UNIT 1 DIVIDE-AND-CONQUER

Structure   **Page Nos.**

1.0 INTRODUCTION

We have already mentioned that solving (a general) problem, with or without computers, is quite a complex and difficult task. We also mentioned a large number of problems, which we may encounter, even in a formal discipline like Mathematics, may not have any algorithmic/computer solutions. Out of the problems, which theoretically can be solved algorithmically, designing a solution for such a problem is, in general, quite difficult. In view of this difficulty, a number of standard techniques, which are found to be helpful in solving problems, have become popular in computer science. Out of these techniques Divide-and-Conquer is probably the most well-known one.

The general plan for Divide-and-Conquer technique has the following three major steps:

Step 1: An instance of the problem to be solved, is divided into a number of smaller instances of the (same) problem, generally of equal sizes. Any sub-instance may be further divided into its sub-instances. A stage reaches when either a direct solution of a sub-instance at some stage is available or it is not further sub-divisible. In the latter case, when no further sub-division is possible, we attempt a direct solution for the sub-instance.

Step 2: Such smaller instances are solved.

Step 3: Combine the solutions so obtained of the smaller instances to get the solution of the original instance of the problem.

In this unit, we will study the technique, by applying it in solving a number of problems.

1.1 OBJECTIVES

After going through this Unit, you should be able to:

- explain the essential idea behind the Divide-and-Conquer strategy for solving problems with the help of a computer, and
- use Divide-and-Conquer strategy for solving problems.

1.2 GENERAL ISSUES IN DIVIDE-AND-CONQUER

Recalling from the introduction, Divide-and-Conquer is a technique of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved, split this into more than one sub-instances (*of the given problem*). If possible, divide each of the sub-instances into smaller instances, till a sub-instnace has a direct solution available or no further subdivision is possible. Then independently solve each of the sub-instances and then combine solutions of the sub-instances so as to yield a solution for the original instance.

The methods by which sub-instances are to be independently solved play an important role in the overall efficiency of the algorithm.

Example 1.2.1:

We have an algorithm, *alpha* say, which is known to solve all instances of size *n*, of a given problem, in at most $c \, n^2$ steps (where c is some constant). We then discover an algorithm, *beta* say, which solves the same problem by:

- Dividing an instance into 3 sub-instances of size n/2.
- Solve these 3 sub-instances.
- Combines the three sub-solutions taking *d n* steps in combining.

Suppose our original algorithm *alpha* is used to carry out the Step 2, viz., 'solve these sub-instances'. Let

T (alpha) (n)	=	Running time of alpha on an instance of size n.
T (beta) (n)	=	Running time of beta on an instance of size n.

Then,

T (alpha) (n) = $c \, n^2$ (by definition of alpha)

But

T (beta) (n)	=	3 T (alpha) (n/2) + d n
	=	$(3/4) \, (cn^2) + dn$

So if $dn < (cn^2)/4$ (i.e., $d/4c < n$) then beta is faster than alpha.

In particular, for all large enough n's, (viz., for n > 4d/c = Constant), beta is faster than alpha.

The algorithm beta improves upon the algorithm *alpha* by just a constant factor. But if the problem size *n* is large enough such that for some i > 1, we have

n	>	4d/c	and also
n/2	>	4d/c	and even
$n/2^i$	>	4d/c	

which suggests that using *beta* instead of *alpha* for the Step 2 repeatedly until the sub-sub-sub…sub-instances are of size $n_0 < = (4d/c)$, will yield a still faster algorithm.

So consider the following new algorithm for instances of size n

Procedure gamma (n : problem size),

If $n < = n_0$ then

Solve problem using Algorithm alpha;
else

Split the problem instance into 3 sub-instances of size n/2;
Use gamma to solve each sub-instance;
Combine the 3 sub-solutions;
end if ;
end gamma;

Let T (gamma) (n) denote the running time of this algorithm. Then

$$
T \text{ (gamma) } (n) = \begin{cases} cn^2 & \text{if } n <= n_0 \\[2em] 3T \text{ (gamma) } (n/2) + dn, & \text{otherwise} \end{cases}
$$

we shall show how relations of this form can be estimated. Later in the course, with these methods it can be shown that

$T(gamma) (n) = O (n^{(log3)}) = O(n^{1.59})$

This is a significant *improvement* upon algorithms *alpha* and *beta*, in view of the fact that as n becomes larger the differences in the values of $n^{1.59}$ and n^2 becomes larger and larger.

The improvement that results from applying algorithm *gamma* is due to the fact that it maximizes the savings achieved through *beta*. The (relatively) inefficient method *alpha* is applied only to "*small*" problem sizes.

The precise form of a divide-and-conquer algorithm is characterised by:

(i)   The *threshold* input size, n_0, below which the problem size is not further sub-divided.

(ii)   The *size* of sub-instances into which an instance is split.

(iii)   The *number* of such sub-instances.

(iv)   The method for solving instances of size $n <= n_0$.

(iv)   The algorithm used to combine sub-solutions.

In (ii), it is more usual to consider the *ratio* of initial problem size to sub-instance size. In our example, the ration was 2. The *threshold* in (i) is sometimes called the *(recursive) base value*. In summary, the generic form of a divide-and-conquer algorithm is:

Procedure D-and-C   (n : input size);
  begin
    read (n_0)   ;   read the threshold value.
    if $n <= n_0$ **then**
     solve problem without further sub-division;
   else
     Split into sub-instances each of size n/k;
     for each of the r sub-instances do
        D-and-C (n/k);
     Combine the resulting sub-solutions to produce the solution to the original
     problem;
end if;
end D-and-C;

Such algorithms are naturally and easily realised as recursive procedures in (suitable) high-level programming languages.

1.3 INTEGER MULTIPLICATION

The following problem is a classic example of the application of Divide-and-Conquer technique in the field of Computer Science.

Input: Two n-digit decimal numbers $\underline{x}$ and $\underline{y}$ represented as

$$\underline{x} = x_{n-1}\, x_{n-2}\, ...x_1\, x_0 \qquad \text{and}$$

$$\underline{y} = y_{n-1}\, y_{n-2}\, ...y_1\, y_0$$

where x_i, y_i are decimal digits.

Output: The (*2n*)-digit decimal representation of the product x * y.

$$\underline{z} = z_{2n-1}\ \ z_{2n-2}\ \ z_{2n-3}\, ...z_1\, z_0$$

Note: *The algorithm given below works for any number base, e.g., binary, decimal, hexadecimal, etc. We use decimal simply for convenience.*

The classical algorithm for multiplication requires $O(n^2)$ steps to multiply two n-digit numbers.

A step is regarded as a single operation involving two single digit numbers, e.g., 5+6, 3* 4, etc.

