VIII DYNAMIC PROGRAMMING

Divide and conquer is a top-down method. When a problem is solved by D&C, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses. Dynamic programming on the other hand is a bottom up technique. We usually start with the smallest, and hence the simplest, subinstances. By combining their solutions, we obtain the answers to subinstances of increasing size, until finally we arrive at the solution of the original instance.

8.1 Calculating the binomial coefficient:

Consider the problem of calculating the binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k=0 \text{ or } k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0<k<n \\ 0 & \text{otherwise} \end{cases}$$

Suppose $0 \leq k \leq n$. If we calculate by

```
function C(n, k)
    if k = 0 or k = n then return 1
    else return C(n-1, k-1) + C(n-1, k)
```

many of the values $C(i,j)$, $i<n$, $j<k$ are calculated over and over. For example, the algorithm calculates $C(5,3)$ as the sum of $C(4,2)$ and $C(4,3)$. Both these intermediate results require us to calculate $C(3,2)$. Similarly the value of $C(2,2)$ is used several times; the execution time of this algorithm is sure to be in $\Omega\left(\binom{n}{k}\right)$. If on the other hand, we use a table of intermediate results - Pascal's triangle - we obtain a more efficient algorithm - See the following table, where it should be filled line by line.

In fact, it is not even necessary to store the entire table: it suffices to keep a vector of length k, representing the current line, and to update this vector from left to right. Thus to calculate $\binom{n}{k}$ the algorithm takes a time in $\Theta(nk)$ and space in $\Theta(k)$; if we assume that addition is an elementary operation.

	0	1	2	3	4	...	k-1	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
⋮	⋮							
n-1	1						C(n-1,k-1)	C(n-1,k)
n	1							+ ↓ C(n,k)

- Pascal's triangle -

8.2 Making Change (2)

We described a greedy algorithm before, that was very efficient but works only in a limited number of instances, with certain systems of coinage, or when coins of a particular denomination are missing or in short supply, the algorithm may either find a suboptimal answer, or not find an answer at all. Let's approach the problem using DP.

Let the currency we use has available coins of n different denominations, let a coin of denomination i, $1 \leq i \leq n$, has value d_i units. We suppose as is usual, that each $d_i > 0$. For now, we suppose we have an unlimited supply of coins of each denomination. Suppose we have to give the customer coins worth N units, using as few coins as possible. Let's set up a table $c[1..n, 0..N]$, one row for each available denomination and one column for each amount from 0 to N units. In this table $c[i,j]$ will be the minimum number of coins required

to pay an amount of j units, $0 \leq j \leq N$ using only coins of denominations 1 to i, $1 \leq i \leq n$. The solution to the instance is then given by $c[n, N]$, if all we want to know is how many coins are needed. To fill in the table, note first that $c[i,0]$ is zero for every value of i. After this initialization the table can be filled in either left to right row by row, or column by column from top to bottom. To pay an amount j using coins of denominations 1 to i, we have in general 2 choices.

a. Coins of denomination i, even though this is now permitted, in which case $c[i,j] = c[i-1,j]$

b. At least one coin of denomination i. Amount remained to be paid an amount of $j - d_i$ units.

To pay this takes $c[i, j-d_i]$ coins, so $c[i,j] = 1 + c[i, j-d_i]$. Since we want to minimize the number of coins used, we use whichever alternative is better. In general therefore

$$c[i,j] = \min\left(c[i-1,j],\ c[i, j-d_i]\right)$$

When $i = 1$ one of the elements to be compared falls outside the table. The same is true when $j < d_i$. It is convenient to think of these elements as having the value $+\infty$. If $i = 1$ and $j < d_1$, then both elements to be compared fall outside the table. In this case we set $c[i,j]$ to $+\infty$, to indicate that it is impossible to pay an amount j using only coins of type 1.

Consider the following example, where we have to pay 8 units with coins worth 1, 4 and 6 units. For example, $c[3,8]$ is obtained in this case as the smaller of $c[2,8] = 2$ and $1 + c[3, 8-d_3] = 1 + c[3,2] = 3$. The entries elsewhere in the table are obtained similarly. The answer to this

particular instance is that we can pay 8 units using only 2 coins.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

formal algorithm.

```
function coins(N)
  {Gives the minimum number of coins needed to make
   change for N units. Array d[1..n] specifies the
   coinage: in the example these are coins for 1, 4 and 6 units.}

  array d[1..n] = [1, 4, 6]
  array c[1..n, 0..N]
  for i ← 1 to n do c[i, 0] ← 0
  for i ← 1 to n do
      for j ← 1 to N do
          c[i,j] ← if i = 1 and j < d[i] then +∞
                   else if i = 1 then 1 + c[1, j - d[1]]
                   else if j < d[i] then c[i-1, j]
                   else min(c[i-1, j], 1 + c[i, j - d[i]])
  return c[n, N]
```

8.3 Chained Matrix Multiplication:

Recall that the product C of $p \times q$ matrix A and a $q \times r$ matrix B is the $p \times r$ matrix given by

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj} \qquad 1 \leq i \leq p, \; 1 \leq j \leq r$$

Algorithmically, we can express this as the following from which it is clear that a total of pqr scalar multiplic

are required to calculate the product.

```
for i ← 1 to p do
    for j ← 1 to r do
        c[i,j] ← 0
        for k ← 1 to q do
            c[i,j] ← c[i,j] + A[i,k]B[k,j]
```

Suppose we want to calculate the following product

$$M = M_1 M_2 M_3 \dots M_n$$

Since matrix multiplication is associative, this can be done in a number of ways

$$M = (\dots((M_1 M_2) M_3) \dots M_n)$$
$$= (M_1 (M_2 (M_3 \dots (M_{n-1} M_n) \dots)))$$
$$= (\dots((M_1 M_2)(M_3 M_4)) \dots) \quad \text{and so on.}$$

However matrix multiplication is not commutative, so we're not allowed to change the order of the matrices. The choice of a method has a considerable influence on the time required, let's consider the following example,

Example: Calculate the product $ABCD$ where A is 13×5, B is 5×89, C is 89×3 and D is 3×34. To measure the efficiency of the different methods, we count the number of scalar multiplications involved, there will be approximately the same number of additions, so the number of multiplications is a good indicator of overall efficiency, for instance, using $M = ((AB)C)D$ we get

AB	5785	multiplications
$(AB)C$	3471	"
$((AB)C)D$	1326	"

for a total of 10,582 multiplications.
There are 5 different ways of calculating the product in this case (when we do (AB)(CD) we do not differentiate between the cases which one is done first).

$((A\,B)\,C)\,D \longrightarrow 10,582$

$(A\,B)(C\,D) \longrightarrow 54,201$ ← almost 19 times more than

$(A\,(B\,C))\,D \longrightarrow 2,856$

$(A\,((B\,C)\,D)) \longrightarrow 4,055$

$A\,(B\,(C\,D)) \longrightarrow 26,418$

To find directly the best way to calculate the product, we could simply parenthesize the expression in all possible ways and each time count how many multiplications are required. Let $T[n]$ be the number of essentially different ways to parenthesize a product of n matrices. Suppose we make the first cut at the i^{th} - $(i+1)$st interval

$$M = (M_1 M_2 \ldots M_i)(M_{i+1} \ldots M_n)$$

There are $T(i)$ ways to parenthesize the LHS and $T(n-i)$ ways to parenthesize the RHS, so there are:

$$T(i)\,T(n-i)$$

to parenthesis the whole expression. Since i can take any value from 1 to $n-1$, we obtain the following recurrence:

$$T(n) = \sum_{i=1}^{n-1} T(i)\,T(n-i)$$

where some values are:

n	1	2	3	4	5	10	15
$T(n)$	1	1	2	5	14	4862	2,675,440

Values of $T(n)$ are called the Catalan numbers.
A solution: $T(n) = \frac{1}{n}\binom{2n-2}{n-1}$

To insert parenthesis in M, it takes a time in $\Omega(n)$.
combining $T(n) = \frac{1}{n}\binom{2n-2}{n-1}$, and the fact that $\binom{2n}{n} \geq \frac{4^n}{(2n+1)}$

$T(n)$ is in $\Omega\left(\frac{4^n}{n^2}\right)$

This method is obviously very impratical for lenge n, too many parenthesis to insert.
The greedy algorithm (or money of the known ones) will not provide an optimal way to compute matrix multiplication
But we can try the fundamental principle of optimality which states (when it applies of course),
" The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its subinstances."
(The difficulty in turning this principle into an algorithm is that it is not usually obvious which subinstances are relevant to the instance under consideration.)
Coming back to our case, the best way of multiplying all the matrices requires us to make the first cut between the ith and the $(i+1)$-st matrices of the product, then both subproducts $M_1 M_2 \dots M_i$ and $M_{i+1} M_{i+2} \dots M_n$ must also be calculated in an optimal way.
We should consider DP and the construction of a table m_{ij}, $1 \leq i \leq j \leq n$, where m_{ij} gives the optimal solution for the part $M_i \dots M_j$. The solution to the original problem is m_{1n}

Suppose the dimensions of the matrices are given by a vector $d[0..n]$ such that, matrix M_i $1 \leq i \leq n$ is of dimension $d_{i-1} \times d_i$. We build the table m_{ij} diagonal by diagonal: diagonal s contains elements of m_{ij} such that $j - i = s$. The diagonal $s = 0$, therefore contains the elements m_{ii}, $1 \leq i \leq n$, corresponding to the "Products" M_i. Since no

multiplication to be done, so $m_{ii} = 0 \ \forall i$. Diagonal $s = 1$ contains $m_{i,i+1}$ which corresponds to $M_i M_{i+1}$. Here we have no choice but to compute the product directly, which we can do using $d_{i-1} d_i d_{i+1}$ multiplications. When $s > 1$ the diagonal s contains the elements $m_{i,i+s}$ corresponding to products of the form $M_i, M_{i+1}, \ldots, M_{i+s-1}$. If we make the cut after M_k, $i \le k \le i+s$, we need m_{ik} multiplications the left hand term, $m_{k+1,i+s}$ to calculate the right hand term, and then $d_{i-1} d_k d_{i+s}$ to multiply the 2 resulting matrices to obtain the final result. To find the optimum, we choose the cut that minimizes the required number of scalar multiplications.

Summing up, fill in the table m_{ij} using the following rules for $s = 0, 1, \ldots, n-1$

$$s = 0: \quad m_{ii} = 0 \qquad i = 1, 2, \ldots, n$$

$$s = 1: \quad m_{i,i+1} = d_{i-1} d_i d_{i+1} \qquad i = 1, 2, \ldots, n-1$$

$$1 < s < n: \quad m_{i,i+s} = \min_{i \le k < i+s} \left(m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s} \right) \qquad i = 1, 2, \ldots, n-s$$

Example: A B C D

A: 13×5, B: 5×89, C: 89×3, D: 3×34

vector $d = [13, 5, 89, 3, 34]$.

for $s = 1$, we find $m_{12} = 5785$, $m_{23} = 1335$, and $m_{34} = 9078$.

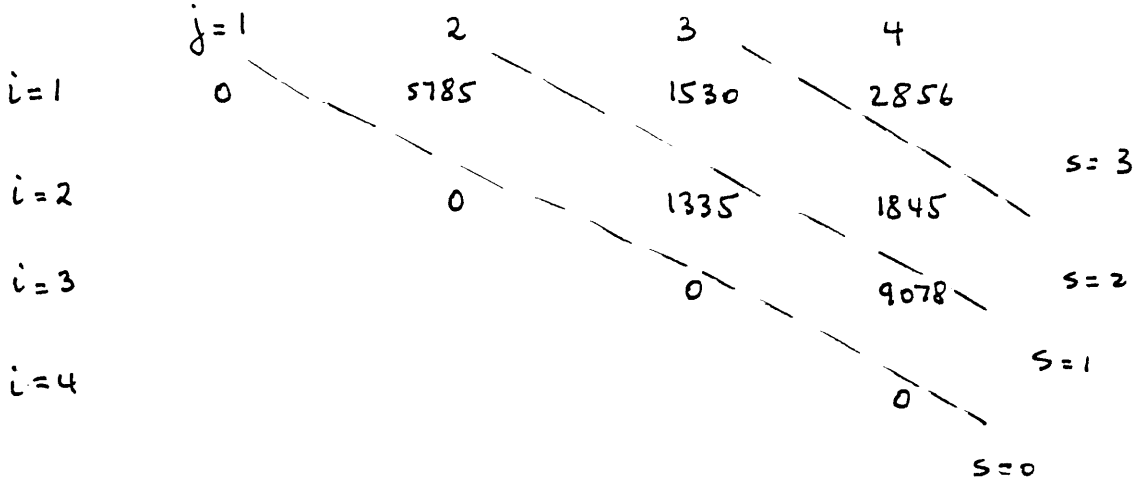
Next; for $s = 2$ we obtain

$$m_{24} = \min \left(m_{22} + m_{34} + 5 \times 89 \times 34, \; m_{23} + m_{44} + 5 \times 3 \times 34 \right)$$
$$= \min (24208, 1845) = 1845$$

$$m_{13} = \min \left(m_{11} + m_{23} + 13 \times 5 \times 3, \; m_{12} + m_{33} + 13 \times 89 \times 3 \right)$$
$$= \min (1530, 9256) = 1530$$

finally $s = 3$

$$m_{14} = \min\begin{cases} m_{11} + m_{24} + 13\times 5\times 34, & \{k\}=1 \\ m_{12} + m_{34} + 13\times 89\times 34, & \{k\}=2 \\ m_{13} + m_{44} + 13\times 3\times 34 & \{k\}=3 \end{cases}$$
$$= \min(4055,\ 54201,\ 2856) = 2856$$



One again we usually want to know not only the number of multiplicati necessary to compute the product M, but also how to perform this computation efficiently. We do this by adding a second array to keep track of the choices we have made. Let this array be "bestk". Now when we compute m_{ij} we save in bestk[i,j] the value of k that corresponds to the minimum term among those compared. When the algorithm stops, bestk[1..n] tells us where to make the first cut in the product. Proceeding recursively on both the terms thus produced, we can reconstruct the optimal way of parenthesizing M.

For $s>0$, there are $n-s$ elements to be computed in the diagonal s; for each of these we must choose between s possibilities given by the different values of k. The execution time of the algorithm is therefore in the exact order of:

$$\sum_{s=1}^{n-1}(n-s)s = n\sum_{s=1}^{n-1}s - \sum_{s=1}^{n-1}s^2 = \frac{(n^3-n)}{6} \in \theta(n^3).$$