UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER

2.1 BRUTE FORCE

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.

Examples:

1. Computing a^n : a * a * a * … * a (n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Selection Sort

- First scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
- Then scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position in the sorted list.
- Generally, on the ***i***th pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with A_i:

$$A_0 \le A_1 \le \ldots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions | *the last n – i elements*

- After n − 1 passes, the list is sorted.

ALGORITHM SelectionSort(A[0..n − 1])

//Sorts a given array by selection sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in nondecreasing order

for *i* ← *0 to n − 2* **do**

min ← *i*

for j ← *i* + 1 to *n* − 1 do

if A[*j*]<A[*min*] *min* ← *j*

swap A[*i*] and A[*min*]

| 89 | 45 | 68 | 90 | 29 | 34 | **17**

17 | 45 | 68 | 90 | **29** | 34 | 89

17 | 29 | 68 | 90 | 45 | **34** | 89

17 | 29 | 34 | 90 | **45** | 68 | 89

17 | 29 | 34 | 45 | 90 | **68** | 89

17 | 29 | 34 | 45 | 68 | 90 | **89**

17 29 34 45 68 89 | 90

The sorting of list 89, 45, 68, 90, 29, 34, 17 is illustrated with the selection sort algorithm.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Note: The number of key swaps is only **Θ(n),** or, more precisely n – 1.

Bubble Sort

The bubble sorting algorithm is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after n − 1 passes the list is sorted. Pass i ($0 \le i \le n-2$) of bubble sort can be represented by the following: $A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$

ALGORITHM BubbleSort(A[0..n − 1])

//Sorts a given array by bubble sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in nondecreasing order

for *i* ← 0 **to** n − 2 **do**

  for *j* ← 0 **to** *n* − 2 − *i* **do**

    if A[*j* + 1]<A[*j*] swap A[*j*] and A[*j* + 1]

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example.

89 ?↔ 45 68 90 29 34 17
45 ?↔ 89 ?↔ 68 90 29 34 17
45 68 ?↔ 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 ?↔ 90 ?↔ 34 17
45 68 89 29 34 ?↔ 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 ?↔ 29 89 ?↔ 34 17 | 90
45 68 29 34 ?↔ 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons.

$C_{worst}(n) \in \Theta(n^2)$

2.2 CLOSEST-PAIR AND CONVEX-HULL PROBLEMS

We consider a straight forward approach (Brute Force) to two well-known problems dealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

Closest-Pair Problem

The closest-pair problem finds the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

Consider the two-dimensional case of the closest-pair problem. The points are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance.

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The following algorithm computes the distance between each pair of distinct points and finds a pair with the smallest distance.

ALGORITHM *BruteForceClosestPair(P)*

//Finds distance between two closest points in the plane by brute force

//Input: A list *P* of *n* $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$

//Output: The distance between the closest pair of points

$d \leftarrow \infty$

for $i \leftarrow 1$ **to** $n - 1$ **do**

  for $j \leftarrow i + 1$ **to** n **do**

    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root

return *d*

The basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} . \sum_{j=(i+1)}^{n} 2$$

$$= 2 \sum_{i=1}^{n-1} (n - i)$$

$$= 2[(n - 1) + (n - 2) + \ldots + 1]$$

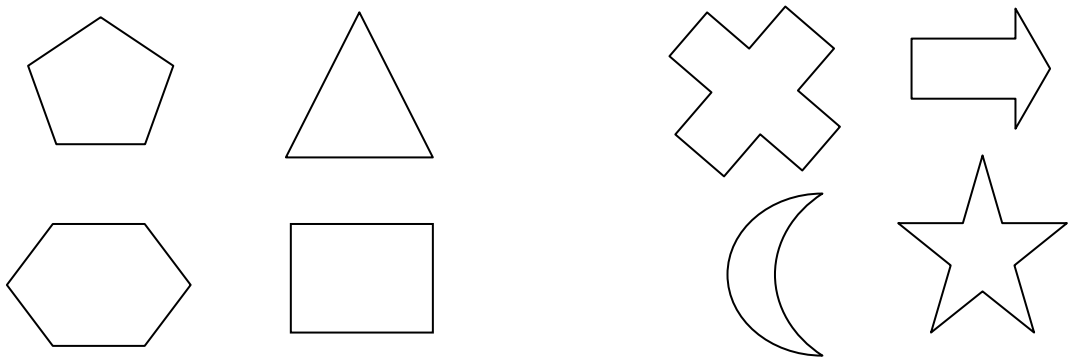
$$= (n - 1)n \in \Theta(n^2).$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor, but it cannot improve its asymptotic efficiency class.

