[Chapter 3]

Dynamic Programming (Full)

Dynamic Programming

- An instance of a problem into one or more smaller instances, like DAC
  - Solve small instances first.
  - Store the results.
  - Reuse the stored results, instead of re-computing.
- Bottom-up approach, unlike DAC.
  - Establish a recursive property that gives the solution to an instance of the problem.
  - Solve an instance of a problem in a *bottom-up* fashion by solving smaller instances first.

The Binomial Coefficient

- $$\begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k!(n-k)!} \text{ for } 0 \le k \le n$$

- Pascal's Triangle

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

Algorithm 3.1 Binomial Coefficient using Divide-and-Conquer

Problem: Compute the binomial coefficient $\begin{bmatrix} n \\ k \end{bmatrix}$

Inputs: nonnegative integers *n* and *k*, where $k \leq n$.

Outputs: *bin*, the binary coefficient of *n* and *k*.

```
void bin (int n, int k) {
        if ( k == 0 || n == k)
            return 1;
        else
            return bin(n – 1, k – 1) + bin(n – 1, k);
    }
```

Time Complexity for Algorithm 3.1

Basic operation: the number of terms to compute.

Input size: n k .

$$T(n,k) = 2\begin{bmatrix} n \\ k \end{bmatrix} - 1$$

Proof by Induction:

Very inefficient!!!

Proof by Induction

$$T(n,k) = 2\begin{bmatrix} n \\ k \end{bmatrix} - 1$$

Induction Basis: If n = 1, $2\begin{bmatrix} n \\ k \end{bmatrix} - 1 = 2 \times 1 - 1 = 1$

Induction Hypothesis: Suppose the above formula is true.

Induction Step: Compute $\begin{bmatrix} n+1 \\ k \end{bmatrix}$.

$$\begin{aligned}
&2\begin{bmatrix} n \\ k-1 \end{bmatrix} - 1 + 2\begin{bmatrix} n \\ k \end{bmatrix} - 1 + 1 \\
&= 2\left(\tfrac{n!}{(k-1)!(n-k+1)!} + \tfrac{n!}{k!(n-k)!}\right) - 1 \\
&= 2\left(\tfrac{n!(k+n+1-k)}{k!(n+1-k)!}\right) - 1 \\
&= 2\left(\tfrac{n!(n+1)}{k!(n+1-k)!}\right) - 1 \\
&= 2\left(\tfrac{(n+1)!}{k!(n+1-k)!}\right) - 1 \\
&= 2\begin{bmatrix} n+1 \\ k \end{bmatrix} - 1
\end{aligned}$$

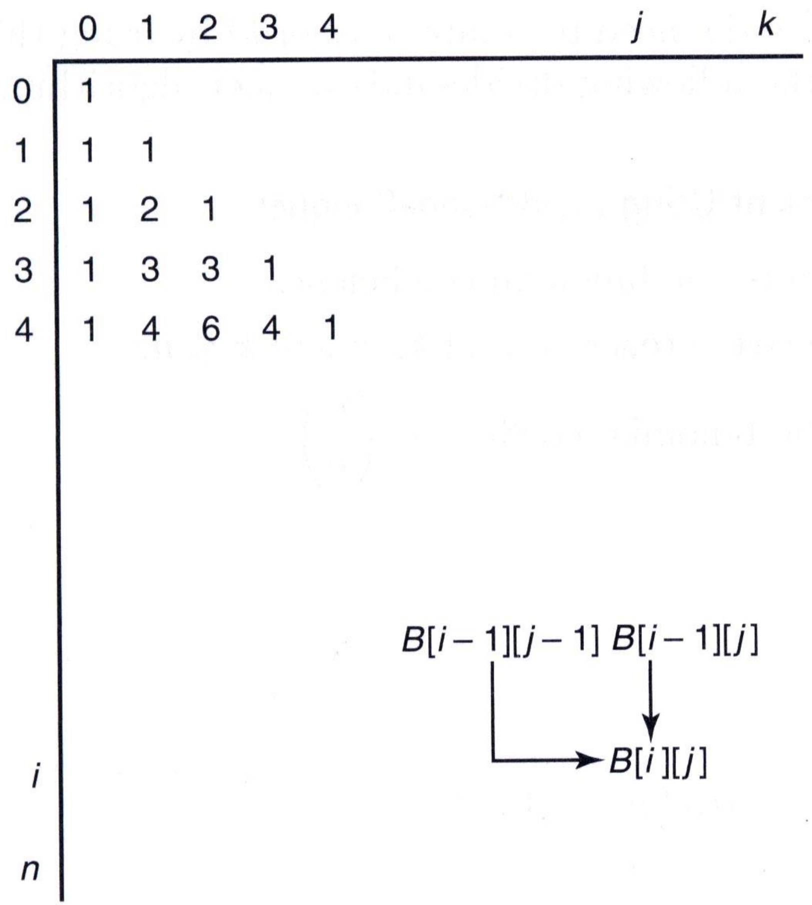
Dynamic Programming for Binomial Coefficient