In 1962, A. A. Karatsuba discovered an *asymptotically faster* algorithm for multiplying two numbers by using a divide-and-conquer approach.

The *values* x and y of the numbers with representations

$$\underline{x} = x_{n-1}\, x_{n-2}\, ...x_1\, x_0 \quad \text{and}$$

$$\underline{y} = y_{n-1}\, y_{n-2}\, ...y_1\, y_0$$

are clearly given by,

$$x = \sum_{i=0}^{n-1} (x_i) * 10^i \; ; \text{ and}$$

$$y = \sum_{i=0}^{n-1} (y_i) \ * \ 10^i.$$

Then, the resultant value z = x * y

with representation

$$\underline{z} = z_{2n-1}\ \ z_{2n-2}\ \ z_{2n-3}\, ...z_1\, z_0$$

is given by

$$z = \sum_{i=0}^{2n-1} (z_i) * 10^i = \left(\sum_{i=0}^{n-1} x_i * 10^i \right) * \left(\sum_{i=0}^{n-1} y_i * 10^i \right).$$

For example:

581	=	$5 * 10^2 + 8 * 10^1 + 1 * 10^0$
602	=	$6 * 10^2 + 0 * 10^1 + 2 * 10^0$
581*602	=	$349762 = 3 * 10^5 + 4 * 10^4 + 9 \times 10^3 + 7 \times 10^2 + 6 \times 10^1$
		$+ 2 \times 10^0$

Let us denote

$\underline{a}$	=	$x_{n-1} x_{n-2} \dots x_{[n/2]+1} x_{[n/2]}$
$\underline{b}$	=	$x_{[n/2]-1} x_{[n/2]-2} \dots x_1 . x_0$
$\underline{c}$	=	$y_{n-1} y_{n-2} \dots y_{[n/2]+1} y_{[n/2]}$
$\underline{d}$	=	$y_{[n/2]-1} y_{[n/2]-2} \dots y_1 y_0$

Where [n/2] = largest integer less than or equal to n/2.

Then, if a, b, c, and d are the numbers whose decimal representations are $\underline{a}$, $\underline{b}$, $\underline{c}$ and $\underline{d}$ then

$$x = a * 10^{[n/2]} + b \quad ; \quad y = c * 10^{[n/2]} + d$$

For example, if n = 4, x = 1026 and y = 7329 then a = 10, b = 26, c = 73 and d = 29, and,

x	=	1026	=	$10 * 10^2 + 26$	=	$a * 10^2 + b$
y	=	7329	=	$73 * 10^2 + 29$	=	$c * 10^2 + d$

From this we also know that the result of multiplying x and y (i.e., z) is

z	= x * y	=	$(a * 10^{[n/2]} + b) * (c * 10^{[n/2]} + d)$
		=	$(a * c) * 10^{2[n/2]} + (a * d + b * c) * 10^{[n/2]} + (b * d)$

where

$$2[n/2] = \begin{cases} n, & \text{if n is even} \\ n+1 & \text{if n is odd} \end{cases}$$

e.g., 1026 * 7329 is

$$= (10 * 73) * 10^4 + (10 * 29 + 26 * 73) * 10^2 + (26 * 29)$$

$$= 730 * 10^4 + 2188 * 10^2 + 754 = 7,519,554$$

Each of the terms $(a * c)$, $(a * d)$, $(b * c)$ and $(b * d)$ is a product of **two [n/2]-digit numbers.**

Thus the expression for the multiplication of x and y in terms of the numbers a, b, c and d tells us that:

1. Two single digit numbers can be multiplied immediately.
    (**Recursive base: step 1**)

2. If n > *1* then the product of two n-digit numbers can be expressed in terms of
    4 products of two numbers[*] (**Divide-and-conquer stage**)

---

[*] For a given n-digit number, whenever we divides the sequence of digits into two subsequences, one of which has [n/] digits, the other subsequence has $n - [n/2]$ digits, which is

(n/2) digits if n is even and $\left(\dfrac{n+1}{2} \right)$ if n is odd. However, because of the convenience we

may call both as $(n/2) -$ digit sequences/numbers.

3. Given the four returned products, the calculation of the result of multiplying x and y involves only *additions* (can be done in O(n) steps) and multiplications by a *power of 10* (also can be done in O(n) steps, since it only requires placing the appropriate number of 0s at the end of the number). (**Combine stage**).

Steps 1−3, therefore, describe a Divide-and-Conquer algorithm for multiplying two n-digit numbers represented in decimal. However, Karatsuba discovered how the product of two n-digit numbers could be expressed in terms of **three** products each of two (n/2)-digit numbers, instead of the **four** products that a conventional implementation of the Divide-and-Conquer schema, as above, uses.

This saving is accomplished at the expense of a slightly more number of steps taken in the 'combine stage' (Step 3) (although, this will still uses O(n) operations).

We continue with the earlier notations in which z is the product of two numbers x and y having respectively the decimal representations

$$\underline{x} = x_{n-1} x_{n-2} ... x_1 x_0$$

$$\underline{y} = y_{n-1} y_{n-2} ... y_1 y_0$$

Further a,b,c,d are the numbers whose decimal representations are given by

$$\underline{a} = x_{n-1} x_{n-2} .... x_{[n/2]+1} x_{[n/2]}$$

$$\underline{b} = x_{[n/2]-1} x_{[n/2]-2} ... x_1 . x_0$$

$$\underline{c} = y_{n-1} y_{n-2} ... y_{[n/2]+1} y_{[n/2]}$$

$$\underline{d} = y_{[n/2]-1} y_{[n/2]-2} ... y_1 y_0$$

Let us compute the following 3 products (of two (n/2)-digit numbers):

$$U = a * c$$
$$V = b * d$$
$$W = (a + b) * (c + d)$$

Then

$$W = a * c + a * d + b * c + b * d$$
$$= U + a * d + b * c + V$$

Therefore,

$$a * d + b * c = W - U - V.$$

Therefore,

$$z = x * y$$
$$= (a * 10^{[n/2]} + b) * (c * 10^{[n/2]} + d)$$
$$= (a * c) * 10^{2[n/2]} + (a * d + b * c) * 10^{[n/2]} + b * d$$
$$= U * 10^{2[n/2]} + (W - U - V) * 10^{[n/2]} + V.$$

This algorithm is formalised through the following function.

function Karatsuba   (xunder, yunder : **n-digit integer;** n : integer)
a, b, c, d: (n/2)−digit integer
U, V, W: n−digit integer;

begin
   if n = 1 then
      return $x_0 * y_0$;

else

$a := x_{(n-1)}\ldots x_{[n/2]}$;
$b := x_{[n/2]-1} \ldots x_0$;
$c := y_{(n-1)} \ldots y_{[n/2]}$;
$d := y_{[n/2]-1} \ldots y_0$;

U   :=   Karatsuba   (a, c, [n/2]);
V   :=   Karatsuba   (b, d, [n/2]);
W   :=   Karatsuba   (a+b, c+d, [n/2]);

Return $U*10^{2[n/2]}$ + (W—U—V) * $10^{[n/2]}$ + V;
; where 10^m stands for 10 raised to power m.
end if;
end Karatsuba;

Performance Analysis

One of the reasons why we study analysis of algorithms, is that **if** there are more than one algorithms that solve a given problem **then,** through analysis, we can find the running times of various available algorithms. And then we may choose the one be which takes least/lesser running time.

This is useful in allowing comparisons between the performances of two algorithms to be made.

For Divide-and-Conquer algorithms the running time is mainly affected by 3 criteria:

- The **number of sub-instances** (*let us call the number as* $\propto$) into which a problem is split.

- The **ratio of initial problem size to sub-problem size.** (*let us call the ration as β*)

- The **number of steps** required to **divide** the initial instance into substances and to **combine** sub-solutions, expressed as a function of the input size, n.

Suppose, *P*, is a divide-and-conquer algorithm that instantiates *alpha* sub-instances, each of size n/beta.

Let TP (n) denote the number of steps taken by P on instances of size n. Then

$T (P (n_0))$ =   Constant   (*Recursive-base*)
$T (P (n))$ = $\propto T (P (n \mid \beta)$ + gamma (n)

In the case when $\propto$ and β are both constant (as mentioned earlier, in all the examples we have given, there is a general method that can be used to solve such *recurrence relations* in order to obtain an asymptotic bound for the running time *Tp* (n). These methods were discussed in Block 1.

In general:

$T (n) = \propto T (n/\beta) + O (n ^ { gamma})$

$$T(n) = \begin{cases} O(n^{gamma}) & if \quad alpha \quad < \quad beta \,^{gamma} \\ O (n^{gamma} \log n) & if \quad alpha \quad = \quad beta \,^{gamma} \\ O (n^{\log^{-beta} (alpha)}) if & alpha \quad > \quad beta \,^{gamma} \end{cases}$$

In general:

$T (n) = \propto T (n/\beta) + O (n ^ { gamma})$

(where gamma is constant) has the solution

$$O(n \text{ ^ gamma}) \qquad \text{if} \quad \text{alpha} \quad < \quad \text{beta}$$

^gamma

$$T(n) \quad = \quad O(n \text{ ^ gamma log n}) \qquad \text{if} \quad \text{alpha} \quad = \quad \text{beta}$$

^gamma

$$O(n \text{ ^ log^-beta (alpha))if} \qquad \text{if} \quad \text{alpha} \quad > \quad \text{beta}$$

^gamma

Ex. 1) Using Karatsuba's Method, find the value of the product 1026732×732912

1.4   BINARY SEARCH

Binary Search algorithm searches a given value or element in an *already sorted* array by repeatedly dividing the search interval in half. It first compares the value to be searched with the item in the middle of the array. If there is a match, then search is successful and we can return the required result immediately. But if the value does not match with the item in middle of the array, then it finds whether the given value is less than or greater than the value in the middle of array. If the given value is less than the middle value, then the value of the item sought must lie in the lower half of the array. However, if the given value is greater than the item sought, must lie in the upper half of the array. So we repeat the procedure on the lower or upper half of the array according as the given value is respectively less than or greater than the middle value. The following C++ function gives the Binary Search Algorithm.

int Binary Seach (int * A, int low, int high, int value)

```
{      int  mid:
     while (low < high)
     {        mid = (low + high) / 2;
              if (value = = A [mid])
                     return mid;
              else if (value < A [mid])
                     high = mid − 1;
              else low = mid + 1;
     }
     return − 1;

}
```

Explanation of the Binary Search Algorithm

It takes as parameter the array A, in which the value is to be searched. It also takes the lower and upper bounds of the array as parameters viz., *low* and *high* respectively. At each step of the interation of the while loop, the algorithm reduces the number of elements of the array to be searched by half. If the value is found then its index is returned. However, if the value is not found by the algorithm, then the loop terminates if the value of the low exceeds the value of high, there will be no more items to be searched and hence the function returns a negative value to indicate that item is not found.

Analysis

As mentioned earlier, each step of the algorithm divides the block of items being searched in half. The presence or absence of an item in an array of n elements, can be established in at most lg n steps.

Thus the running time of a binary search is proportional to **lg *n*** and we say this is a **O(lg n)** algorithm.

Ex. 2) Explain how Binary Search method finds or fails to find in the given sorted array:

8	12	75	26	35	48	57	78	86
93	97	108	135	168	201			

the following values
(i)   15
(ii)   93
(iii)   43

1.5   SORTING

We have already discussed the two sorting algorithms viz., Merge Sort and Quick Sort. The purpose of repeating the algorithm is mainly to discuss, not the design but, the analysis part.
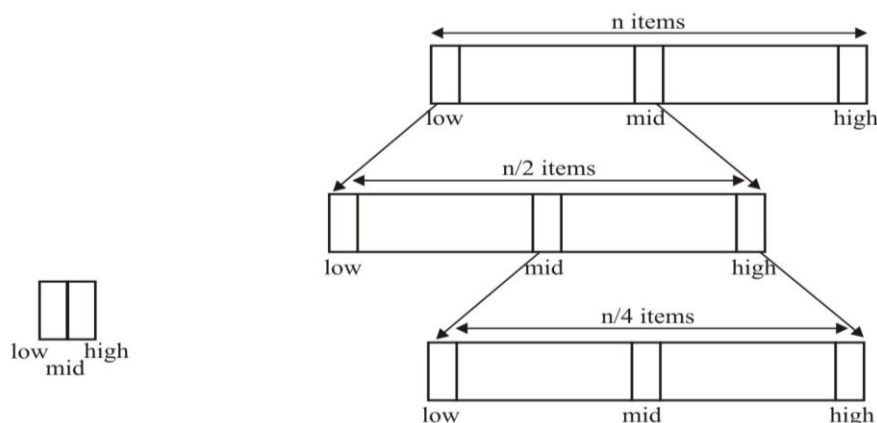
1.5.1 Merge Sort

As discussed in Block 1, Merge Sort is a sorting algorithm which is based on the divide-and-conquer technique or paradigm. The Merge-Sort can be described in general terms as consisting of 3 major steps namely, a divide step, recursive step and merge step. These steps are described below in more detail.

Divide Step: If given array A has zero or 1 element then return the array A, as it is trivially sorted. Otherwise, chop the given array A in almost the middle to give two subarrays A_1 and A_2, each of which containing about half of the elements in A.

Recursive Step: Recursively sort arry A1 and A2.

