Mastering Data Structures and Algorithms: A Practical Approach

Divide and Conquer

By Dr. Juan C. Gomez

Fall 2018

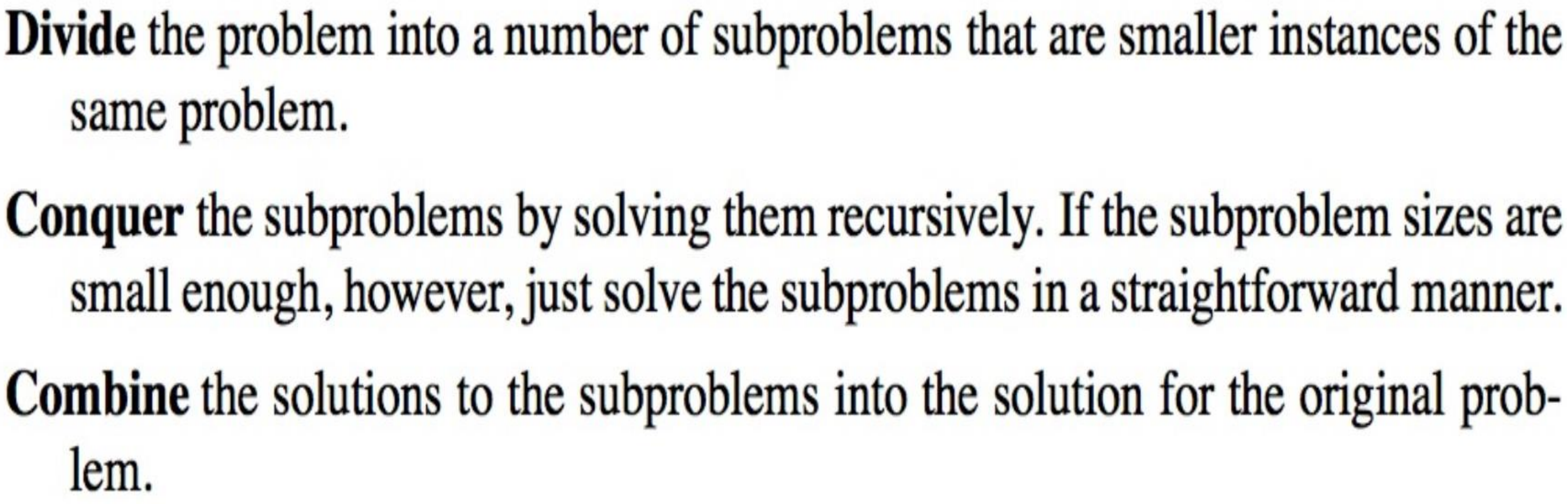
UC Santa Cruz Extension in Silicon Valley

Overview

- Concept
- Performance
- Sample Problems
  - Merge Sort
  - Quick Sort
  - Maximum Subarray Problem
  - Binary Tree Traversal
  - Selection Problem
  - Multiplication of Large Integers
  - Strassen’s Algorithm (Matrix Multiplication)
  - Closest Points
  - Convex Hull Problems

Divide and Conquer: Concept

- Solve the problem recursively, applying the following steps at each level of recursion:



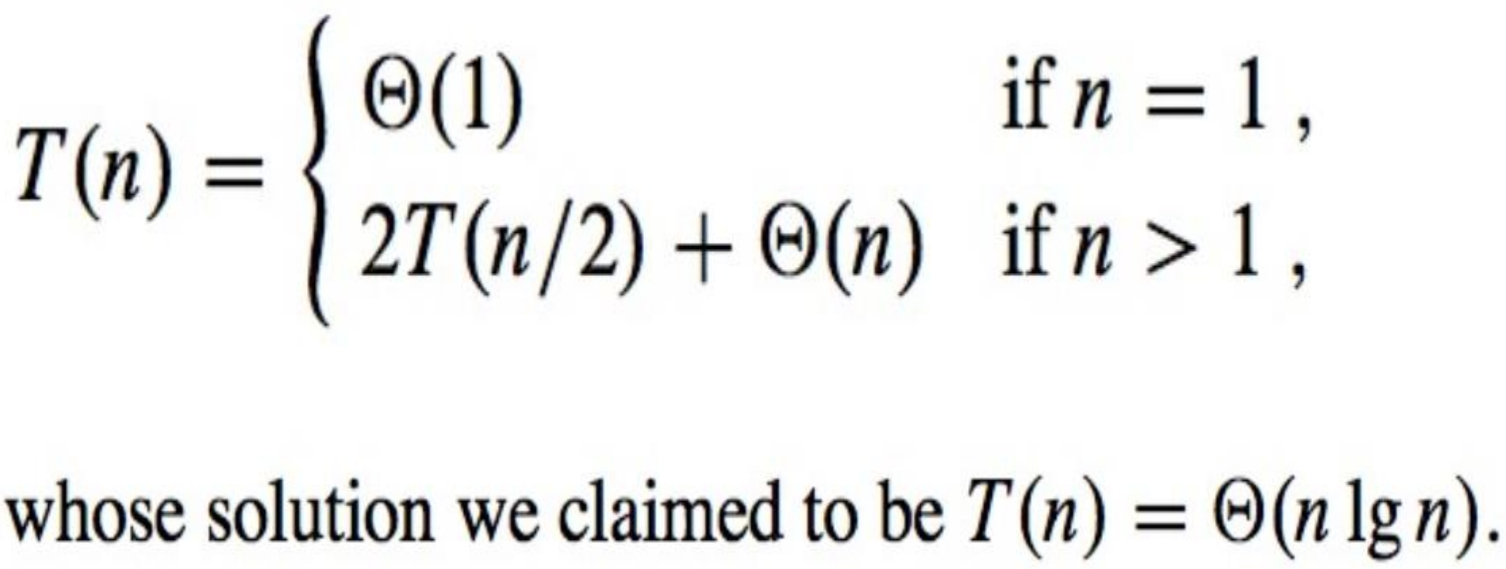
Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Divide and Conquer: Performance

- Characterized by a recursion:
  - As an example, for Merge sort we have worst case:

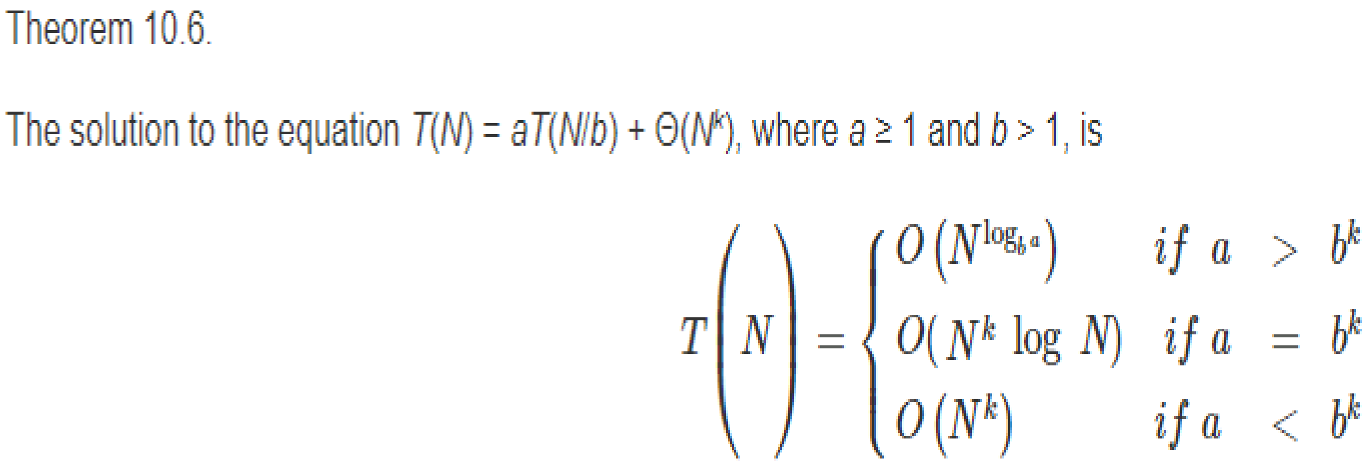


$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,, \end{cases}$$

whose solution we claimed to be $T(n) = \Theta(n \lg n)$.

Divide and Conquer: Performance

- General Equation for divide and conquer algos:



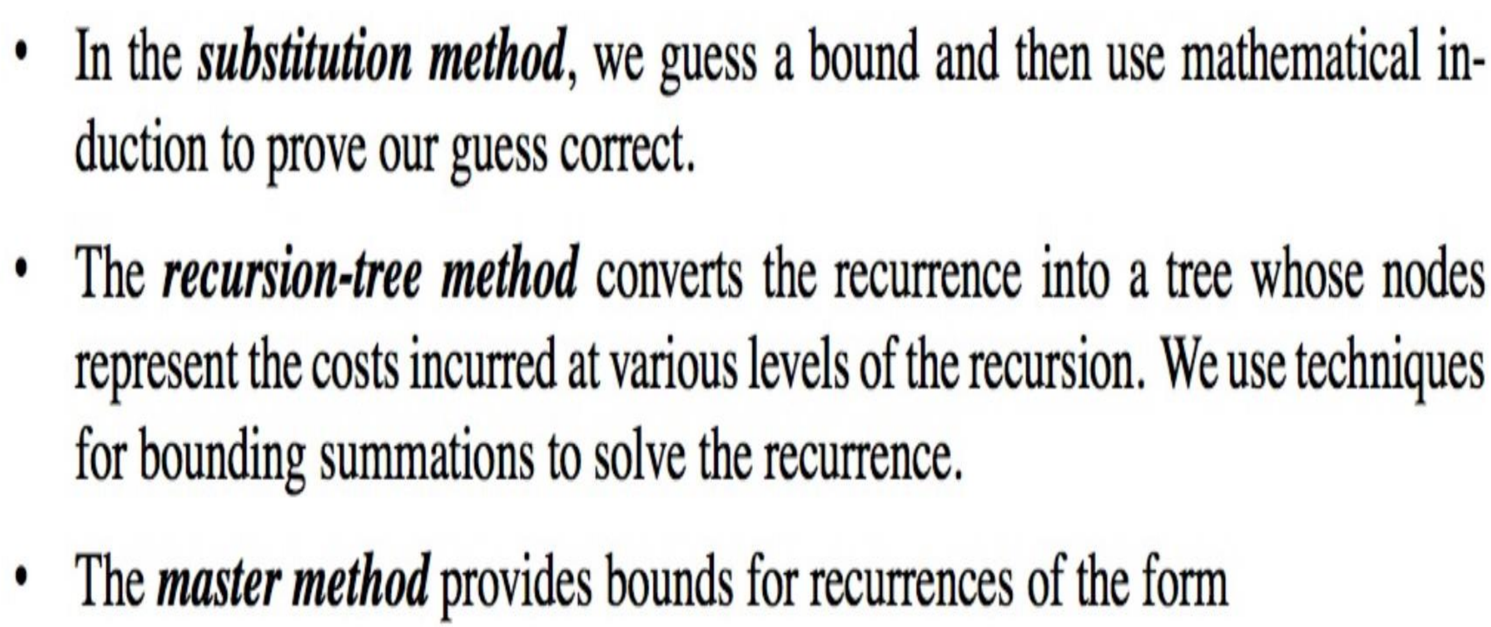
Theorem 10.6.

The solution to the equation $T(N) = aT(N/b) + \Theta(N^k)$, where $a \geq 1$ and $b > 1$, is

$$T\left(N\right) = \begin{cases} O\left(N^{\log_b a}\right) & if\ a > b^k \\ O(N^k \log N) & if\ a = b^k \\ O\left(N^k\right) & if\ a < b^k \end{cases}$$

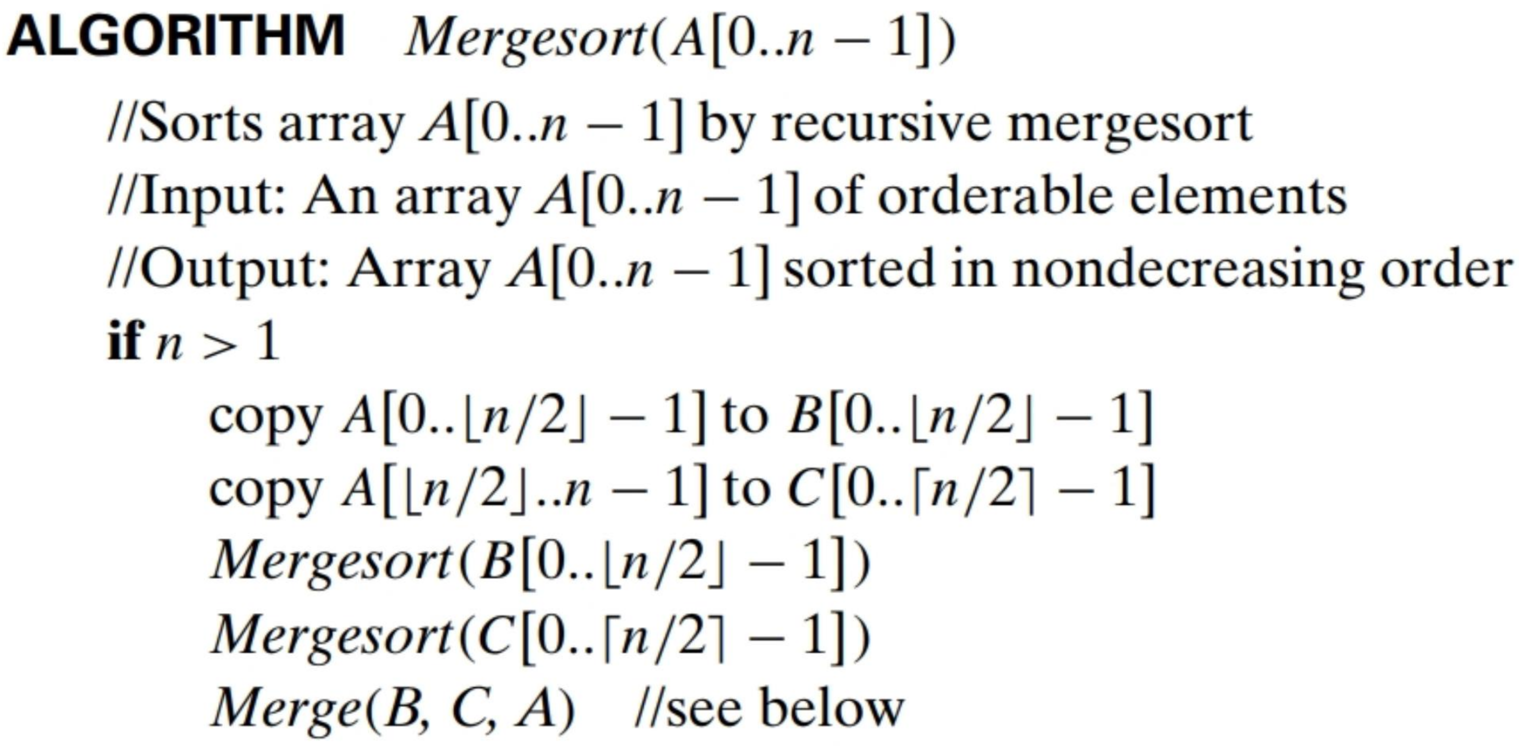
Divide and Conquer: Performance

- Methods for solving recurrences:



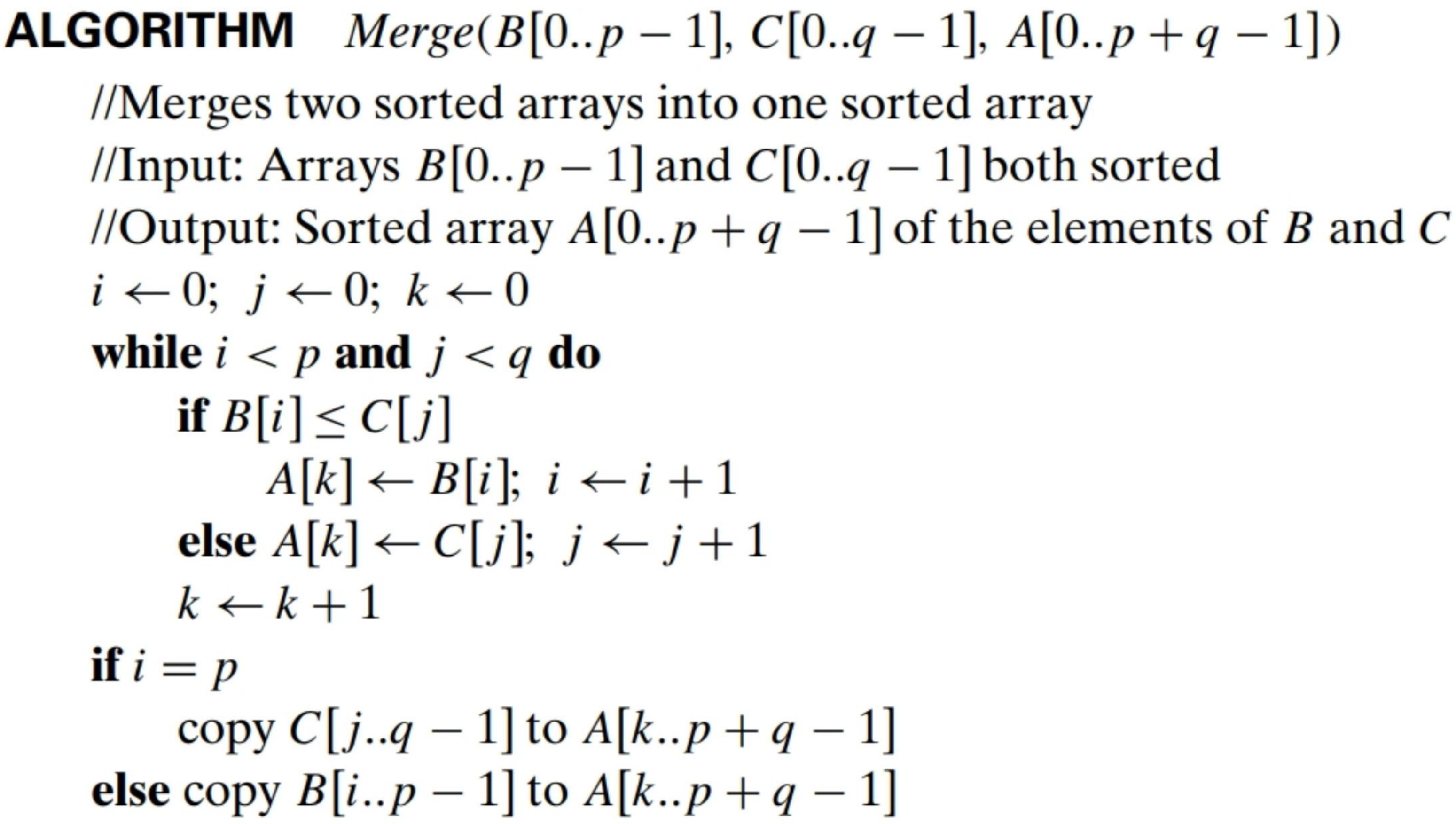
- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

Merge-Sort



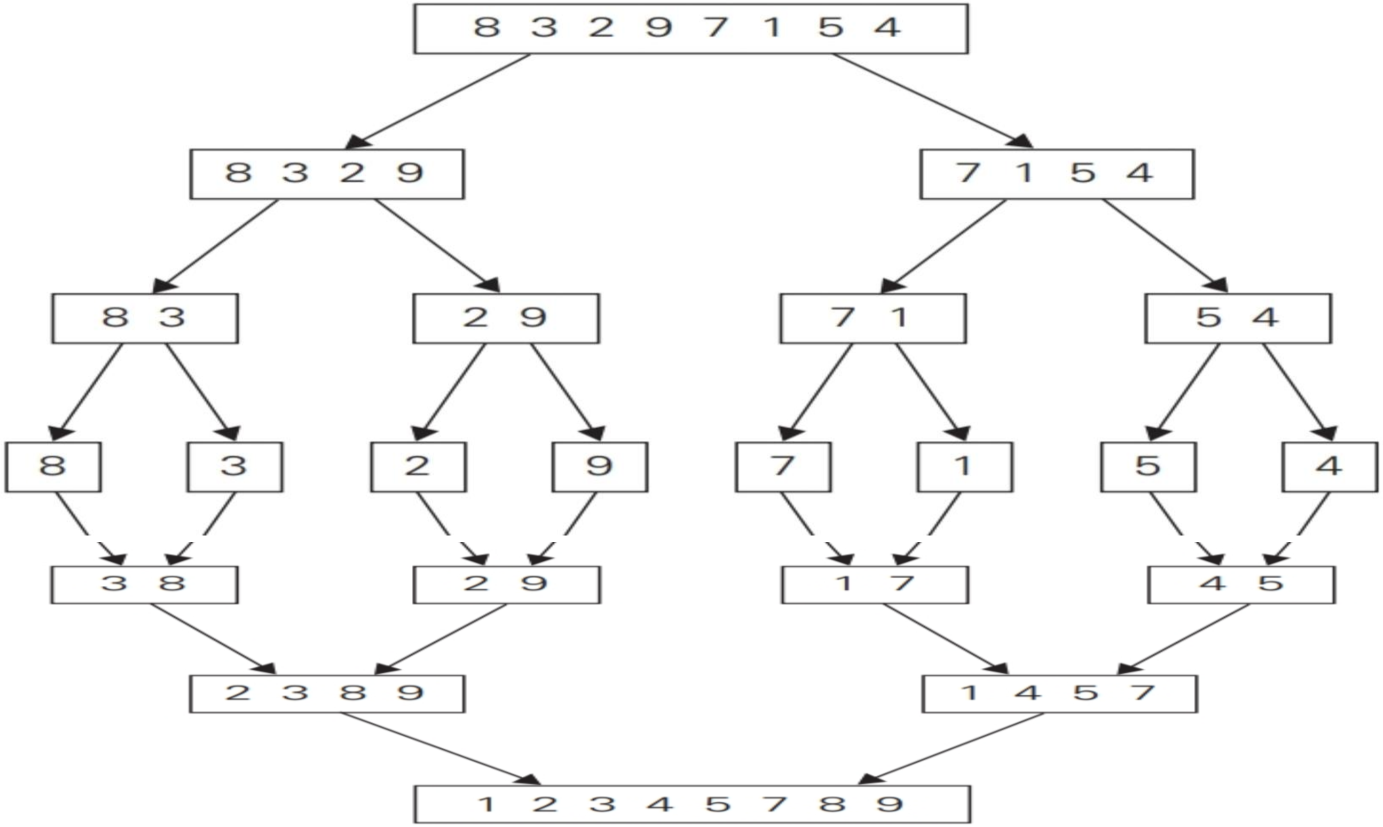
```
ALGORITHM  $Mergesort(A[0..n-1])$
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$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

Merge-Sort



```
ALGORITHM  $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Merge-Sort


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Merge-Sort

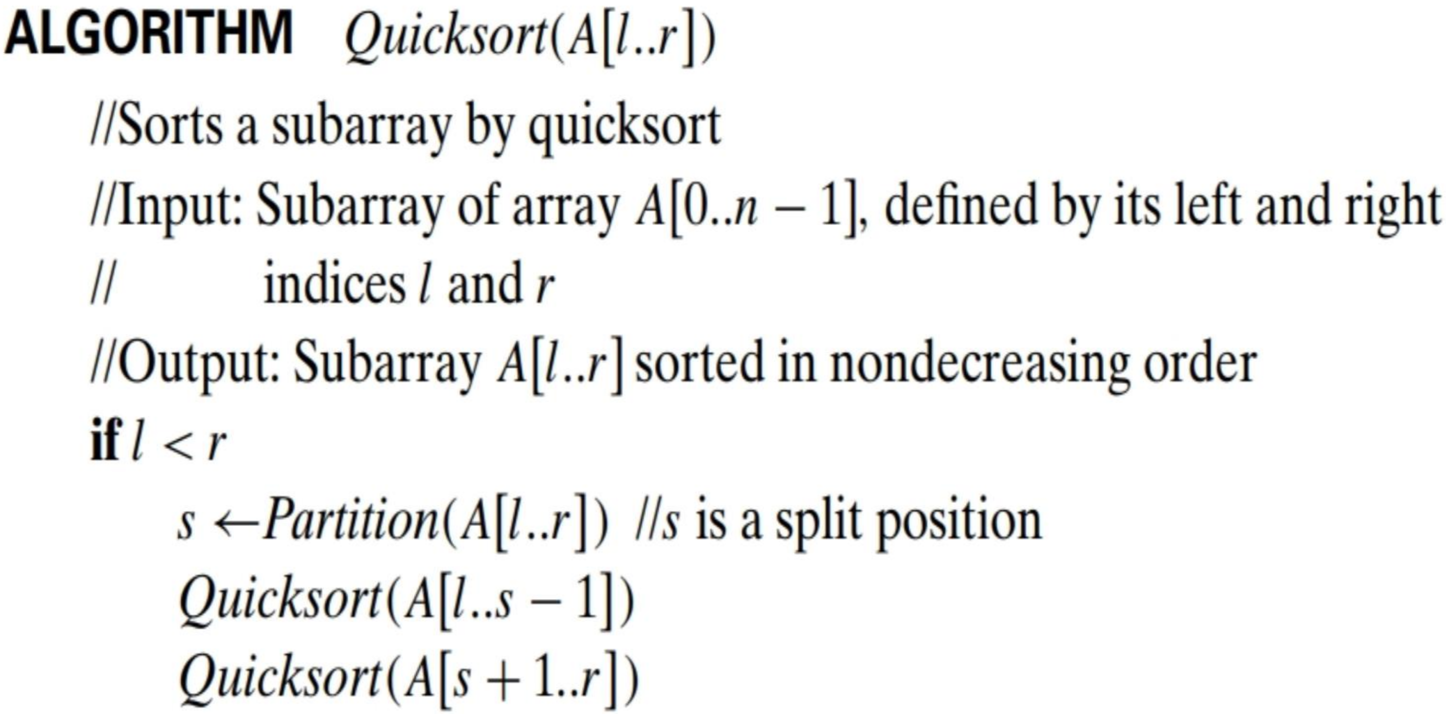
$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

$$C_{worst}(n) = n \log_2 n - n + 1.$$

Quick-Sort

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

Quick-Sort



```
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 $l < r$
        $s \leftarrow Partition(A[l..r])$  //$s$ is a split position
        $Quicksort(A[l..s-1])$
        $Quicksort(A[s+1..r])$
```

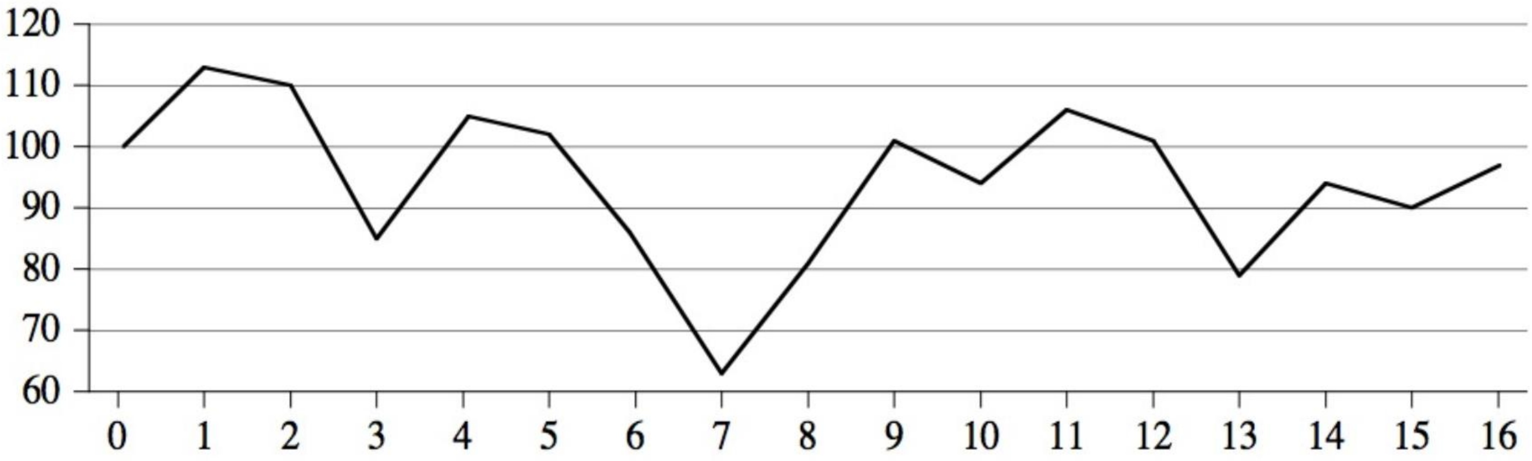
Quick-Sort

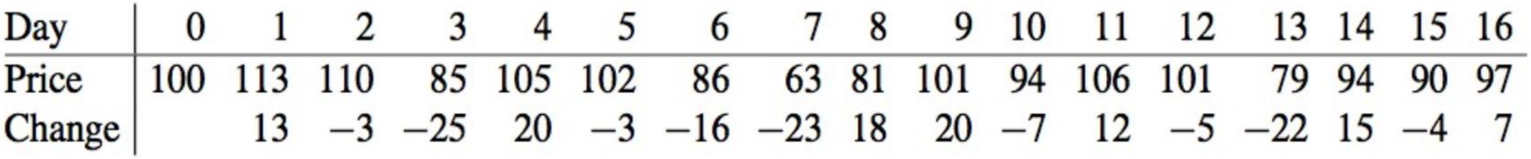
$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

$$C_{best}(n) = n \log_2 n.$$

Maximum Subarray Problem

- Origin: optimizing stock profit if you know future values.

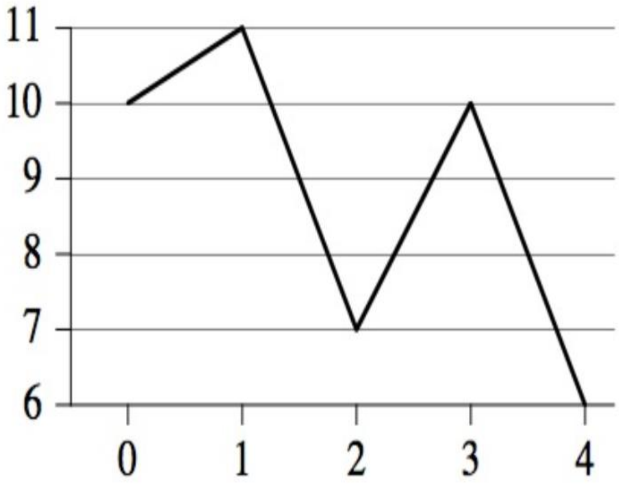


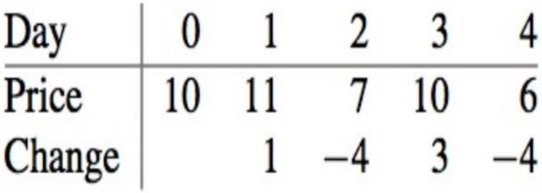


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Maximum Subarray Problem

- Origin: optimizing stock profit if you know future values.



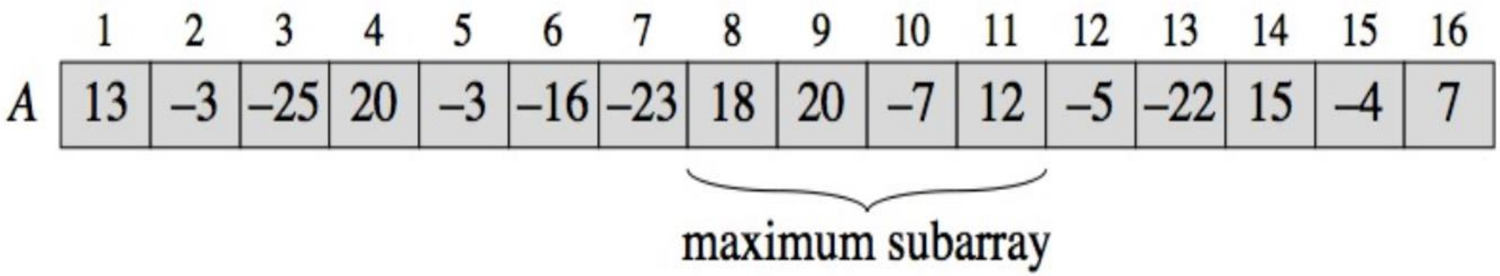


Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Maximum Subarray Problem

- Brute Force Approach: try all possible pair of buy/sell days:
  - (n-1) + (n-2) …=> O(n^2)

- Transform the problem: maximum contiguous subarray in the delta array
  - Brute Force: try all possible subarrays, still O(n^2)

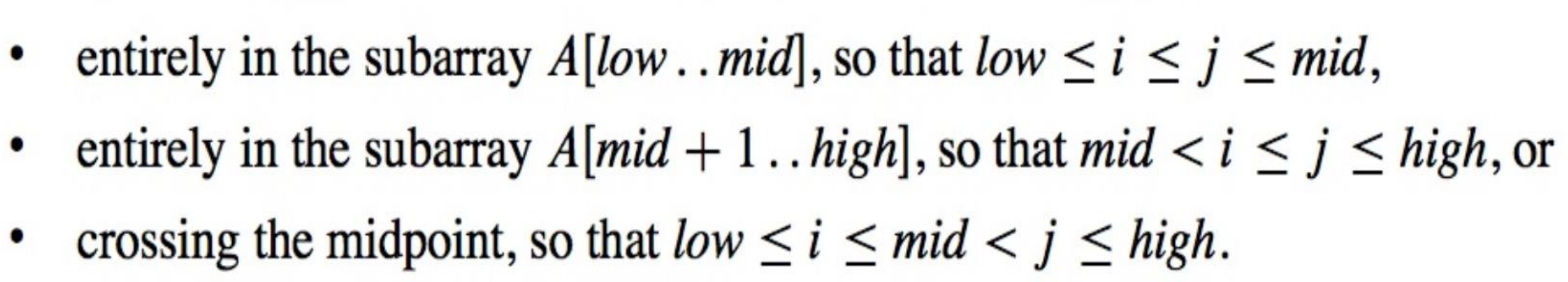


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Maximum Subarray Problem

- Divide and conquer solution:
  - Solution must be in:



- entirely in the subarray $A[low \,..\, mid]$, so that $low \le i \le j \le mid$,
- entirely in the subarray $A[mid + 1 \,..\, high]$, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.

