USN | | | | | | | | | | |

Internal Assessment Test 1 – March 2025

Sub:	ANALYSIS & DESIGN OF ALGORITHMS					Sub Code:	BCS401	Branch:	AIML/CSE-AIML
Date:	27.03.25	Duration:	90 mins	Max Marks:	50	Sem/Sec:	IV -A, B & C		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.	Explain the general framework for analyzing the efficiency of algorithms. Ans:	10M	CO1	L1

2.1 The Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. We already mentioned in Section 1.2 that there are two kinds of efficiency: time efficiency and space efficiency. ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs. ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output. In the early days of electronic computing, both resources—time and space—were at a premium. Half a century of relentless technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now the amount of extra space required by an algorithm is typically not of as much concern, with the caveat that there is still, of course, a difference between the fast main memory, the slower secondary memory, and the cache. The time issue has not diminished quite to the same extent, however. In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, following a well-established tradition of algorithm textbooks, we primarily concentrate on time efficiency, but the analytical framework introduced here is applicable to analyzing space efficiency as well.

Measuring an Input's Size

Let's start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.[1] In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists. For the problem of evaluating a polynomial $p(x) = a_n x^n + \cdots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree. You'll see from the discussion that such a minor difference is inconsequential for the efficiency analysis.

There are situations, of course, where the choice of a parameter indicating an input size does matter. One such example is computing the product of two $n \times n$ matrices. There are two natural measures of size for this problem. The first and more frequently used is the matrix order n. But the other natural contender is the total number of elements N in the matrices being multiplied. (The latter is also more general since it is applicable to matrices that are not necessarily square.) Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of these two measures we use (see Problem 2 in this section's exercises).

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring input size for algorithms solving problems such as checking primality of a positive integer n. Here, the input is just one number, and it is this number's magnitude that determines the input

1. Some algorithms require more than one parameter to indicate the size of their inputs (e.g., the number of vertices and the number of edges for algorithms on graphs represented by their adjacency lists).

size. In such situations, it is preferable to measure size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1. \tag{2.1}$$

This metric usually gives a better idea about the efficiency of algorithms in question.

Units for Measuring Running Time

The next issue concerns units for measuring an algorithm's running time. Of course, we can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm. There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the program. Since we are after a measure of an *algorithm*'s efficiency, we would like to have a metric that does not depend on these extraneous factors.

One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.[2]

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n. We will find out how to compute such a count for nonrecursive and recursive algorithms in Sections 2.3 and 2.4, respectively.

Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate

2. On some computers, multiplication does not take longer than addition/subtraction (see, for example, the timing data provided by Kernighan and Pike in [Ker99, pp. 185–186]).

the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op}C(n).$$

Of course, this formula should be used with caution. The count $C(n)$ does not contain any information about operations that are not basic, and, in fact, the count itself is often computed only approximately. Further, the constant c_{op} is also an approximation whose reliability is not always easy to assess. Still, unless n is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time. It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is 10 times faster than the one we have?" The answer is, obviously, 10 times. Or, assuming that $C(n) = \frac{1}{2}n(n-1)$, how much longer will the algorithm run if we double its input size? The answer is about four times longer. Indeed, for all but very small values of n,

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Note that we were able to answer the last question without actually knowing the value of c_{op}: it was neatly cancelled out in the ratio. Also note that $\frac{1}{2}$, the multiplicative constant in the formula for the count $C(n)$, was also cancelled out. It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's ***order of growth*** to within a constant multiple for large-size inputs.

Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms discussed in Section 1.1 or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important. For large values of n, it is the function's order of growth that counts: just look at Table 2.1, which contains values of a few functions particularly important for analysis of algorithms.

The magnitude of the numbers in Table 2.1 has a profound significance for the analysis of algorithms. The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect a program

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes. Also note that although specific values of such a count depend, of course, on the logarithm's base, the formula

$$\log_a n = \log_a b \log_b n$$

makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant. This is why we omit a logarithm's base and write simply $\log n$ in situations where we are interested just in a function's order of growth to within a multiplicative constant.