Merge Step: Though recursive application of we reach a stage when subarrays of sizes 2 and then of sizes 1 are obtained. At this stage two sublists of sizes 1 each are combined by placing the elements of the lists in sorted order. The process of this type of combinations of sublists is repeated on lists of larger and large sizes. To accomplish this step we will define a C++ function void merge (int A []. int p, int r).

The recursion stops when the subarray has just only 1 element, so that it is trivially sorted. Below is the Merge Sort function in C++.

```
void merge_sort (int A[], int p, int r)
    {

        if (p < r)          //Check for base case
        {
                int q = (p + r)/2;          //Divide step
                merge_sort (A, p,q);                //Conquer step
                merge_sort (A, q + 1, r);//Conquer step
                merge (A, p, q, r);       }          //Combine step
```

Next, we define merge function which is called by the program merge-sort At this stage, we have an Array *A* and indices *p,q,r* such that $p < q < r$. Subarray $A[p .. q]$ is sorted and subarray $A[q + 1 .. r]$ is sorted and by the restrictions on *p, q, r,* neither subarray is empty. We want that the two subarrays are merged into a single sorted subarray in $A[p .. r]$. We will implement it so that it takes O(n) time, where $n = r - p + 1$ = the number of elements being merged.

Let us consider two piles of cards. Each pile is sorted and placed face-up on a table with the smallest card on top of each pile. We will merge these into a single sorted pile, face-down on the table. A basic step will be to choose the smaller of the two top cards, remove it from its pile, thereby exposing a new top card and then placing the chosen card face-down onto the output pile. We will repeatedly perform these basic steps until one input becomes empty. Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile. Each basic step should take constant time, since we check just the two top cards and there are n basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles. Therefore, this procedure should take O(n) time. We don't actually need to check whether a pile is empty before each basic step. Instead we will put on the bottom of each input pile a special ***sentinel*** card. It contains a special value that we use to simplify the code. We know in advance that there are exactly $r - p + 1$ non sentinel cards. We will stop once we have performed $r - p + 1$ basic step. Below is the function merge which runs in O(n) time.

Void merge (int A[]. int p. int q. int r)

```
    {

        int n 1 = q – p + 1
        int n 2 = r – q:
        int* L = new int[n1 + 1];
        int * R = new int [ n2 + 1];
        for (int i = 1; i < = n1; i ++)
                L [i] = A [ p + i – 1];

        for (int j = 1; j < = n2; j++)
                R[ j] = A [q + j];
        L[0] = INT_MIN;          //negative infinity
        R[0] = INT_MIN;          //negative infinity
        L[n1 + 1] = INT_MAX;          // positive infinity
        R[n2 + 1] = INT_MAX;// positive infinity

        i = j = 1;
for (int k = p;k <=r; k++)
'
'

        if (L[i] < = R [j])

                '
                '

                        A[k] = L[i];
```

```
                    i + = 1;

              ‘

              ‘

              else
              ‘

              ‘

                        A[k] = R[j];
                        j + = 1;
              ‘

              ‘                    }
```

Analysing merge sort

For simplicity, we will assume that n is a power of 2. Each divide step yields two subproblems, both of size exactly n/2. The base case occurs when n = 1. When n ≥ 2, time for merge sort steps are given below:

Divide: Just compute q as the average of p and r. $D(n) = O(1)$.

Conquer: Recursively solve 2 subproblems, each of size n/2. Therefore, total time is $2T(n/2)$.

Combine: MERGE on an n-element subarray takes $O(n)$ time. Therefore, $C(n) = O(n)$. Since $D(n) = O(1)$ and $C(n) = O(n)$, summed together they give a function that is linear in n: $O(n)$. Recurrence for merge sort running time is

$T(n) = O(1)$ if $n = 1$,
$T(n) = 2T(n/2) + O(n)$ if $n ≥ 2$.

Solving the merge-sort recurrence: By the master theorem, this recurrence has the solution $T(n) = O(n \lg n)$. Compared to insertion sort ($O(n^2)$ worst-case time), merge sort is faster. Trading a factor of n for a factor of lg n is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

1.5.2   Quick Sort

As mentioned earlier, the purpose of discussing the Quick Sort Algorithm is to discuss its analysis

- In the previous section, we discussed how the divide-and-conquer technique can be used to design sorting **algorithm Merge-sort**, as summarized below:

  - Partition *n* elements array *A* into two subarrays of *n/2* elements each
  - Sort the two subarrays recursively
  - Merge the two subarrays

Running time: $T(n) = 2T(n/2) + \theta(n \log n)$.

The Quick-Sort algorithm is obtained by exploring another possibility of dividing the elements such that there is no need of merging, that is

Partition *a* [1…*n*] into subarrays A′ = A [1..q] and A″ = A[q + 1…n] such that all elements in A″ are larger than all elements in A′.
Recursively sort A′ and A″.

(*nothing to combine/merge. A is already sorted after sorting A′ and A″*)

Pseudo code for QUICKSORT:

```
QUICKSORT (A, p, r)
If p < r THEN
        q = PARTITION (A, p, r)
        QUICKSORT (A, p, q − 1)
        QUICKSORT (A, q + 1, r)
end if
```

The algorithm PARTITION, which is called by QUICKSORT, is defined after a short while.

Then, in order to sort an array A of n elements, we call QUICKSORT with the three parameters A, 1 and n QUICKSORT (*A, 1, n*).

If q = n/2 and is $\theta(n)$ time, we again get the recurrence. If T(n) denotes the time taken by QUICKSORT in sorting an array of n elements.

T(n) = $2T$(n/2) + θ (n). Then after solving the recurrence we get the running time as
$\Rightarrow T(n) = \theta$ (n log n)

The problem is that it is hard to develop partition algorithm which always divides *A* in two halves.

```
PARTITION (A, p, r)
x = A [ r]
i = p − 1
FOR j = p TO  r − 1 DO
        IF A [J] ≤ r THEN
                i = i + 1
                Exchange A [ i] and A[j]
        end if

end Do
Exchange A[I + 1] and A [r]
RETURN i + 1
```

QUICKSORT correctness:

- Easy to show inductively, if PARTITION works correctly

Example:

‖2	8	7	1	3	5	6	4	i = 0, j = 1
2 ‖	8	7	1	3	5	6	4	i = 1, j = 2
2 ‖	8 ‖	7	1	3	5	6	4	i = 1, j = 3
2 ‖	8	7 ‖	1	3	5	6	4	i = 1, j = 4
2	1 ‖	7	8 ‖	3	5	6	4	i = 2, j = 5
2	1	3 ‖	8	7 ‖	5	6	4	i = 3, j = 6
2	1	3 ‖	8	7	5 ‖	6	4	i = 3, j = 7
2	1	3 ‖	8	7	5	6	4	i = 3, j = 8
2	1	3 ‖	4 ‖	7	5	6	8	q = 4

Average running time

The natural question: what is the average case running time of QUICKSORT?
Is it close to worst case ($\theta(n^2)$, or to the best case θ (*n lg n*) ? Average time

depends on the distribution of inputs for which we take the average.

