BCO 002A	DATA STRUCTURES AND ALGORITHMS	4-0-0 [4]

OBJECTIVE:

- To study various data structure concepts like Stacks, Queues, Linked List, Trees and Files
- To overview the applications of data structures.
- To be familiar with utilization of data structure techniques in problem solving.
- To have a comprehensive knowledge of data structures and algorithm.
- To carry out asymptotic analysis of algorithm.

UNIT 1	Introduction: Notions of data type, abstract data type and data structures. Importance of algorithms and data structures in programming. Notion of Complexity covering time complexity, space complexity, Worst case complexity & Average case complexity. BigOh Notation, Omega notation, Theta notation. Examples of simple algorithms and illustration of their complexity. Sorting- Bubble sort, selection sort, insertion sort, Quick sort; Heap sort; Merge sort; Analysis of the sorting methods. Selecting the top k elements. Lower bound on sorting.
UNIT 2	Stack ADT, Infix Notation, Prefix Notation and Postfix Notation. Evaluation of Postfix Expression, conversion of Infix to Prefix and Postfix Iteration and Recursion- Problem solving using iteration and recursion with examples such as binary search, Fibonacci numbers, and Hanoi towers. Tradeoffs between iteration and recursion.
UNIT 3	List ADT. Implementation of lists using arrays and pointers. Stack ADT. Queue ADT. Implementation of stacks and queues. Dictionaries, Hash tables: open tables and closed tables. Searching technique- Binary search and linear search. link list- single link list, double link list. Insertion and deletion in link list.
UNIT 4	Binary Trees- Definition and traversals: preorder, post order, in order. Common types and properties of binary trees. Binary search trees: insertion and deletion in binary search tree worst case analysis and average case analysis. AVL trees. Priority Queues -Binary heaps: insert and delete min operations and analysis.
UNIT 5	Graph: Basic definitions, Directed Graphs- Data structures for graph representation. Shortest path algorithms: Dijkstra (greedy algorithm) and Operations on graph. Worshall's algorithm . Depth first search and Breadth-first search. Directed acyclic graphs. Undirected Graphs, Minimal spanning trees and algorithms (Prims and Kruskal) and implementation. Application to the travelling salesman problem.

OUTCOME:

Module-1 DSA

Data Structure:

- Arrangement of data in computer's memory.
- Structural rep. of logical relationships between elements of data.
- Way of representing data in computer memory.

Need of Data Structure:

- Data structure is a way of organizing all data items with their relationship to each other.

Areas
- DBMS
- Compiler Design
- Network Analysis
- Numerical Analysis
- A I.
- OS
- Graphics.

Types.

Data Structure.
- Logical
  - Linear
    - Eg. List
      - Linear
        - ↳ Queue? Array, Stack
      - Linked.
        - ↳ one way., Two way, Circular
  - Non-Linear.
    - ↳ Tree
    - ↳ Graph.
- Physical
  - ↳ int
  - ↳ char
  - ↳ float.

Teacher's Signature

Algorithm:

Analysis of Algo: (TCYPW

- Correctness
- Implementation
- Simplicity.
- execution time & memory req.

Types of Analysis

↳ Best

↳ Worst

↳ Average.

ADT - An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them and the types of parameters of the operations.

It specify what each operation does, but not how it does. Typically an ADT can be implemented using one of many diff. data structures.

Steps to Create ADT -
→ understand and clarify the nature of the target info" unit.
- identify and determine which data objects and operations to include in models.
- Express this property somewhat formally, so that it can be understood & communicate well
- Translate this formal specification into proper lang

Data Structure: ADT and its Implementation -

* A data structure and its operations can be packaged together into an entity called an ADT."

When we use abstract data types, our Prog. divides into two parts:
→ Application part
→ Implementation part

App part - describe use of ADT.
Inple - implement abstract data type.

"It is specification of logical and mathematics properties of a data type or data structure.

Ex –

The 'int' data type, available in 'C' prog lang. provides an implementation of mathematical concept of integer number. The int data type in 'C' can be considered as an implementation of ADT INTEGER-ADT.

INTEGER-ADT. defines set of numbers given by the union of the set (-1, -2 .. ∞) and set of whole number. (0, 1, 2 ... +∞).