On the other end of the spectrum are the exponential function 2^n and the factorial function $n!$ Both these functions grow so fast that their values become astronomically large even for rather small values of n. (This is the reason why we did not include their values for $n > 10^2$ in Table 2.1.) For example, it would take about $4 \cdot 10^{10}$ years for a computer making a trillion (10^{12}) operations per second to execute 2^{100} operations. Though this is incomparably faster than it would have taken to execute 100! operations, it is still longer than 4.5 billion ($4.5 \cdot 10^9$) years—the estimated age of the planet Earth. There is a tremendous difference between the orders of growth of the functions 2^n and $n!$, yet both are often referred to as "exponential-growth functions" (or simply "exponential") despite the fact that, strictly speaking, only the former should be referred to as such. The bottom line, which is important to remember, is this:

> Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Another way to appreciate the qualitative difference among the orders of growth of the functions in Table 2.1 is to consider how they react to, say, a twofold increase in the value of their argument n. The function $\log_2 n$ increases in value by just 1 (because $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$); the linear function increases twofold, the linearithmic function $n \log_2 n$ increases slightly more than twofold; the quadratic function n^2 and cubic function n^3 increase fourfold and

eightfold, respectively (because $(2n)^2 = 4n^2$ and $(2n)^3 = 8n^3$); the value of 2^n gets squared (because $2^{2n} = (2^n)^2$); and $n!$ increases much more than that (yes, even mathematics refuses to cooperate to give a neat answer for $n!$).

Worst-Case, Best-Case, and Average-Case Efficiencies

In the beginning of this section, we established that it is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. Consider, as an example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. It also assumes that the second condition $A[i] \neq K$ will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//         or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Clearly, the running time of this algorithm can be quite different for the same list size n. In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: $C_{worst}(n) = n$.

The ***worst-case efficiency*** of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$. (For sequential search, the answer was obvious. The methods for handling less trivial situations are explained in subsequent sections of this chapter.) Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other

words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

The ***best-case efficiency*** of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. Accordingly, we can analyze the best-case efficiency as follows. First, we determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.) Then we ascertain the value of $C(n)$ on these most convenient inputs. For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, $C_{best}(n) = 1$ for this algorithm.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either. Though we should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms a good best-case performance extends to some useful types of inputs close to being the best-case ones. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast. Moreover, the best-case efficiency deteriorates only slightly for almost-sorted arrays. Therefore, such an algorithm might well be the method of choice for applications dealing with almost-sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

It should be clear from our discussion, however, that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the ***average-case efficiency*** seeks to provide. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

Let's consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal to p $(0 \le p \le 1)$ and (b) the probability of the first match occurring in the ith position of the list is the same for every i. Under these assumptions—the validity of which is usually difficult to verify, their reasonableness notwithstanding—we can find the average number of key comparisons $C_{avg}(n)$ as follows. In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$. Therefore,

$$\begin{aligned} C_{avg}(n) &= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1 - p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p). \end{aligned}$$

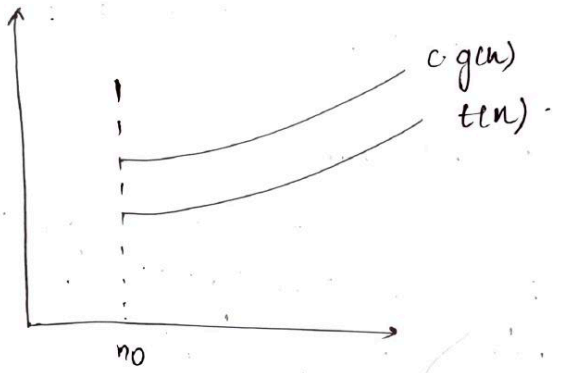
	Recapitulation of the Analysis Framework Before we leave this section, let us summarize the main points of the framework outlined above. ■ Both time and space efficiencies are measured as functions of the algorithm's input size. ■ Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm. ■ The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies. ■ The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity. In the next section, we look at formal means to investigate orders of growth. In Sections 2.3 and 2.4, we discuss particular methods for investigating nonrecursive and recursive algorithms, respectively. It is there that you will see how the analysis framework outlined here can be applied to investigating the efficiency of specific algorithms. You will encounter many more examples throughout the rest of the book.			
2.	Define Big_Oh, Big_Omega and Big_Theta definitions with examples and neat diagrams. Ans:	10M	CO1	L2