- If we run QUICKSORT on a set of inputs that are already sorted, the average running time will be close to the worst-case.

- Similarly, if we run QUICKSORT on a set of inputs that give good splits, the average running time will be close to the best-case.

- If we run QUICKSORT on a set of inputs which are picked uniformly at random from the space of all possible input permutations, then the average case will also be close to the best-case. Why? Intuitively, if any input ordering is equally likely, then we expect at least as many good splits as bad splits, therefore on the average a bad split will be followed by a good split, and it gets "absorbed" in the good split.

So, under the assumption that all input permutations are equally likely, the average time of QUICKSORT is θ (*n lg n*) (intuitively). Is this assumption realistic?

- Not really. In many cases the input is almost sorted: think of rebuilding indexes in a database etc.

The question is: how can we make QUICKSORT have a good average time irrespective of the input distribution?

- Using randomization.

1.6   RANDOMIZATION QUICK SORT[*]

Next, we consider what we call *randomized algorithms*, that is, algorithms that make some random choices during their execution.

Running time of normal *deterministic* algorithm only depend on the input.

Running time of a randomized algorithm depends not only on input but also on the random choices made by the algorithm.

Running time of a randomized algorithm is not fixed for a given input!

Randomized algorithms have best-case and worst-case running times, but the inputs for which these are achieved are not known, they can be any of the inputs.
We are normally interested in analyzing the *expected* running time of a randomized algorithm, that is the expected (average) running time for all inputs of size *n*

$$T_c\,(\mathrm{n}) = E_{|x|\ =n}\,\big|\,T\,(X)\,\big|$$

Randomized Quicksort

We can enforce that all *n*! permutations are equally likely by randomly permuting the input before the algorithm.

- Most computers have pseudo-random number generator *random* (1, n) returning "random" number between 1 and *n*.
- Using pseudo-random number generator we can generate a random permutation (such that all n! permutations equally likely) in O(*n*) time:

Choose element in *A*[1] randomly among elements in *A*[1..n], choose element in *A*[2] randomly among elements in *A* [2..n], choose element in *A*[3] randomly among elements in *A* [3..n] and so on.

---

[*] This section may be omitted after one reading.

- Alternatively we can modify PARTITION slightly and exchange last element in *A* with random element in *A* before partitioning.

```
RAND PARTITION  (A. p. r)
i = RANDOM (p, r)
Exchange A ⌐r⌐ and A [i ]
RETURN PARTITION (A, p, r)
```

```
RANDQUICKSORT (A, p, r)
IF p < r  THEN
        q = RANDPARTITION (A,p,r)
    RANDQUICKSORT (A, p, q − 1, r)

END IF
```

Expected Running Time of Randomized Quicksort

Let $T(n)$ be the running of RANDQUICKSORT for an input of size *n*.

- Running time of RANDQUICKSORT is the total running time spent in all PARTITION calls.

- PARTITION is called *n* times

    – The pivot element *r* is not included in any recursive calls.

One call of PARTITION takes $O(1)$ time plus time proportional to the number of iterations FOR-loop.

    – In each iteration of FOR-loop we compare an element with the pivot element.

If X is the number of comparisons $A[j] \leq r$ performed in PARTITION over the entire execution of RANDQUICKSORT then the running time is $O(n + X)$.

$E[T(n)] = E[O(n + X)] = n + E[X]$
To analyse the expected running time we need to compute $E[X]$
To compute X we use $z_1, z_2 \ldots z_n$ to denote the elements in *A* where z_i is the *i*th smallest element. We also use Z_{ij} to denote $\{z_i, z_{i+1} \ldots z_j\}$.

Each pair of elements z_i and z_j are compared at most once (when either of them is the pivot)

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \quad \text{where}$$

$$X_{ij} = \begin{cases} 1 & \text{If } z_i \text{ comparedto } z_i \\ 0 & \text{If } z_i \text{ notcomparedto } z_i \end{cases}$$

$$E|X| = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij} \right]$$

$$= \quad \Sigma_{i=1}^{n-1} \, \Sigma_{j=i+1}^{n} \, \Pr\left[z_i \, \text{comparedto} z_j\right]$$

To compute *Pr* [z_i compared to z_j] it is useful to consider when two elements are *not* compared.

Example: Consider an input consisting of numbers 1 through *n*.

Assume first pivot it $7 \Rightarrow$ first partition separates the numbers into sets

{1, 2, 3, 4, 5, 6} and {8, 9, 10}.

In partitioning. 7 is compared to all numbers. No number from the first set will ever be compared to a number from the second set.

In general once a pivot *r,* $z_i < r < z_j$, is chosen we know that z_i and z_j cannot later be compared.

On the other hand if z_i is chosen as pivot before any other element in Z_{ij} then it is compared to each element in Z_{ij}. Similar for z_j.

In example 7 and 9 are compared because 7 is first item from $Z_{7,9}$ to be chosen as pivot and 2 and 9 are not compared because the first pivot in $Z_{2,9}$ is 7.

Prior to an element in Z_{ij} being chosen as pivot the set Z_{ij} is together in the same partition $\Rightarrow$ any element in Z_{ij} is equally likely to be first element chosen as pivot $\Rightarrow$ the probability that z_i or z_j is chosen first in Z_{ij} is $\dfrac{1}{j-i+1}$

$$Pr \, [z_i \text{ compared to } z_j] = \dfrac{2}{j-i+1}$$

- We now have:

$$
\begin{aligned}
E\,|X| \quad = & \quad \Sigma_{i=1}^{n-1} \, \Sigma_{j=i+1}^{n} \, \Pr\left[z_i \, \text{comparedto} z_j\right] \\
= & \quad \Sigma_{i=1}^{n-1} \, \Sigma_{j=i+1}^{n} \, \dfrac{2}{j-i+1} \\
\\
= & \quad \Sigma_{i=1}^{n-1} \, \Sigma_{k=1}^{n-i} \, \dfrac{2}{k+1} \\
= & \quad \Sigma_{i=1}^{n-1} \, \Sigma_{k=1}^{n-i} \, \dfrac{2}{k} \\
= & \quad \Sigma_{i=1}^{n-1} \, O\,(\log n) \\
= & \quad O(n \log n)
\end{aligned}
$$

Since best case is 0(n lg n) $\Rightarrow E \,|X| = \theta \,(n \lg n)$ and therefore $E\,|T(n)| = \theta \,(n \lg n)$.