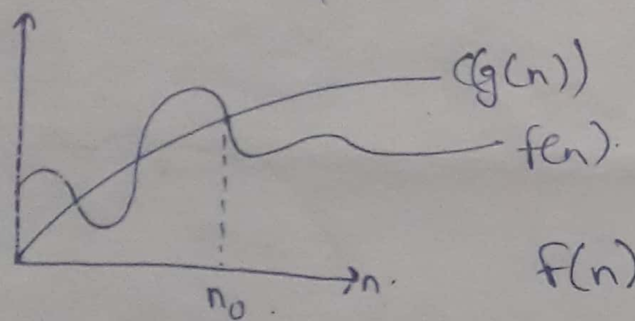
Asymptotic notations

- They are used to describe the time complexity in numerical form.
- They are lang. that allow us to analyze an algo's running time by identifying its behaviour as the i/p size for the algo increase. (growth rate)

1. Big Oh(O) Notation -

→ Method of expressing upper bound of an algo's running time

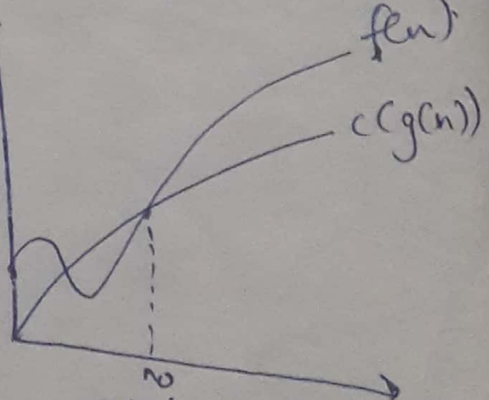
→ It is measure of longest amount of time it could possibly take for the algorithm to complete.



It takes linear time in best case and quadratic time in worst case.

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

2. Omega (Ω) Notations :-



$$f(n) \geq cg(n)$$

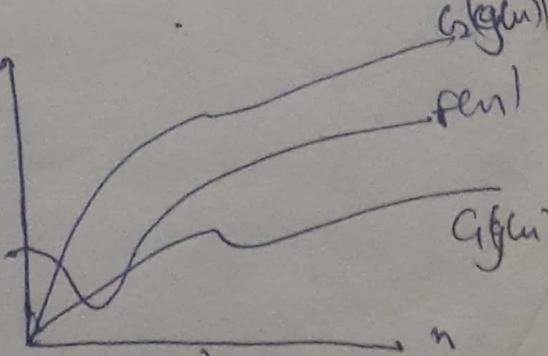
→ Method of expressing lower bound of an algorithm's running time.

→ measure of lowest amount of time ~~taken~~ it could possibly take for an algo to complete

$$f(n) = \Omega(g(n))$$

3. Theta (Θ) Notations:

n lies b/w $C_1(g(n))$ and $C_2(g(n))$



$$f(n) = \Theta(g(n))$$

Scanned by CamScanner

Complexity: The Complexity of an algo is a function describing the efficiency of the algorithm in terms of the amount of data the algo must process.

There are two main complexity measures : time and space.

Time Complexity: It is a function describing the amount of time an algo takes in terms of the amount of i/p to the algo. "Time" can mean the number of memory accesses performed, no of comparison b/w integers, no. of times some inner loop executed.

Space Complexity: It is a f^n describing the amount of memory an algo takes in terms of the amount of i/p to the algo

"Algo complexity is a rough approximation of the number of steps, which will be executed depending on the size of the i/p data. Complexity gives the order of steps count, not their exact count".

Time Complexity

→ Comparison of diff. time complexities:

(1.) → Constant time - O(1).

→ def. constant (n)

result = n*n

return result

There is always a fixed number of operations.

2 Logarithmic time - O(log n)

```
def lograrithmic (n):
    result = 0
    while n > 1:
        n //= 2
        result += 1
    return result.
```

3 Linear Time - O(n)

```
def linear (n, A):
for i in xrange (n):
    if A[i] == 0
        return 0
```

If the first value of array A is 0, then the program will end immediately.

4. Quadratic time - O(n²)

```
def quadratic (n):
    result = 0
    for i in xrange (n)
    for j in xrange (i, n)
        result += 1
    return result.
```

Linear Time - O(n+m)

```
def linear2 (n, m)
result = 0
for i in xrange (n):
    result += i
for j in xrange (m)
    result += j
    return result.
```

Space and Time Complexity:

Space Complexity: To calculate space complexity:

$$S(p) = C + Sp(n).$$

C - Constant.

n - instance characteristics.

Ex - f^n to compute the sum of elements of an array having N elements

```
sum(ARR, N)
{
    S = 0;
    for I = 0 to N-1 by 1
        set S = S + ARR[I]
    returns;
}
```

→ n+3

Time Complexity: Time taken by algo or running time algo depends on following factors -

- → machine is single or multi processor
- → Read/write speed to memory.
- → 32 bit / 64 bit architecture.
- → Configuration
- → Input given to Prog.