Maximum Subarray Problem

- Divide and conquer solution:
  - Solution must be in lower half (recurse on n/2 elements), upper half (recurse on n/2 elements) or crossing the middle.

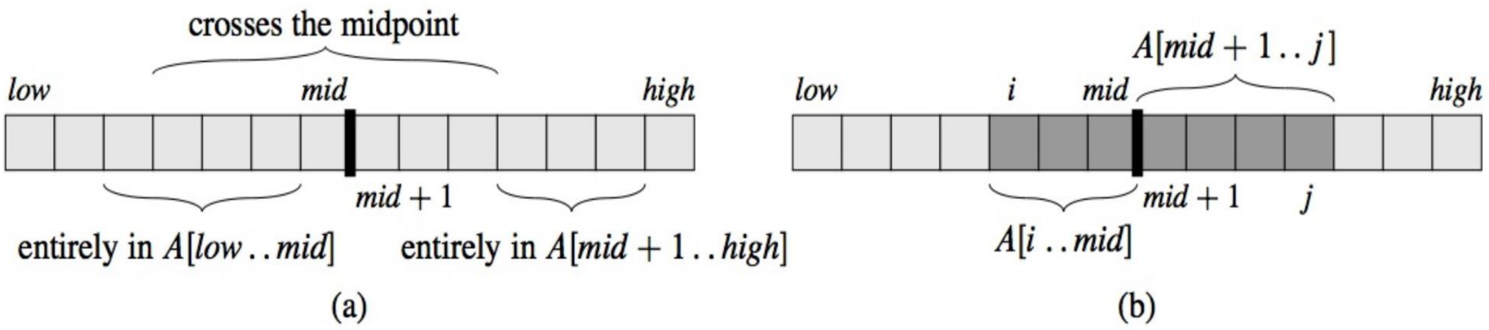
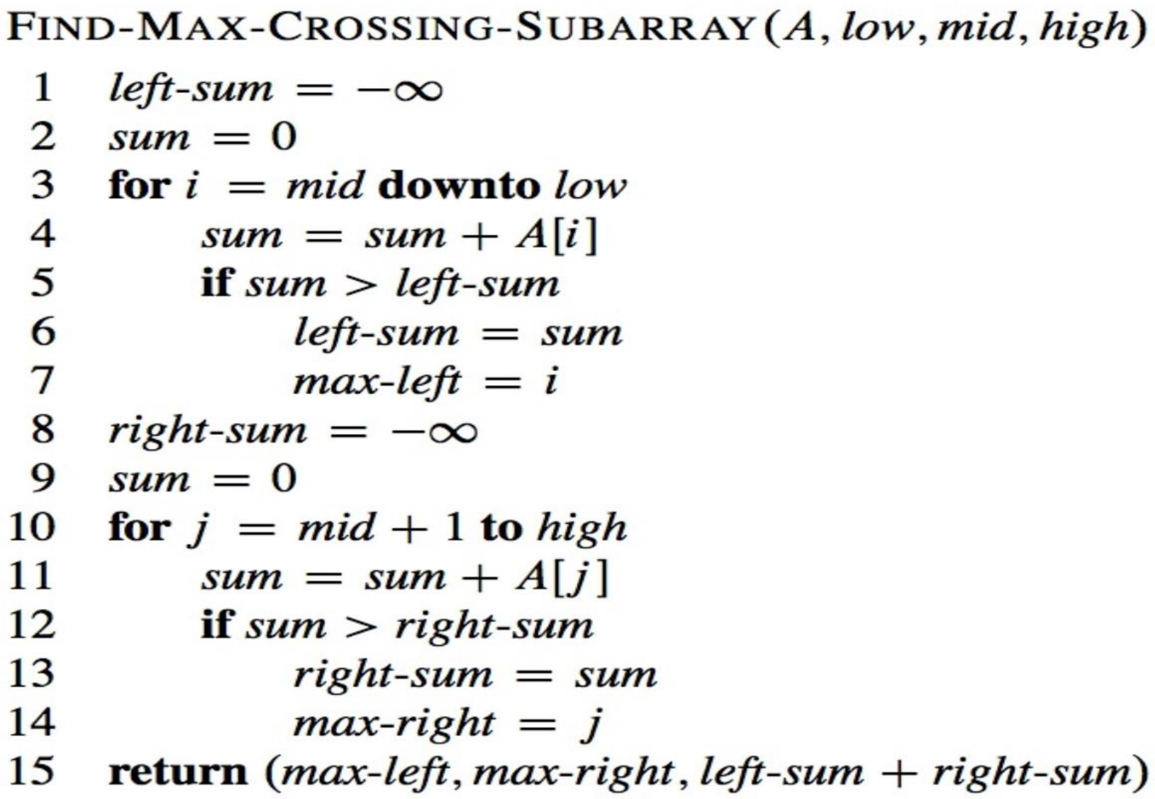


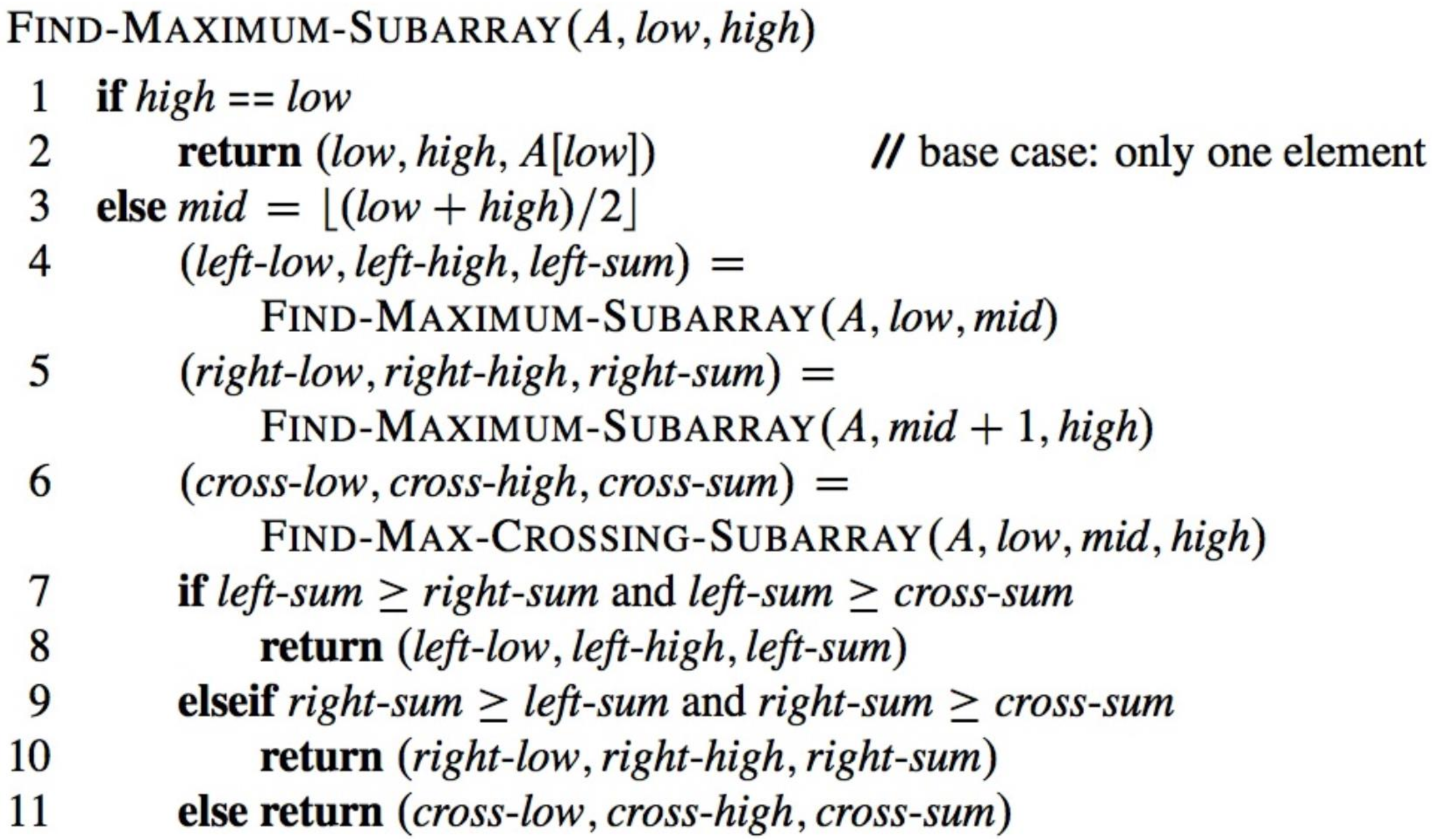
Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid + 1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid + 1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

Maximum Subarray Problem



```
FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
$left\text{-}sum = -\infty$
$sum = 0$
for $i = mid$ downto $low$
    $sum = sum + A[i]$
    if $sum > left\text{-}sum$
        $left\text{-}sum = sum$
        $max\text{-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
for $j = mid + 1$ to $high$
    $sum = sum + A[j]$
    if $sum > right\text{-}sum$
        $right\text{-}sum = sum$
        $max\text{-}right = j$
return ($max\text{-}left$, $max\text{-}right$, $left\text{-}sum + right\text{-}sum$)
```

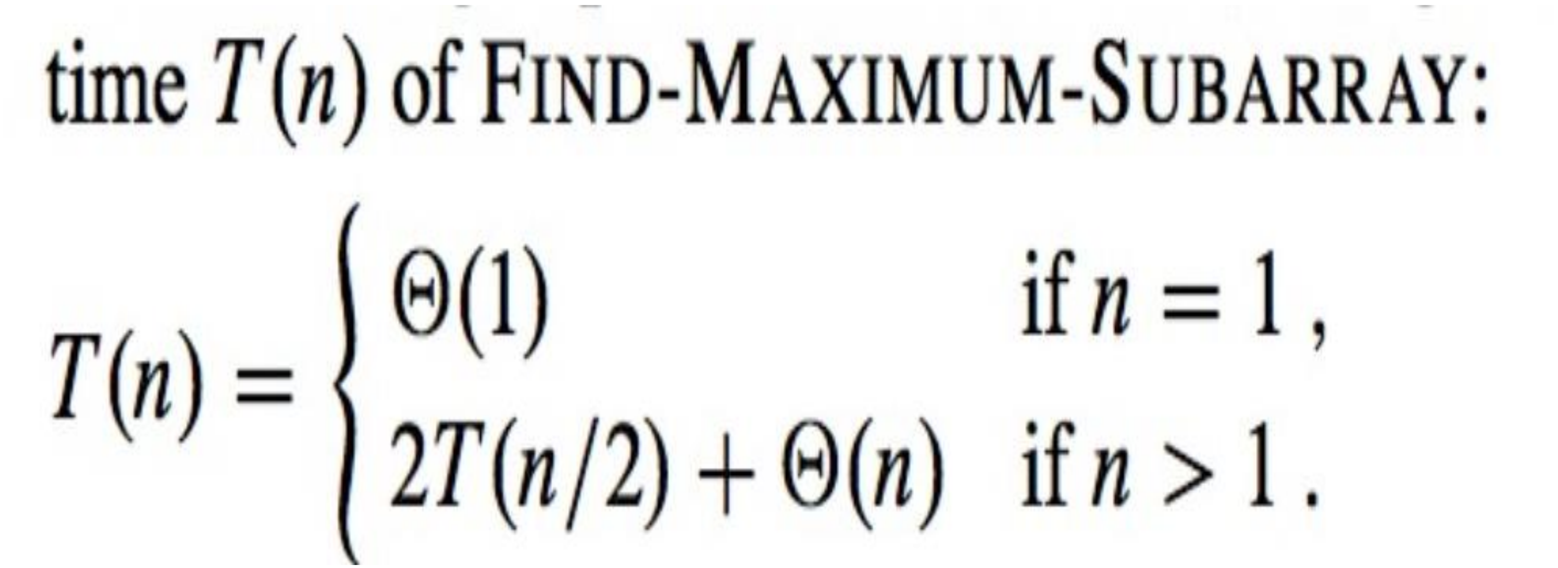
Maximum Subarray Problem



```
FIND-MAXIMUM-SUBARRAY($A$, $low$, $high$)
if $high == low$
    return ($low$, $high$, $A[low]$)                // base case: only one element
else $mid = \lfloor (low + high)/2 \rfloor$
    ($left\text{-}low$, $left\text{-}high$, $left\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A$, $low$, $mid$)
    ($right\text{-}low$, $right\text{-}high$, $right\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A$, $mid + 1$, $high$)
    ($cross\text{-}low$, $cross\text{-}high$, $cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low$, $left\text{-}high$, $left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low$, $right\text{-}high$, $right\text{-}sum$)
    else return ($cross\text{-}low$, $cross\text{-}high$, $cross\text{-}sum$)
```

UC Santa Cruz Extension in Silicon Valley

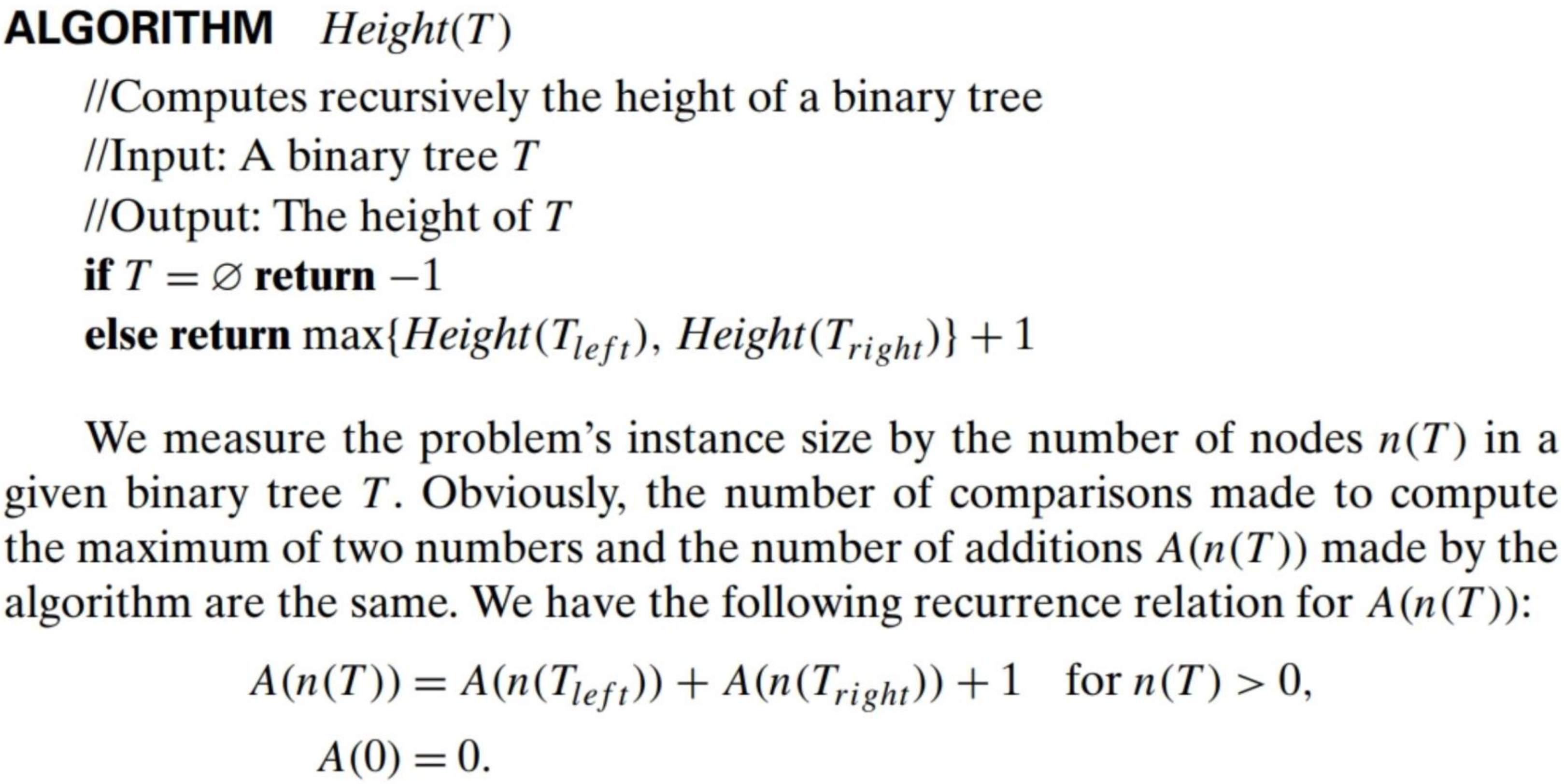
Maximum Subarray Problem



time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

Binary Tree Traversal



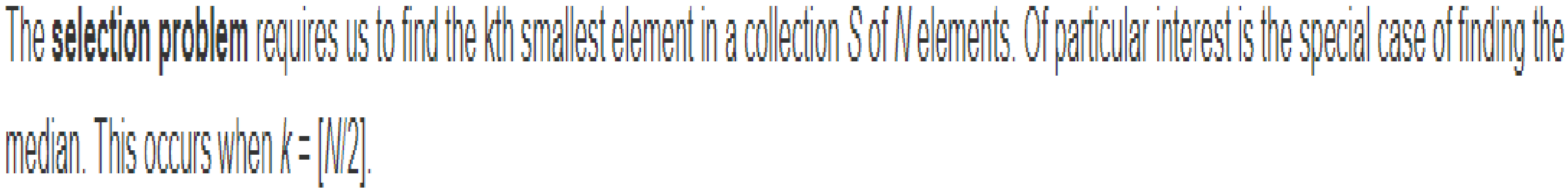
```
ALGORITHM  $Height(T)$
    //Computes recursively the height of a binary tree
    //Input: A binary tree $T$
    //Output: The height of $T$
    if $T = \varnothing$ return $-1$
    else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$
```

