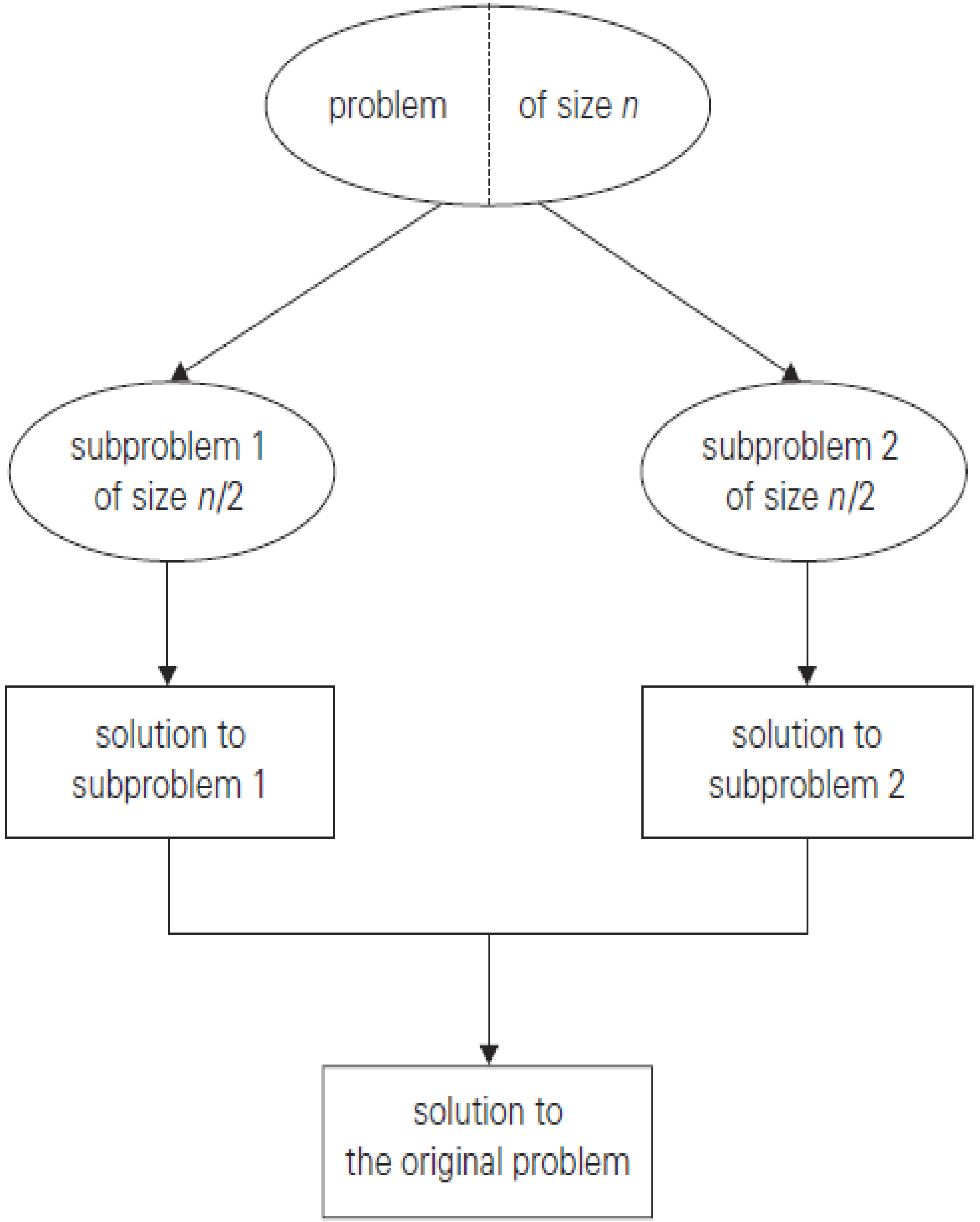
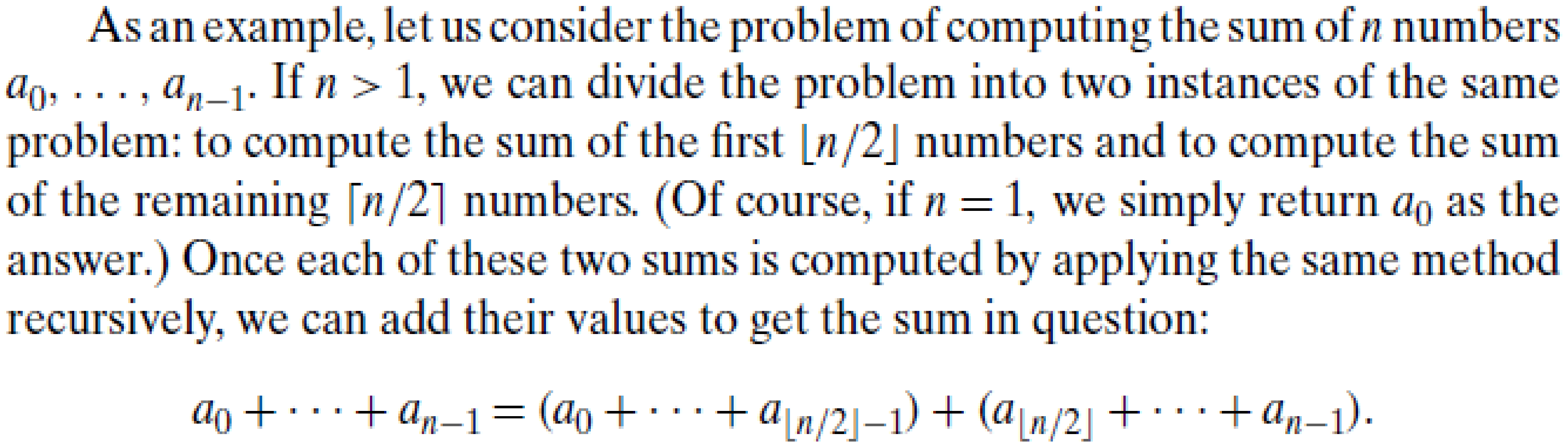
3- DIVIDE AND CONQUER

