \sum_{k=1}^{N-1} (T(k) + T(N-k) + 1) \end{aligned}$$

This can be rewritten as (since each $T(i)$ occurs twice in the sum)

$$T(N) \geq 2 \sum_{k=1}^{N-1} T(k) + N$$

Claim. $T(N) \geq 2^{N-1}$. (Proved by induction on N).

Dynamic Programming

Careful with the indices as arrays in Java run from zero onward.

```
public void dynamic_programming () {
     int i, j, k, l, q;

     for ( i = 0 ; i < N ; i++ ) m[i][i] = 0;

     for ( l = 2 ; l <= N ; l++ ) {
         for ( i = 0 ; i < N - l + 1; i++ ) {
             j = i + l - 1;
             for ( k = i ; k < j ; k++ ) {
                 q = m[i][k] + m[k+1][j] + n[i]*n[k+1]*n[j+1];
                 if (( k == i ) || (q < m[i][j]))
                     m[i][j] = q;
             }
         }
         displaym();
         System.out.println();
     }
 }
```


Comparing Running times

I run both alternatives on a Linux 200MHz PC.

The input were N square matrices of size 3.

cost$(1) = 0$ and cost$(N) =$ cost$(N-1) + 27$ giving cost$(N) = 27(N-1)$.

Divide and conquer

N	time	cost
6	1.20	135
8	1.04	189
10	1.10	243
12	1.15	297
14	1.78	351
16	6.86	405
18	51.40	459
20	7:32.80	513
22	1:11:20.64	567

Dynamic programming

N	time
6	1.05
8	1.04
10	1.09
12	1.15
14	1.08
16	1.11
18	1.12
20	1.07
22	1.09

Constructing an optimal solution

We can use another table s to determine the best way to multiply the matrices. Each entry $s[i,j]$ records the value k such that the optimal ordering to compute $A_i \times \ldots \times A_j$ splits the product at A_k. Thus we know that the final matrix multiplication in computing $A_{1..N}$ optimally is $A_{1..s[1,N]} \times A_{s[1,N]+1..N}$. Earlier matrix multiplications can be computed recursively.

On the next slide a slightly more detailed pseudo-code for MATRIX-CHAIN-ORDER including the handling of s.

MATRIX-CHAIN-ORDER $(n_1, n_2, \ldots, n_{N+1}, N)$

for $i \leftarrow 1$ **to** N
  $m[i,i] \leftarrow 0$
for $l \leftarrow 1$ **to** $N-1$
  for $i \leftarrow 1$ **to** $N-l$
    $j \leftarrow i+l$
    $m[i,j] \leftarrow +\infty$
    for $k \leftarrow i$ **to** $j-1$
      $q \leftarrow m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1}$
      if $q < m[i,j]$
        $m[i,j] \leftarrow q$
        $s[i,j] \leftarrow k$

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
    if j > i
        X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
        Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
        return MATRIX-MULTIPLY(X, Y)
    else return A_i
```


Another Problem?

Golf playing. A target value N is given, along with a set $S = \{s_1, s_2, \ldots, s_n\}$ of *admissible segment lengths* and a set $B = \{b_1, \ldots, b_m\}$ of *forbidden values*. The aim is to choose the shortest sequence $\sigma_1, \sigma_2, \ldots, \sigma_u$ of elements of S such that

1. $\sum_{i=1}^{u} \sigma_i = N$ and
2. $\sum_{i=1}^{j} \sigma_i \notin B$ for each $j \in \{1, \ldots, u\}$.

Claim. Any instance of this problem is solvable optimally in time polynomial in N and n.

Yet another problem?

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of length $l_1, \ldots, l_n$ (number of characters). We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is

$$M - j + i - \sum_{k=i}^{j} l_k,$$

(which must be non-negative so that the word fit on the line). We wish to minimise the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines.

Dynamic programming solution?

Run-time? Space requirements?