- *Establish* a recursive property

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & \text{if } 0 < j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \end{cases}$$

- *Solve* an instance in a bottom-up fashion
  - Solve, store and keep going until we get to the point by reusing the stored results.
  - See Fig. 3.1

Figure 3.1 The array *B* used to compute the binomial coefficient.



Algorithm 3.2 Binomial Coefficient using Dynamic Programming

Problem: Compute the binomial coefficient $\begin{bmatrix} n \\ k \end{bmatrix}$

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: *bin2*, the binary coefficient of n and k.

```
void  bin2 (int n, int k) {
     index i, j;
     int B[0..n][0..k];
     for ( i = 0; i <= n; i++)
              for ( j = 0; j <= min(i, k); j++)
                 if (j == 0 || j == i) B[i][j] = 1;
                 else B[i][j] =B[i – 1][j – 1] + B[i – 1][j];
      return B[n][k];
}
```

Time Complexity for Algorithm 3.2

Basic operation: the number of terms to compute.

Input size: $n\ k$.

$$1+2+3+\cdots+k+\overbrace{(k+1)+\cdots+(k+1)}^{n-k+1\text{ times}} = \frac{k(k+1)}{2}+(n-k+1)(k+1)$$

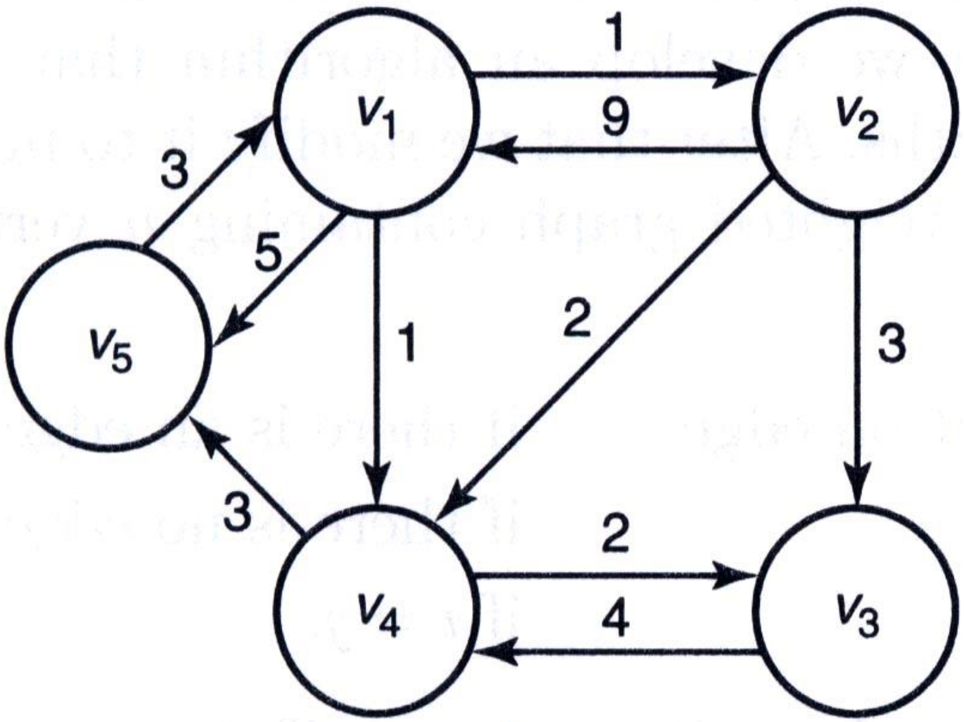
$$= \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$$

Very good!!!

Graph Theory

- Glossary
  - Graph consists of two elements: $G = (V, E)$.
  - E is a set of edges. Every edge has two endpoints in V.
  - If an edge in E can be defined as a set of ordered pairs, G is a directed graph or *digraph* in short.
  - If the edges have values associated with them, the values are called *weights* and G is a weighted graph.
  - In a digraph, a *path* is a sequence of vertices such that there is an edge from each vertex to its successor.
  - A path from a vertex to itself is called a *cycle*.
  - If G contains a cycle, G is *cyclic*; otherwise, it is *acyclic*.
  - A path is *simple*, if it never passes through the same vertex twice.
  - A *length* of a path in a weighted graph is the sum of the weights on the path.

Example: A weighted, directed graph.



Floyd's algorithm for Shortest Paths Problem

Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices.

Inputs: A weight digraph and *n,* the number of vertices. W[*i*][*j*] is the weight on the edge from the *i*-th vertex to the *j*-th vertex.

Outputs: A two dimensional array D, which has both its rows and columns indexed from 1 to n, where D[*i*][*j*] is the length of a shortest path from the *i*-th vertex to the *j*-th vertex.

The shortest paths problem is **an optimization problem**, which is to find a solution with an optimal value among multiple solutions to an instance of a problem.

Brute-force algorithm for Shortest Paths

- Strategy
  - Find all possible paths, compute their lengths, and select the path with a minimal length.
- Analysis
  - Suppose there are n vertices in the graph.
  - The total number of paths from v_i to v_j is $(n\text{-}2)!$.
  - This is much worse than exponential.
- Our goal is to find a more efficient algorithm.
  - Let's apply DP strategy instead.
  - The DP algorithm for SP is formed by Robert Floyd in 1962.
  - But it is essentially same as the algorithm by Bernard Roy in 1959, and by Stephen Warshall in 1962. for finding a transitive closure.
  - Floyd-Warshall algorithm.

Dynamic Programming Strategy for Shortest Paths

- Adjacency matrix representation

$$W[i][j] = \begin{cases} weight & \text{If there is an edge from } v_i \text{ to } v_j \\ \infty & \text{If there is no edge from } v_i \text{ to } v_j \\ 0 & \text{If } i = j. \end{cases}$$

- Distance matrix for establishing recursive property

$$D^{(k)}[i][j] = \{v_1, v_2, \ldots, v_k\}$$

  - Length of a shortest path from v_i to v_j using only vertices in the set

    $$\{v_1, v_2, \ldots, v_k\}$$

    as intermediate vertices.

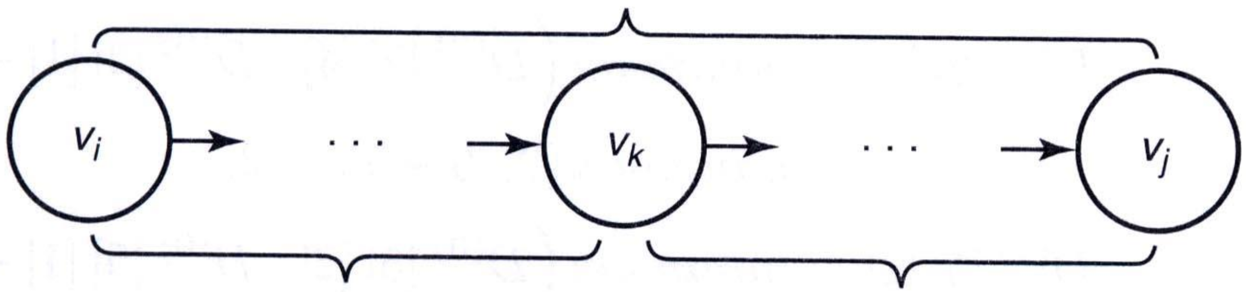
Designing an algorithm for Shortest Paths

- Establish a recursive property

$$D^{(k)}[\mathrm{i}][\mathrm{j}] = minimum(\underbrace{D^{(k-1)}[i][j]}_{Case1}, \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{Case2})$$

  - Case 1: At least one shortest path from v_i to v_j, using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, does not use v_k. Then $D^{(k)}[\mathrm{i}][\mathrm{j}] = D^{(k\text{-}1)}[\mathrm{i}][\mathrm{j}]$.
    - (e.g.) $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$
  - Case 2: All shortest paths from v_i to v_j, using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, do use v_k.

Case 2: The shortest path uses V_k.



$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

Figure 3.3 W represents the graph in Figure 3.2 and D contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in *D* from those in *W*.

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Floyd's algorithm I

- Algorithm

```
void floyd(int n, const number W[][],
                          number D[][]) {
   int i, j, k;
   D = W;
   for(k=1; k <= n; k++)
      for(i=1; i <= n; i++)
         for(j=1; j <= n; j++)
                D[i][j] =
minimum(D[i][j], D[i][k]+D[k][j]);
```

- Every-Case Time Complexity

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

Floyd's algorithm II

- Problem: Same as in Floyd's algorithm I, except shortest paths are also created.
- Additional outputs: an array P, which has both its rows and columns indexed from 1 to *n*, where

$$P[i][j] = \begin{cases} \text{Highest index of an intermediate vertex on the shortest path from } v_i \text{ to } v_j\text{, if at least one intermediate vertex exists.} \\ \text{0, if no intermediate vertex exists.} \end{cases}$$

Floyd’s algorithm II

```
void floyd2(int n, const number W[][],
        number D[][], index P[][]) {
   index i, j, k;
   for(i=1; i <= n; i++)
     for(j=1; j <= n; j++)
         P[i][j] = 0;
   D = W;
   for(k=1; k<= n; k++)
     for(i=1; i <= n; i++)
        for(j=1; j<=n; j++)
          if (D[i][k]+D[k][j] < D[i][j]) {
              P[i][j] = k;
              D[i][j] = D[i][k] + D[k][j];
          }
}
```

Figure 3.5 The array *P* produced when Algorithm 3.4 is applied to the graph on Figure 3.2.

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Print Shortest Path

```
void path(index q,r) {
   if (P[q][r] != 0) {
      path(q,P[q][r]);
      cout << " v" << P[q][r];
      path(P[q][r],r);
   }
}
 (e.g.) Using P, solve path(5, 3)
        path(5,3) = 4
            path(5,4) = 1
                path(5,1) = 0
                v1
                path(1,4) = 0
            v4
            path(4,3) = 0
```

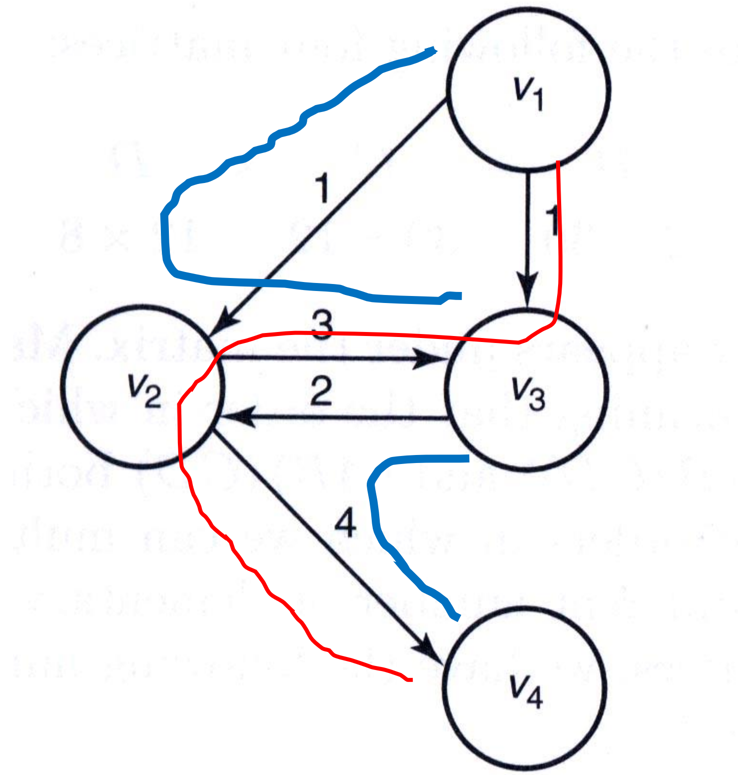
Result: `v1 v4`.

I.e., the shortest path from v_5 to v_3 is v_5, v_1, v_4, v_3.

The Principle of Optimality

- The principle of optimality is said to apply in a problem if an optimal solution to an instance of a problem always contains optimal solutions to all substances.
  - Although it may seem that any optimization problem can be solved using dynamic programming, this is not the case.
  - The principle of optimality must apply in the problem.
- Longest Paths problem is to find the longest simple paths from each vertex to all other vertices.
  - Can we solve the problem using dynamic programming?

Figure 3.6 A weighted, directed graph with a cycle.



