CSE 102
Introduction to Analysis of Algorithms
Dynamic Programming (CLRS sections 15.1-15.5)

Dynamic Programming is an algorithm design technique that can sometimes provide a more efficient solution than the Divide and Conquer approach. We begin with an elementary example.

Problem Computing Binomial Coefficients

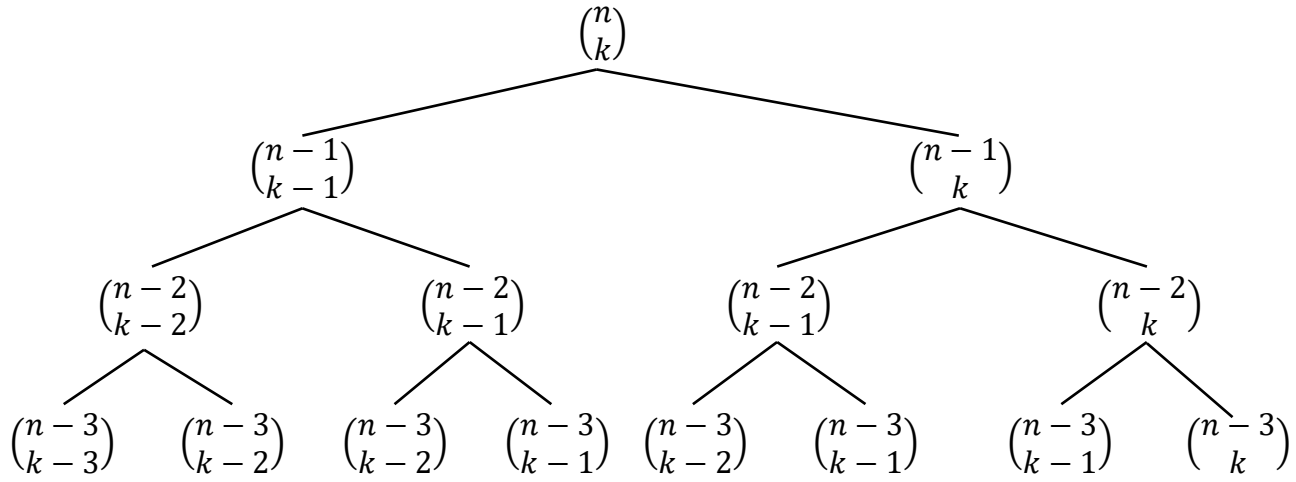
The Binomial Coefficient $\binom{n}{k}$ denotes the number of *k*-element subsets of an *n*-element set, where $0 \leq k \leq n$. Pascal's identity says that if $0 < k < n$, then $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. This identity provides the basis for a recursive algorithm for computing $\binom{n}{k}$. The initial terms for the recurrence are $\binom{n}{0} = \binom{n}{n} = 1$, so we can write the recurrence as

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

Expressing the formula in pseudo-code we have:

BinomialCoefficient(n, k)
1. if $k = 0$ or $k = n$
2. return 1
3. else
4. return BinomialCoefficient$(n-1, k-1)$ + BinomialCoefficient$(n-1, k)$

Observe that at the bottom level, BinomialCoefficient( ) always returns 1, so ultimately it just adds up a lot of 1's. If we take additions our basic operation, then BinomialCoefficient(n, k) runs in time $\Theta\left(\binom{n}{k}\right)$. Recall that if $n = 2k$, Stirling's formula gives $\binom{n}{k} = \Theta\left(\frac{2^n}{\sqrt{n}}\right) = \Theta\left(\frac{4^k}{\sqrt{k}}\right)$, which is exponential time. The recursion tree below shows the problem.



Observe that many of the same values are computed multiple times. This problem arises frequently in the Divide and Conquer approach. The obvious and natural way of dividing a problem into subinstances leads to overlapping (or identical) subinstances that are solved multiple times, leading to an inefficient

algorithm. A more efficient approach is to maintain a table of intermediate results. When a subinstance value is needed, we first look it up in the table to see if it has already been computed. Only when it has not, do we compute that subinstance value.