ex: $100n^3 + 4n \in \Omega(n^3)$.

which is valid

as $100n^3 + 4n \geq cg(n)$

$100n^3 + 4n \geq 100n^3 + 4n^3$ ✓

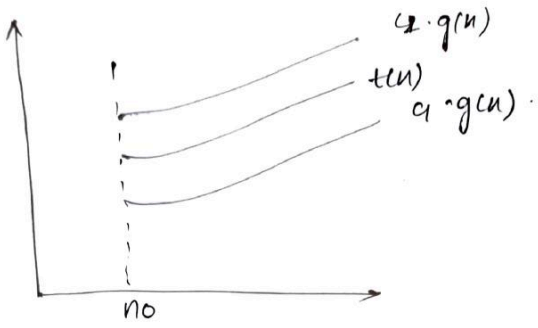


Big Theta notation (Θ).

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted by $t(n) \in \Theta(g(n))$, when it is bounded between or sandwiched between an upper bound (Ω) and lower bound (O); such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n).$$

where c_1 and c_2 are constants and $g(n)$ is the order of growth of the function



it is also known as remainder notation and is used to denote ~~avo~~ average case scenarios, when an element is in between the array.

ex: $n^3 \in \theta(n^3)$.

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

$g(n) = n^3$.

$f(n) = n^3$.

$\therefore$

$$c_1 n^3 \leq n^3 \leq c_2 n^3.$$

lets take c_1 as 1 and c_2 as 2.

then

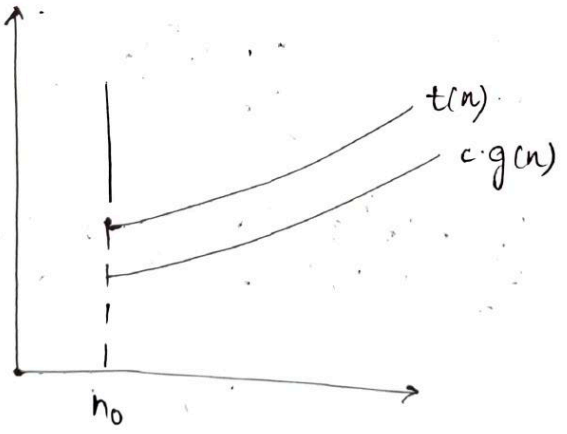
$n^3 \leq n^3 \leq 2n^3$ which is satisfied.

2) Big O notation:

A function $t(n)$ is said to be in $O(g(n))$, denoted by $t(n) \in O(g(n))$, when it is Bounded ABOVE the function such that:

$$f(n) \leq c \cdot g(n)$$

the function $f(n)$ is lesser that or equal to $c \cdot g(n)$, where $g(n)$ is the order of growth and c is some constant for all $n \geq n_0$.



Big O notation is usually used for denoting worst case scenarios in an algorithm, were, ~~the~~ for example, the element to be searched is at the end of the array.

ex: $n^3 \in O(n^3)$

or $n^2 \in O(n^3)$.

Here $g(n) = n^3$.

and $c \cdot g(n) = n^3$

means $c = 1 \quad \forall n \geq n_0$.

$\therefore n^2 \in O(n^3)$.

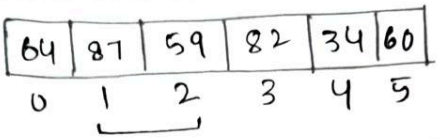
Big Omega (Ω) notation:

A function $t(n)$ is said to be in $\Omega(g(n))$, ~~when~~ denoted by $t(n) \in \Omega(g(n))$, when it is Bounded BELOW the function such that

$f(n) \geq c \cdot g(n)$, the function $f(n)$ is greter than or equal to $c \cdot g(n)$, where $g(n)$ is the order of growth and c is some constant for all $n \geq n_0$.

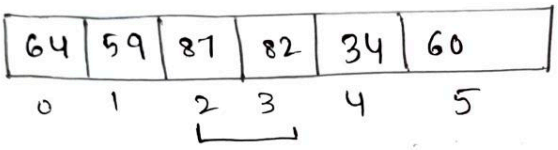
usually its used to denote Best case scenarios in an algorithm, were, for example, the element to be searched, is at the first position or not even present in the array.

3.	Write the algorithm for Bubble Sort and apply the same on the following set of numbers to arrange them in ascending order: 87, 64, 59, 82, 34, 60. Ans:	10M	CO2	L3



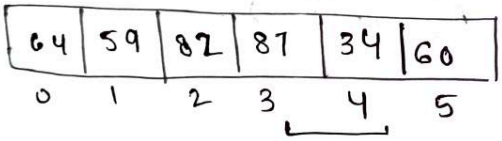
now we compare 1 & 2 index elements :

87 > 59 → swap



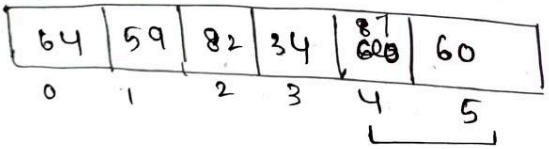
now compare indexes 2 & 3 elements.

87 > 82 → swap



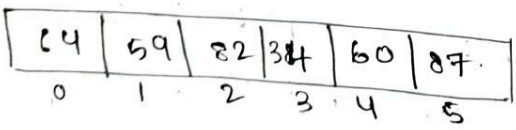
compare 3 and 4 index elements.

87 > 34 → swap.



compare 87 & 60

87 > 60 → swap.



end of itteration 1, we have put the last element in its correct position

CMR

starting 2nd iteration

compare 64 & 59

64 > 59 → swap

59	64	82	34	60	87
0	1	2	3	4	5

comparing 64 and 82 , 64 < 82 , no swap

comparing 82 & 34 82 > 34 → swap

59	64	34	82	60	87
0	1	2	3	4	5

comparing 82 and 60

82 > 60 → swap

59	64	34	60	82	87
0	1	2	3	4	5

Here iteration 2 completes and second Last element is in its correct position

starting iteration 3.

compare 59 & 64 → 59 < 64 → no swap

compare 64 & 34 → 64 > 34 → swap.

59	34	64	60	82	87

64 > 60 → swap

59	34	60	64	82	87

64 < 82 → no swap

or iteration 3 completes

64 is in correct place.

iteration 4:

compare 59 and 34, 34 < 59, swap.

34	59	60	64	82	87
0	1	2	3	4	5

Here it is ordered.

//.

3) Bubble sort :

```
Algorithm Bubblesort ( n, A[]) {
 // input  n: size of array , A[]: input unordered array
 // output sorted array .
for (i ← 0 to n) {
    for ( j ← 0 to  n-i-1) {
          if (A[j] > A[j+1]) {
               swap ( A[j], A[j+1]) .
    }     }
}
```

given array :

87	64	59	82	34	60
0	1	2	3	4	5

- Bubble sort technique sorts elements in pairs and at each iteration the last element gets sorted.
- starting first iteration :

comparing first 2 indexes elements :

87	64
0	1

Here A[j] = 87. A[j+1] = 64

since A[j] > A[j+1] we swap.