Chained Matrix Multiplication

- In general, to multiply an $i \times j$ matrix times a $j \times k$ matrix using the standard method, it is necessary to do $i \times j \times k$ elementary multiplications.
- (e.g.) $A_1 \times A_2 \times A_3$.
  - Suppose A_1 is 10×100, A_2 is 100×5, and A_3 is 5×50.
  - $(A_1 \times A_2) \times A_3$

    $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7{,}500$
  - $A_1 \times (A_2 \times A_3)$

    $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75{,}000$

Chained Matrix Multiplication

- Brute-force algorithm
  - Consider all possible orders and take the minimum.
  - Let t_n be the number of different orders in which we can multiply n matrices: $A_1, A_2, \ldots, A_n$.
  - $(A_1 \ldots A_{n-1})\, A_n$ will have t_{n-1} different orders.
  - $A_1\, (A_2 \ldots A_n)$ will have t_{n-1} different orders.
  - In other words, $t_n \geq t_{n-1} + t_{n-1} = 2\, t_{n-1}$ and $t_2 = 1$.
  - Therefore, $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$

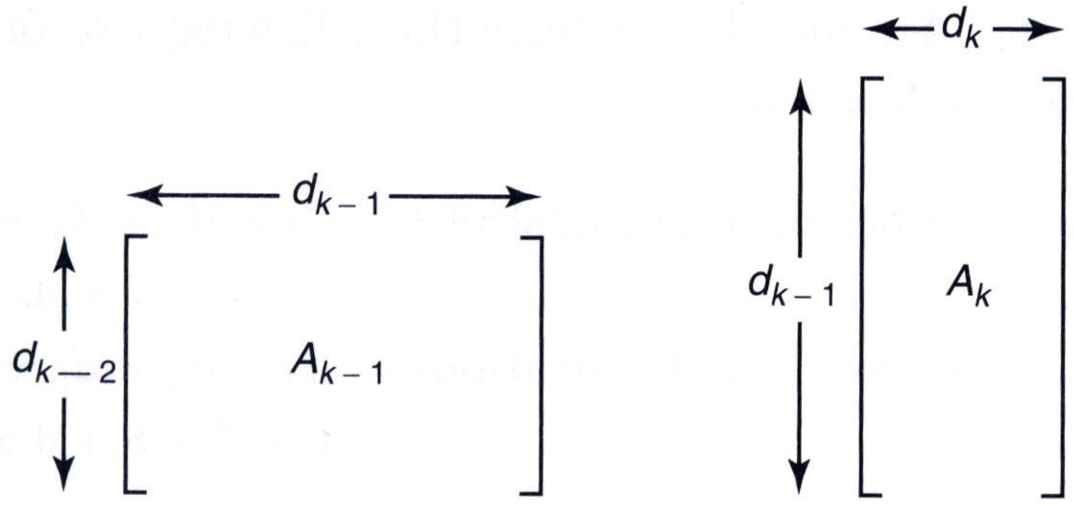
Chained Matrix Multiplication

- Dynamic Programming Design
  - Let d_k be the number of columns in A_k for $1 \le k \le n$.
  - Let d_0 be the number of rows in A_1.
  - In other words, $A_1\, A_2 \ldots A_n$ will have be represented as $d_0 \times d_1 \times \ldots \times d_n$.
  - Suppose $1 \le i \le j \le n$.
  - $M[i][j]$ = minimum number of multiplications needed to multiply A_i through A_j, if $i < j$.

    $$MIN_{i \le k \le j-1}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j)$$

  - $M[i][i]$ = 0.

Figure 3.7 The number of columns in A_{k-1} is the same as the number of rows in A_k.



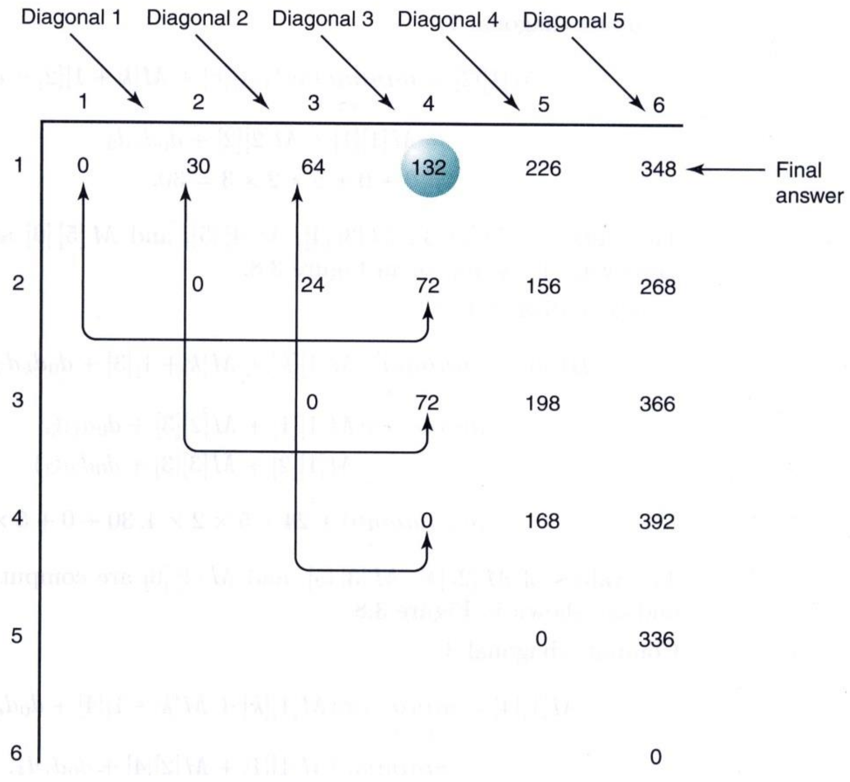
Example 3.5: Solving the recursive formula.

$$\begin{array}{cccccc} A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\ 5\times 2 & 2\times 3 & 3\times 4 & 4\times 6 & 6\times 7 & 7\times 8 \end{array}$$

$$\begin{aligned} M[4][6] &= minimum_{4\le k\le 5}(M[4][4]+M[5][6]+4\times 6\times 8, M[4][5]+M[6][6]+4\times 7\times 8) \\ &= minimum(0+6\times 7\times 8+4\times 6\times 8, 4\times 6\times 7+0+4\times 7\times 8) \\ &= minimum(528,392)=392 \end{aligned}$$

$M[i][j]$	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Figure 3.8 The array M developed in Example 3.5. M [1] [4], which is circled, is computed from the pairs of entries indicated.



Minimum Multiplication

- Problem: Determine the minimum number of multiplications needed to multiply n matrices and an order that produces that minimum number.
- Inputs: The number of matrices n, and an array of integers d_k, indexed from 0 to n, where $d_{i-1} \times d_i$ is the dimension of the i-th matrix.
- Outputs: the minimum number of elementary multiplications needed to multiply the n matrices; a two-dimensional array P from which the optimal order can be obtained. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying the matrices.
- See Algorithm 3.6 in p. 113.
- Check if the principle of optimality works for this case.

Minimum Multiplication Algorithm

```
int minmult(int n, const int d[], index P[][]) {
   index i, j, k, diagonal;
   int M[1..n, 1..n];
      for(i=1; i <= n; i++)
        M[i][j] = 0;
      for(diagonal = 1; diagonal <= n-1; diagonal++)
        for(i=1; i <= n-diagonal; i++) {
           j = i + diagonal;
           M[i][j] = minimum(M[i][k]+M[k+1][j]+
                  d[i-1]*d[k]*d[j]);
                           where i <= k <= j-1
           P[i][j] = a value of k that gave the min;
      }
      return M[1][n];
}
```

Figure 3.9 The array P produced when Algorithm 3.6 is applied to the dimensions in Example 3.5.

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

$P[1][6] = 1$

$(A_1((((A_2\, A_3)\, A_4)\, A_5)\, A_6)).$

Every-Case Time Complexity: Minimum Multiplication

- Basic operation: The instructions executed for each value of n. Included is a comparison to test for the minimum.
- Input size: n, the number of matrices to be multiplied.
- Analysis:
  - $j = i + diagonal$.
  - For a given values of $diagonal$ and i, the number of passes through the k-loop =

    $$(j-1)-i+1 = i + diagonal - 1 - i + 1 = diagonal$$
  - For a given values of $diagonal$, the number of passes through the i-loop = $n - diagonal$
  - Therefore,

$$\sum_{diagonal=1}^{n-1} [(n-diagonal)\times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Comments: Minimum Multiplication

- See Algorithm 3.7, which is to print the optimal order for multiplying n matrices.
  - $Order(i, j)$ prints the optimal order for multiplying $A_i \times \ldots \times A_j$ with parentheses.
- Our algorithm $\Theta(n^3)$ for chained matrix multiplication is from Godbole (1973).
- Other algorithms
  - Yao(1982) - $\Theta(n^2)$ by speeding up certain dynamic programming solutions.
  - Hu and Shing(1982, 1984) - $\Theta(n \lg n)$