	0	1	2	3	……	$k-1$	k
0	**1**	**0**	**0**	**0**	……	**0**	**0**
1	**1**	**1**	**0**	**0**	……	**0**	**0**
2	**1**	**2**	**1**	**0**	……	**0**	**0**
3	**1**	**3**	**3**	**1**	……	**0**	**0**
⋮	⋮	⋮	⋮	⋮		⋮	⋮
⋮	⋮	⋮	⋮	⋮		⋮	⋮
$n-1$	**1**					$\binom{\boldsymbol{n-1}}{\boldsymbol{k-1}}$	$\binom{\boldsymbol{n-1}}{\boldsymbol{k}}$
n	**1**						$\binom{\boldsymbol{n}}{\boldsymbol{k}}$

Once the table is completely filled, we return the value in row n column k. In fact, it is not necessary to store the whole table, just a single row at a time.

DynamicBinomialCoefficient(n, k)
1. $C[0] = 1$
2. for $i = 1$ to k
3. $\quad C[i] = 0$
4. for $j = 1$ to n
5. $\quad$ for $i = k$ down to 1
6. $\quad\quad C[i] = C[i-1] + C[i]$
7. return $C[k]$

The run time of the above algorithm is obviously $\Theta(nk)$. Note there is still some efficiency to be gained by deleting all zeros and by not calculating parts of certain rows. We leave this as an exercise for the reader.

Actually, there are two versions of this technique. The iterative version (as above) is called *Dynamic Programming*. (The word "programming" is used here in an archaic sense, in which "program" is synonymous with the word "table".) The fully recursive version is called *Memoization*. All of the algorithms in this handout are of the iterative version.

Problem Coin Changing
Suppose we have coins in n denominations $\{d_1, d_2, d_3, \ldots, d_n\}$ where each $d_i \geq 1$ is a positive integer. We wish to pay an amount N using the least number of coins possible. We assume that there is an unlimited supply of coins in each denomination. There are two questions in this optimization problem.

- What is the least number of coins needed to pay N monetary units. This is called the *value of an optimal solution*.
- Exactly which coins, i.e. how many coins in each denomination, are to be disbursed. This is the *optimal solution* itself.

To answer the first question, we create a 2-dimensional table $C[1 \cdots n; 0 \cdots N]$, where

$C[i,j]$ = the minimum number of coins necessary to pay amount j using only coins in the set $\{d_1, d_2, \ldots, d_i\}$

for $1 \le i \le n$ and $0 \le j \le N$. We seek the value $C[n,N]$. First observe that $C[i,0] = 0$ for $1 \le i \le n$. Next, notice that to pay the amount j using only denominations $\{d_1, d_2, \ldots, d_i\}$ we have in general two choices.

(1) Use no coins of value d_i, even though this is allowed. The least number of coins needed for this choice is $C[i-1,j]$.
(2) Use at least one coin of value d_i. After paying one such coin, there are $j - d_i$ units left to pay from denominations $\{d_1, d_2, \ldots, d_i\}$. The least number of coins for this choice is $1 + C[i, j-d_i]$.

The table entry $C[i,j]$ should be whichever alternative uses the least coins. Thus, we have the recurrence relation

$$C[i,j] = \min(\, C[i-1,j], 1 + C[i,j-d_i]\,).$$

Note that if either $i = 1$ or $j < d_i$, then one of the values on the right falls outside the table. It is convenient to regard such values as being $+\infty$ (since this is the identity element with respect to the min operation.) If both $i = 1$ and $j < d_1$, then we set

$$C[i,j] = +\infty$$

indicating that it is impossible to pay amount j using only coins of value d_1.

Example $n = 4$, $d_1 = 1$, $d_2 = 3$, $d_3 = 5$, $d_4 = 6$ and $N = 8$.

		j								
i	d_i	0	1	2	3	4	5	6	7	8
1	1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	3	**0**	**1**	**2**	**1**	**2**	**3**	**2**	**3**	**4**
3	5	**0**	**1**	**2**	**1**	**2**	**1**	**2**	**3**	**2**
4	6	**0**	**1**	**2**	**1**	**2**	**1**	**1**	**2**	**2**

If we have an unlimited supply of coins of value 1 (i.e. if $d_1 = 1$), then it is possible to disburse any amount. If not, it may be impossible to pay certain amounts. This is indicated by ∞ as a table entry.

Example $n = 3$, $d_1 = 2$, $d_2 = 4$, $d_3 = 5$ and $N = 8$.

		j								
i	d_i	0	1	2	3	4	5	6	7	8
1	2	**0**	∞	**1**	∞	**2**	∞	**3**	∞	**4**
2	4	**0**	∞	**1**	∞	**1**	∞	**2**	∞	**2**
3	5	**0**	∞	**1**	∞	**1**	**1**	**2**	**2**	**2**

The following algorithm takes as input the array of monetary values $d[1 \cdots n]$, and a positive integer N giving the amount to be paid. It uses a local array $C[1 \cdots n; 0 \cdots N]$.

```
CoinChange(d, N)
1.  n = length[d]
2.  for i = 1 to n
3.      C[i, 0] = 0
4.  for i = 1 to n
5.      for j = 1 to N
6.          if i = 1 and j < d[1]
7.              C[1, j] = ∞
8.          else if i = 1
9.              C[1, j] = 1 + C[1, j − d[1]]
10.         else if j < d[i]
11.             C[i, j] = C[i − 1, j]
12.         else
13.             C[i, j] = min(C[i − 1, j], 1 + C[i, j − d[i]])
14. return C[n, N]
```

If necessary, the algorithm can be altered to return the entire table $C[1 \cdots n; 0 \cdots N]$ of intermediate results. The run time of this algorithm is obviously $\Theta(nN)$ since each of the $n(N+1)$ table entries must be filled.

Exercise
Modify this algorithm to deal with the situation in which the supply of coins in some denominations is limited. Let $l[1 \cdots n]$ be another input array, where $0 \leq l[i] \leq \infty$, and require that at most $l[i]$ coins of value $d[i]$ be used for $1 \leq i \leq n$.

Once $C[1 \cdots n; 0 \cdots N]$ has been filled, we can by backtracking through the table, solve the second problem, i.e. determine exactly which coins should be used. Observe that if $C[i, j] = C[i-1, j]$, then no coins of value d_i are needed to pay j units when restricted to the denominations $\{d_1, \ldots, d_i\}$. In this case we move up one row in the table to cell $C[i-1, j]$. On the other hand, if $C[i, j] = 1 + C[i, j - d_i]$, we pay one coin of value d_i, then move left to cell $C[i, j - d_i]$ to see what to do next. If $C[i, j] = C[i-1, j] = 1 + C[i, j - d_i]$, then either action is acceptable.

Exercise
Write a recursive algorithm that, given the filled table $C[1 \cdots n; 0 \cdots N]$ as input, prints a sequence of $C[n, N]$ coin values whose total value is N. If it happens that $C[n, N] = \infty$, print a message to the effect that no such disbursement of coins is possible.

Exercise
Modify your solution to the above exercise so as to print out ***all*** sequences of coin values that pay N units using the least number of coins from denominations $\{d_1, d_2, \ldots, d_n\}$.

Problem Discrete knapsack
A thief wishes to steal n objects labeled 1 to n. For each $i = 1$ to n, let

v_i = value of object i
w_i = weight of object i

The thief has a knapsack that can carry a maximum weight of W. His goal is to fill the knapsack in a way that maximizes the total value of the goods stolen, while not exceeding its weight constraint. Note each object has a unique identity and cannot be stolen twice. Define a bit string $x = x_1 x_2 \cdots x_n$ by

$$x_i = \begin{cases} 1 & \text{if object } i \text{ is stolen} \\ 0 & \text{if object } i \text{ is not stolen} \end{cases}$$

The problem is then to choose $x \in \{0,1\}^n$ so as to

Maximize: $\sum_{i=1}^{n} x_i v_i$

Subject to: $\sum_{i=1}^{n} x_i w_i \leq W$

Any bit string $x \in \{0,1\}^n$ that satisfies the second condition is known as a *feasible solution*, while a bit string satisfying both conditions is called an *optimal solution*. As before there are really two problems here.

- Find the value of an optimal solution: the maximum possible value $\sum_{i=1}^{n} x_i v_i$ of the goods stolen.
- Find the optimal solution itself: exactly which objects should be stolen in order to achieve this maximum value.

To solve the first problem, we create a table $V[1 \cdots n; 0 \cdots W]$ where

$V[i,j] =$ the maximum value of the objects in the set $\{1, \ldots, i\}$ whose total weight does not exceed j

To determine $V[i,j]$ we have, as before, two alternatives

(1) Do not include object i. In this case at most value $V[i-1,j]$ can be stolen.
(2) Include object i. This increases the value of the load by v_i and effectively reduces the capacity by w_i. Thus, in this case, at most value $v_i + V[i-1, j-w_i]$ can be stolen.

Choosing the best of these alternatives yields the following recurrence relation

$$V[i,j] = \max(V[i-1,j], v_i + V[i-1, j-w_i])$$

By including boundary and out of bounds entries we have

$$V[i,j] = \begin{cases} 0 & i = 0 \text{ and } j \geq 0 \\ \max(V[i-1,j], v_i + V[i-1, j-w_i]) & i > 0 \text{ and } j \geq 0 \\ -\infty & j < 0 \end{cases}$$

It is understood that when $i = 0$, the set $\{1, \ldots, i\}$ is empty, hence the boundary value of 0 in that case. The value $-\infty$ is chosen for $j < 0$ since it is the identity element with respect to the max operation. Also, the above discussion seems to assume that the object weights $(w_1, w_2, \ldots, w_n)$ and the weight limit W are integers. If this is not the case, we can make it (at least approximately) true by a strategic choice of units. We approximate all weights by fractions (in lowest terms), compute the least common multiple (LCM) of each of their denominators, then use the reciprocal of that LCM as our basic unit of weight.

Example $n = 5, w = (1, 3, 5, 6, 7), v = (1, 5, 12, 25, 30), W = 10$

			j										
i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10
1	1	1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	3	5	**0**	**1**	**1**	**5**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
3	5	12	**0**	**1**	**1**	**5**	**6**	**12**	**13**	**13**	**17**	**18**	**18**
4	6	25	**0**	**1**	**1**	**5**	**6**	**12**	**25**	**26**	**26**	**30**	**31**
5	7	30	**0**	**1**	**1**	**5**	**6**	**12**	**25**	**30**	**31**	**31**	**35**

Exercise
Write an algorithm that, given the input arrays $v[1 \cdots n]$ and $w[1 \cdots n]$, and the weight limit W, fills in the table $V[1 \cdots n; 0 \cdots W]$ and returns the maximum value $V[n, W]$ (or returns the whole table if necessary.)

Exercise
Write an algorithm that backtracks through the table created in the previous exercise and prints out a list of exactly which objects must be stolen to achieve the maximum value $V[n, W]$.

The Principle of Optimality
An optimization problem is said to satisfy the *Principle of Optimality* or to exhibit *Optimal Substructure* if an optimal solution to any (non-trivial) instance is a combination of optimal solutions to some of its subinstances. Another way to say this is that in an optimal sequence of choices, each subsequence is also optimal. The optimal substructure in the last two examples can be seen from their recurrence relations, and the reasoning that led to them.

- Coin Change Problem:
  $C[i, j] = \min(\ C[i-1, j], 1 + C[i, j - d_i]\ )$

- Discrete Knapsack Problem:
  $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i])$

Optimal substructure is necessary for a problem to be solvable by dynamic programming. Perhaps the quintessential example of optimal substructure is in the problem of finding shortest paths in a graph.

Definition
Given a graph G and $u, v \in V(G)$, a u-v *path* in G is a sequence of alternating vertices and incident edges starting at u and ending at v, in which no vertex is repeated (except possibly when $u = v$). The *length* of a path is the number of edges in the sequence. We denote by $d(u, v)$ and $l(u, v)$ the lengths of a shortest u-v path, and of a longest u-v path, respectively.

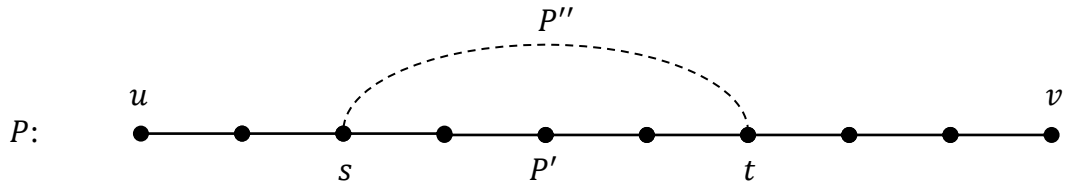
Problem
Given a graph G and $u, v \in V(G)$, determine a shortest u-v path. In particular, find (1) the length $d(u, v)$ of a shortest u-v path, and (2) the shortest path itself.

Problem
Given a graph G and $u, v \in V(G)$, determine a longest u-v path. In particular, find (1) the length $l(u, v)$ of a longest u-v path, and (2) the longest path itself.

Although these problems may appear nearly identical, one is in fact much more difficult than the other. Several efficient solutions are known for the shortest paths problem, but no efficient solution is known for longest paths.

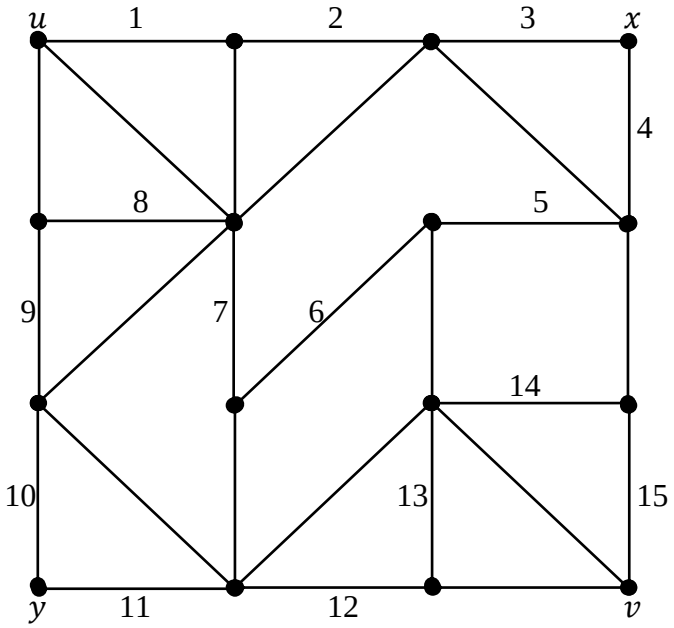
Claim
Any subsequence of a shortest path is also a shortest path.

Proof:
Let P be a shortest u-v path in G, let s and t be two intermediate vertices on P, and let P' be the subsequence of P from s to t. Assume, to get a contradiction, that P' is not a shortest s-t path. Then there exists a shorter s-t path in G, call it P''.



We can splice P' out of P and replace it by the sequence P'' to obtain a u-v path shorter than P. This contradicts that P was shortest to begin with. This contradiction shows that our assumption was false, i.e. the subsequence P' must itself be a shortest path joining its two end vertices. ■

Thus, the shortest paths problem exhibits the optimal substructure necessary for a dynamic programming solution. Indeed, most algorithms for this problem (and its variations) are based on the dynamic programming model. The following example illustrates why longest paths does not satisfy the optimality principle.



As one can easily verify $d(u, v) = 5$ while $l(u, v) = 15$. A longest u-v path in the above graph follows edges 1 through 15 in order. One also checks that $l(x, y) \geq 14$, but our longest u-v path travels from x to y in only 7 steps. Hence a subsequence of a longest path is not necessarily a longest path, and this problem violates the principle of optimality.

We now summarize the general procedure for finding a dynamic programming solution to an optimization problem.

1. Characterize the structure of an optimal solution as consisting of a combination of optimal subinstance solutions. In other words, show that solving the problem entails making some choices that leave one or more subinstances of the same problem.

2. Recursively define an optimal solution in terms of optimal subinstance solutions, i.e. write down a recurrence relation for the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion, i.e. fill in the corresponding table, either recursively or iteratively.

4. Construct an optimal solution by backtracking through the table created in (3).

What problem features indicate a dynamic programming solution? First, the problem must exhibit optimal substructure, so you *can* use dynamic programming. Second, the straightforward recursive approach entails overlapping subproblems, so you *should* use dynamic programming.

Matrix Chain Multiplication
Consider the problem of multiplying two rectangular matrices A of size $p \times q$, and B of size $q \times r$. The product AB will have size $p \times r$. Its ij^{th} entry is $\sum_{k=1}^{q} a_{ik} b_{kj}$ which involves q multiplications for each i and j satisfying $1 \leq i \leq p$ and $1 \leq j \leq r$. Thus, the total number of multiplications performed in computing AB is pqr.

Now consider the product of three matrices A, B and C of sizes $p \times q$, $q \times r$ and $r \times s$ respectively. There are two possible parenthesizations of this product:

(1) $A(BC)$ with number of multiplications $= pqs + qrs$

or

(2) $(AB)C$ with number of multiplications $= pqr + prs$

For instance, if $p = 10$, $q = 100$, $r = 10$ and $s = 100$, then (1) yields 200,000 multiplications and (2) yields 20,000. It therefore pays to choose the parenthesization carefully.

Exercise
Write the 5 parenthesizations of the product of 4 matrices. Determine the number of parenthesizations of the product of 5 matrices, then write them.

Exercise
Let $P(n)$ denote the number of distinct parenthesizations of a product of n matrices. Show that $P(n)$ satisfies the recurrence

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n \geq 2 \end{cases}$$

It is a challenging exercise to show, using the above recurrence, that $P(n) = \frac{1}{n}\binom{2n-2}{n-1}$. The sequence $C_n = P(n+1)$ is known as the *Catalan Numbers*. See https://en.wikipedia.org/wiki/Catalan_number to learn more about this interesting sequence, and for hints on how to prove the above identity.

Exercise
Use Stirling's formula and the above identity to show $P(n) = \Theta\left(\frac{4^n}{n^{3/2}}\right)$

Problem Matrix Chain Multiplication
Given $n \geq 1$ and a *matrix-chain product* $A_1 A_2 \cdots A_n$, where A_i has size $p_{i-1} \times p_i$ $(1 \leq i \leq n)$, determine a parenthesization of the product that minimizes the total number of multiplication operations performed. As usual there are really two problems.

- Find the value of an optimal solution, the minimum number of multiplications necessary
- Find an optimal solution itself, i.e. a particular parenthesization that achieves the minimum.

We can see from the last exercise that the brute force approach is highly inefficient, and intractable for large values of n.

To develop a dynamic programming solution, we consider the general subproblem of finding an optimal parenthesization of the product $A_i \cdots A_j$ where $1 \leq i \leq j \leq n$. Observe that any parenthesization will split $A_i \cdots A_j$ into two top-level subproducts

$$(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$$

for some k in the range $i \leq k < j$.

Claim
If the product $A_i \cdots A_j$ is optimally parenthesized, then so are both factors $(A_i \cdots A_k)$ and $(A_{k+1} \cdots A_j)$.

$$\underset{p_{i-1} \times p_k}{\text{optimal}} \quad \underset{p_k \times p_j}{\text{optimal}} \quad \underset{p_{i-1} \times p_j}{\text{optimal}}$$
$$(A_i \cdots \cdots A_k) \cdot (A_{k+1} \cdots \cdots A_j) = A_i \cdots \cdots \cdots A_j$$

Proof:
Assume, to get a contradiction, that the parenthesization of $(A_i \cdots A_k)$ is not optimal. Then we can replace it with an optimal one, yielding a parenthesization of $A_i \cdots A_j$ with fewer multiplication operations, contradicting that the original parenthesization of $A_i \cdots A_j$ was optimal. Likewise, if $(A_{k+1} \cdots A_j)$ is not optimally parenthesized, we reach a similar contradiction. Therefore, both factors must be optimally parenthesized, if $A_i \cdots A_j$ is. ■

Exercise
Show that the converse of the above claim is false, i.e. show by example that it is possible to concatenate two optimal parenthesizations to produce a non-optimal one.

Now that we know this problem exhibits the required optimal substructure, we let $m[i,j]$ denote the minimum number of multiplications necessary to compute the product $A_i \cdots A_j$, for $1 \leq i \leq j \leq n$. Our

goal is to find the value $m[1,n]$. Note that if $i=j$, then the "product" has only one factor, and therefore $m[i,j]=0$. If $i \le k < j$, and k is the split position of an optimal parenthesization, then by the above discussion

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j.$$

Thus we add the cost of optimally computing the products $(A_i \cdots \cdots A_k)$ and $\left(A_{k+1} \cdots \cdots A_j\right)$, to the cost of a single matrix product on sizes $p_{i-1} \times p_k$ and $p_k \times p_j$. Since position k is unknown, we define

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \right) & i < j \end{cases}$$

Observe cell $m[i,j]$ depends on the cells $m[i;\ i \cdots (j-1)]$ and $m[(i+1) \cdots j;\ j]$. To systematically compute the full table, we initialize the main diagonal to 0, then successively fill each off diagonal above the main by using the above formula.

$$m[i,i] = 0 \qquad (1 \le i \le n)$$

$$m[i,i+1] = \min\{m[i,i] + m[i+1,i+1] + p_{i-1}p_i p_{i+1}\} \qquad (1 \le i \le n-1)$$

$$m[i,i+2] = \min \begin{Bmatrix} m[i,i] + m[i+1,i+2] + p_{i-1}p_i p_{i+2} \\ m[i,i+1] + m[i+2,i+2] + p_{i-1}p_{i+1}p_{i+2} \end{Bmatrix} \qquad (1 \le i \le n-2)$$

$$m[i,i+3] = \min \begin{Bmatrix} m[i,i] + m[i+1,i+3] + p_{i-1}p_i p_{i+3} \\ m[i,i+1] + m[i+2,i+3] + p_{i-1}p_{i+1}p_{i+3} \\ m[i,i+2] + m[i+3,i+3] + p_{i-1}p_{i+2}p_{i+3} \end{Bmatrix} \qquad (1 \le i \le n-3)$$

$$\vdots$$

$$m[i,i+q] = \cdots \text{minimum of } q \text{ numbers} \cdots \qquad (1 \le i \le n-q)$$

Once the table is filled for $1 \le i \le j \le n$, we return the value $m[1,n]$. Note that the values below the main diagonal remain undefined.

Exercise
Write an algorithm that takes as input the array $p[0 \cdots n]$ of matrix sizes, fills the table $m[i,j]$ as described above for $1 \le i \le j \le n$, then returns $m[1,n]$.

Example $n = 5$ and $p[0 \cdots 5] = (10, 20, 30, 10, 40, 10)$

		j 1	2	3	4	5
i	1	**0**	**6000**	**8000**	**12000**	**13000**
	2	-	**0**	**6000**	**14000**	**12000**
	3	-	-	**0**	**12000**	**7000**
	4	-	-	-	**0**	**4000**
	5	-	-	-	-	**0**

For instance, to compute $m[2,5]$ in this example:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 7000 + 6000 & = 13000 \quad (k = 2) \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 6000 + 4000 + 2000 & = 12000 \quad (k = 3) * \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 14000 + 0 + 8000 & = 22000 \quad (k = 4) \end{cases}$$

The minimum value occurs at the split position $k = 3$, so $m[2,5] = 12000$.

To construct the optimal parenthesization itself, we need to know for each cell in the above table, the split points (or k-values) at which the minimum occurred. The most efficient way to do this is to simultaneously construct a parallel table $s[i,j]$ (for $1 \leq i < j \leq n$) that containing the split points.

Exercise
Modify the algorithm you wrote in the last exercise to also fill $s[i,j]$ ($1 \leq i < j \leq n$), the table of k-values at which the minima occur. (See page 375 of CLRS for the complete solution.)

Exercise
Write an algorithm that backtracks through the table $s[i,j]$ ($1 \leq i < j \leq n$) and prints out the optimal parenthesization of the matrix product $A_1 \cdots \cdots A_n$. (See page 377 of CLRS for the solution.)

Example $n = 5$ and $p[0 \cdots 5] = (10, 20, 30, 10, 40, 10)$
We found $s[2,5] = 3$ above. The remaining cells for the table of split points are computed similarly.

		j				
		1	2	3	4	5
i	1	*	**1**	**1**	**3**	**3**
	2	-	*	**2**	**3**	**3**
	3	-	-	*	**3**	**3**
	4	-	-	-	*	**4**
	5	-	-	-	-	*

From this table we can construct the optimal parenthesization: $\big(A_1(A_2A_3)\big)(A_4A_5)$. ■

Further Reading
- All-Pairs Shortest-Paths problem (section 25.2 in CLRS)
- Longest Common Subsequence problem (section 15.4 in CLRS)
- Optimal Binary Search Trees problem (section 15.5 in CLRS)