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

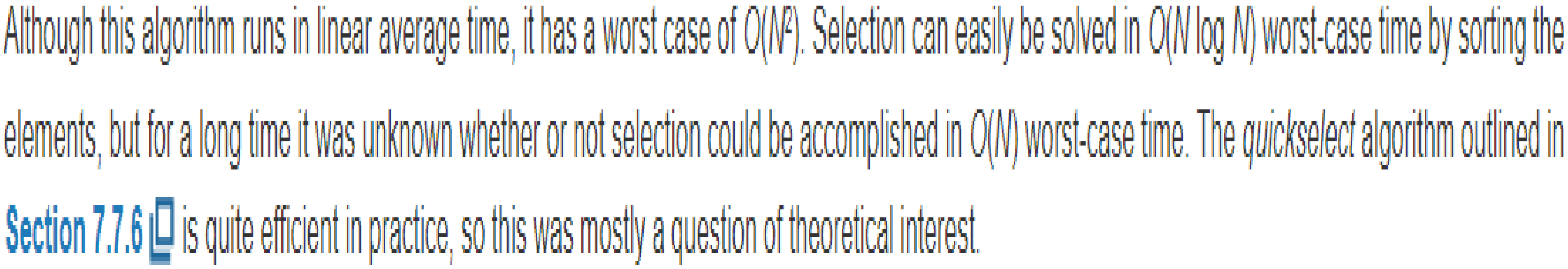
$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \quad \text{for } n(T) > 0,$$

$$A(0) = 0.$$

Selection Problem



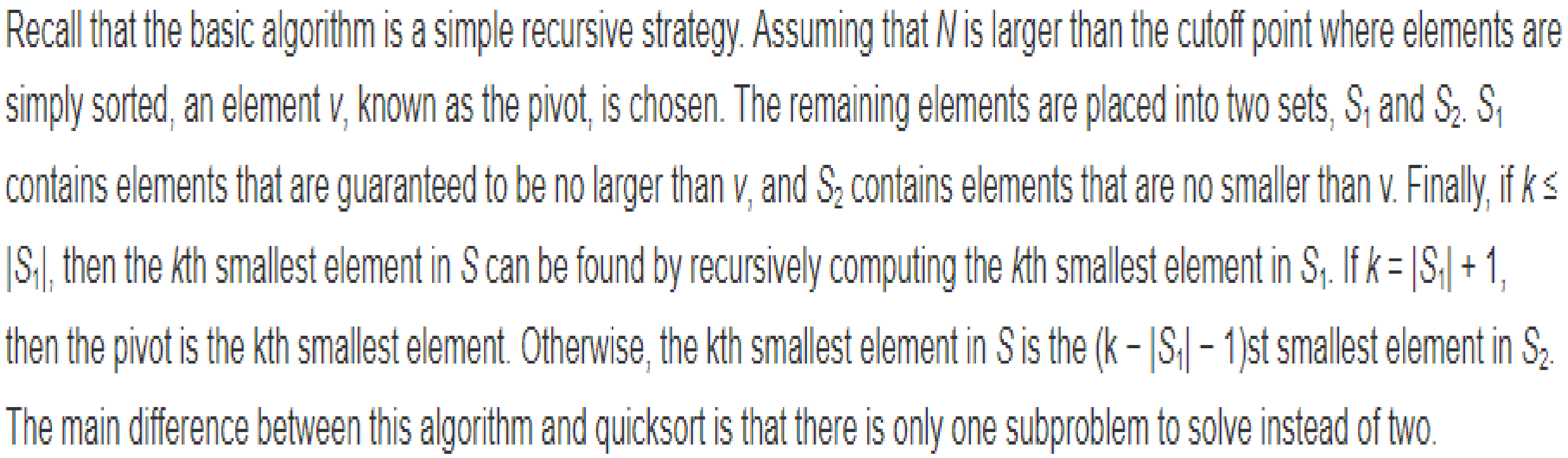
The **selection problem** requires us to find the *k*th smallest element in a collection *S* of *N* elements. Of particular interest is the special case of finding the median. This occurs when $k = \lceil N/2 \rceil$.



Although this algorithm runs in linear average time, it has a worst case of $O(N^2)$. Selection can easily be solved in $O(N \log N)$ worst-case time by sorting the elements, but for a long time it was unknown whether or not selection could be accomplished in $O(N)$ worst-case time. The *quickselect* algorithm outlined in Section 7.7.6 is quite efficient in practice, so this was mostly a question of theoretical interest.

Selection Problem

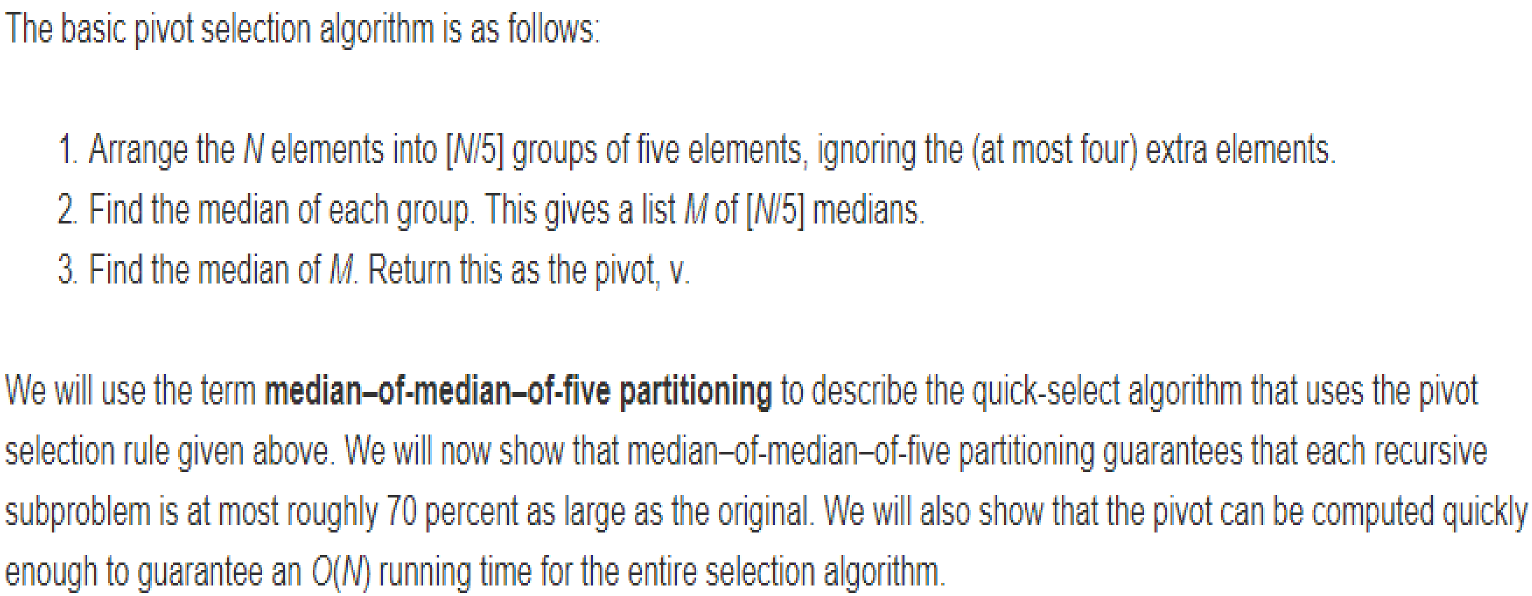
- The divide and conquer algorithm:



Recall that the basic algorithm is a simple recursive strategy. Assuming that N is larger than the cutoff point where elements are simply sorted, an element v, known as the pivot, is chosen. The remaining elements are placed into two sets, S_1 and S_2. S_1 contains elements that are guaranteed to be no larger than v, and S_2 contains elements that are no smaller than v. Finally, if $k \leq |S_1|$, then the kth smallest element in S can be found by recursively computing the kth smallest element in S_1. If $k = |S_1| + 1$, then the pivot is the kth smallest element. Otherwise, the kth smallest element in S is the $(k - |S_1| - 1)$st smallest element in S_2. The main difference between this algorithm and quicksort is that there is only one subproblem to solve instead of two.

Selection Problem

- Selecting a Pivot the right way:

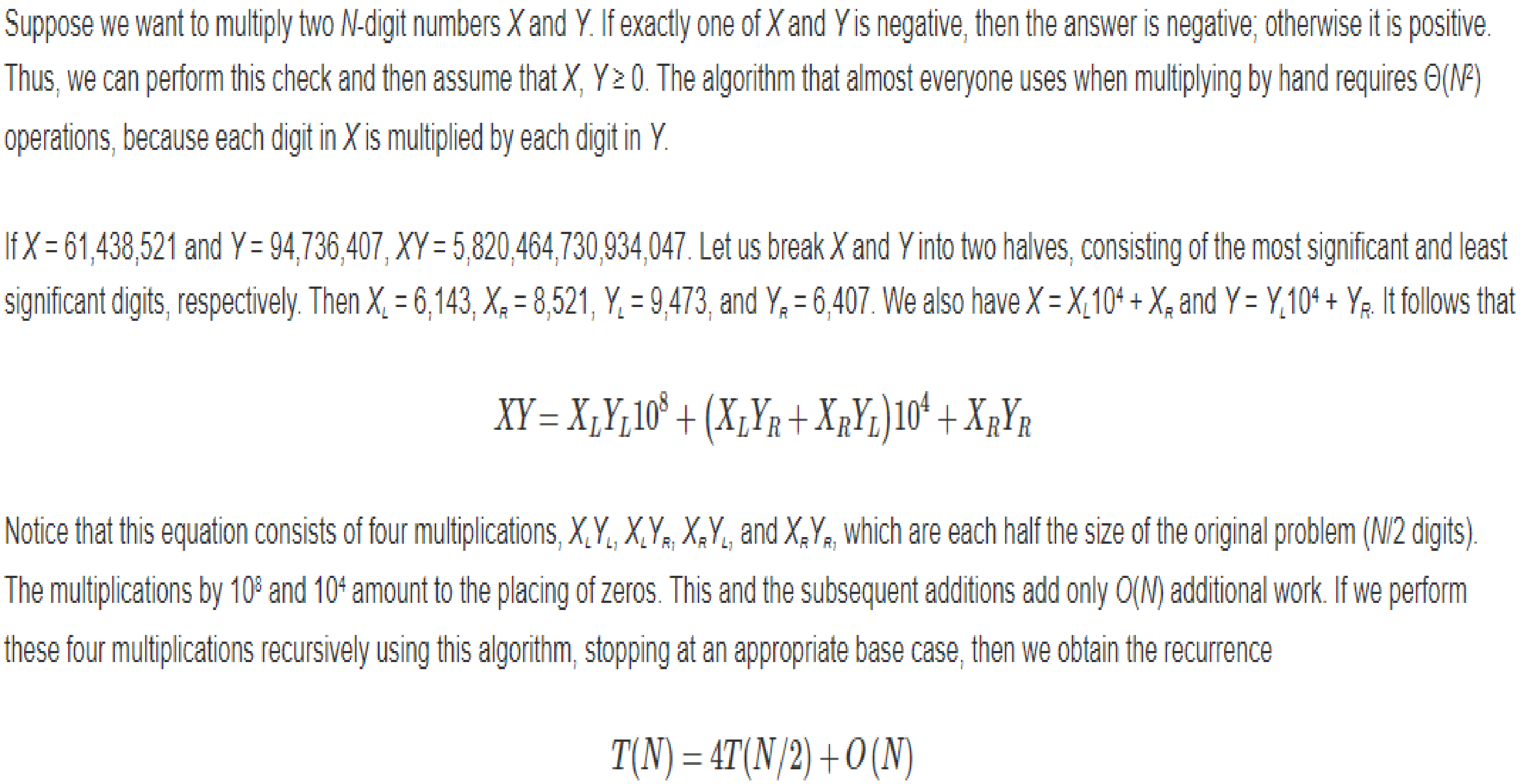


The basic pivot selection algorithm is as follows:

1. Arrange the *N* elements into [*N*/5] groups of five elements, ignoring the (at most four) extra elements.
2. Find the median of each group. This gives a list *M* of [*N*/5] medians.
3. Find the median of *M*. Return this as the pivot, v.

We will use the term **median–of-median–of-five partitioning** to describe the quick-select algorithm that uses the pivot selection rule given above. We will now show that median–of-median–of-five partitioning guarantees that each recursive subproblem is at most roughly 70 percent as large as the original. We will also show that the pivot can be computed quickly enough to guarantee an *O*(*N*) running time for the entire selection algorithm.

Multiplication of Large Integers



Suppose we want to multiply two *N*-digit numbers *X* and *Y*. If exactly one of *X* and *Y* is negative, then the answer is negative; otherwise it is positive. Thus, we can perform this check and then assume that $X, Y \geq 0$. The algorithm that almost everyone uses when multiplying by hand requires $\Theta(N^2)$ operations, because each digit in *X* is multiplied by each digit in *Y*.

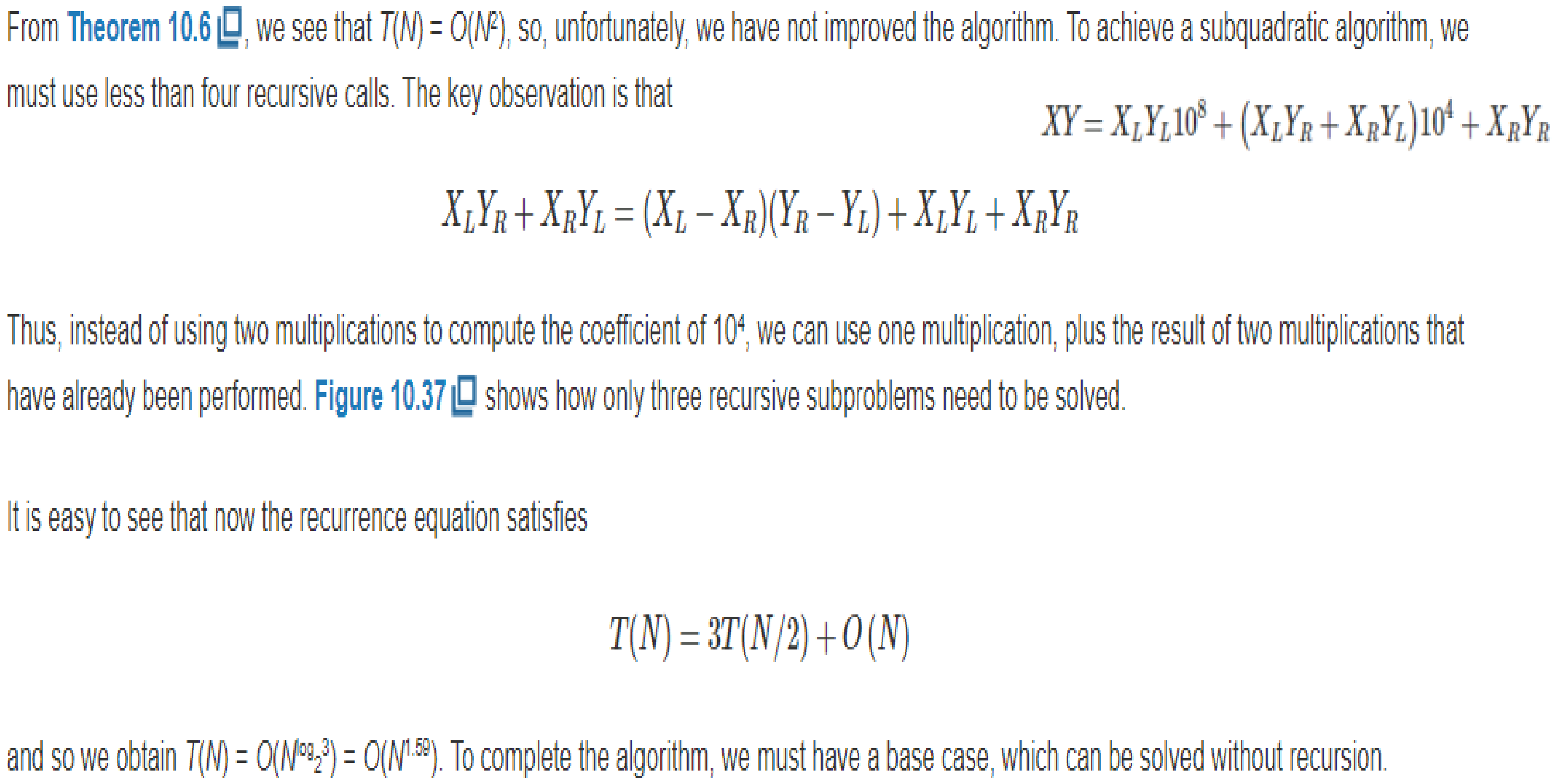
If $X = 61{,}438{,}521$ and $Y = 94{,}736{,}407$, $XY = 5{,}820{,}464{,}730{,}934{,}047$. Let us break *X* and *Y* into two halves, consisting of the most significant and least significant digits, respectively. Then $X_L = 6{,}143$, $X_R = 8{,}521$, $Y_L = 9{,}473$, and $Y_R = 6{,}407$. We also have $X = X_L 10^4 + X_R$ and $Y = Y_L 10^4 + Y_R$. It follows that

$$XY = X_L Y_L 10^8 + \left(X_L Y_R + X_R Y_L\right) 10^4 + X_R Y_R$$

Notice that this equation consists of four multiplications, $X_L Y_L$, $X_L Y_R$, $X_R Y_L$, and $X_R Y_R$, which are each half the size of the original problem (*N*/2 digits). The multiplications by 10^8 and 10^4 amount to the placing of zeros. This and the subsequent additions add only $O(N)$ additional work. If we perform these four multiplications recursively using this algorithm, stopping at an appropriate base case, then we obtain the recurrence

$$T(N) = 4T(N/2) + O(N)$$

Multiplication of Large Integers



From Theorem 10.6, we see that $T(N) = O(N^2)$, so, unfortunately, we have not improved the algorithm. To achieve a subquadratic algorithm, we must use less than four recursive calls. The key observation is that

$$XY = X_L Y_L 10^8 + (X_L Y_R + X_R Y_L) 10^4 + X_R Y_R$$

$$X_L Y_R + X_R Y_L = (X_L - X_R)(Y_R - Y_L) + X_L Y_L + X_R Y_R$$

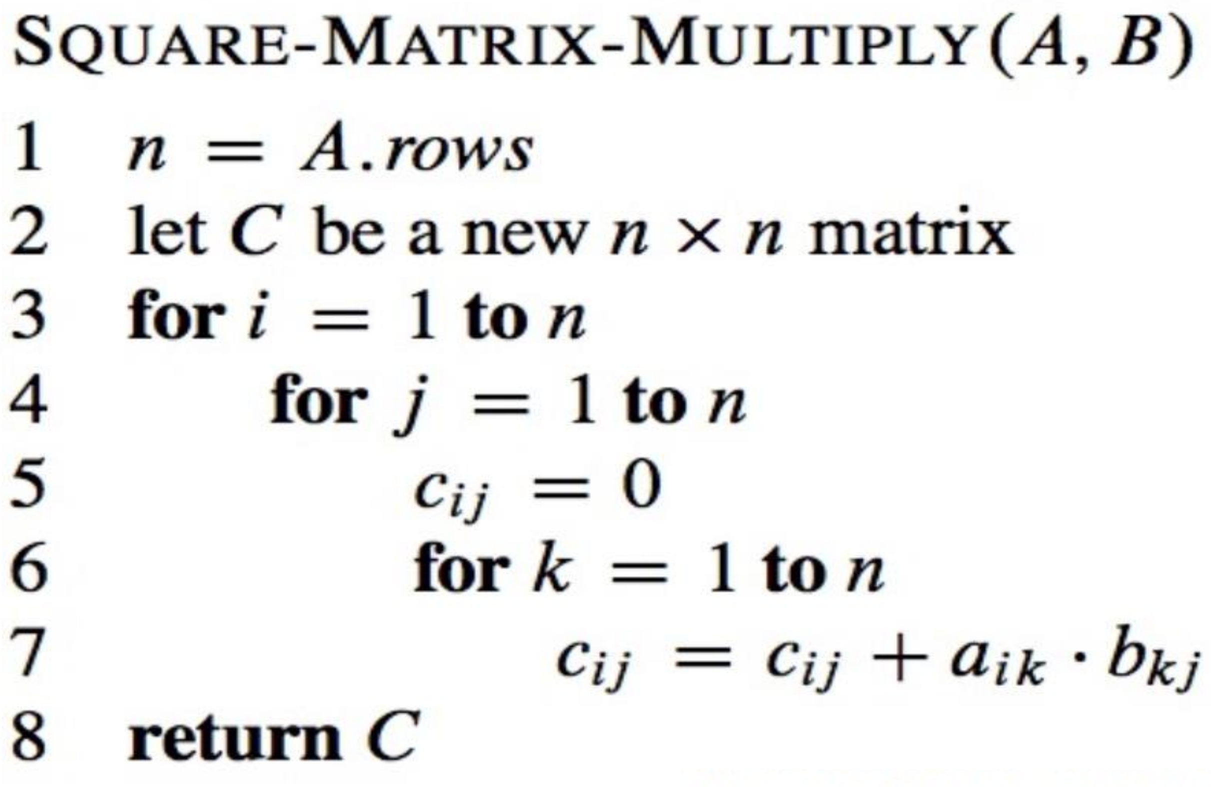
Thus, instead of using two multiplications to compute the coefficient of 10^4, we can use one multiplication, plus the result of two multiplications that have already been performed. Figure 10.37 shows how only three recursive subproblems need to be solved.

It is easy to see that now the recurrence equation satisfies

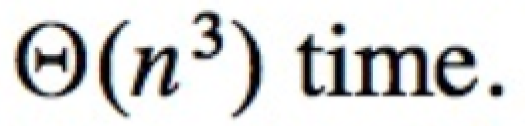
$$T(N) = 3T(N/2) + O(N)$$

and so we obtain $T(N) = O(N^{\log_2 3}) = O(N^{1.59})$. To complete the algorithm, we must have a base case, which can be solved without recursion.

Strassen’s Algorithm

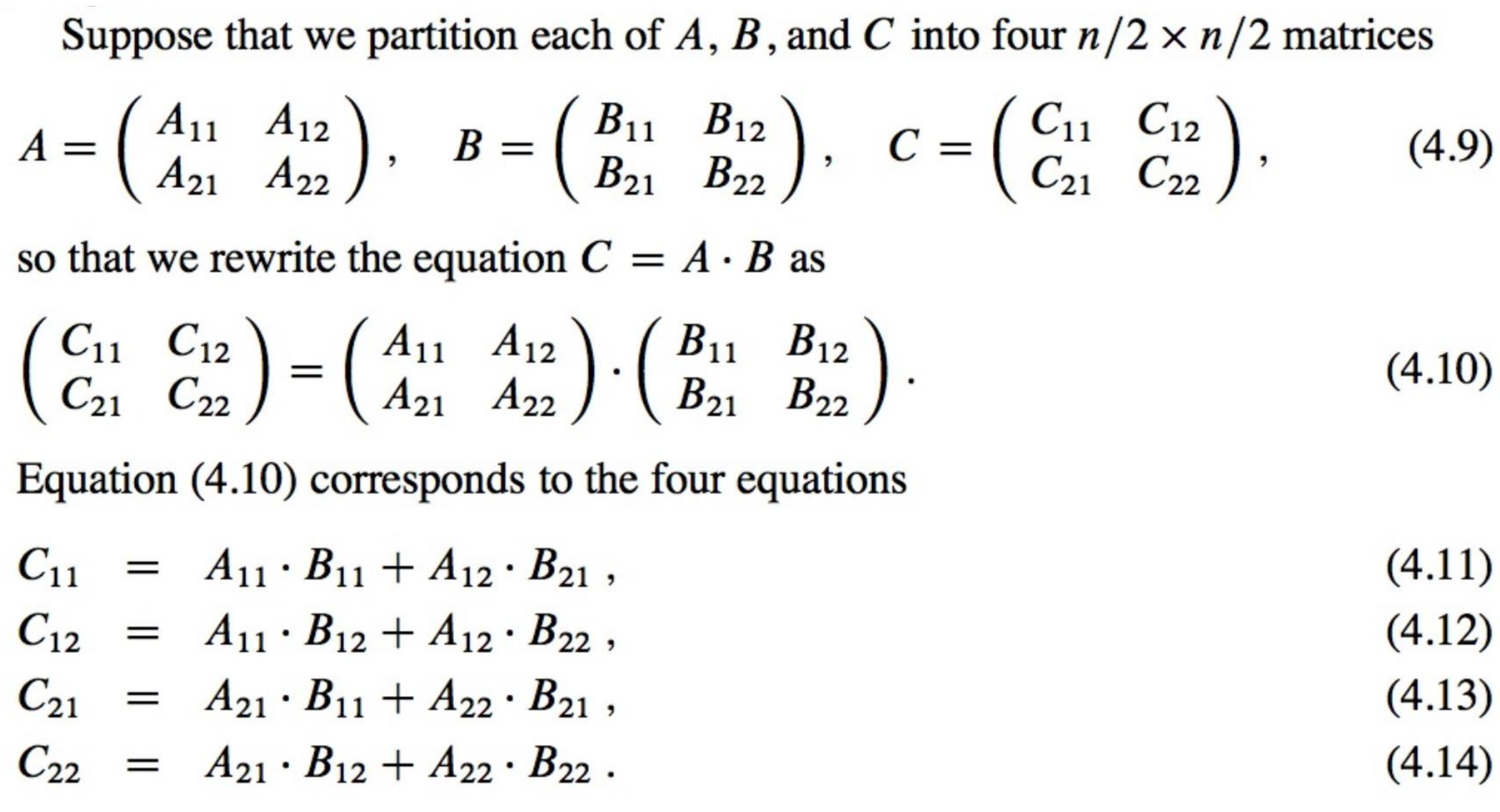


```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        $c_{ij} = 0$
        for k = 1 to n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C
```



$\Theta(n^3)$ time.

Strassen's Algorithm



Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

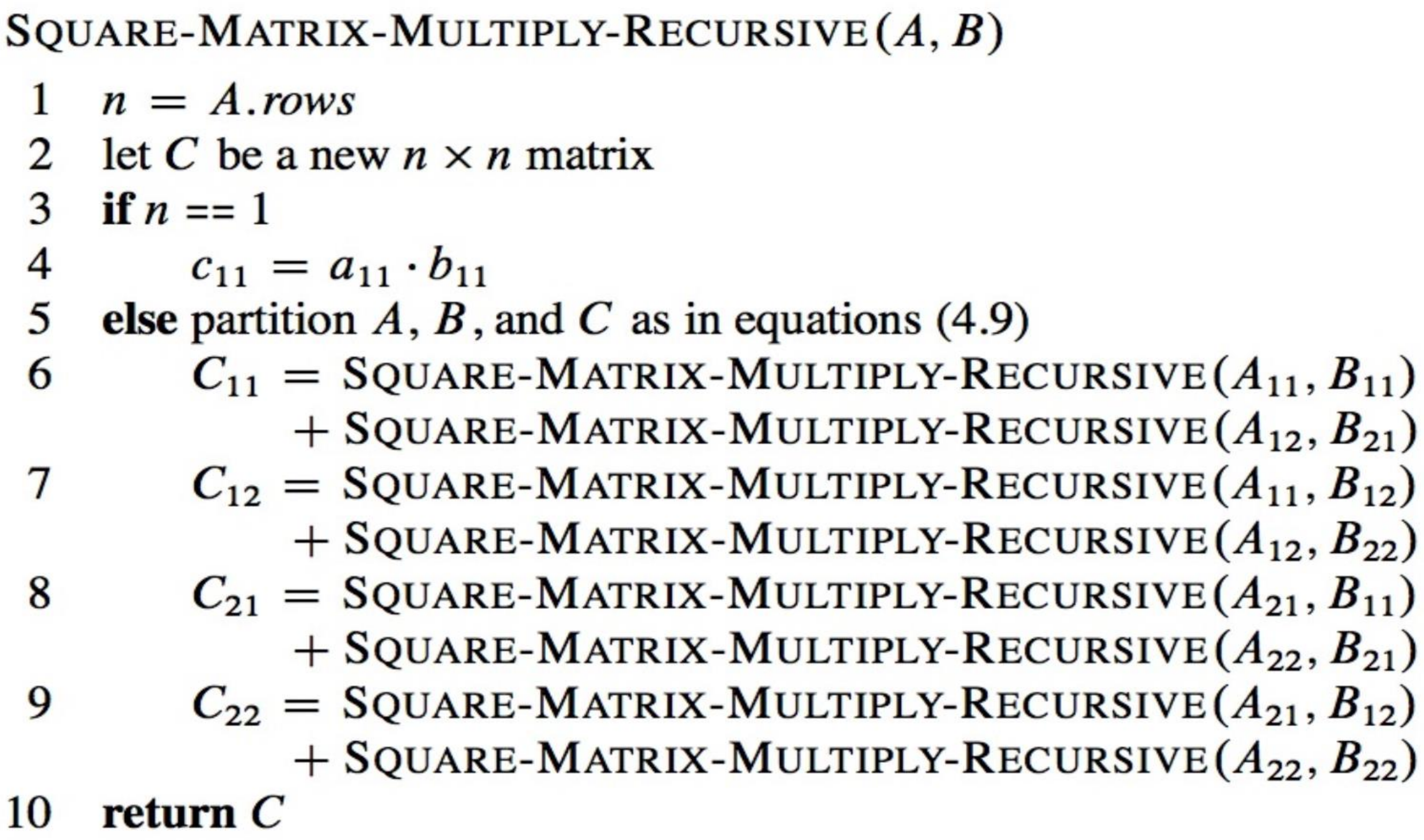
so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Strassen’s Algorithm



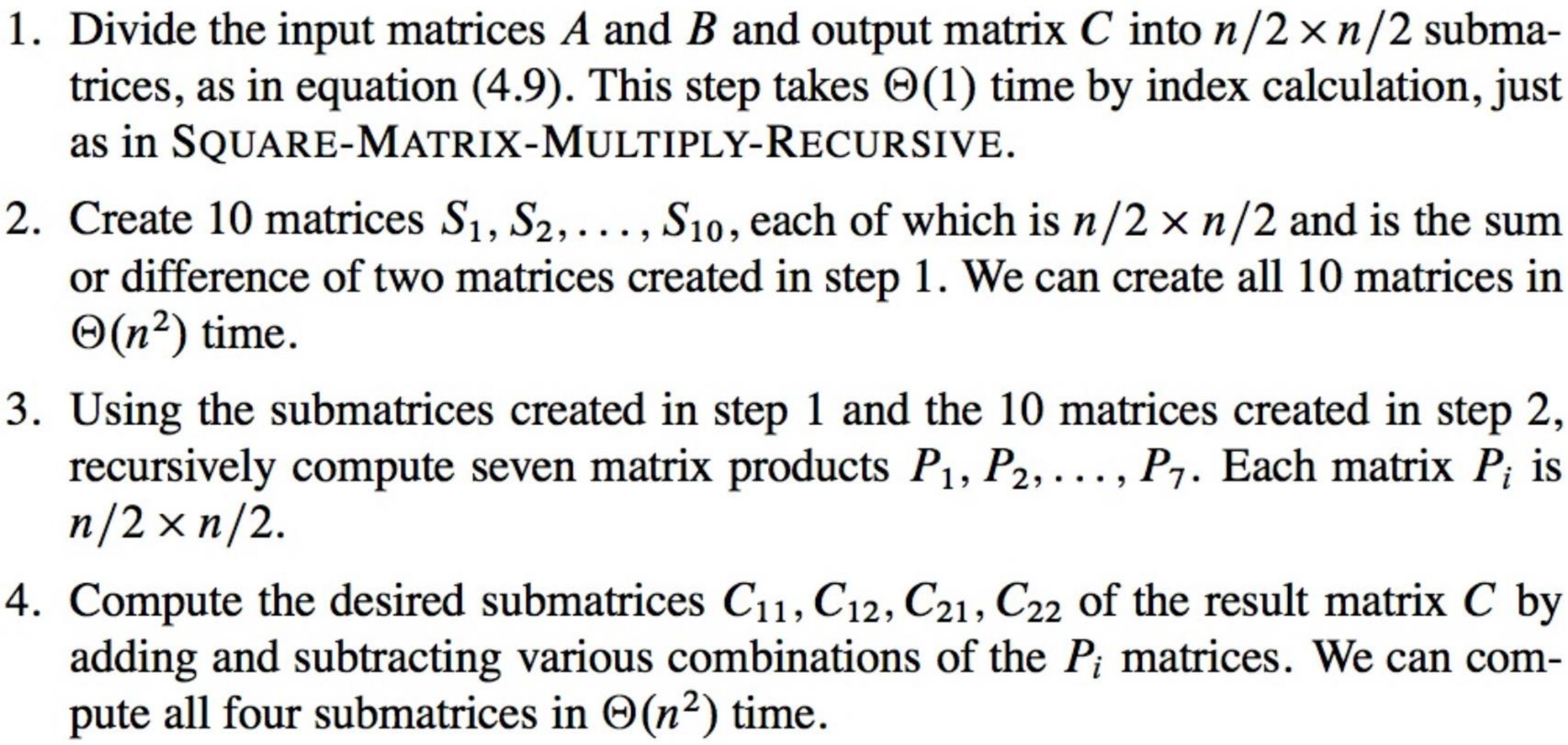
```
SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A, B)$
$n = A.rows$
let $C$ be a new $n \times n$ matrix
**if** $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$
**else** partition $A$, $B$, and $C$ as in equations (4.9)
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21})$
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22})$
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21})$
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22})$
**return** $C$
```

Strassen’s Algorithm

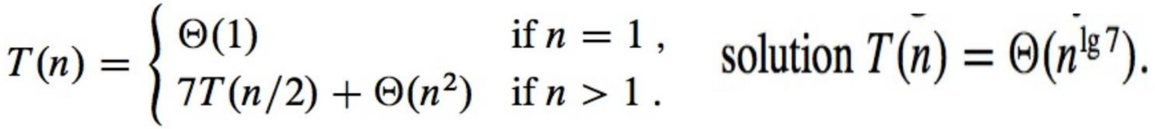
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

$$\text{solution } T(n) = \Theta(n^3).$$

Strassen’s Algorithm

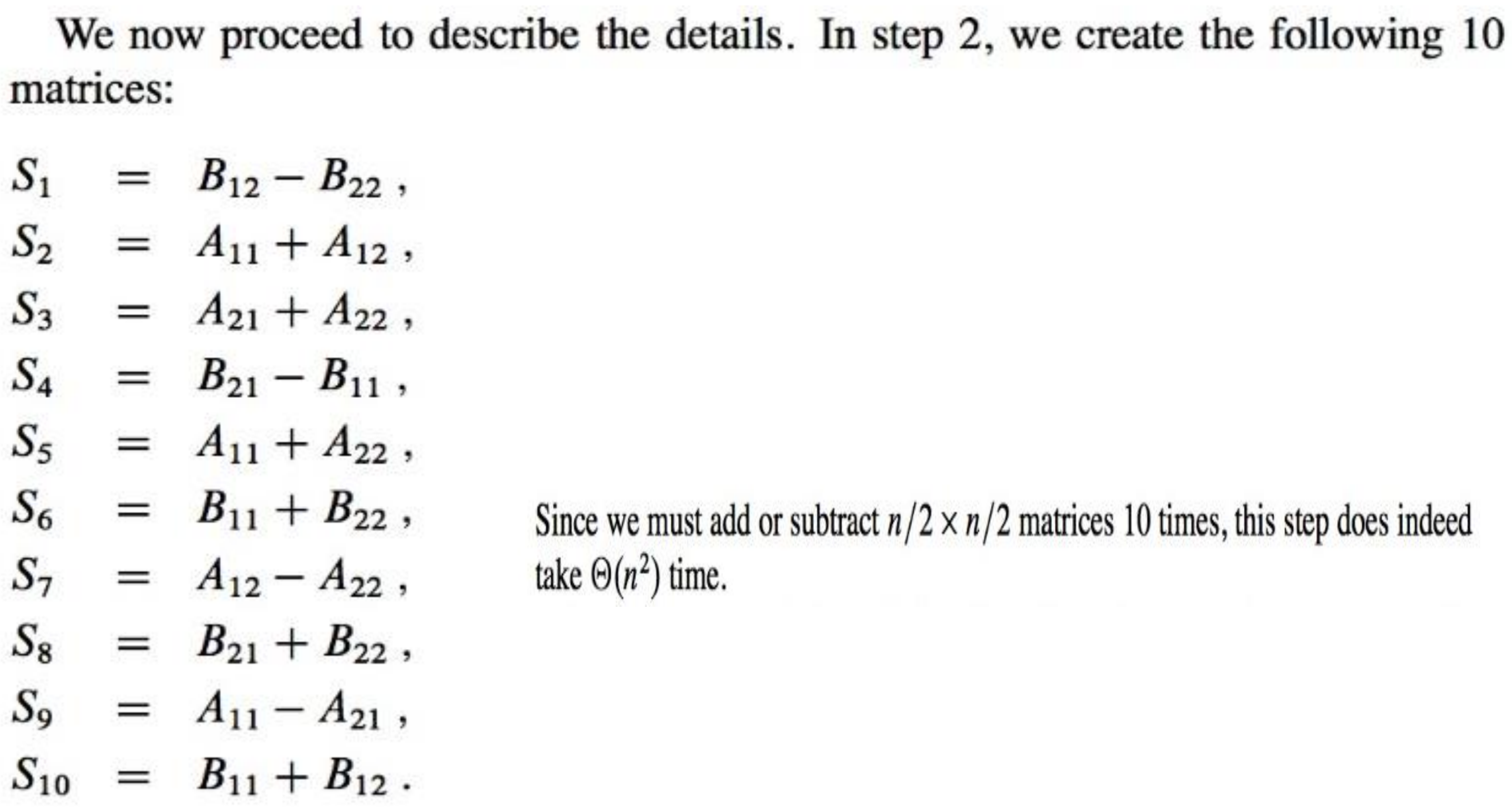


1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.



$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases} \quad \text{solution } T(n) = \Theta(n^{\lg 7}).$$

Strassen’s Algorithm

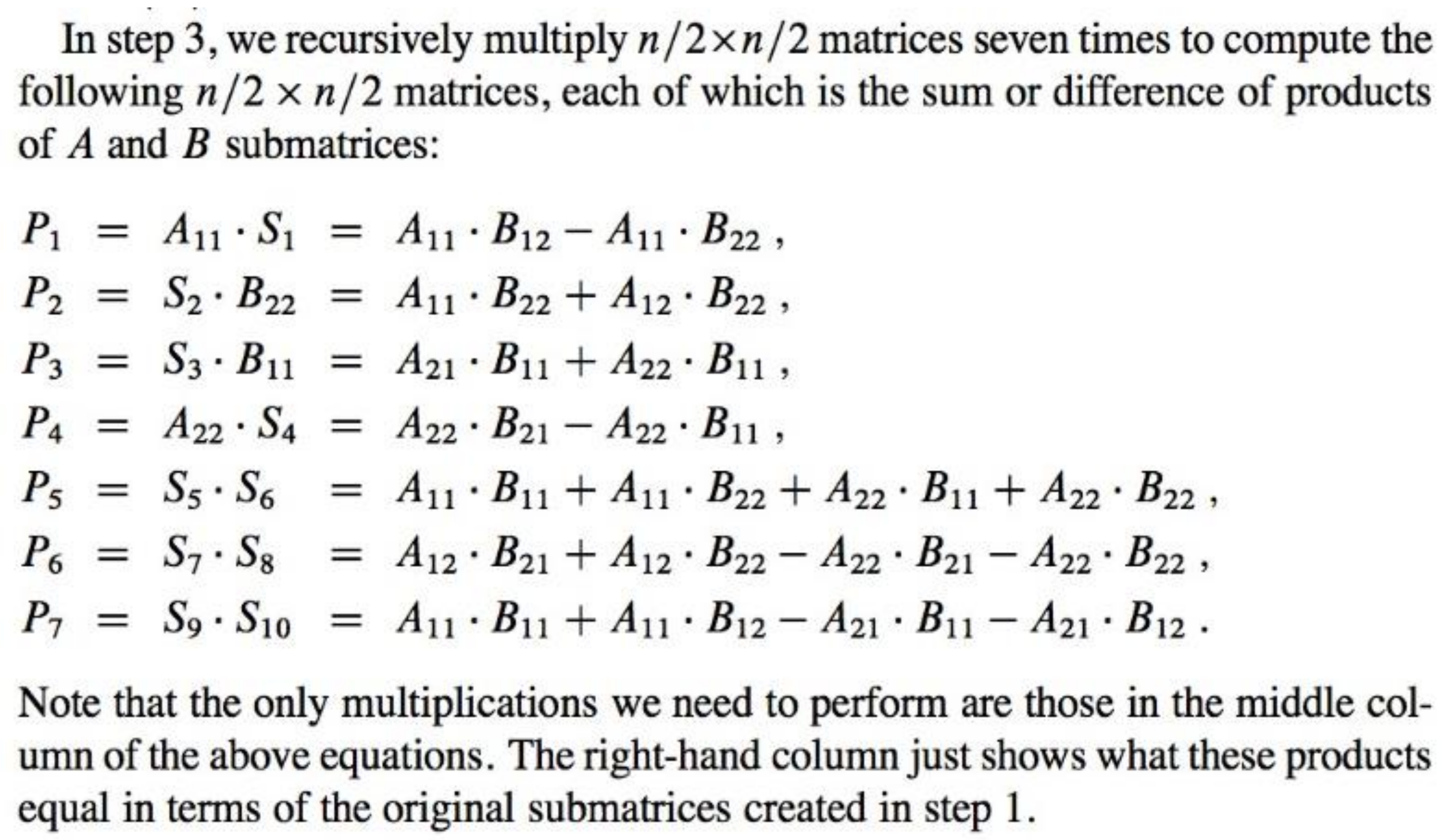


We now proceed to describe the details. In step 2, we create the following 10 matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step does indeed take $\Theta(n^2)$ time.

Strassen’s Algorithm

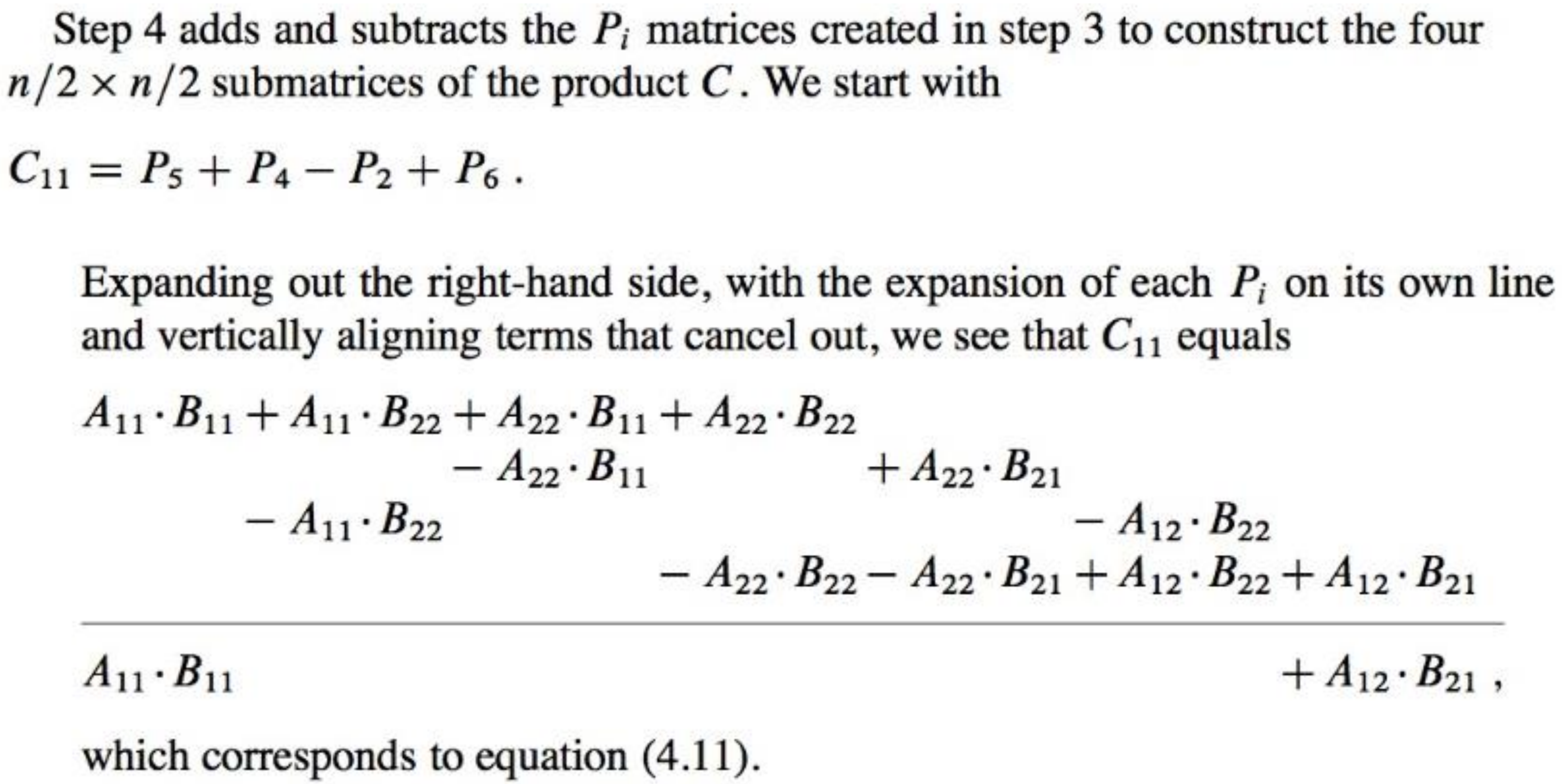


In step 3, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}$$

Note that the only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1.

Strassen's Algorithm



Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product C. We start with

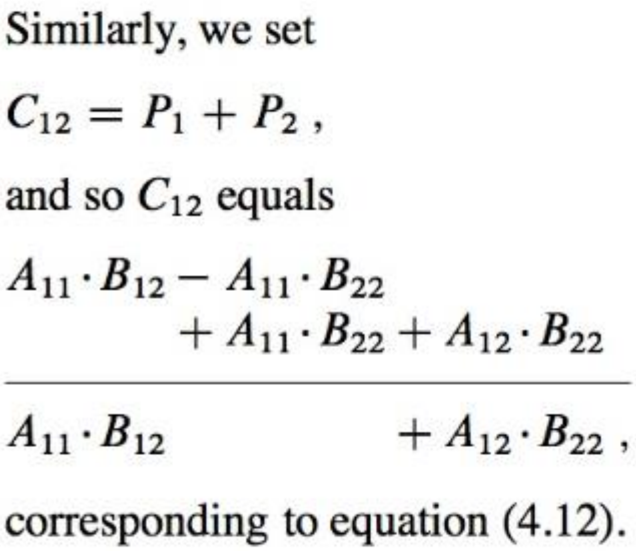
$$C_{11} = P_5 + P_4 - P_2 + P_6 \,.$$

Expanding out the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that C_{11} equals

$$
\begin{array}{lllllll}
A_{11}\cdot B_{11} & + A_{11}\cdot B_{22} & + A_{22}\cdot B_{11} & + A_{22}\cdot B_{22} & & & \\
 & & - A_{22}\cdot B_{11} & & + A_{22}\cdot B_{21} & & \\
 & - A_{11}\cdot B_{22} & & & & - A_{12}\cdot B_{22} & \\
 & & & - A_{22}\cdot B_{22} & - A_{22}\cdot B_{21} & + A_{12}\cdot B_{22} & + A_{12}\cdot B_{21} \\
\hline
A_{11}\cdot B_{11} & & & & & & + A_{12}\cdot B_{21} \,,
\end{array}
$$

which corresponds to equation (4.11).

Strassen’s Algorithm



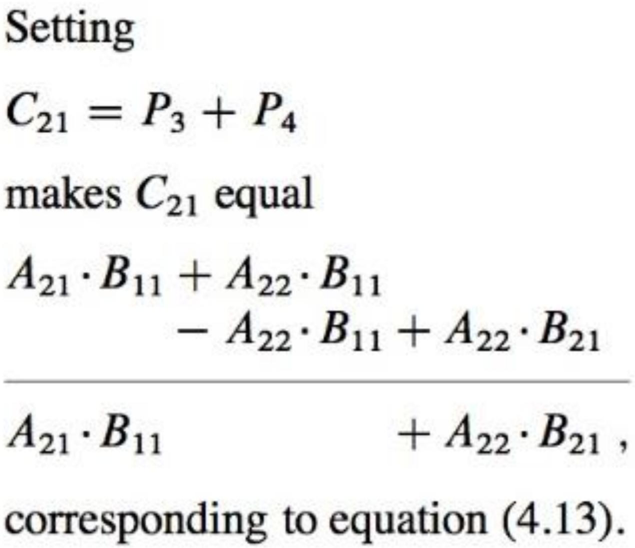
Similarly, we set

$$C_{12} = P_1 + P_2 ,$$

and so C_{12} equals

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad\qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad\qquad + A_{12} \cdot B_{22} , \end{array}$$

corresponding to equation (4.12).



Setting

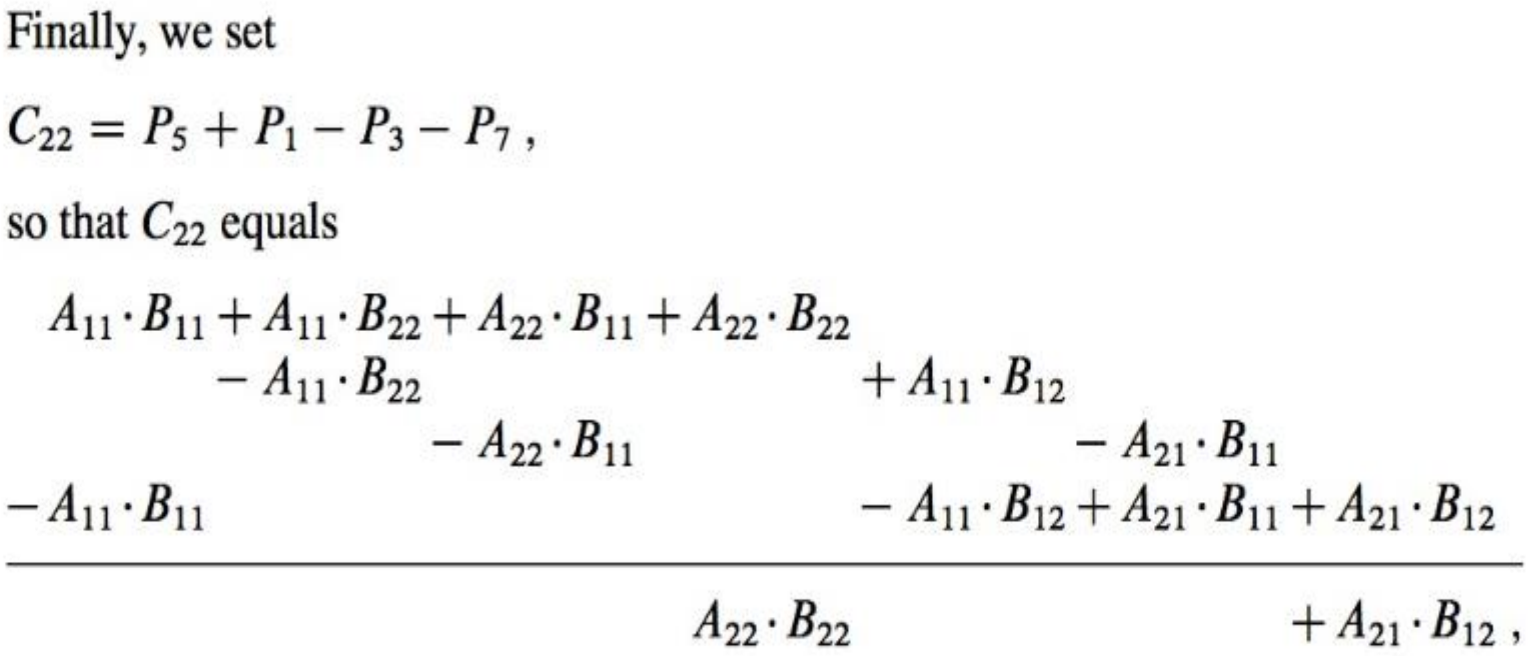
$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad\qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad\qquad + A_{22} \cdot B_{21} , \end{array}$$

corresponding to equation (4.13).

Strassen’s Algorithm



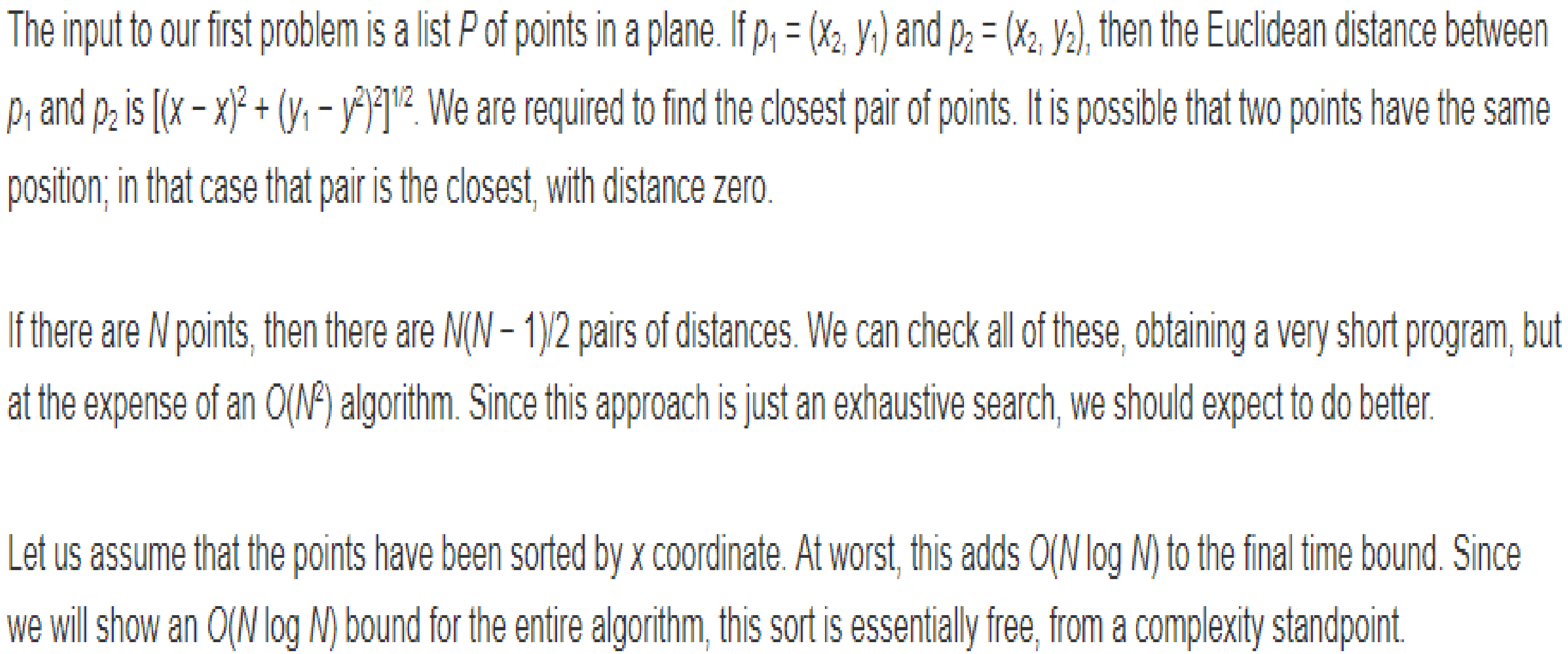
Finally, we set

$$C_{22} = P_5 + P_1 - P_3 - P_7 \,,$$

so that C_{22} equals

$$\begin{array}{rrrrrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12} \,,
\end{array}$$

Closest Points



The input to our first problem is a list *P* of points in a plane. If $p_1 = (x_{2,} y_1)$ and $p_2 = (x_{2,} y_2)$, then the Euclidean distance between p_1 and p_2 is $[(x - x)^2 + (y_1 - y^2)^2]^{1/2}$. We are required to find the closest pair of points. It is possible that two points have the same position; in that case that pair is the closest, with distance zero.

If there are *N* points, then there are $N(N - 1)/2$ pairs of distances. We can check all of these, obtaining a very short program, but at the expense of an $O(N^2)$ algorithm. Since this approach is just an exhaustive search, we should expect to do better.

Let us assume that the points have been sorted by *x* coordinate. At worst, this adds $O(N \log N)$ to the final time bound. Since we will show an $O(N \log N)$ bound for the entire algorithm, this sort is essentially free, from a complexity standpoint.

Closest Points

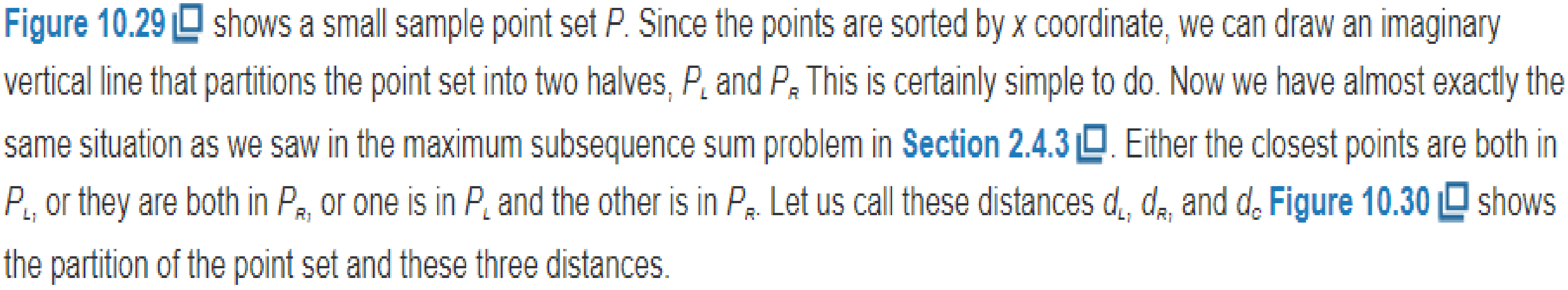


Figure 10.29 shows a small sample point set *P*. Since the points are sorted by *x* coordinate, we can draw an imaginary vertical line that partitions the point set into two halves, P_L and P_R This is certainly simple to do. Now we have almost exactly the same situation as we saw in the maximum subsequence sum problem in **Section 2.4.3**. Either the closest points are both in P_L, or they are both in P_R, or one is in P_L and the other is in P_R. Let us call these distances d_L, d_R, and d_C **Figure 10.30** shows the partition of the point set and these three distances.

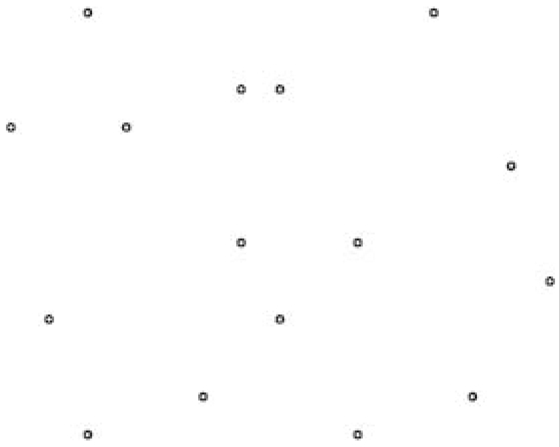


Figure 10.29 A small point set

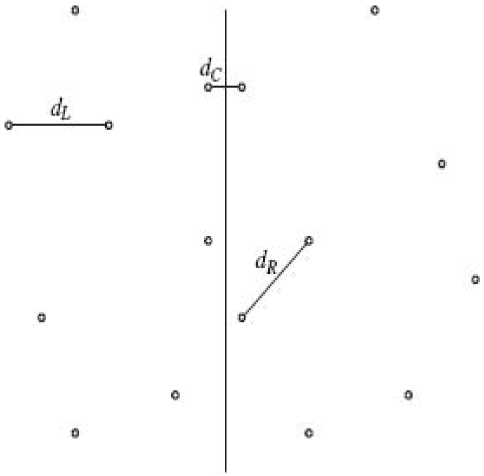
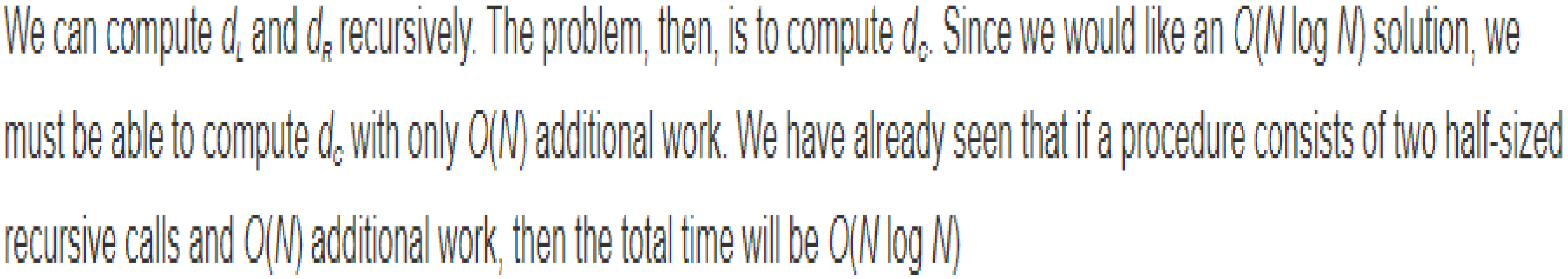


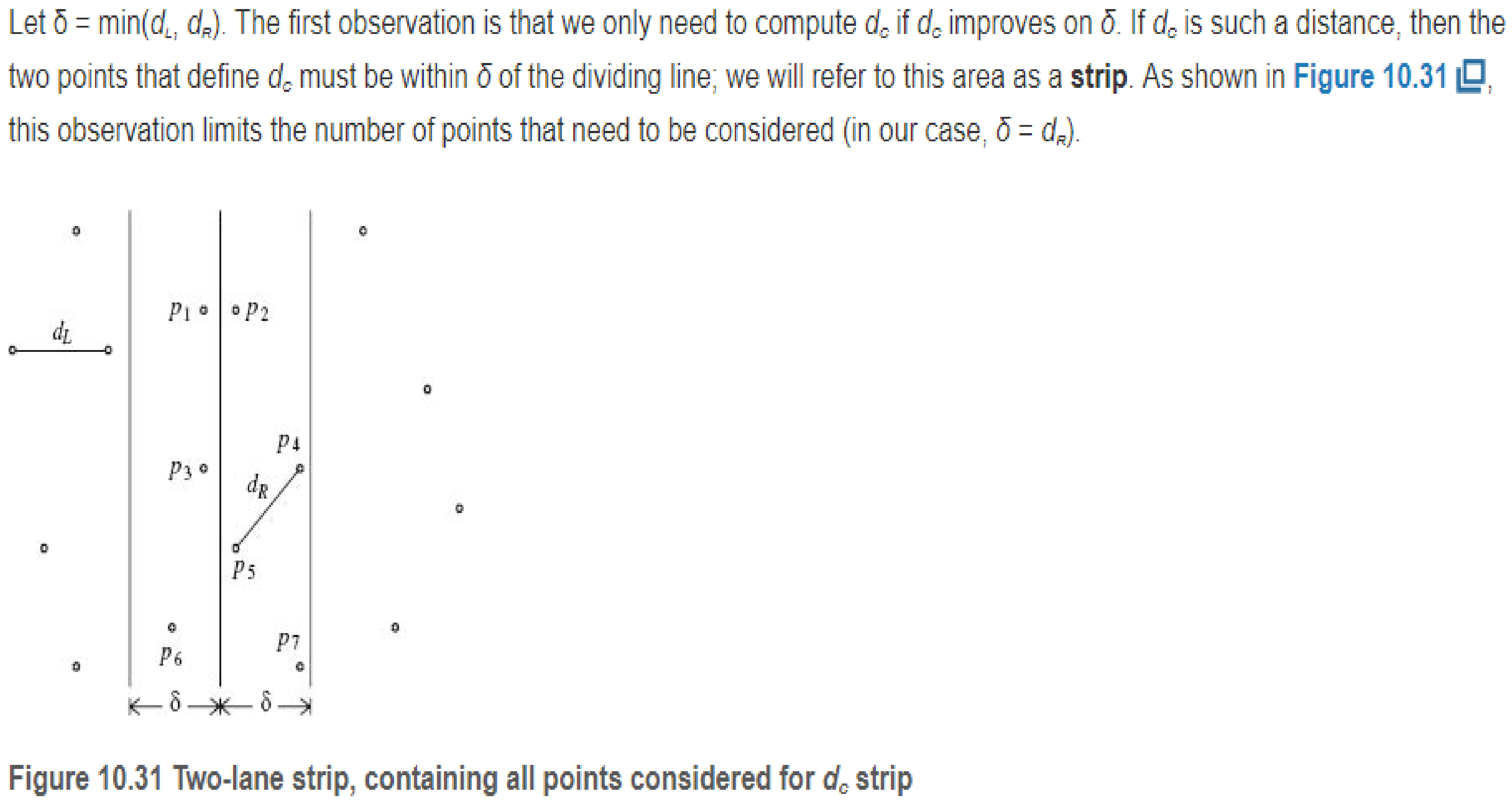
Figure 10.30 ***P*** **partitioned into** ***P_L*** **and** ***P_R*****; shortest distances are shown**

Closest Points



We can compute d_L and d_R recursively. The problem, then, is to compute d_C. Since we would like an $O(N \log N)$ solution, we must be able to compute d_C with only $O(N)$ additional work. We have already seen that if a procedure consists of two half-sized recursive calls and $O(N)$ additional work, then the total time will be $O(N \log N)$

Closest Points

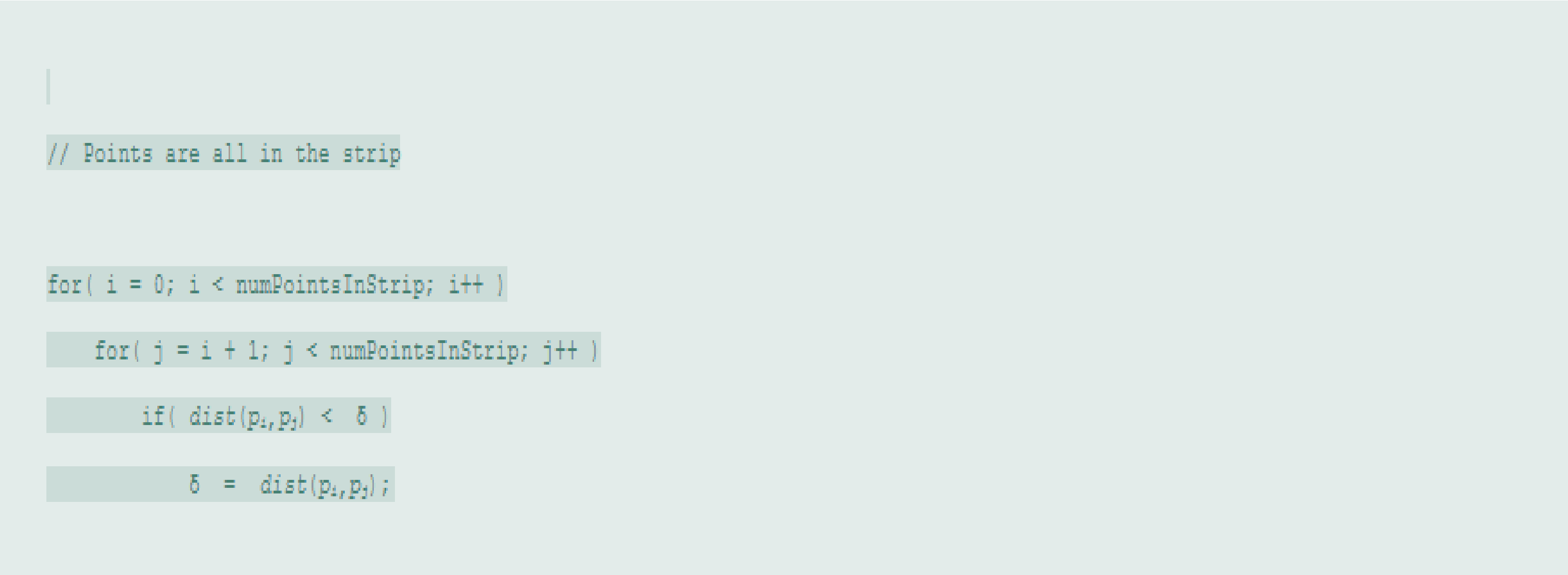


Let δ = min(d_L, d_R). The first observation is that we only need to compute d_c if d_c improves on δ. If d_c is such a distance, then the two points that define d_c must be within δ of the dividing line; we will refer to this area as a **strip**. As shown in Figure 10.31, this observation limits the number of points that need to be considered (in our case, δ = d_R).

Figure 10.31 Two-lane strip, containing all points considered for d_c strip

Closest Points

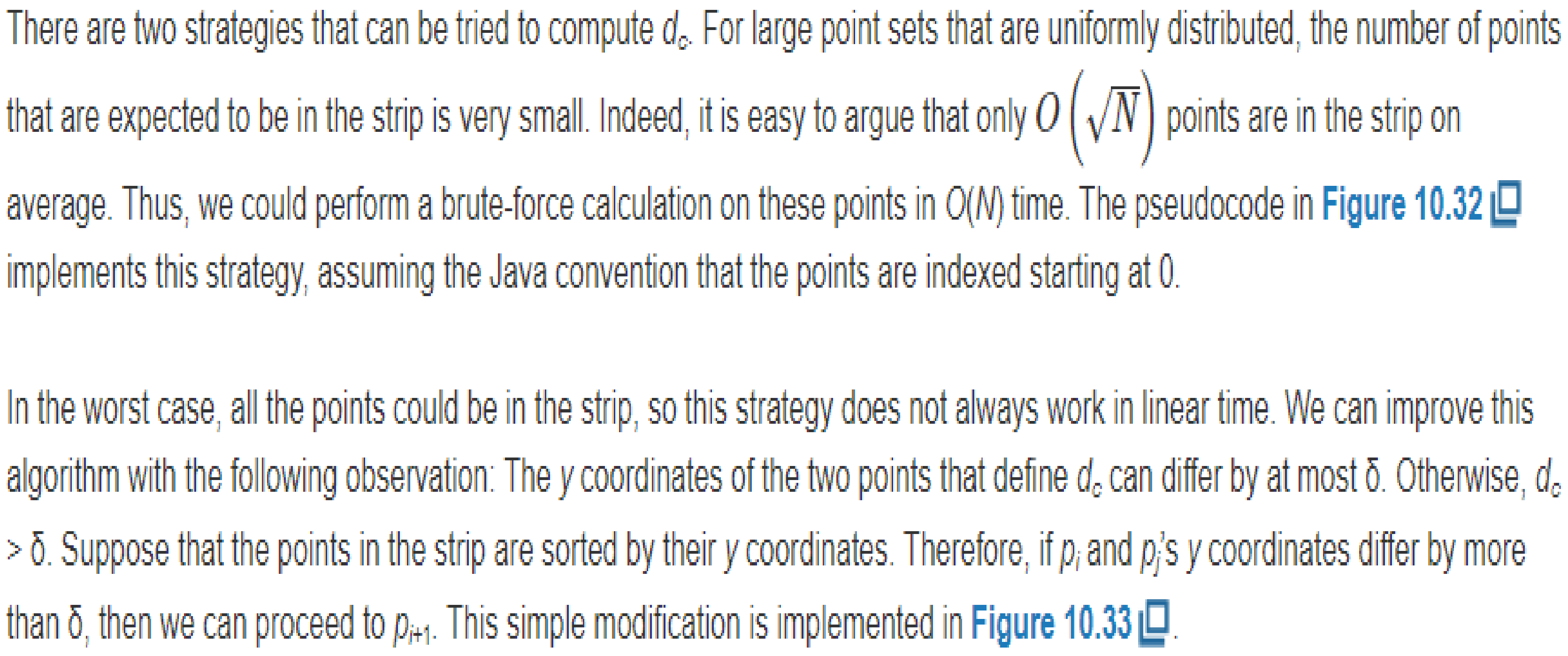
Figure 10.32 Brute-force calculation of min(δ, d_c)



```
// Points are all in the strip

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( dist(pi,pj) <  δ )
            δ  =  dist(pi,pj);
```

Closest Points

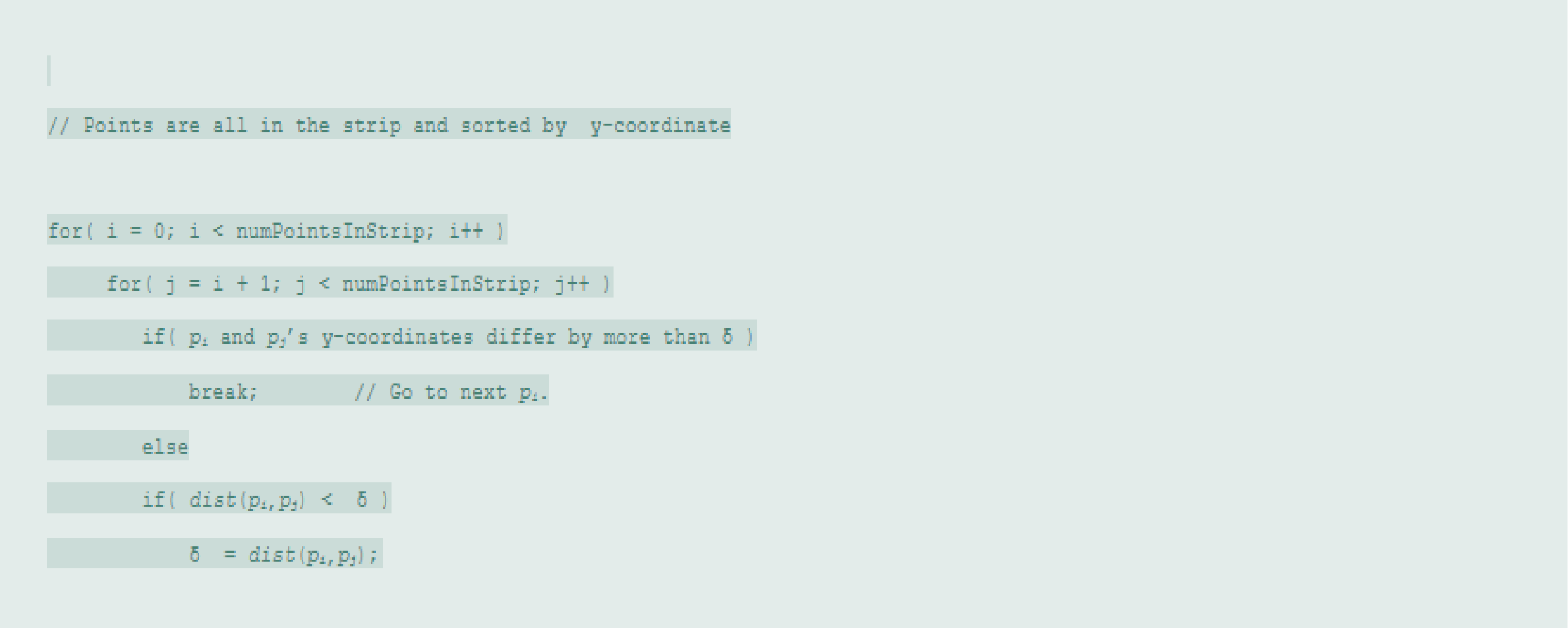


There are two strategies that can be tried to compute d_c. For large point sets that are uniformly distributed, the number of points that are expected to be in the strip is very small. Indeed, it is easy to argue that only $O\left(\sqrt{N}\right)$ points are in the strip on average. Thus, we could perform a brute-force calculation on these points in $O(N)$ time. The pseudocode in Figure 10.32 implements this strategy, assuming the Java convention that the points are indexed starting at 0.

In the worst case, all the points could be in the strip, so this strategy does not always work in linear time. We can improve this algorithm with the following observation: The *y* coordinates of the two points that define d_c can differ by at most δ. Otherwise, $d_c > \delta$. Suppose that the points in the strip are sorted by their *y* coordinates. Therefore, if p_i and p_j's *y* coordinates differ by more than δ, then we can proceed to p_{i+1}. This simple modification is implemented in Figure 10.33.

Closest Points

Figure 10.33 Refined calculation of min(δ, d_c)



```
// Points are all in the strip and sorted by  y-coordinate

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( p_i and p_j's y-coordinates differ by more than δ )
            break;         // Go to next p_i.
        else
        if( dist(p_i,p_j) <  δ )
            δ  = dist(p_i,p_j);
```

Closest Points

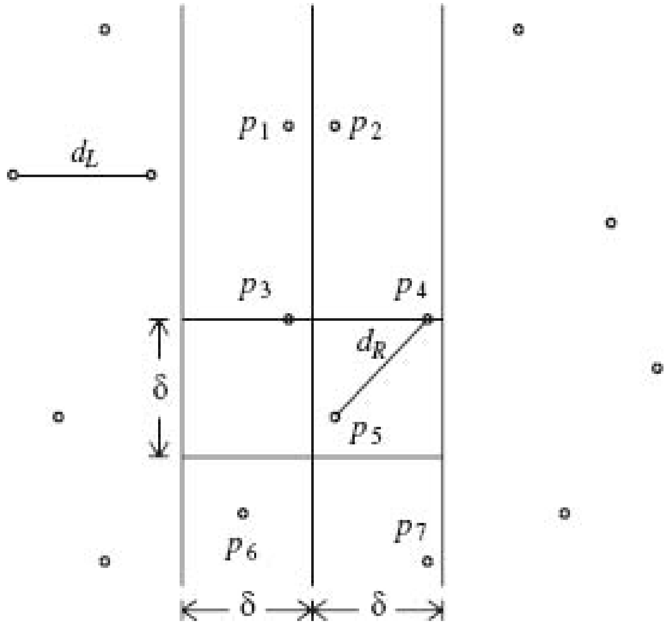
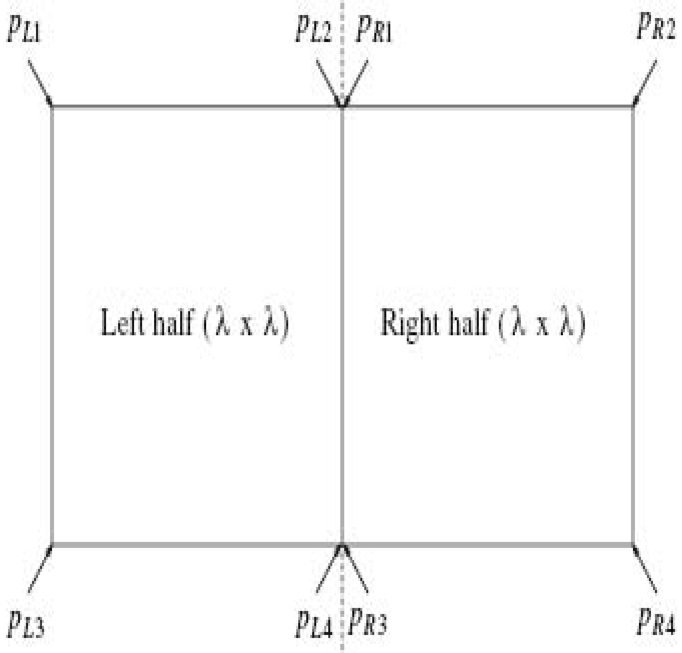


Figure 10.34 Only p_4 and p_5 are considered in the second for loop

Closest Points



p_{L1}
p_{L2}
p_{R1}
p_{R2}
Left half (λ x λ)
Right half (λ x λ)
p_{L3}
p_{L4}
p_{R3}
p_{R4}

Figure 10.35 At most eight points fit in the rectangle; there are two coordinates shared by two points each

Convex Hull Problem