3- DIVIDE AND CONQUER

- Divide-and-conquer is probably the best-known general algorithm design technique.
- Divide-and-conquer algorithms work according to the following general plan:
  - **1.** A problem is divided into several subproblems of the same type, ideally of about equal size.
  - **2.** The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
  - **3.** If necessary, the solutions to the subproblems are combined to get a solution to the original problem.


problem
of size n
subproblem 1
of size $n/2$
subproblem 2
of size $n/2$
solution to
subproblem 1
solution to
subproblem 2
solution to
the original problem

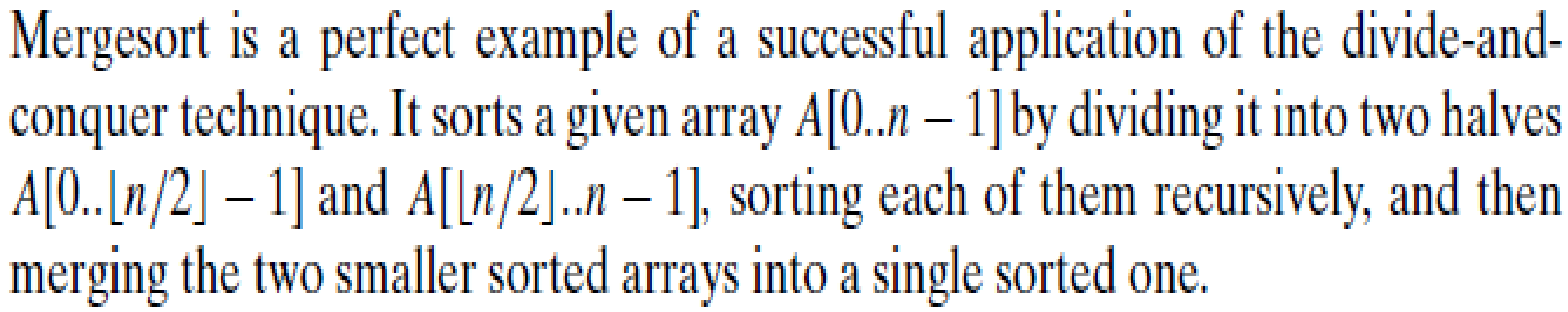
3- DIVIDE AND CONQUER



As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

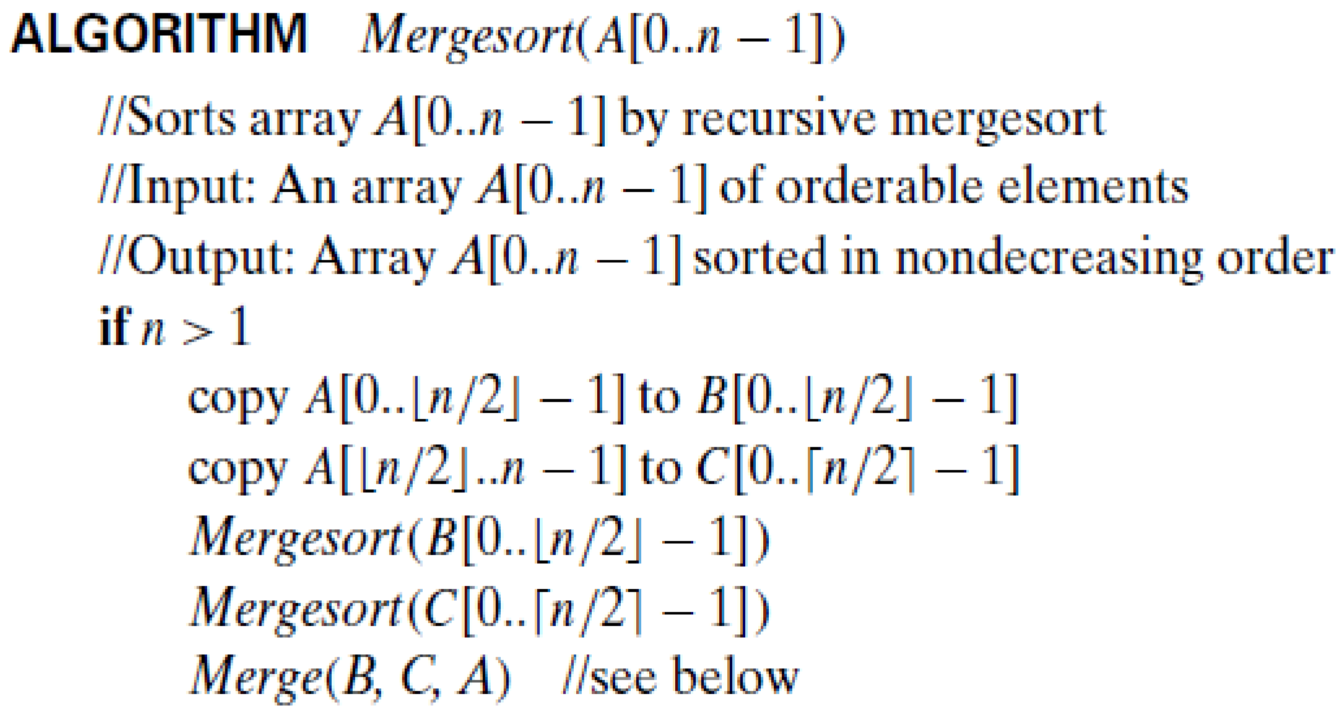
$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

3.1 Mergesort



Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor ..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

3.1 Mergesort

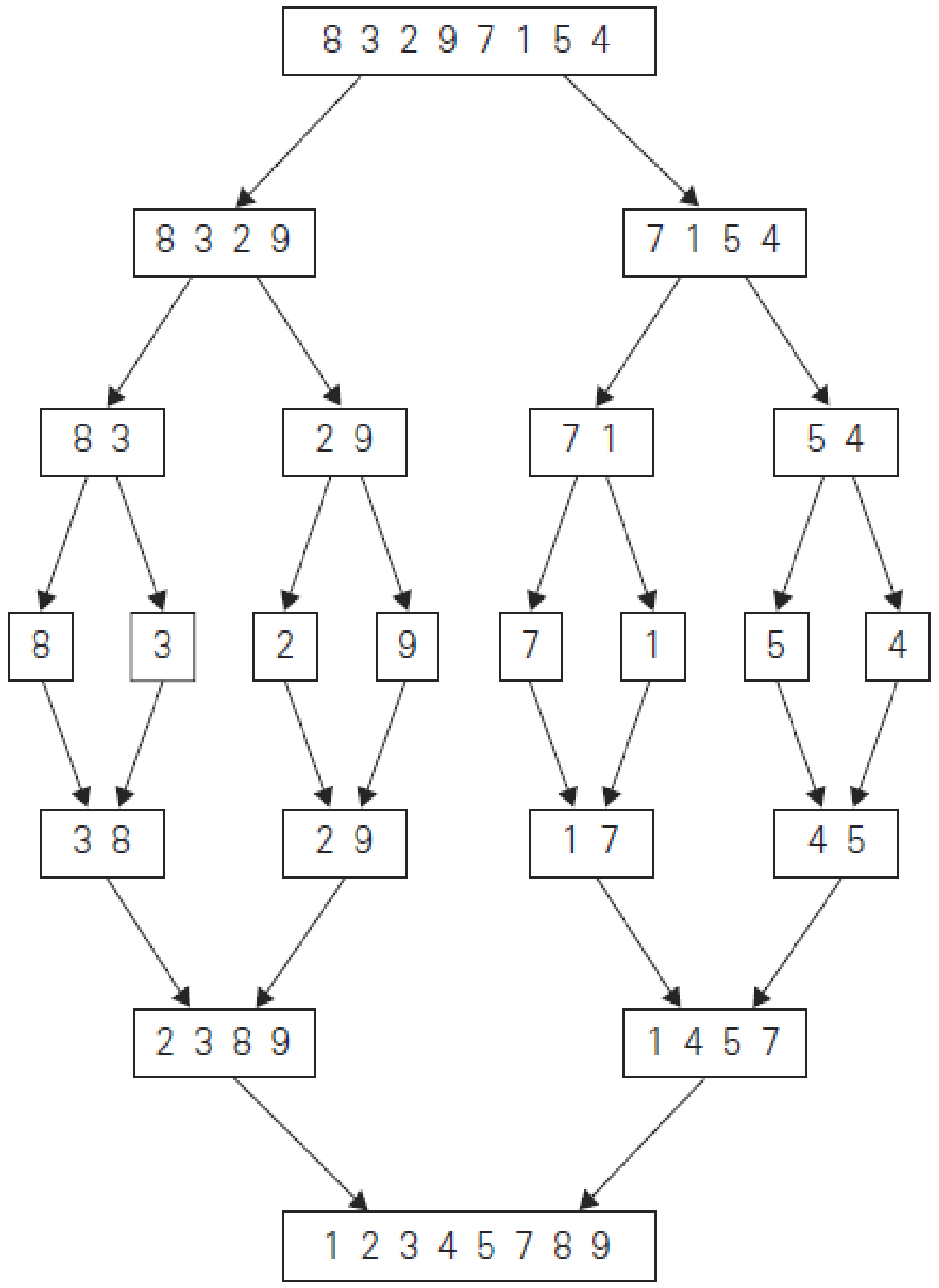


```
**ALGORITHM** *Mergesort*($A[0..n-1]$)
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    **if** $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        *Mergesort*($B[0..\lfloor n/2 \rfloor - 1]$)
        *Mergesort*($C[0..\lceil n/2 \rceil - 1]$)
        *Merge*($B, C, A$)    //see below
```

3.1 Mergesort

- The merging of two sorted arrays can be done as follows.
  - Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
  - The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
  - This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