→ Rate of growth of time taken w.r.t input

→ one unit of time is taken for
- → operations
- → Assignment / return
- →

Ex - sum(a, b)
```
{
    return a+b;   → 1 unit
}
```
1 unit

T_{sum} = 2 units.
(Const)

Ex2 sum of all elements in a list

sum of list (A, n)

	Cost	no of line
{		
1 total = 0	→ 1 unit (executed once)	1
2 for i = 0 to n-1	2 unit	n+1
3 total = total + Ai	2	n
4 return total	1	1
}.		

$T_{\text{sum of list}} = 1 \times 1 + 2(n+1) + 2 \times n + 1$

$1 + 2n + 2 + 2n + 1$

$T(n) = C(n) + C'$ $\quad \underline{4 + 4n}$: Linear function

$C = C_2 + C_3$

$C' = C_1 + C_2 + C_4$

Ex

for (int i = 0; i < N; i++) → N. (① ; n+1)

{
printf ("Hello");
}.

$1 + n + 1 + n \rightarrow 2n + 2$.

① O(1): Time Complexity of a function is considered as O(1) if it do not contain loop, recursion and call to any other non-constant time function. Ex - swap.

A loop or recursion that runs a const. number of times is also considered as O(1).

Ex - for (int i = 1; i <= c; i++) $\quad$ c = 1, 2, 3, 4 ...

{
=
}.

(2) O(n) - If loop variable is incremented/decremented by a constant amount.

```
for (i=1;i<=n; i+=c)
{
  ≡
}
```

(3) O(n²) - Time complexity of nested loops is equal to the number of times the innermost stmt is executed.

Ex -
```
for (int i=1; i<=n; i+=c)
{
  for (int j=1; j<=n; j+=c)
  {
    ≡
  }
}
```
$O(n^2)$

(4) O(log n) - Time complexity of a loop is considered as O(log n) if the loop variable is divided/multiplied by a constant amount.

Sorting:

1

External sorting: when unsorted and sorted array are stored at diff. locations i.e, primary & in secondary storage.

Iternal sorting: when both array are placed in same memory.

1. Bubble sort:

This technique is easy to implement but not efficient.

Strategy: Each element is compared with adjacent element. If first element is greater than second then interchange their locations otherwise no change. Then next element is compared with its adjacent element & same process is repeated for all elements in array.

Ex. 25, 9, 41, 130, 15

Pass 1: (a) Compare A[0] & A[1] (25>9) yes, interchange.

9 25 41 130 15

(b) A[1] & A[2]   25>41, No change.

(c) A[2] & A[3]   41>130, ".

(d). A[3] & A[4]   130>15. Interchange.

9 25 41 15 130.

same for all passes, until all conditions becomes false.

Sorted Array: 9 15 25 41 130.

Algorithm:

1. for i = 0 to (A.length − 1) or N−1
2. for j = 0 to N−I−1
3. If ARR[J] > ARR[J+1] then
4. Interchange ARR[J] and ARR[J+1] using temp variable.

Analysis:

There are n−1 comparisons during pass 1 to find largest element. for pass 2 → n−2 comparisons. & so on.

$f(n) = (n-1)+(n-2)+(n-i)+ \dots +2+1$

$= \frac{n*(n-1)}{2} = O(n^2)$.

for both avg. & worst case.

Selection Sort: Strategy: Take an element & keep it at its appropriate position. find the smallest element & keep it at first location of array. & Repeat the process for next smallest element.

Ex. 65, 30, 22, 80, 47.

A[0] A[1] [2] [3] [4].

Pass1. select smallest element in list.

· A[0] = Min = 65.

Min > A[1], yes, set Min = 30.

A[1] > A[2] yes set Min = 22.

A[2] > A[3] no change.

A[2] > A[4] No change.

Interchange 65 and 22 i.e, A[0]=22, A[2]=65.

22 30 65 80 47.

In Pass 2 No element is less than 30, so list will be unchanged.

Pass 3. Now min = 65.

(a) Min > A[3] 65>80 no change.
(b) Min > A[4] 65->47 Yes., Min = 47

Interchange 65 & 47.

22 30 47 80 65.

Pass 4. Min = 80.

Min > A[4] Yes. Min = 65.

Interchange. 80 & 65.

22 30 47 65 80.

Algorithm

1. Read Array.
2. Repeat step 3 to 6 for I=0 to N-1
3. