Introduction to

The Design and Analysis of Algorithms

3rd Edition

6

Transform-and-Conquer

This chapter deals with a group of design methods that are based on the idea of transformation(轉換). We call this general technique *transform-and-conquer* because these methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution(更容易解決). Then, in the second or conquering stage, it is solved.

There are three major variations of this idea that differ by what we transform a given instance to (Figure 6.1):

- Transformation to a simpler or more convenient instance of the same problem—we call it *instance simplification* (實例簡化).
- Transformation to a different representation of the same instance—we call it *representation change* (表示改變).
- Transformation to an instance of a different problem for which an algorithm is already available—we call it *problem reduction* (問題簡化).

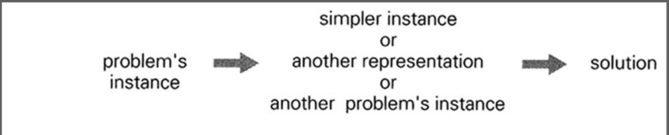


FIGURE 6.1 Transform-and-conquer strategy.

The chapter concludes with a review of several applications of the third variety of transform-and-conquer: problem reduction. This variety should be considered the most radical of the three: one problem is reduced to another, i.e., transformed into an entirely different problem. This is a very powerful idea, and it is extensively used in the complexity theory (Chapter 11). Its application to designing practical algorithms is not trivial, however. First, we need to identify a new problem into which the given problem should be transformed.

6.1 Presorting

EXAMPLE 1 *Checking element uniqueness in an array*
If this element uniqueness problem looks familiar to you, it should; we considered a brute-force algorithm for the problem in Section 2.3 (see Example 2). The brute-force algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in $\Theta(n^2)$.

Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other, and vice versa.

```
ALGORITHM    PresortElementUniqueness(A[0..n − 1])
    //Solves the element uniqueness problem by sorting the array first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Returns "true" if A has no equal elements, "false" otherwise
    sort the array A
    for i ←0 to n − 2 do
        if A[i]= A[i + 1] return false
    return true
```

The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements. Since the former requires at least $n \log n$ comparisons and the latter needs no more than $n - 1$ comparisons, it is the sorting part that will determine the overall efficiency of the algorithm. So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one. But if we use a good sorting algorithm, such as mergesort, with worst-case efficiency in $\Theta(n \log n)$, the worst-case efficiency of the entire presorting-based algorithm will be also in $\Theta(n \log n)$:

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

EXAMPLE 2 *Computing a mode*

A *mode* is a value that occurs most often in a given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.) The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list. On each iteration, the ith element of the original list is compared with the values already encountered by traversing this auxiliary list.

If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

It is not difficult to see that the worst-case input for this algorithm is a list with no equal elements. For such a list, its *i*th element is compared with *i* − 1 elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1. As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is

$$C(n) = \sum_{i=1}^{n}(i-1) = 0+1+\cdots+(n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The additional *n* − 1 comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of the algorithm.

As an alternative, let us first sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

```
ALGORITHM    PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode
    sort the array A
    $i \leftarrow 0$                              //current run begins at position i
    $modefrequency \leftarrow 0$    //highest frequency seen so far
    while $i \le n - 1$ do
        $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
        while $i + runlength \le n - 1$ and $A[i + runlength] = runvalue$
            $runlength \leftarrow runlength + 1$
```

```
        if $runlength > modefrequency$
            $modefrequency \leftarrow runlength$;    $modevalue \leftarrow runvalue$
        $i \leftarrow i + runlength$
    return modevalue
```

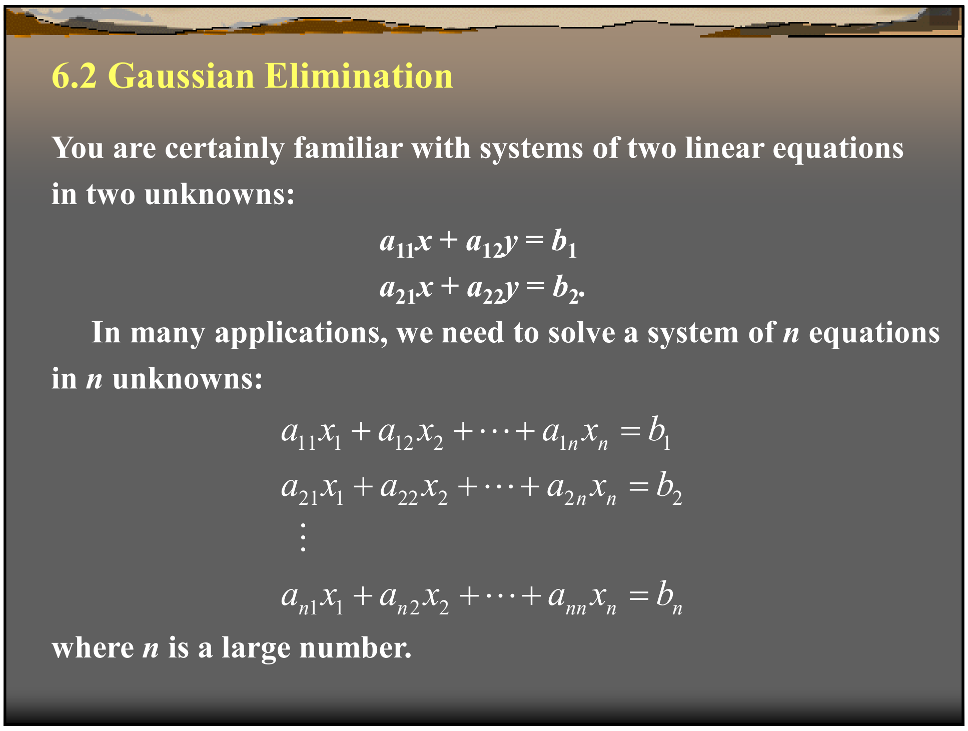
The analysis here is similar to the analysis of Example 1: the running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time (why?). Consequently, with an $n \log n$ sort, this method's worst-case efficiency will be in a better asymptotic class than the worstcase efficiency of the brute-force algorithm.

EXAMPLE 3 ***Searching problem***

Consider the problem of searching for a given value *v* in a given array of *n* sortable items. The brute-force solution here is sequential search (Section 3.1), which needs *n* comparisons in the worst case. If the array is sorted first, we can then apply binary search, which requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case. Assuming the most efficient *n* log *n* sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

which is inferior to sequential search. The same will also be true for the averagecase efficiency. Of course, if we are to search in the same list more than once, the time spent on sorting might well be justified. (Problem 4 in this section's exercises asks to estimate the minimum number of searches needed to justify presorting.)



6.2 Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2.$$

In many applications, we need to solve a system of *n* equations in *n* unknowns:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

where *n* is a large number.

Fortunately, there is a much more elegant algorithm for solving systems of linear equations called *Gaussian elimination*.[2] The idea of Gaussian elimination is to transform a system of *n* linear equations in *n* unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal:

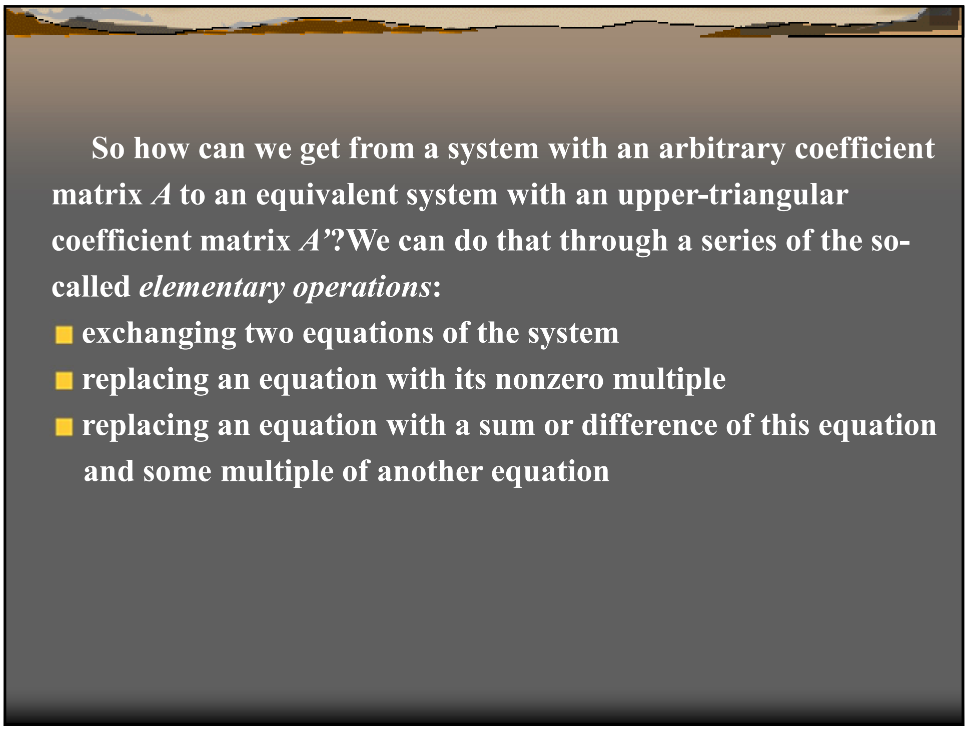
$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \quad \Longrightarrow \quad \begin{array}{r} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n = b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2 \\ \vdots \\ a'_{nn}x_n = b'_n. \end{array}$$

In matrix notations, we can write this as

$$Ax = b \quad \Rightarrow \quad A'x = b',$$

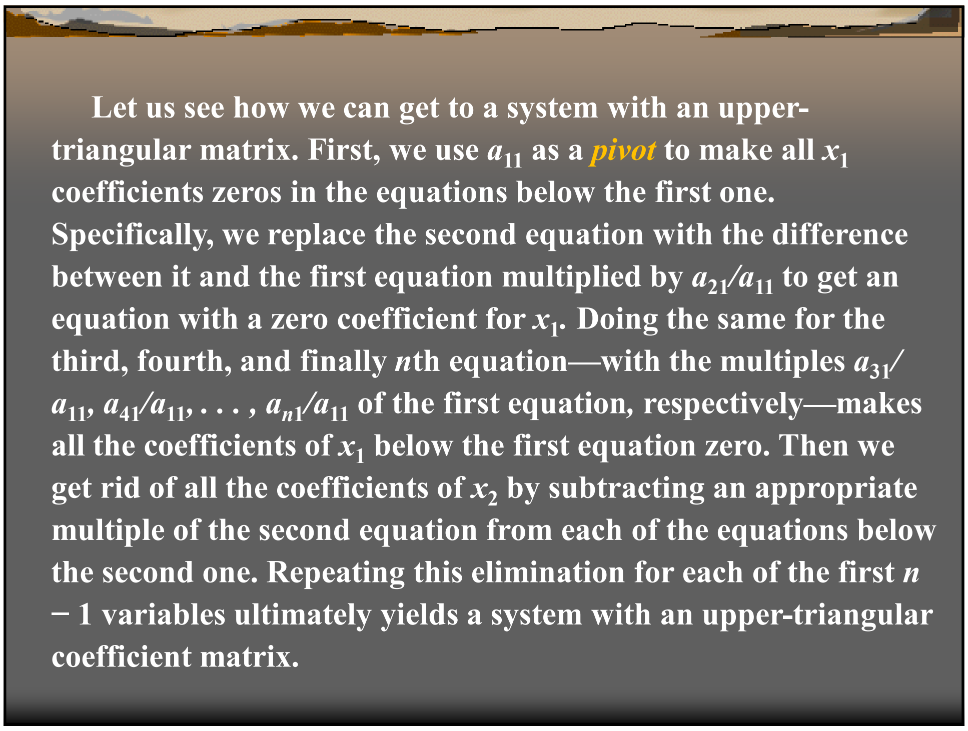
where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$



So how can we get from a system with an arbitrary coefficient matrix *A* to an equivalent system with an upper-triangular coefficient matrix *A'*?We can do that through a series of the so-called *elementary operations*:

- exchanging two equations of the system
- replacing an equation with its nonzero multiple
- replacing an equation with a sum or difference of this equation and some multiple of another equation



Let us see how we can get to a system with an upper-triangular matrix. First, we use a_{11} as a *pivot* to make all x_1 coefficients zeros in the equations below the first one. Specifically, we replace the second equation with the difference between it and the first equation multiplied by a_{21}/a_{11} to get an equation with a zero coefficient for x_1. Doing the same for the third, fourth, and finally *n*th equation—with the multiples a_{31}/a_{11}, a_{41}/a_{11}, . . . , a_{n1}/a_{11} of the first equation, respectively—makes all the coefficients of x_1 below the first equation zero. Then we get rid of all the coefficients of x_2 by subtracting an appropriate multiple of the second equation from each of the equations below the second one. Repeating this elimination for each of the first $n - 1$ variables ultimately yields a system with an upper-triangular coefficient matrix.

EXAMPLE 1 Solve the system by Gaussian elimination.

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0. \end{aligned}$$

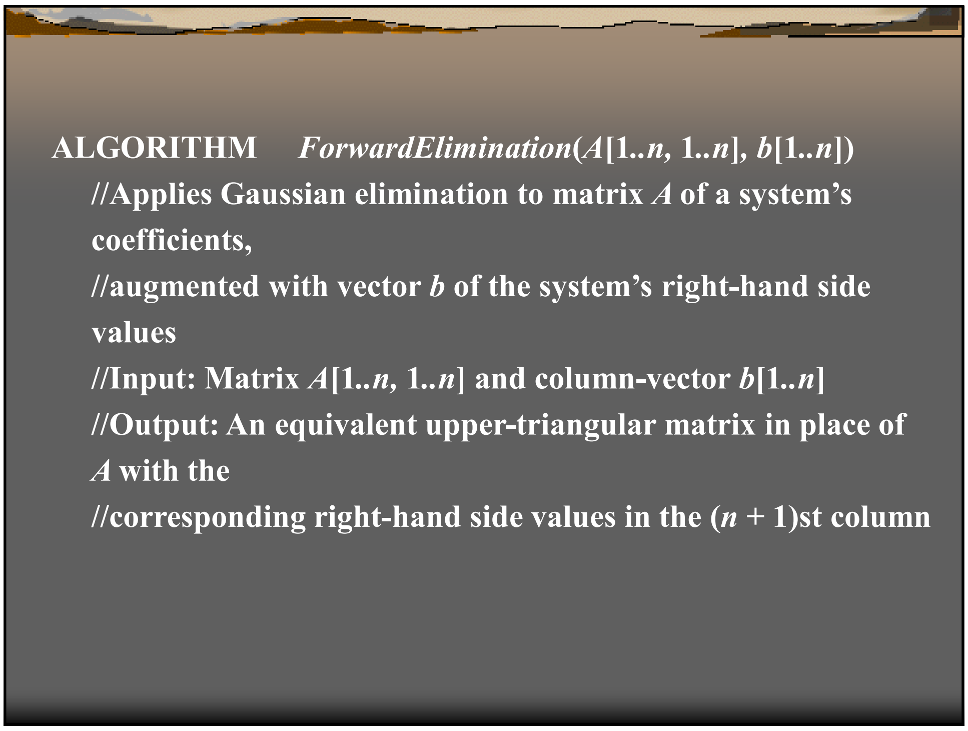
$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{array}{l} \\ \text{row } 2 - \tfrac{4}{2}\text{ row } 1 \\ \text{row } 3 - \tfrac{1}{2}\text{ row } 1 \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \tfrac{3}{2} & \tfrac{1}{2} & -\tfrac{1}{2} \end{bmatrix} \begin{array}{l} \\ \\ \text{row } 3 - \tfrac{1}{2}\text{ row } 2 \end{array}$$

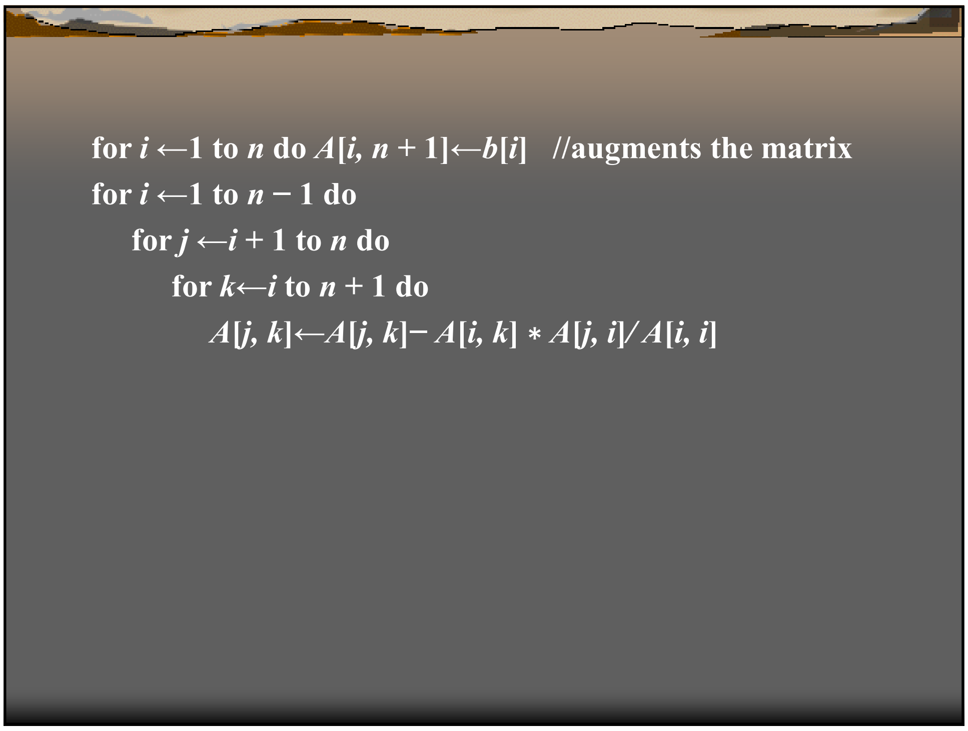
$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Now we can obtain the solution by back substitutions: $x_3 = (-2)/2 = -1$, $x_2 = (3 - (-3)x_3)/3 = 0$, and $x_1 = (1 - x_3 - (-1)x_2)/2 = 1$.

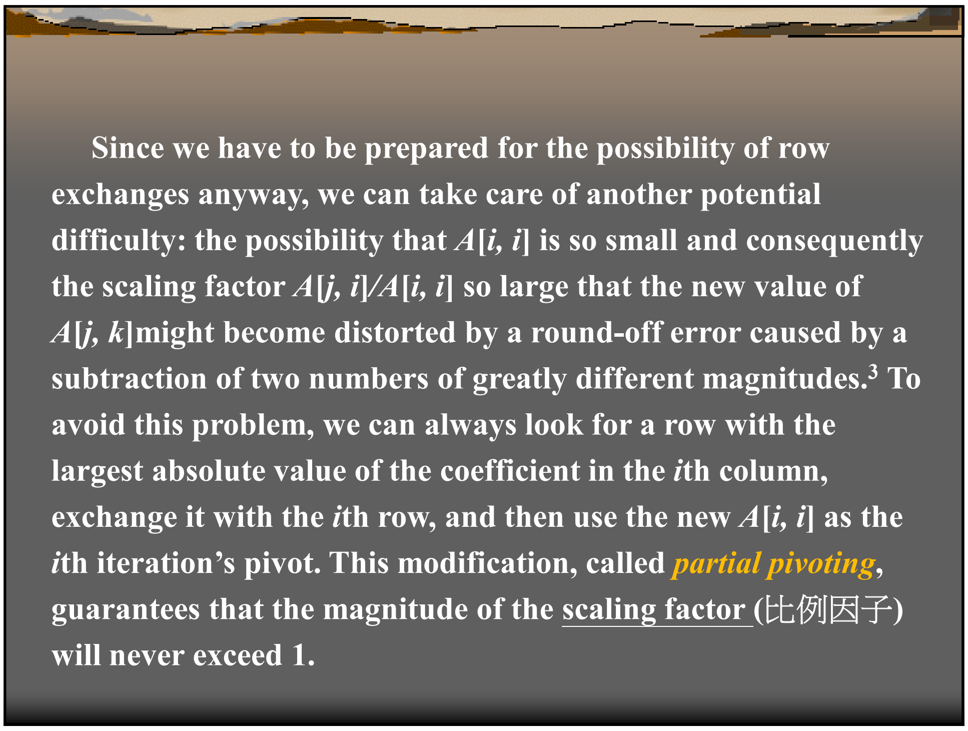
Here is pseudocode of the first stage, called *forward elimination*, of the algorithm.



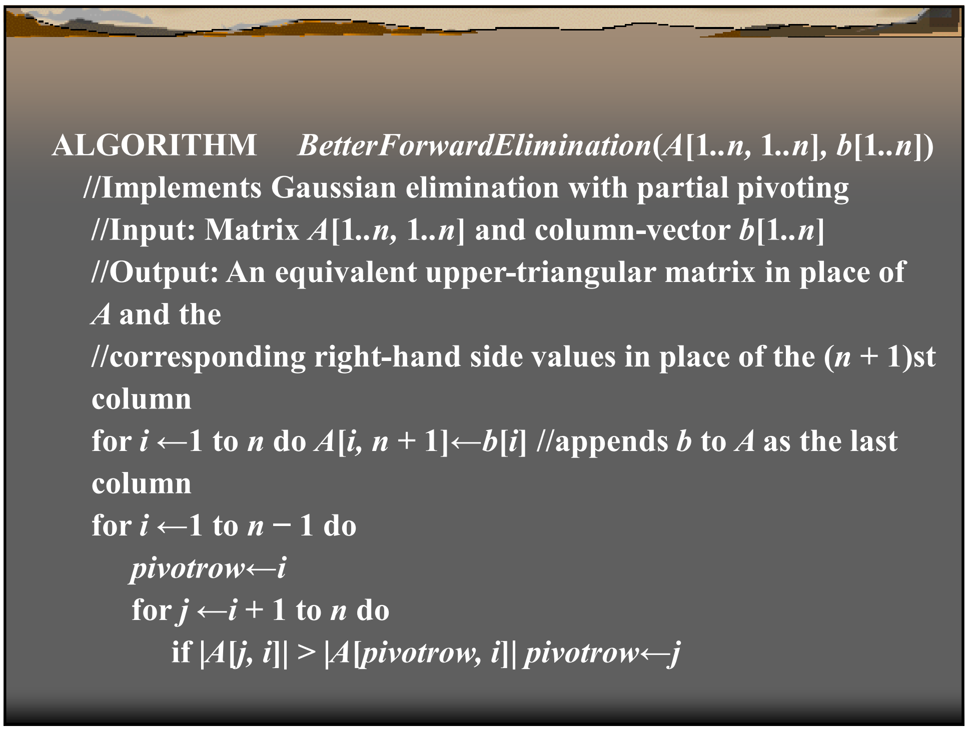
```
ALGORITHM    ForwardElimination($A[1..n, 1..n], b[1..n]$)
    //Applies Gaussian elimination to matrix $A$ of a system's coefficients,
    //augmented with vector $b$ of the system's right-hand side values
    //Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ with the
    //corresponding right-hand side values in the $(n + 1)$st column
```



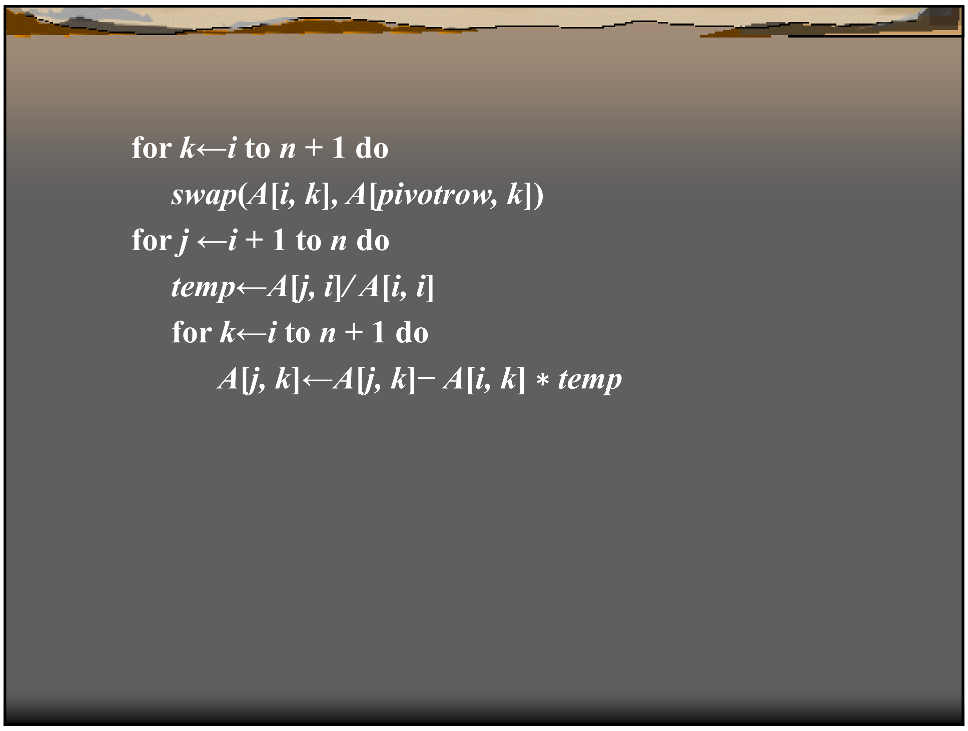
```
    for $i \leftarrow 1$ to $n$ do $A[i, n + 1] \leftarrow b[i]$    //augments the matrix
    for $i \leftarrow 1$ to $n - 1$ do
        for $j \leftarrow i + 1$ to $n$ do
            for $k \leftarrow i$ to $n + 1$ do
                $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$
```



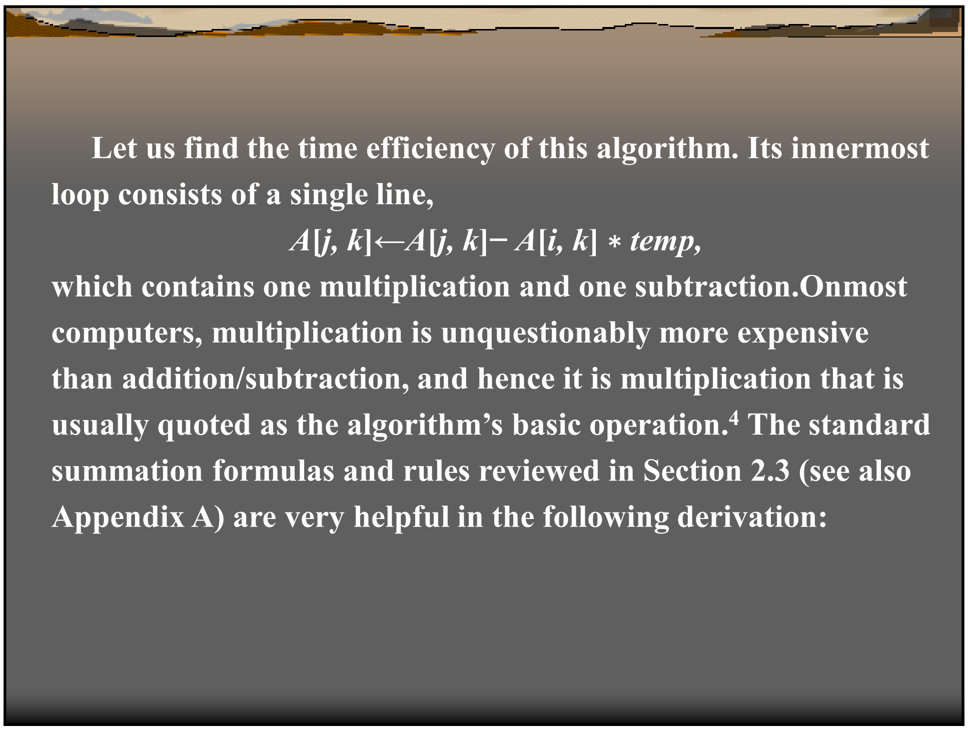
Since we have to be prepared for the possibility of row exchanges anyway, we can take care of another potential difficulty: the possibility that $A[i, i]$ is so small and consequently the scaling factor $A[j, i]/A[i, i]$ so large that the new value of $A[j, k]$might become distorted by a round-off error caused by a subtraction of two numbers of greatly different magnitudes.[3] To avoid this problem, we can always look for a row with the largest absolute value of the coefficient in the ith column, exchange it with the ith row, and then use the new $A[i, i]$ as the ith iteration's pivot. This modification, called *partial pivoting*, guarantees that the magnitude of the scaling factor (比例因子) will never exceed 1.



```
ALGORITHM    BetterForwardElimination($A[1..n, 1..n], b[1..n]$)
    //Implements Gaussian elimination with partial pivoting
    //Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of
    $A$ and the
    //corresponding right-hand side values in place of the $(n + 1)$st
    column
    for $i \leftarrow 1$ to $n$ do $A[i, n + 1] \leftarrow b[i]$ //appends $b$ to $A$ as the last
    column
    for $i \leftarrow 1$ to $n - 1$ do
        $pivotrow \leftarrow i$
        for $j \leftarrow i + 1$ to $n$ do
            if $|A[j, i]| > |A[pivotrow, i]|$ $pivotrow \leftarrow j$
```



```
for $k \leftarrow i$ to $n + 1$ do
    swap($A[i, k]$, $A[pivotrow, k]$)
for $j \leftarrow i + 1$ to $n$ do
    $temp \leftarrow A[j, i] / A[i, i]$
    for $k \leftarrow i$ to $n + 1$ do
        $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$
```



Let us find the time efficiency of this algorithm. Its innermost loop consists of a single line,

$$A[j, k] \leftarrow A[j, k] - A[i, k] * temp,$$

which contains one multiplication and one subtraction.Onmost computers, multiplication is unquestionably more expensive than addition/subtraction, and hence it is multiplication that is usually quoted as the algorithm's basic operation.[4] The standard summation formulas and rules reviewed in Section 2.3 (see also Appendix A) are very helpful in the following derivation:

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+2-i) \\
&= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
&= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1 \\
&= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} \\
&= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).
\end{aligned}
$$

Since the second (*back substitution*) stage of Gaussian elimination is in $\Theta(n^2)$, as you are asked to show in the exercises, the running time is dominated by the cubic elimination stage, making the entire algorithm cubic as well.

LU Decomposition

Gaussian elimination has an interesting and very useful byproduct called *LU decomposition* of the coefficient matrix. In fact, modern commercial implementations of Gaussian elimination are based on such a decomposition rather than on the basic algorithm outlined above.

EXAMPLE2 Let us return to the example in the beginning of this section, where we applied Gaussian elimination to the matrix

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

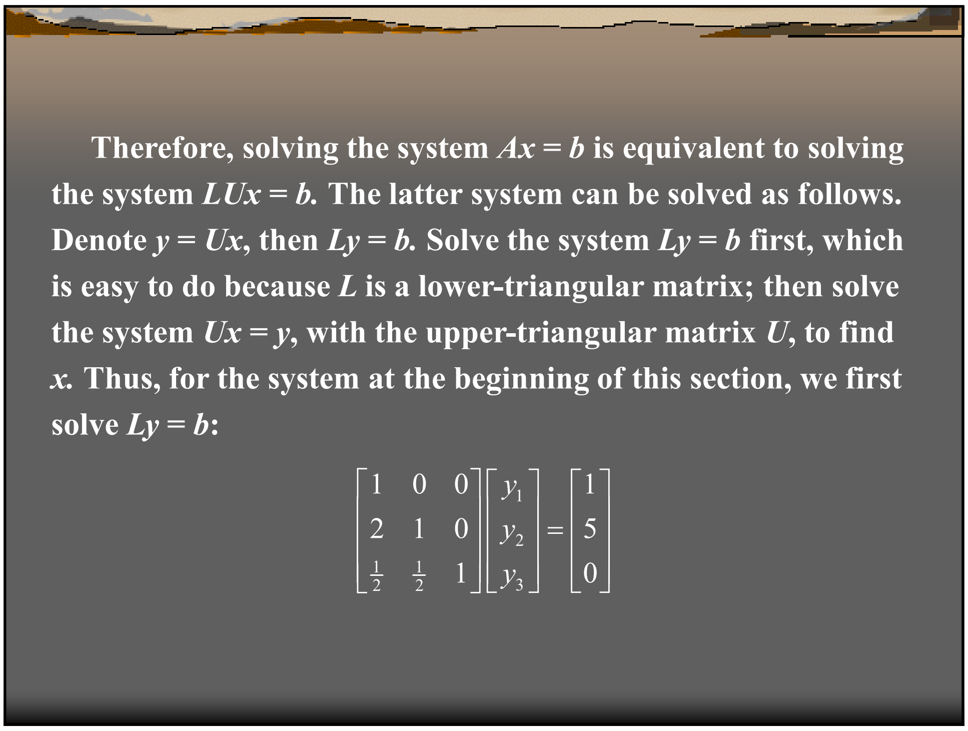
Consider the lower-triangular matrix L made up of 1's on its main diagonal and the row multiples used in the forward elimination process

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$

and the upper-triangular matrix U that was the result of this elimination

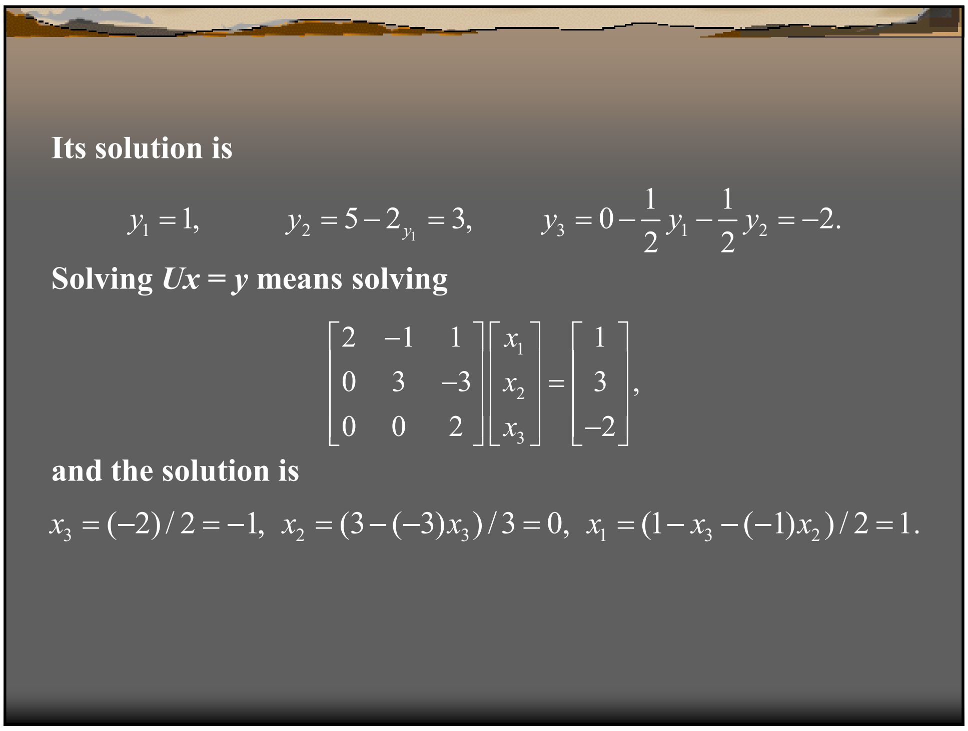
$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$

It turns out that the product LU of these matrices is equal to matrix A. (For this particular pair of L and U, you can verify this fact by direct multiplication, but as a general proposition, it needs, of course, a proof, which we omit here.)



Therefore, solving the system $Ax = b$ is equivalent to solving the system $LUx = b$. The latter system can be solved as follows. Denote $y = Ux$, then $Ly = b$. Solve the system $Ly = b$ first, which is easy to do because L is a lower-triangular matrix; then solve the system $Ux = y$, with the upper-triangular matrix U, to find x. Thus, for the system at the beginning of this section, we first solve $Ly = b$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}$$



Its solution is

$$y_1 = 1, \qquad y_2 = 5 - 2_{y_1} = 3, \qquad y_3 = 0 - \frac{1}{2} y_1 - \frac{1}{2} y_2 = -2.$$

Solving $Ux = y$ means solving

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix},$$

and the solution is

$$x_3 = (-2)/2 = -1, \;\; x_2 = (3 - (-3)x_3)/3 = 0, \;\; x_1 = (1 - x_3 - (-1)x_2)/2 = 1.$$

Computing a Matrix Inverse

Gaussian elimination is a very useful algorithm that tackles one of the most important problems of applied mathematics: solving systems of linear equations. In fact, Gaussian elimination can also be applied to several other problems of linear algebra, such as computing a matrix *inverse*. The inverse of an $n \times n$ matrix A is an $n \times n$ matrix, denoted A^{-1}, such that

$$AA^{-1} = I,$$

where I is the $n \times n$ identity matrix (the matrix with all zero elements except the main diagonal elements, which are all ones).

Not every square matrix has an inverse, but when it exists, the inverse is unique. If a matrix A does not have an inverse, it is called *singular*. One can prove that a matrix is singular if and only if one of its rows is a linear combination (a sum of some multiples) of the other rows. A convenient way to check whether a matrix is nonsingular is to apply Gaussian elimination: if it yields an upper-triangular matrix with no zeros on the main diagonal, the matrix is nonsingular; otherwise, it is singular. So being singular is a very special situation, and most square matrices do have their inverses.

According to the definition of the inverse matrix for a nonsingular $n \times n$ matrix A, to compute it we need to find n^2 numbers x_{ij}, $1 \leq i, j \leq n$, such that

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

We can find the unknowns by solving n systems of linear equations that have the same coefficient matrix A, the vector of unknowns x^j is the jth column of the inverse, and the right-hand side vector e^j is the jth column of the identity matrix ($1 \leq j \leq n$):

$$Ax^j = e^j .$$

We can solve these systems by applying Gaussian elimination to matrix A augmented by the $n \times n$ identity matrix. Better yet, we can use forward elimination to find the LU decomposition of A and then solve the systems $LUx^j = e^j$, $j = 1, \ldots, n$, as explained earlier.

Computing a Determinant

Another problem that can be solved by Gaussian elimination is computing a determinant. The *determinant* of an $n \times n$ matrix A, denoted det A or $|A|$, is a number whose value can be defined recursively as follows. If $n = 1$, i.e., if A consists of a single element a_{11}, det A is equal to a_{11}; for $n > 1$, det A is computed by The recursive formula

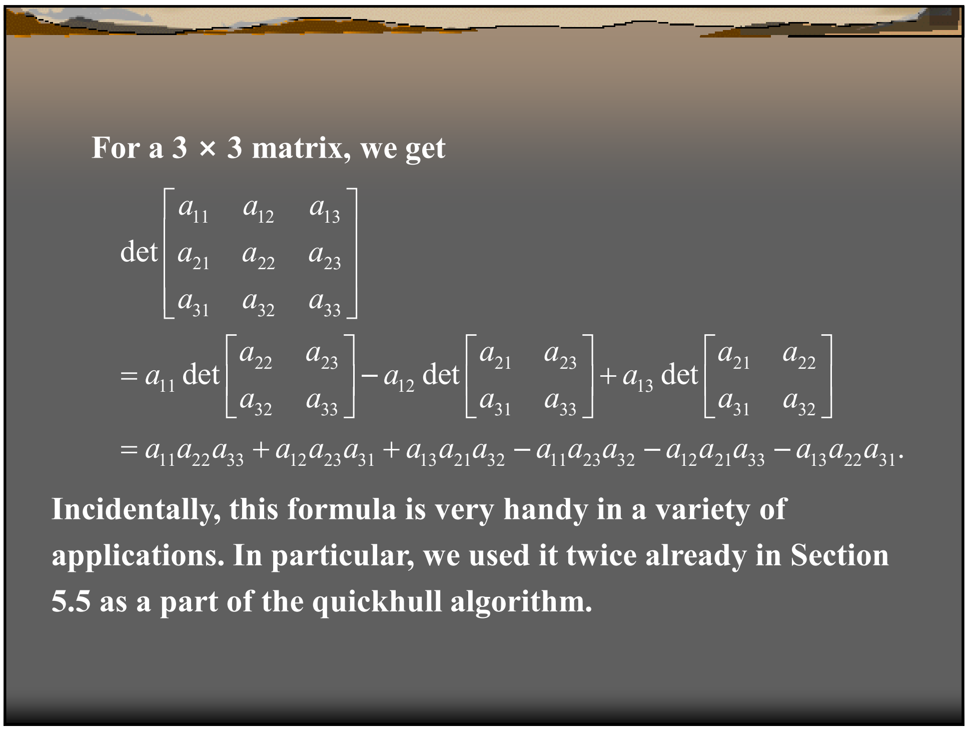
$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j,$$

where s_j is +1 if j is odd and −1 if j is even, a_{1j} is the element in row 1 and column j , and A_j is the $(n-1) \times (n-1)$ matrix obtained from matrix A by deleting its row 1 and column j .

In particular, for a 2 × 2 matrix, the definition implies a formula that is easy to remember:

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \det [a_{22}] - a_{12} \det [a_{21}] = a_{11}a_{22} - a_{12}a_{21}.$$

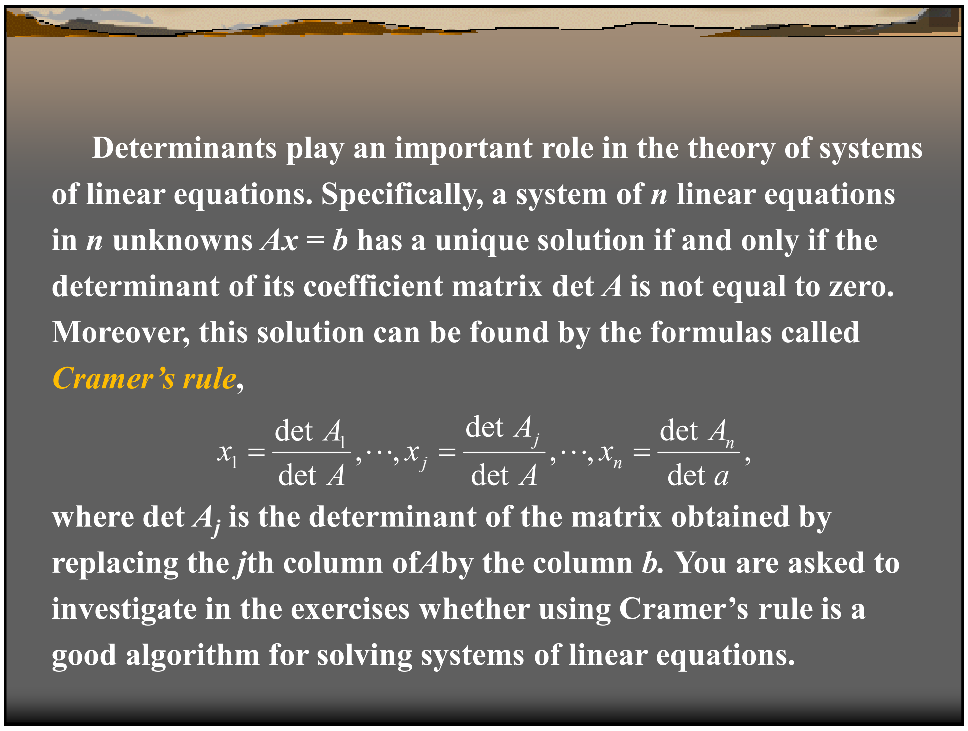
In other words, the determinant of a 2 × 2 matrix is simply equal to the difference between the products of its diagonal elements.



For a 3 × 3 matrix, we get

$$\det\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$
$$= a_{11}\det\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12}\det\begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13}\det\begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$
$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}.$$

Incidentally, this formula is very handy in a variety of applications. In particular, we used it twice already in Section 5.5 as a part of the quickhull algorithm.



Determinants play an important role in the theory of systems of linear equations. Specifically, a system of *n* linear equations in *n* unknowns $Ax = b$ has a unique solution if and only if the determinant of its coefficient matrix det *A* is not equal to zero. Moreover, this solution can be found by the formulas called *Cramer's rule*,

$$x_1 = \frac{\det A_1}{\det A}, \dots, x_j = \frac{\det A_j}{\det A}, \dots, x_n = \frac{\det A_n}{\det a},$$

where det A_j is the determinant of the matrix obtained by replacing the *j*th column of*A*by the column *b.* You are asked to investigate in the exercises whether using Cramer's rule is a good algorithm for solving systems of linear equations.