3.1 Mergesort

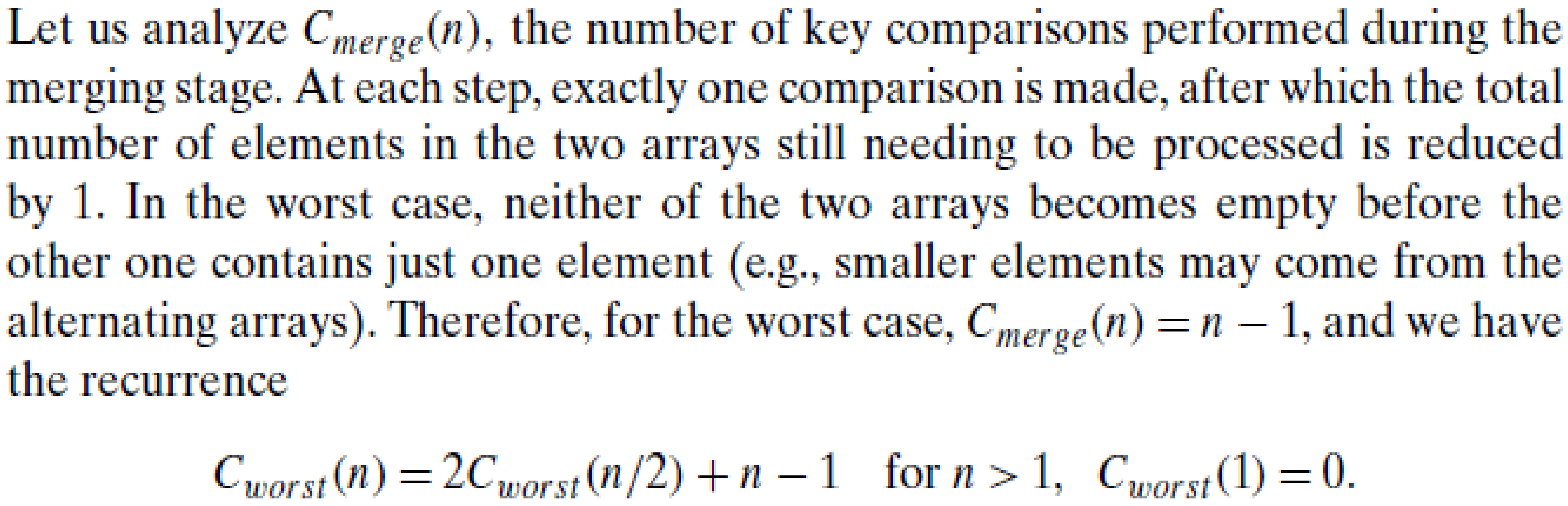


3.1 Mergesort

- How efficient is mergesort?
- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

3.1 Mergesort



Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

3.1 Mergesort – SOME EXERCISES

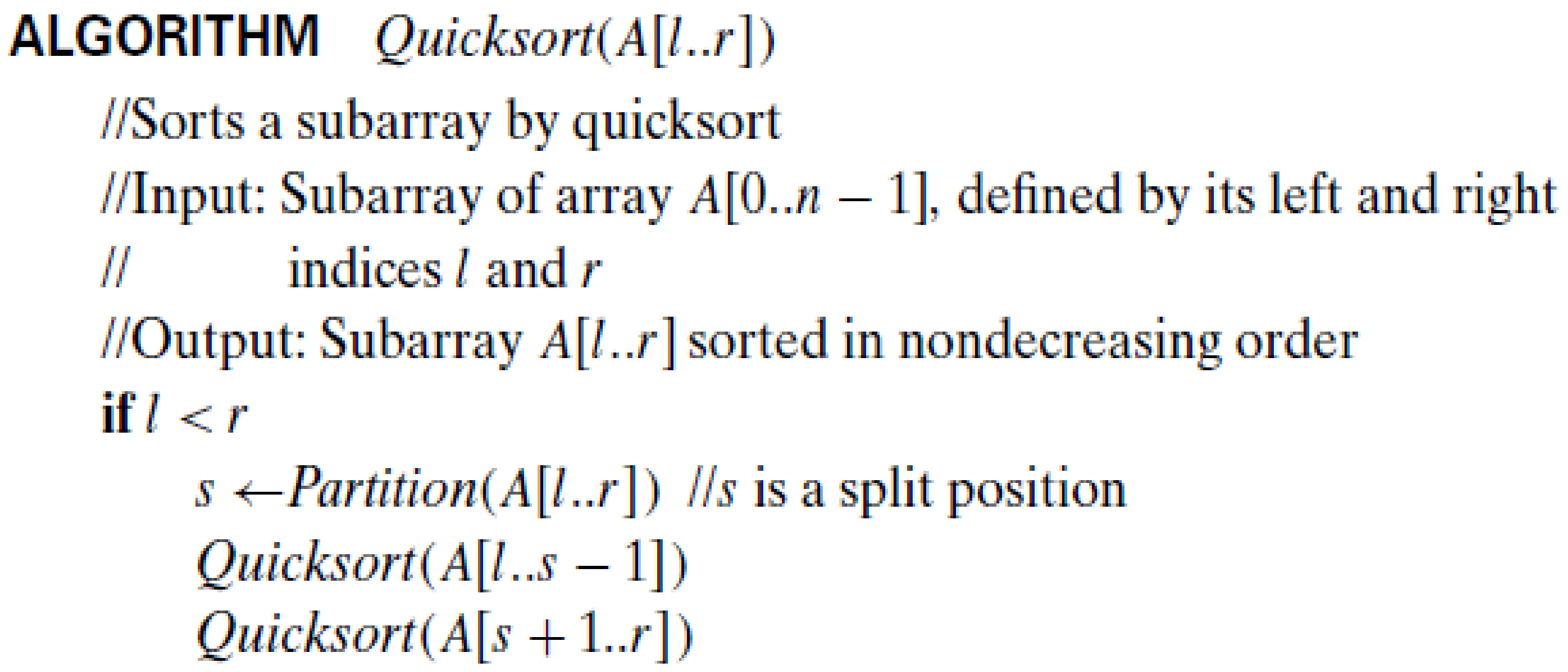
- Write pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers?
- Write pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.
- Write pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing an where n is a positive integer.
- Apply mergesort to sort the list E, X, A, M, P, L, E in alphabetical order.