Convex-Hull Problem

Convex Set

A set of points (finite or infinite) in the plane is called ***convex*** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.



(a) (b)

FIGURE 2.1 (a) Convex sets. (b) Sets that are not convex.

All the sets depicted in Figure 2.1 (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.

On the other hand, the sets depicted in Figure 2.1 (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in Figure 2.2

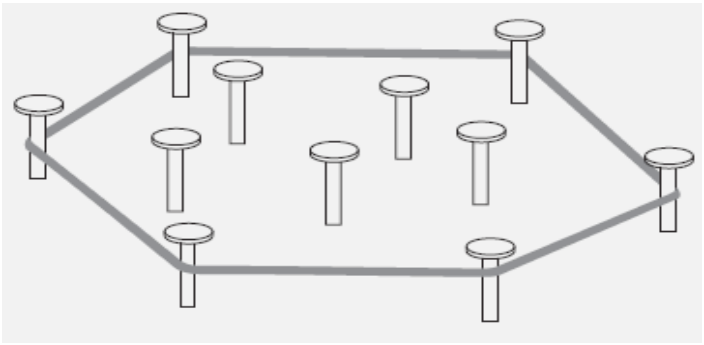


FIGURE 2.2 Rubber-band interpretation of the convex hull.

Convex hull

The ***convex hull*** of a set S of points is the smallest convex set containing S. (The smallest convex hull of S must be a subset of any convex set containing S.)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 2.3.

THEOREM

The convex hull of any set *S* of *n*>2 points not all on the same line is a convex polygon with the vertices at some of the points of *S*. (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of *S*.)

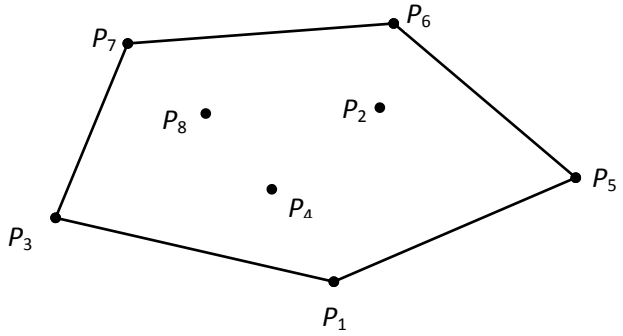


FIGURE 2.3 The convex hull for this set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, and p_3.

The ***convex-hull problem*** is the problem of constructing the convex hull for a given set *S* of *n* points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon "extreme points." By definition, an ***extreme point*** of a convex set is a point of this set that is *not a middle point of any line segment with endpoints in the set*. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 2.3 are p_1, p_5, p_6, p_7, and p_3.

Application

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the ***simplex method***, This algorithm solves ***linear programming*** **Problems.**

We are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of *n* points of a given set are extreme points of the set's convex hull. we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counterclockwise order.

We can solve the convex-hull problem by brute-force manner. The convex hull problem is one with no obvious algorithmic solution. there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points *pi* and *pj* of a set of *n* points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

Facts

A few elementary facts from analytical geometry are needed to implement the above algorithm.

- First, the straight line through two points $(x_1, y_1), (x_2, y_2)$ in the coordinate plane can be defined by the equation $ax + by = c$,where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.
- Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.

Time efficiency of this algorithm.

Time efficiency of this algorithm is in $O(n^3)$: for each of $n(n-1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.

2.3 EXHAUSTIVE SEARCH

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete search or*** **direct *search.***

Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only **general approach** for which it is more difficult to point out problems it *cannot* tackle.
2. Second, for some important problems, e.g., sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustive Search is applied to the important problems like

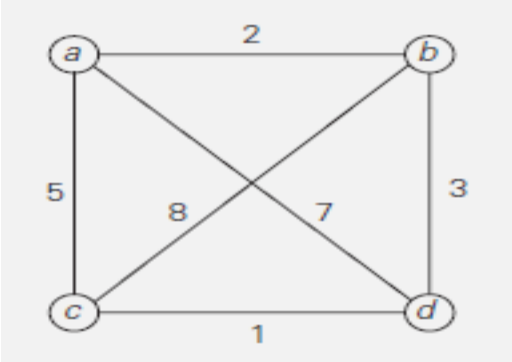
- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

2.4 TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $vi_0, vi_1, \ldots, vi_{n-1}, vi_0$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	Length
a ---> b ---> c ---> d ---> a	I = 2 + 8 + 1 + 7 = 18
a ---> b ---> d ---> c ---> a	**I = 2 + 3 + 1 + 5 = 11 optimal**
a ---> c ---> b ---> d ---> a	I = 5 + 8 + 3 + 7 = 23
a ---> c ---> d ---> b ---> a	**I = 5 + 1 + 3 + 2 = 11 optimal**
a ---> d ---> b ---> c ---> a	I = 7 + 3 + 8 + 5 = 23
a ---> d ---> c ---> b ---> a	I = 7 + 1 + 8 + 2 = 18

FIGURE 2.4 Solution to a small instance of the traveling salesman problem by exhaustive search.

Time efficiency

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. **(n - 1)!**
- Consider two intermediate vertices, say, *b* and *c*, and then only permutations in which *b* precedes *c*. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

2.5 KNAPSACK PROBLEM

Given *n* items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.

Real time examples:

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

FIGURE 2.5 Instance of the knapsack problem.

Subset	Total weight	Total value
Φ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65 (Maximum-Optimum)**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{ 2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

FIGURE 2.6 knapsack problem's solution by exhaustive search. The information about the optimal selection is in bold.

Time efficiency: As given in the example, the solution to the instance of Figure 2.5 is given in Figure 2.6. Since the *number of subsets of an n-element set is* 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Note: Exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***NP-hard problems***. **No polynomial-time** algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

2.6 ASSIGNMENT PROBLEM.

There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i,j]$ for each pair $i,j = 1,2,\ldots,n$. The problem is to find an assignment with the minimum total cost.

Assignment problem solved by exhaustive search is illustrated with an example as shown in figure 2.8. A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

FIGURE 2.7 Instance of an Assignment problem.

An instance of the assignment problem is completely specified by its cost matrix *C*.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

The problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

We can describe feasible solutions to the assignment problem as *n*-tuples $<j_1, \ldots, j_n>$ in which the *i*th component, $i = 1,\ldots,n$, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person). For example, for the cost matrix above, <2, 3, 4, 1> indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. Similarly we can have $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24, i.e., 24$ permutations.

The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers $1, 2, \ldots, n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are given below.

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = **18**
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

<2, 1, 3, 4>	**cost = 2 + 6 + 1 + 4 = 13 (Min)**
<2, 1, 4, 3>	cost = 2 + 6 + 8 + 9 = 25
<2, 3, 1, 4>	cost = 2 + 3 + 5 + 4 = 14
<2, 3, 4, 1>	cost = 2 + 3 + 8 + 7 = 20
<2, 4, 1, 3>	cost = 2 + 7 + 5 + 9 = 23
<2, 4, 3, 1>	cost = 2 + 7 + 1 + 7 = 17, etc

FIGURE 2.8 First few iterations of solving a small instance of the assignment problem by exhaustive search.

Since the number of permutations to be considered for the general case of the assignment problem is ***n*!,** exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the ***Hungarian method.***

2.7 DIVIDE AND CONQUER METHODOLOGY

A **divide and conquer algorithm** works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**).

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.

The divide-and-conquer technique as shown in Figure 2.9, which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved separately. Finally solution to the original problem is done by combining the solutions of subproblems.

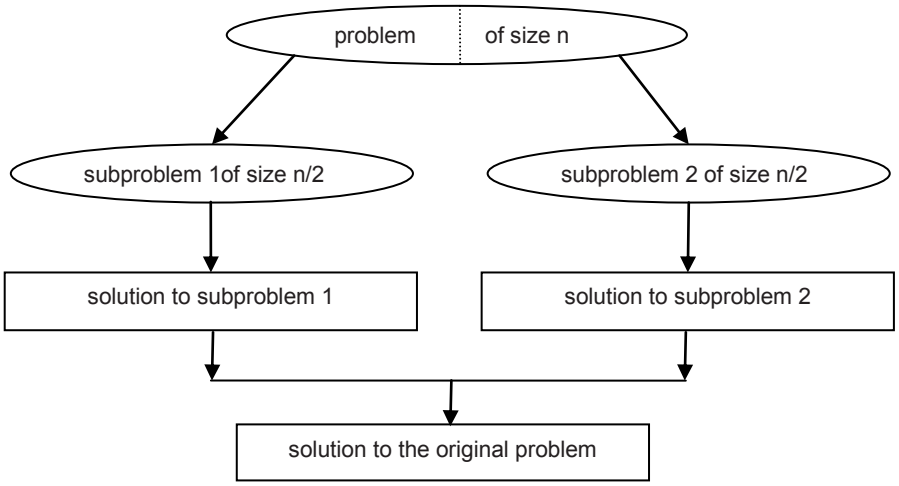


FIGURE 2.9 Divide-and-conquer technique.

Divide and conquer methodology can be easily applied on the following problem.

1. Merge sort
2. Quick sort
3. Binary search

2.8 MERGE SORT

Mergesort is based on 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.

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

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.

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*[$k..p+q-1$]

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 2.10.

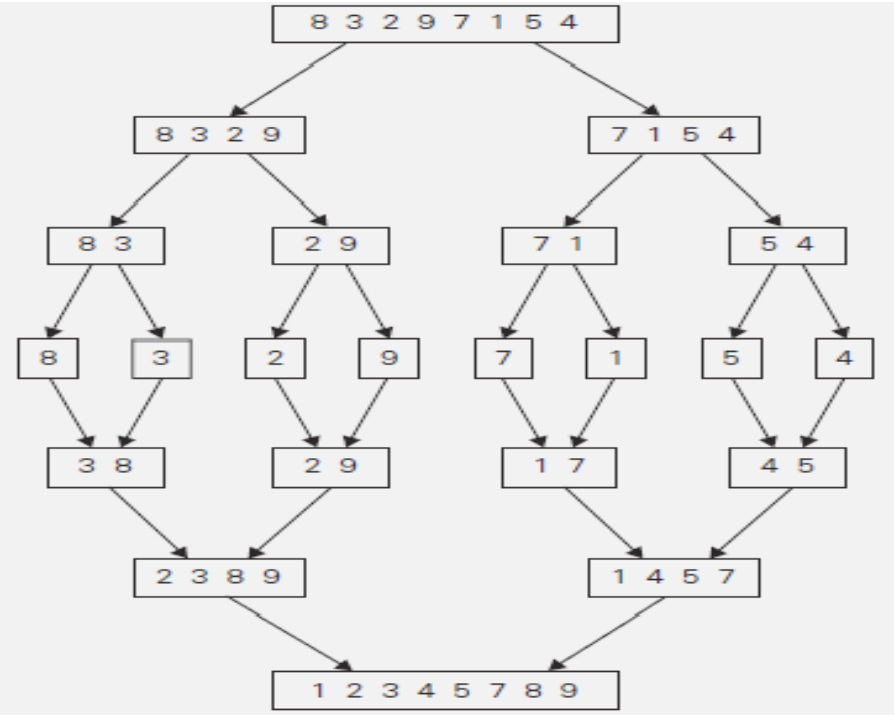


FIGURE 2.10 Example of mergesort operation.

The recurrence relation for the number of key comparisons $C(n)$ is

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

In the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

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

By Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$
the exact solution to the worst-case recurrence for $n = 2^k$
$C_{worst}(n) = n \log_2 n - n + 1$.

For large n, the number of comparisons made by this algorithm in the average case turns out to be about $0.25n$ less and hence is also in **$\Theta(n \log n)$.**

First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.

2.9 QUICK SORT

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. quicksort divides input elements 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:

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

Sort the two subarrays to the left and to the right of $A[s]$ independently. No work required to combine the solutions to the subproblems.

Here is pseudocode of quicksort: call *Quicksort*($A[0..n-1]$) where As a partition algorithm use the *HoarePartition*

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

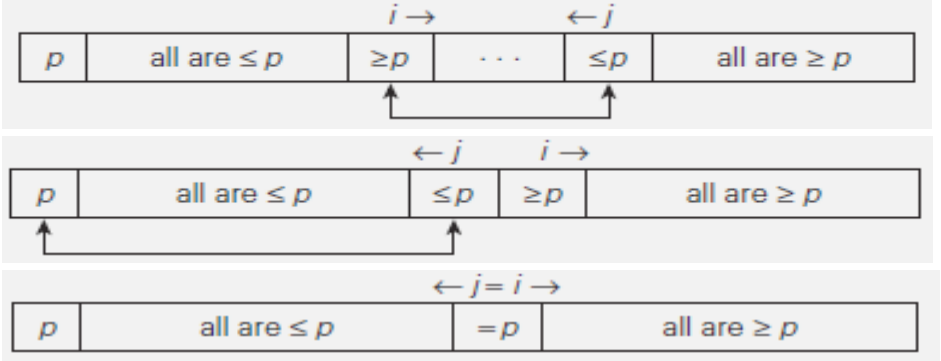
//Output: Subarray $A[l..r]$ sorted in nondecreasing order

if $l < r$

$s \leftarrow$*HoarePartition*($A[l..r]$) //s is a split position

Quicksort($A[l..s-1]$)

Quicksort($A[s+1..r]$)



ALGORITHM *HoarePartition(A[l..r])*

//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r $(l<r)$
//Output: Partition of $A[l..r]$, with the split position returned as this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
    repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9

FIGURE 2.11 Example of quicksort operation of Array with pivots shown in bold.

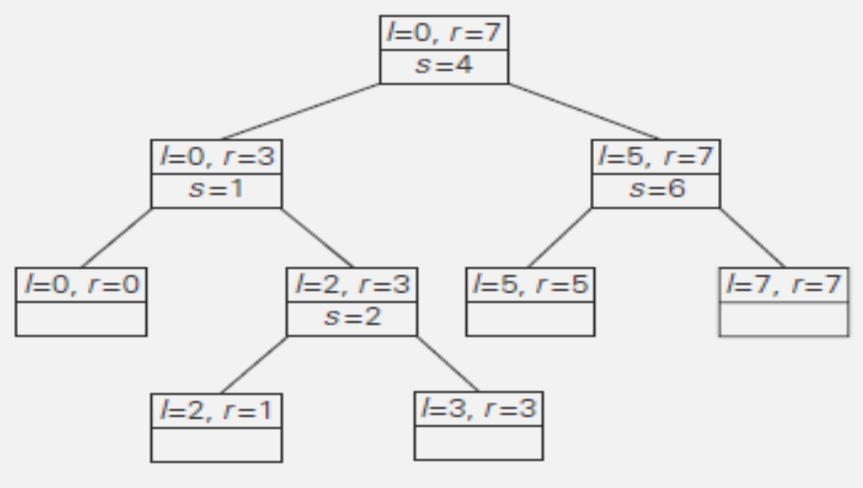


FIGURE 2.12 Tree of recursive calls to *Quicksort* with input values *l* and *r* of subarray bounds and split position *s* of a partition obtained.

The number of key comparisons in the best case satisfies the recurrence

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

By Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

The total number of key comparisons made will be equal to

$Cworst(n) = (n + 1) + n + \ldots + 3 = ((n + 1)(n + 2))/2 - 3 \in \Theta(n^2)$.

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

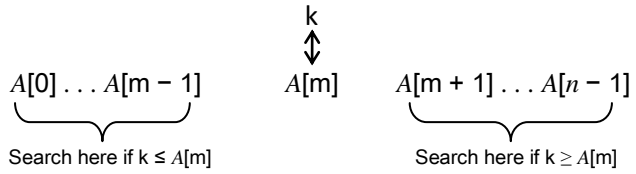
$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

2.10 BINARY SEARCH

A binary search is efficient algorithm to find the position of a target (key) value within a sorted array.

- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished.
- If the target value is less than the middle element's value, then the search continues on the lower half of the array.
- if the target value is greater than the middle element's value, then the search continues on the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (position is returned).

Binary search is a remarkably efficient algorithm for searching in a sorted array (Say A). It works by comparing a search key K with the array's middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K <A[m], and for the second half if K >A[m]:



Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

ALGORITHM *BinarySearch(A*[0..*n* − 1]*, K)*

//Implements nonrecursive binary search

//Input: An array *A*[0..*n* − 1] sorted in ascending order and a search key *K*

//Output: An index of the array's element that is equal to *K*/ or −1 if there is no such element

$l \leftarrow 0$; $r \leftarrow n - 1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$

$r \leftarrow m - 1$

else $l \leftarrow m + 1$

return −1

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array (three-way comparisons). One comparison of K with A[m], the algorithm can determine whether K is smaller, equal to, or larger than A[m].

As an example, let us apply binary search to searching for $K = 70$ in the array. The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size,

The number of key comparisons in the worst case $C_{worst}(n)$ by recurrence relation.

$C_{worst}(n) = C_{worst}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \; for \; n > 1, C_{worst}(1) = 1.$

$\therefore C_{worst}(n) = \lfloor log_2 n \rfloor + 1 = \lceil log_2(n + 1) \rceil \quad \because C_{worst}(2^k) = (k + 1) = log_2 k + 1$ for $n = 2^k$

- First, The worst-case time efficiency of binary search is in Θ(log n).
- Second, the algorithm simply reduces the size of the remaining array by half on each iteration, the number of such iterations needed to reduce the initial size *n* to the final size 1 has to be about $\log_2 n$.

- Third, the logarithmic function grows so slowly that its values remain small even for very large values of *n*.

The average case slightly smaller than that in the worst case

$C_{avg}(n) \approx \log_2 n$

The average number of comparisons in a successful is

$\boldsymbol{C_{avg}(n) \approx \log_2 n - 1}$

The average number of comparisons in an unsuccessful is

$\boldsymbol{C_{avg}(n) \approx \log_2(n + 1).}$

2.11 MULTIPLICATION OF LARGE INTEGERS

Some applications like modern cryptography require manipulation of integers 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.

In 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 n^2 digit multiplications.

The divide-and-conquer method does the above multiplication in less than n^2 digit multiplications.

Example: $23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0$

$= 2 \cdot 10^2 + 11 \cdot 10^1 + 12 \cdot 10^0$

$= 3 \cdot 10^2 + 2 \cdot 10^1 + 2 \cdot 10^0$

$= 322$

The term $(2 * 1 + 3 * 4)$ computed as $2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - (2 * 1) - (3 * 4)$. Here $(2 * 1)$ and $(3 * 4)$ are already computed used. So only one multiplication only we have to do.

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.

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

$C = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0,$

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,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

The multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.
Solving it by backward substitutions for $n = 2^k$ yields

$$
\begin{aligned}
M(2^k) &= 3M(2^{k-1}) \\
&= 3[3M(2^{k-2})] \\
&= 3^2M(2^{k-2}) \\
&= \ldots \\
&= 3^iM(2^{k-i}) \\
&= \ldots \\
&= 3^kM(2^{k-k}) \\
&= 3^k.
\end{aligned}
$$

(Since $k = \log_2 n$)
$\boldsymbol{M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}}$.

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Let $A(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides $3A(n/2)$ of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$A(n) = 3 \cdot A(n/2) + cn$ for $n > 1$, $A(1) = 1$.

By using Master Theorem, we obtain $\boldsymbol{A(n) \in \Theta(n^{\log_2 3})}$,

which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Example: For instance: a = 2345, b = 6137, i.e., n=4.

Then C = a * b = $(23*10^2+45)*(61*10^2+37)$

$$
\begin{aligned}
C &= a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\
&= (23 * 61)10^4 + (23 * 37 + 45 * 61)10^2 + (45 * 37) \\
&= 1403 \cdot 10^4 + 3596 \cdot 10^2 + 1665 \\
&= 14391265
\end{aligned}
$$

2.12 STRASSEN'S MATRIX MULTIPLICATION

The Strassen's Matrix Multiplication find the product C of two 2×2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes 7 multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires 8 multiplications and 4 additions. These numbers should not lead us to multiplying 2×2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order n goes to infinity.

Let A and B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four $n/2 \times n/2$ submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

The value C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. The seven products of $n/2 \times n/2$ matrices are computed recursively by Strassen's matrix multiplication algorithm.

The asymptotic efficiency of Strassen's matrix multiplication algorithm

If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n×n matrices, where n is a power of 2, The recurrence relation is M(n) = 7M(n/2) for n > 1, M(1)=1.

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1})$$
$$= 7[7M(2^{k-2})]$$
$$= 7^2M(2^{k-2})$$
$$= \ldots$$

$= 7^i M(2^{k-i})$

$= \ldots$

$= 7^k M(2^{k-k}) = 7^k M(2^0) = 7^k M(1) = 7^k(1)$ (Since M(1)=1)

$M(2^k) = 7^k$.

Since $k = \log_2 n$,

$M(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx \boldsymbol{n^{2.807}}$

which is smaller than n^3 required by the brute-force algorithm.

Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by Strassen's algorithm. To multiply two matrices of order $n>1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions/subtractions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$A(n) = 7A(n/2) + 18(n/2)^2$ for $n > 1$, $A(1) = 0$.

By closed-form solution to this recurrence and the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. which is a better efficiency class than $\Theta(n^3)$of the brute-force method.

Example: Multiply the following two matrices by Strassen's matrix multiplication algorithm.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Answer:

$$\mathbf{C} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \mathbf{x} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Where $\mathbf{A_{00}} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}$ $\mathbf{A_{01}} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$ $\mathbf{A_{10}} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}$ $\mathbf{A_{11}} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$

$\mathbf{B_{00}} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$ $\mathbf{B_{01}} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}$ $\mathbf{B_{10}} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$ $\mathbf{B_{11}} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$

$M_1 = (A_{00}+A_{11})*(B_{00}+B_{11}) = \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$

Similarly apply Strassen's matrix multiplication algorithm to find the following.

$M_2 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$, $M_3 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$, $M_4 = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$, $M_5 = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$, $M_6 = \begin{bmatrix} 2 & -3 \\ -2 & -3 \end{bmatrix}$, $M_7 = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$

$C_{00} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$, $C_{01} = \begin{bmatrix} -7 & 3 \\ 1 & 9 \end{bmatrix}$, $C_{10} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$, $C_{11} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$

$$\mathbf{C} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

2.13 Closest-Pair and Convex-Hull Problems.

The two-dimensional versions of the closest-pair problem and the convex-hull problem problems can be solved by brute-force algorithms in $\theta(n^2)$ and $O(n^3)$ time, respectively. The divide-and-conquer technique provides sophisticated and asymptotically more efficient algorithms to solve these problems.

The Closest-Pair Problem

Let P be a set of $n > 1$ points in the Cartesian plane. The points are ordered in nondecreasing order of their x coordinate. It will also be convenient to have the points sorted (by merge sort) in a separate list in nondecreasing order of the y coordinate and denote such a list by Q.

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we can divide the points into two subsets Pl and Pr of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie to the left of or on the line itself, and $\lceil n/2 \rceil$ points lie to the right of or on the line. Then we can solve the closest-pair problem recursively for subsets P_l and P_r. Let d_l and d_r be the smallest distances between pairs of points in Pl and Pr, respectively, and let $d = \min\{d_l, d_r\}$.

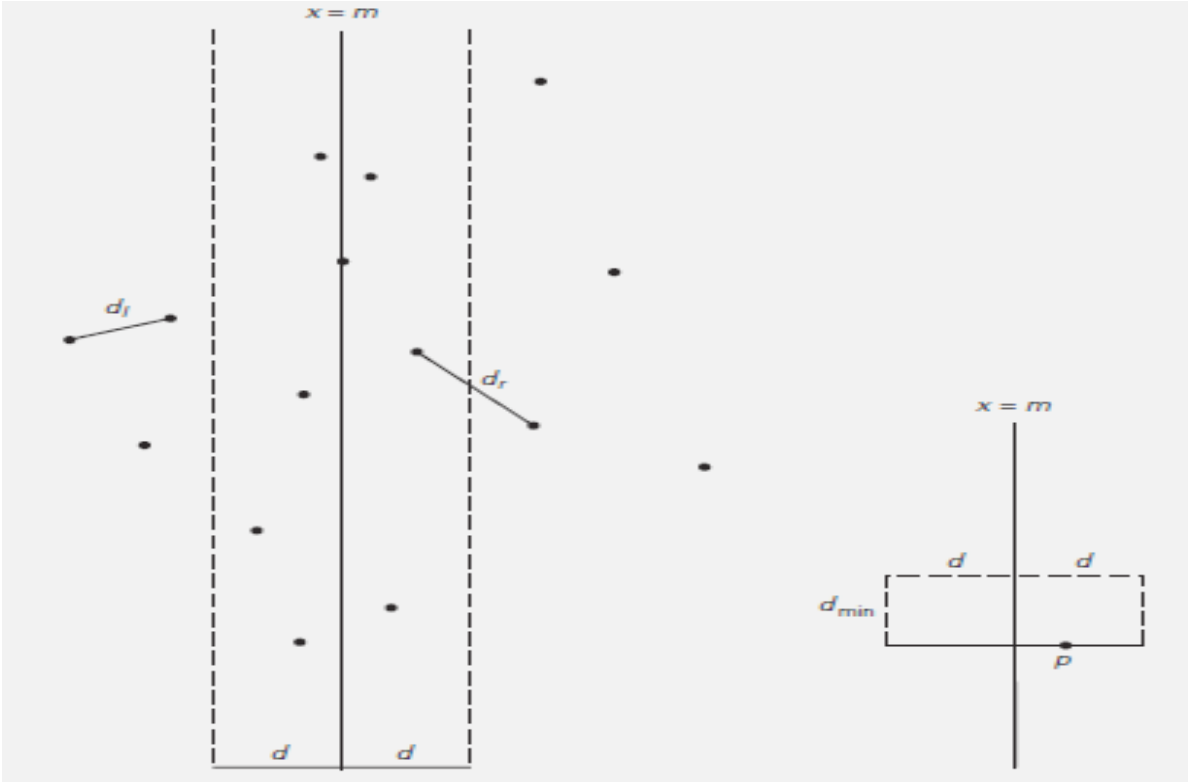


FIGURE 2.13 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) Rectangle that may contain points closer than d_{min} to point p.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width $2d$ around the separating line, since the distance between any other pair of points is at least d (Figure 2.13a).

Let S be the list of points inside the strip of width 2d around the separating line, obtained from Q and hence ordered in nondecreasing order of their y coordinate. We will scan this list, updating the information about dmin, the minimum distance seen so far, if we encounter a closer

pair of points. Initially, $d_{min} = d$, and subsequently $d_{min} \leq d$. Let p(x, y) be a point on this list. For a point p (x, y) to have a chance to be closer to p than d_{min}, the point must follow p on list S and the difference between their y coordinates must be less than d_{min}.

Geometrically, this means that p must belong to the rectangle shown in Figure 2.13b. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart. It is easy to prove that the total number of such points in the rectangle, including p, does not exceed 8. A more careful analysis reduces this number to 6. Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.

Here is pseudocode of the algorithm. We follow the advice given in to avoid computing square roots inside the innermost loop of the algorithm.

ALGORITHM *EfficientClosestPair(P, Q)*

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in nondecreasing
//       order of their x coordinates and an array Q of the same points sorted in
//       nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

$$T(n) = 2T(n/2) + f(n),$$

where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in \Theta(n \log n)$. The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as mergesort. In fact, this is the best efficiency