Next time we will see how to make quicksort run in worst-case O(*n* log *n*) time.

1.7 FINDING THE MEDIAN

The problem of finding the median of n elements is a particular case of the more general selection problem. The selection problem can be stated as finding the ith order statistic in a set of n numbers or in other words the ith smallest element in a set of n element. The minimum is thus the 1st order statistic, maximum is the nth order

statistic and median is the $n/2^{th}$ order statistic. Also note that if n is even then there will be two medians.

We will give here two algorithms for the solution of above problem. One is practical randomized algorithm with O(n) expected running time. Another algorithm which is more of theoretical interest only, with O(n) worst case running time.

Randomized Selection

The key idea is to use the algorithm partition () from quicksort but we only need to examine one subarray and this saving also shows up in the running time O(n). We will use the Randomized Partition (A, p,r) which randomly partitions the Array A around an element A[q] such that all elements from A[p] to A[q−1] are less than A[q] and all elements A[q+1] to A[r] are greater than A[q]. This is shown in a diagram below.

q = Randomized Partition (A, p, r)

≤ A [q]	A[q]	≥ A [q]
p	q	r

We can now give the pseudo code for Randomized Select (A, p, r, i). This procedure selects the ith order statistic in the Array A [p ..r].

Randomized Select (A, p, r, i)

> **if (p = = r) then return A [p];**
> **q = Randomized Partition (A, p, r)**
> **k = q – p + 1;**
> **if (i ==k) then return A [q] |q|;**
> **if (i < k) then**
> > **return Randomized Select (A, p, q-1, i);**
> **else**
> > **return Randomized Select (A, q+1, r, i-k);**

Analyzing **Randomized Select ()**

Worst case: The partition will always occur in 0:n-1 fashion. Therefore, the time required by the Randomized Select can be described by a recurrence given below:

$$T(n) \quad = \quad T(n-1) + O(n)$$
$$= \quad O(n^2) \quad \text{(arithmetic series)}$$

This running time is no better than sorting.
"Best" case: suppose a 9:1 partition occurs. Then the running time recurrence would be

$$T(n) \quad = \quad T(9n/10) + O(n)$$
$$= \quad O(n) \quad \text{(Master Theorem, case 3)}$$

Average case: Let us now analyse the average case running time of Randomized Select.

For upper bound, assume *i*th element always occurs in the larger side of partition:

$$T(n) \quad \leq \quad \frac{1}{n} \sum_{k=0}^{n-1} T(\max(k, n - k - 1)) + \Theta(n)$$

$$\leq \quad \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

Let's show that $T(n) = O(n)$ by substitution.

Assume $T(n) \leq cn$ for sufficiently large c:

$$T(n) \quad \leq \quad \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + \Theta(n) \qquad \textit{The recurrence we started with}$$

$$\leq \quad \frac{2}{n} \sum_{k=n/2}^{n-1} ck + \Theta(n) \qquad \textit{Substitute T(n)} \leq \textit{cn for T(k)}$$

$$= \quad \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n) \qquad \textit{Split" the recurrence}$$

$$= \quad \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n) \quad \textit{Expand arithmetic series}$$

$$= \quad c(n-1) - - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n) \qquad \textit{Multiply it out}$$

$$T(n) \quad \leq \quad c(n-1) - - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n) \qquad \textit{The recurrence so far}$$

$$= \quad cn - c - \frac{cn}{4} - \frac{c}{2} + \Theta(n) \qquad \textit{Subtract c/2}$$

$$= \quad cn - (\frac{cn}{4} - \frac{c}{2} + \Theta(n)) \qquad \textit{Rearrange the arithmetic}$$

$$\leq \quad cn \quad \text{(if c is big enough)} \qquad \textit{What we set out to prove}$$

Worst-Case Linear Time Selection

Randomized algorithm works well in practice. But there exists an algorithm which has a worst-case $O(n)$ time complexity for find the [i]th order statistic but which is only of theoretical significance. The basis idea of worst case linear time selection is to generate a good partitioning element. We will call this element x.

The algorithm in words:

1. Divide n elements into groups of 5

2. Find median of each group

3. Use Select () recursively to find median x of the [n/5] medians

4. Partition the n elements around x. Let $k = $ rank (x)

5. **if** (i == k) **then** return x

6. **if** (i < K) **then** use Select () recursively to find ith smallest element in first partition

7. **else** (i > k) use Select () recursively to find $(i-k)$ th smallest element in last partition.

There are at least ½ of the 5-element medians which are $\leq$ x which equal to $\lfloor\lfloor n/5 \rfloor/2 \rfloor = \lfloor n/10 \rfloor$ and also there are at least $3\lfloor n/10 \rfloor$ elements which are $\leq$ x. Now, for large n, $3\lfloor n/10 \rfloor \geq n/4$. So at least n/4 elements are $\leq$ x and similarly n/4 elements are $\geq x$. Thus after partitioning around x, step 5 will call Select () on at most $3n/4$ elements. The recurrence is therefore.

$$
\begin{aligned}
T(n) &\leq & T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) & \\
&\leq & T(n/5) + T(3n/4) + \Theta(n) & \qquad \lfloor n/5 \rfloor \leq n/5 \\
&\leq & cn/5 + 3cn/\Theta(n)) & \qquad \textit{Substitute } T(n) = cn \\
&= & 19cn/20 + \Theta(n) & \qquad \textit{Combine fractions} \\
&= & cn - (cn/20 - \Theta(n)) & \qquad \textit{Express in desired form} \\
&\leq & cn \text{ if c is big enough} & \qquad \textit{What we set out to prove}
\end{aligned}
$$

1.8 MATRIX MULTIPLICATION

In number of problems, matrix multiplications form core of algorithms to solve the problems under consideration. Any saving in time complexity may lead to significant improvement in the overall algorithm. The conventional algorithm makes $O(n^3)$ integer multiplications. Next, we discuss Strassen's Algorithm which makes only $O(n^{2.8})$ integer multiplications for multiplying 2 n × n matrices.

Strassen's Algorithm

Strassen's recursive algorithm for multiplying n × n matrices runs in
$\theta(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large value of n, therefore, it outperforms the $\theta(n^3)$ matrix-multiplication algorithm.

The idea behind the Strassen's algorithm is to multiply 2 × 2 matrices with only 7 scalar multiplications (instead of 8). Consider the matrices

$$
\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}
$$

The seven submatrix products used are

$$
\begin{aligned}
P_1 &= a \cdot (g - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (f - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (f + h) \\
P_7 &= (a - c) \cdot (e + g)
\end{aligned}
$$

Using these submatrix products the matrix products are obtained by

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

This method works as it can be easily seen that s = (ag − ah) + (ah + bh) = ag + bh. In this method there are 7 multiplications and 18 additions. For (n × n) matrices, it can be worth replacing one multiplication by 18 additions, since multiplication costs are much more than addition costs.

The recursive algorithm for multiplying n × n matrices is given below:

1. Partition the two matrices A, B into n/2 × n/2 matrices.
2. Conquer: Perform 7 multiplications recursively.
3. Combine: Form using + and −.

The running time of above recurrence is given by the recurrence given below:

$$
\begin{aligned}
T(n) &= 7T(n/2) + \theta(n^2) \\
&= \theta(n^{\lg 7}) \\
&= O(n^{2.81})
\end{aligned}
$$

The current best upper bound for multiplying matrices is approximately $O(n^{2.376})$

Limitations of Strassen's Algorithm

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for the following four reasons:

1. The constant factor hidden in the running time of Strassen's algorithm is larger than the constant factor in the naïve $\theta(n^3)$ method.
2. When the matrices are sparse, methods tailored for sparse matrices are faster.
3. Strassen's algorithm is not quite as numerically stable as the naïve method.
4. The sub matrices formed at the levels of recursion consume space.

Ex. 3) Multiply the following two matrices using Strassen's algorithm

$$\begin{bmatrix} 5 & 6 \\ -4 & 3 \end{bmatrix} \text{ and } \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

1.9 EXPONENTIATION

Exponentiating by Squaring is an algorithm used for the fast computation of large powers of number x. It is also known as the **square-and-multiply** algorithm or **binary exponentiation**. It implicitly uses the binary expansion of the exponent. It is of quite general use, for example, in modular-arithmetic.

Squaring Algorithm

The following recursive algorithm computes x^n, for a positive integer n:

$$\text{Power}(x, n) = \begin{cases} x, & \text{if } n = 1 \\ \text{Power}(x^2, n/2), & \text{if } n \text{ is even} \\ x \cdot (\text{Power}(x^2, (n-1)/2)), & \text{if } n > 2 \text{ is odd} \end{cases}$$

Compared to the ordinary method of multiplying x with itself $n-1$ times, this algorithm uses only $O(\log n)$ multiplications and therefore speeds up the computation of x^n tremendously.

Further Applications

The same idea allows fast computation of large exponents modulo a number. Especially in cryptography, it is useful to compute powers in a ring of integers modulo q. It can also be used to compute integer powers in a group, using the rule

$$\text{Power}(x, -n) = (\text{Power}(x, n))^{-1}.$$

The method works in every semigroup and is often used to compute powers of matrices.

Examples 1.9.1:

$$13789^{722341} \pmod{2345}$$

would take a very long time and lots of storage space if the native method is used: compute 13789^{72234} then take the remainder when divided by 2345. Even using a more effective method will take a long time: square 13789, take the remainder when divided by 2345, multiply the result by 13789, and so on. This will take 722340 modular multiplications. The square-and-multiply algorithm is based on the observation that $13789^{722341} = 13789 (13789^2)^{361170}$. So if we computed 13789^2, then the full computation would only take 361170 modular multiplications. This is a gain

of a factor of two. But since the new problem is of the same type, we can apply the same observation *again*, once more approximately having the size.

The repeated application this algorithm is equivalent to decomposing the exponent (by a base conversion to binary) into a sequence of squares and products: for example,

$$x^7 = x^4\, x^2 x^1$$
$$= (x^2)^2 * x^2 * x$$
$$= (x^2 * x)^2 * x \qquad \textit{algorithm needs only 4 multiplications instead of}$$
$$7 - 1 = 6$$
$$\text{where } 7 = (111)_2 = 2^2 + 2^1 + 2^0$$

Some more examples:

$x^{10} = ((x^2)^2 * x)^2$ because $10 = (.1010)_2 = 2^3 + 2^1$, algorithm needs 4 multiplications instead of 9

$x^{100} = ((((x^2 * x)^2)^2)^2 * x)^2$ because $100 = (1\ 100\ 100)2 = 2^6 + 2^5 + 2^2$, algorithm needs 8 multiplications instead of 99

$x^{1,000} = ((((((x^2 * x)^2 * x)^2 * x)^2 * x)^2)^2$ because $10^3 = (1\ 111\ 101\ 000)_2$, algorithm needs 14 multiplications instead of 999

$x^{1,000,.000} = ((((((((((((((x^2 * x)^2 * x)^2 * x)^2)^2)^2)^2 * x)^2)^2)^2)^2$ because $10^6 = (\ 11\ \ 110\ 100\ \ 001\ 001\ 000\ \ 000)_2$, algorithm needs 25 multiplications

$x^{1.000.000.000} = ((((((((((((((((((((((x^2 * x)^2 * x)^2)^2 * x)^2\ )^2)^2 * x)^2)^2 * x)^2)^2 * x)^2 * x)^2)^2\ x)^2 x)^2)^2 * x)^2)^2)^2)^2)^2)^2)^2)^2$ because $10^9 = (111\ \ 0011\ \ 100\ 101\ 101\ 100\ 101\ 000\ 000\ 000)_2$, algorithm needs 41 multiplications

Addition Chain

In mathematics, an **addition chain** is a sequence a_0, a_1,−a_2, a_3… that satisfies

$a_0 = 1$, and
for each $k > 0$:
$a_k = a_i + a_j$ for some $i < j < k$

For example: 1, 2, 3, 6, 12, 24, 30, 31 is an addition chain for 31, of length 7, since

$2 = 1 + 1$
$3 = 2 + 1$
$6 = 3 + 3$
$12 = 6 + 6$
$24 = 12 + 12$
$30 = 24 + 6$
$31 = 30 + 1$

Addition chains can be used for exponentiation: thus, for example, we only need 7 multiplications to calculate 5^{31}:

$5^2 = 5^1 \times 5^1$
$5^2 = 5^2 \times 5^1$
$5^6 = 5^3 \times 5^3$
$5^{12} = 5^6 \times 5^6$
$5^{24} = 5^{12} \times 5^{12}$
$5^{30} = 5^{24} \times 5^6$
$5^{31} = 5^{30} \times 5^1$

Addition chain exponentiation

In mathematics, **addition chain exponentiation** is a fast method of exponentiation. It works by creating a minimal-length addition chain that generates the desired exponent. Each exponentiation in the chain can be evaluated by multiplying two of the earlier exponentiation results.

This algorithm works better than binary exponentiation for high exponents. However, it trades off space for speed, so it may not be good on over-worked systems.

1.10   SUMMARY

The unit discusses various issues in respect of the technique viz., *Divide and Conquer* for designing and analysing algorithms for solving problems. First, the general plan of the *Divide and conquer* technique is explained and then an outline of a formal *Divide-and-conquer* procedure is defined. The issue of whether at some stage to solve a problem directly or whether to further subdivide it, is discussed in terms of the relative efficiencies in the two alternative cases.

The technique is illustrated with examples of its applications to solving problems of (large) integer multiplication, Binary search, Sorting, of finding median of a given data, of matrix multiplication and computing exponents of a given number. Under sorting, the well-known techniques viz., Merge-sort and quick-sort are discussed in detail.

1.11   SOLUTIONS/ANSWERS

Ex.1) 1026732×732912

In order to apply Karatsuba's method, first we make number of digits in the two numbers equal, by putting zeroes on the left of the number having lesser number of digits. Thus, the two numbers to be multiplied are written as

$x = 1026732$ and $y = 0732912$.

As n = 7, therefore [n/2] = 3, we write

$x = 1026 \times 10^3 + 732 = a \times 10^3 + b$
$y = 0732 \times 10^3 + 912 = c \, 10^3 + d$
where   a = 1026,   b = 732
        c = 0732,   d = 912

Then

$$x \times y = (1026 \times 0732) \, 10^{2 \times 3} + 732 \times 912$$
$$+ \, [(1026 + 732) \times (732 + 912)$$
$$- (1026 \times 0732) - (732 \times 912)]10^3$$
$$= (1026 \times 0732) \, 10^6 + 732 \times 912 +$$
$$[(1758 \times 1644) - (1026 \times 0732) - (732 \times 912)]10^3$$
$$\dots (A)$$

Though, the above may be simplified in another simpler way, yet we want to explain Karatsuba's method, therefore, next, we compute the products.

U   = 1026×732
V   = 732×912
P   = 1758×1644

Let us consider only the product 1026×732 and other involved products may be computed similarly and substituted in (A).

Let us write

$$U = 1026 \times 732 = (10 \times 10^2 + 26) \, (07 \times 10^2 + 32)$$
$$= (10 \times 7) \, 10^4 + 26 \times 32 + [(10 + 7) \, (26 + 32)$$
$$- 10 \times 7 - 26 \times 32)] \, 10^2$$

$$= 17 \times 10^4 + 26 \times 32 + (17 \times 58 - 70 - 26 \times 32)\, 10^2$$

At this stage, we do not apply Karatsuba's algorithm and compute the products of 2-digit numbers by conventional method.

Ex. 2) The number of elements in the given list is 15. Let us store these in an array say A[1..15]. Thus, initially low = 1 and high = 15 and, hence, mid = (1+15)/2 = 8.

In the first iteration, the search algorithm compares the value to be searched with A[8] = 78

Part (i): The value to be searched = 15
As 15 < A[8] = 78, the algorithm iterates itself once more. In the second iteration, the new values become

$$\text{low} = 1 \quad \text{high} = \text{mid} - 1 = 7$$

and hence (new) $\text{mid} = \left\lceil \dfrac{1+7}{2} \right\rceil = 4$

As 15 < A[4] = 26, the algorithm iterates once more. In the third iteration, the new values become

$$\text{low} = 1, \quad \text{high } 4 - 1 = 3$$

Therefore, $\text{mid} = \left(\dfrac{1+3}{2} \right) = 2$

As 15 > A[2] = 12, the algorithm iterates itself once more. In fourth iteration, new values become
$$\text{low} = \text{mid} + 1 = 2 + 1 = 3, \qquad \text{high} = 4$$

Therefore

$$(\text{new}) \quad \text{mid} \quad = \left\lceil \dfrac{3+4}{2} \right\rceil = 3$$

As A[3] = 15 (*the value to be searched*)
Hence, the algorithm terminates and returns the index value 3 as output.

Part (ii): The value to be searched = 93

As the first iteration is common for all values to be searched, therefore, in the first iteration

$$\text{low} = 1, \qquad \text{high} = 15 \qquad \text{and mid} = 8$$
As 93 > A[8] = 78,

therefore, the algorithm iterates once more. In the second iteration, new values are

$$\text{low} = \text{mid} + 1 = 9, \qquad \text{high} = 15$$

and (new) mid = $\dfrac{9+15}{2} = 12$, where A[12] = 108

As 93 < A [12] = 108,

Therefore, the algorithm iterates once more. For the third

$$\text{low} = 9 ; \qquad \text{high} = \text{mid} - 1 = 12 - 1 = 11$$

and (new) mid = $\dfrac{9 + 11}{2}$ = 10 with A[10] = 93

As A[10] = 93, therefore, the algorithm terminates and returns the index value 10 of the given array as output.

Part (iii):   The value to be searched is 43. As explained earlier, in the first iteration

low = 1,   high = 15   and   mid = 8

As 43 < A[8] = 78, therefore, as in part (i)

low = 1,   high = 8 − 1 = 7 and   mid = 4

As 43 > A[4] = 26, the algorithm makes another iteration in which

low = mid + 1 = 5   high = 7 and   (new) mid = (5 + 7)/2 = 6

Next, as 43 < A[6] = 48, the algorithm makes another iteration, in which

low = 5   high = 6 − 1 = 5
hence mid = 5, and A[5] = 35
As 43 > A[5], hence value ≠ A[5].

But, at this stage, low is not less than high and hence the algorithm returns − 1, indicating failure to find the given value in the array.

Ex. 3) Let us denote

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

and

$$\begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

Then

$P_1 = a \cdot (g - h)$
  $= 5 (6 - 9) = - 15$
$P_2 = (a + b) \cdot h = (5 + 6) \cdot 9 = 99,$
$P_3 = (c + d) \cdot e = (-4 + 3) \cdot (- 7) = 7$
$P_4 = d \cdot (f - e) = 3 \cdot (5 - (- 7)) = 36;$
$P_5 = (a + d) (e + h) = (5 + 3) (- 7 + 9) = 16$
$P_6 = (b - d) (f + h) = (6 - 3) \cdot (5 + 9) = 42$
$P_7 = (a - c) (e + g) = (5 - (- 4)) (- 7 + 6) = - 9$

Then, the product matrix is

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

where

$r = P_5 + P_4 - P_2 + P_6$
  $= 16 + 36 - 99 + 42 = -5$
$s = P_1 + P_2 = -15 + 99 = 84$
$t = P_3 + P_4 = 7 + 36 = 43$
$u = P_5 + P_1 - P_3 - P_7$
  $= 16 + (-15) - 7 - (-9)$
  $= 16 - 15 - 7 + 9$
  $= 3$

1.12 FURTHER READINGS

1.   *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).

2.   *Algoritmics: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).

3.   *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).

4.   *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).

5.   *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni: (Galgotia Publications).

6.   *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).

7.   *Programming Languages* (*Second Edition*) – *Concepts and Constructs,* Ravi Sethi: (Pearson Education, Asia, 1996).