3.2 Quicksort

- Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach.
- Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- A partition is an arrangement of the array’s elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

3.2 Quicksort



```
ALGORITHM    Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //          indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

3.2 Quicksort- SOME EXERCISES

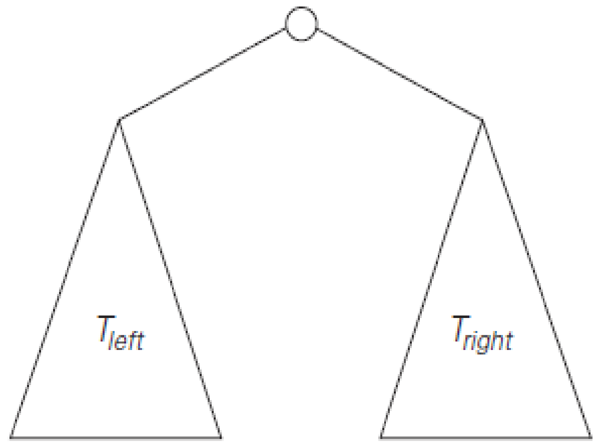
- Apply quicksort to sort the list E, X, A, M, P , L, E in alphabetical order.

- For the version of quicksort given in this section, are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

- True or false: For every n> 1, there are n-element arrays that are sorted faster by insertion sort than by quicksort?

3.3 Binary Trees

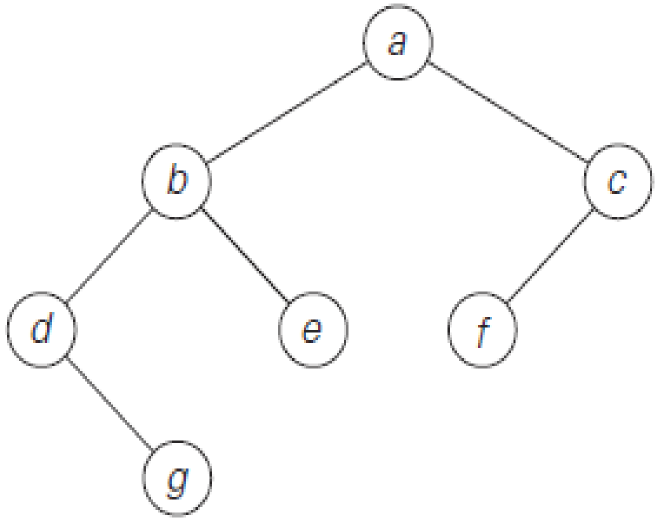
In this section, we see how the divide-and-conquer technique can be applied to binary trees. A ***binary tree*** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root.



3.3 Binary Trees

- The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree’s root and its left and right subtrees. They differ only by the timing of the root’s visit:
  - In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order).
  - In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree.
  - In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order).