∴

64	87
0	1

4.	What is Divide & Conquer technique? Explain with a neat diagram and an example. Ans:	10M	CO2	L2

Let us take an example of mergesort

we need to sort the array using mergesort:

A	2	1	0	4	3.
	0	1	2	3	4.

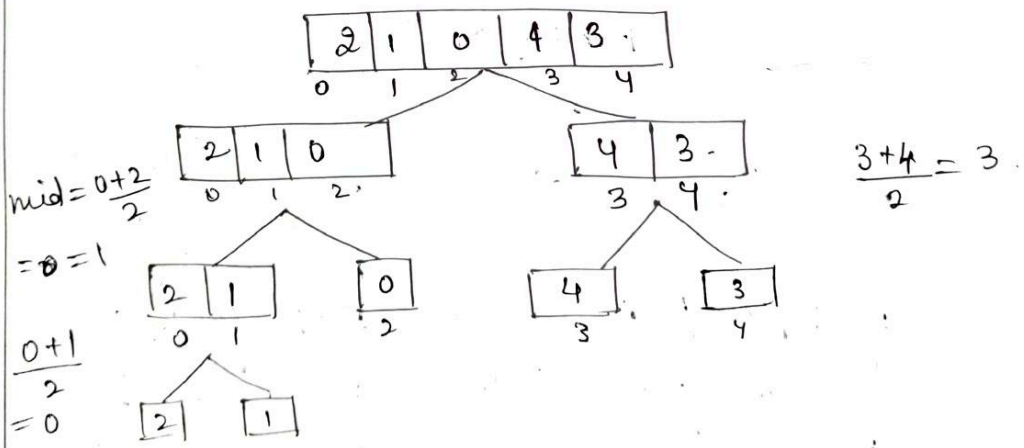
we take low = A[0] = 2

we take high = A[n-1] = A[4] = 3.

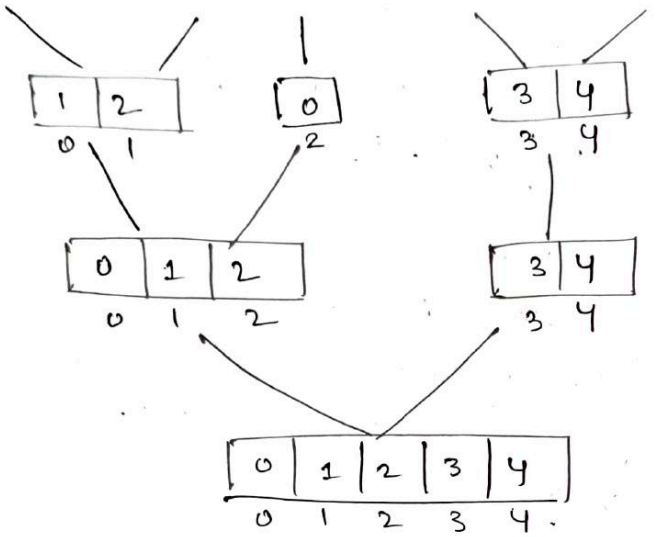
now to divide, we calculate the mid value of the array: mid = (low + high) / 2.

$$mid = \frac{0+4}{2} = 2.$$

the array is divided from low to mid and mid+1 to high

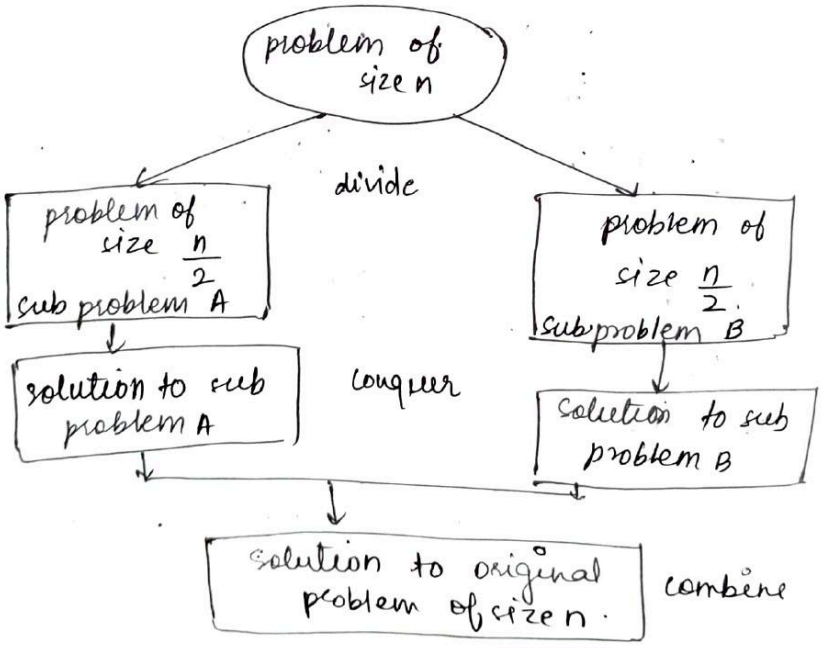


how we ~~combine~~ conquer each value and combine.



∴ the array is sorted by solving sub arrays.

divide & conquer mechanism



4) Divide and conquer:

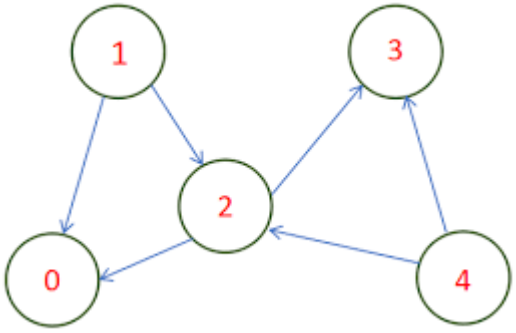
the method divide and conquer is a method of solving problems like sorting, by using 3 main steps:

- divide the big problem into sub problems
- conquer or solve the sub problems
- combine the sub problem solution to achive the final solution for the bigger problem.

The divide and conquer technique usually is used in solving merge sort, quick sort, where the array is divided into 2 parts. and further divided into more 2 parts each. and after solving sub problems, we combine the ~~sub~~ smaller sub problems to get the solution for our main problem.

And the division and conquering of sub probl happens simultaneously.

5.	Write the Warshall's algorithm and find the Transitive Closure for the following graph in neat steps:	10M	CO4	L3

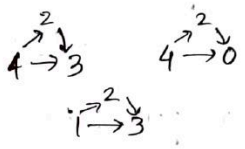


Ans:

lets take (2) as the intermediatary node.

	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	1	0	1	1	0

Here (2) is acting as an intermediatary for



taking (3) as intermediatary node.

	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	1	0	1	1	0

Taking (4) as intermediatary.

	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	1	0	1	1	0

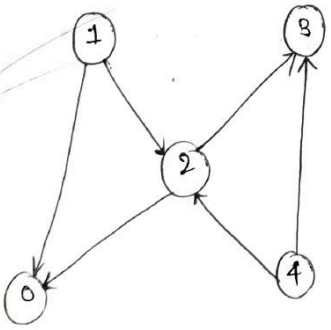
→ Transitive closure.

(5) Algorithm Warshalls (n, A[]) {

// input : size of n or matrix , A[] - adjacency matrix

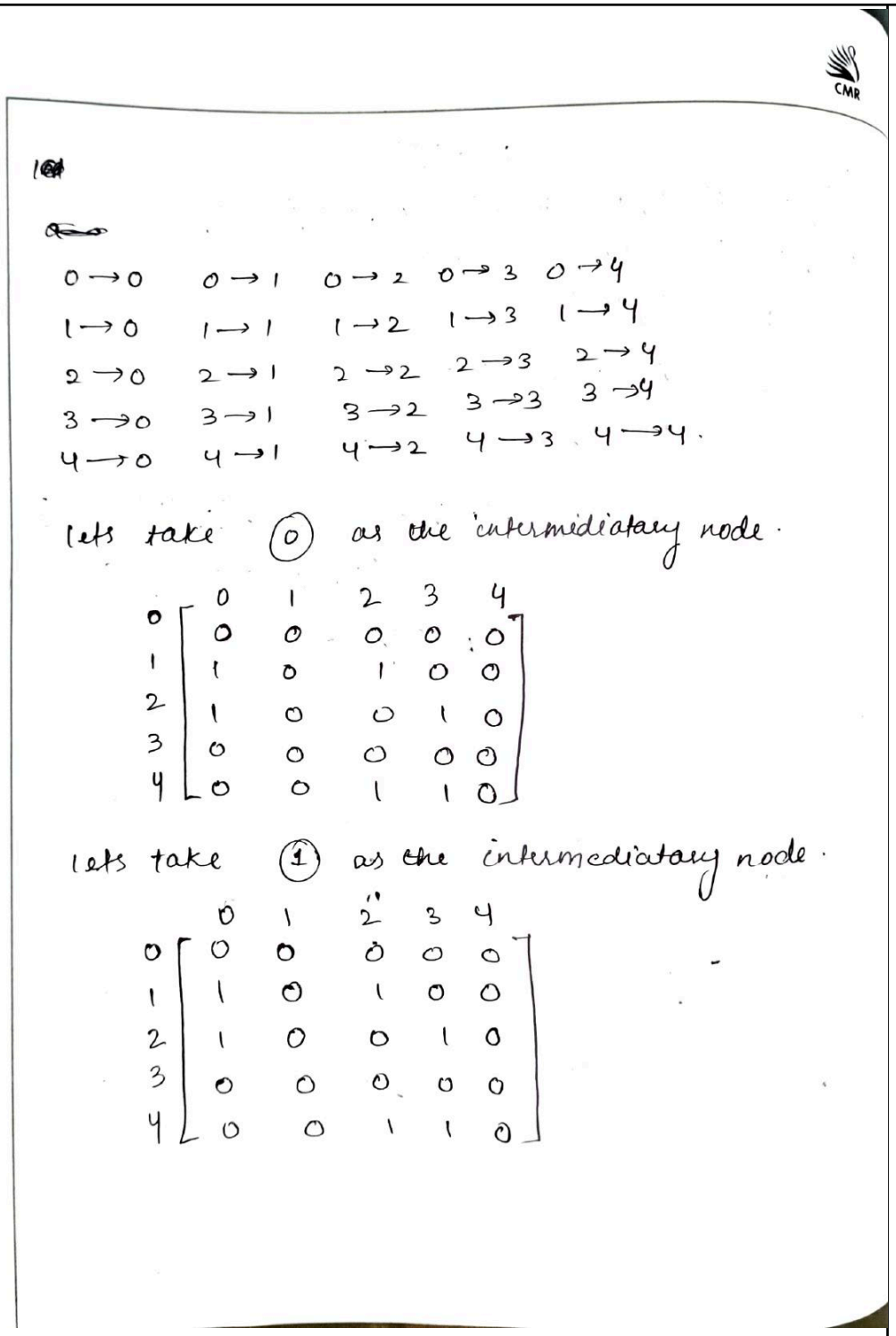
// output : Transitive closure matrix.

```
for ( k ← 0 to n-1 ) {
    for ( i ← 0 to n-1 ) {
        for ( j ← 0 to n-1 ) {
            if ( A[i][j] == 0 && A[i][k] == 1 &&
                 A[k][j] == 1 ) {
                A[i][j] = 1 .
            }
        }
    }
}
```



adjacency matrix :

	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	0	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	0	1	1	0

				
6.	Explain the Merge sort technique with a suitable example and write an algorithm for the same. Ans:	10M	CO2	L2

CI **CCI** **HOD**

---All the Best---