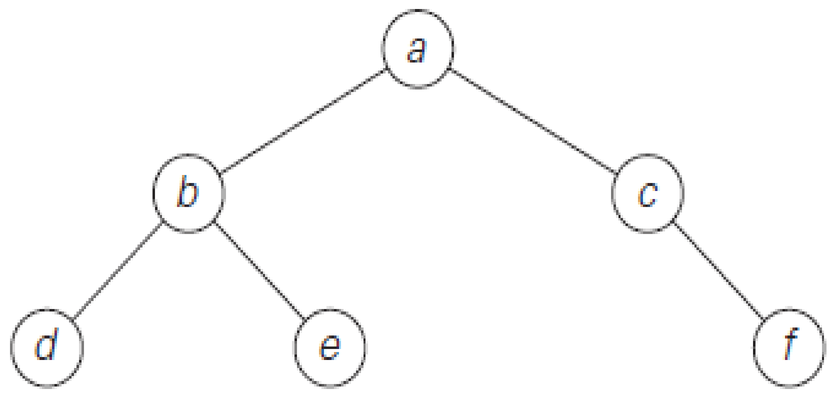
3.3 Binary Trees



preorder:	*a, b, d, g, e, c, f*
inorder:	*d, g, b, e, a, f, c*
postorder:	*g, d, e, b, f, c, a*

3.3 Binary Trees – SOME EXERCISES

- Traverse the following binary tree
  - in preorder.
  - in inorder.
  - in postorder.



3.3 Binary Trees – SOME EXERCISES

- Draw a binary tree with 10 nodes labeled 0, 1,...,9 in such a way that the inorder and postorder traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (inorder) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (postorder).

3.4 Multiplication of Large Integers

- Some applications, notably modern cryptography, require manipulation of inte-gers that are over 100 decimal digits long.
- Since such integers are too long to fit in a single word of a modern computer, they require special treatment.
- This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers.
- If we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n2 digit multiplications.
- If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.
- Though it might appear that it would be impossible to design an algorithm with fewer than n2 digit multiplications, this turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

3.4 Multiplication of Large Integers

- To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0$$

- Now let us multiply them:

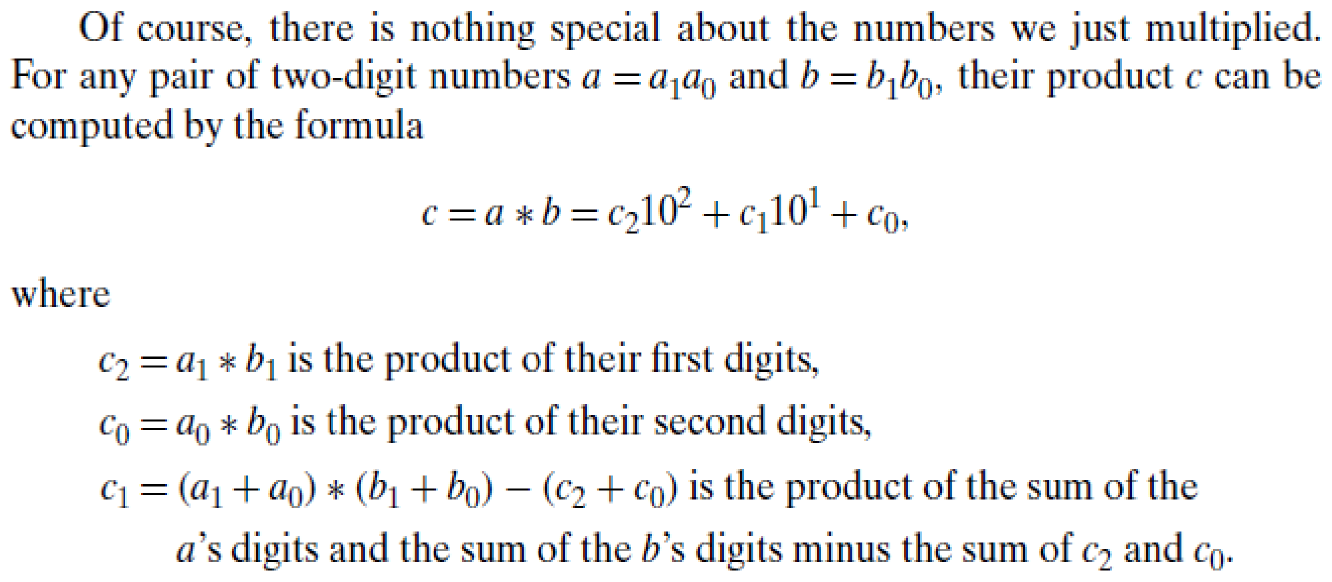
$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0. \end{aligned}$$

3.4 Multiplication of Large Integers

- The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4$$

3.4 Multiplication of Large Integers



Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

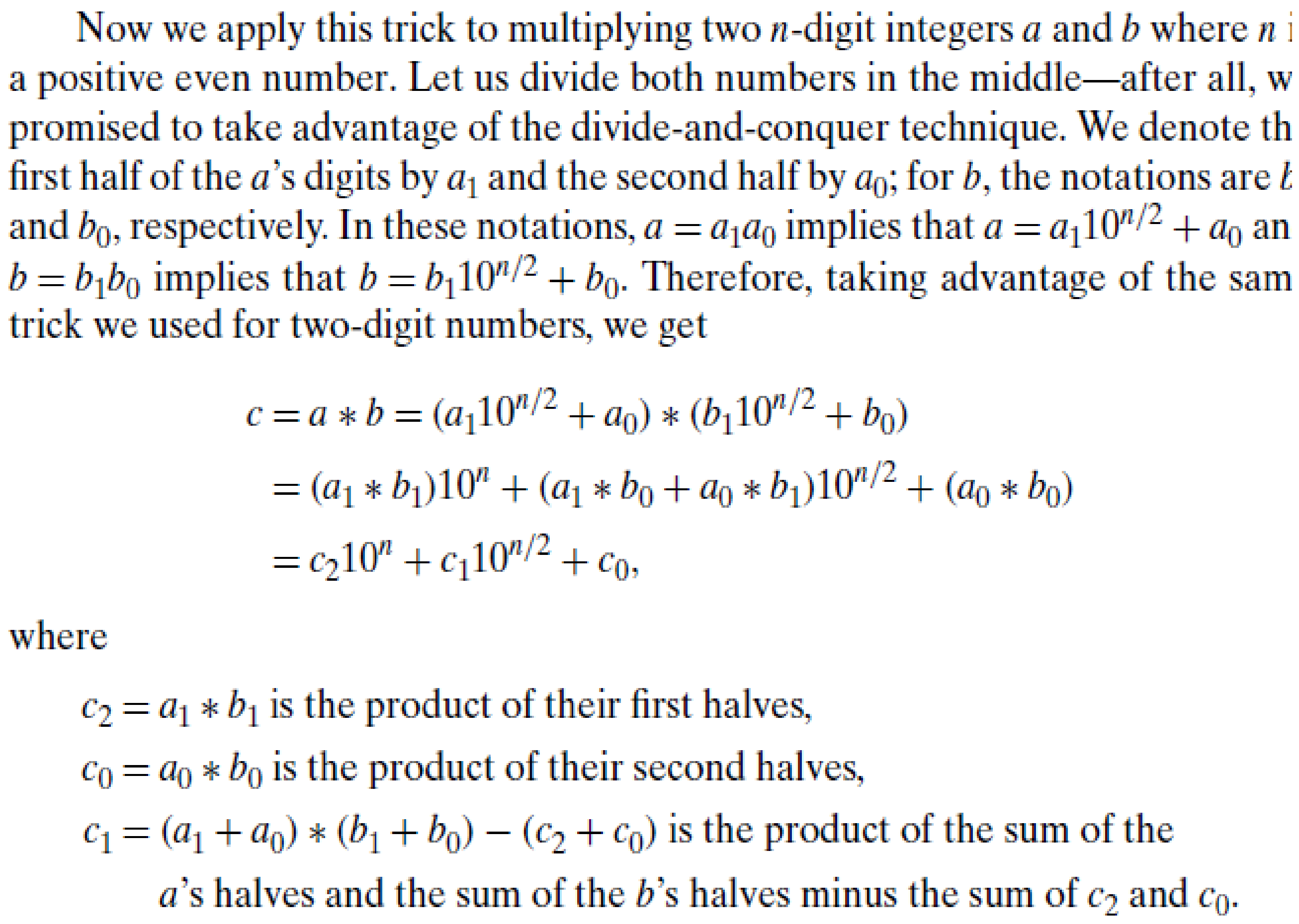
where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

3.4 Multiplication of Large Integers



Now we apply this trick to multiplying two n-digit integers a and b where n i
a positive even number. Let us divide both numbers in the middle—after all, w
promised to take advantage of the divide-and-conquer technique. We denote th
first half of the a's digits by a_1 and the second half by a_0; for b, the notations are
and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ an
$b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the sam
trick we used for two-digit numbers, we get

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,
$c_0 = a_0 * b_0$ is the product of their second halves,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the
a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

3.4 Multiplication of Large Integers – SOME EXERCISES

- What are the smallest and largest numbers of digits the product of two decimal n-digit integers can have?

- Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.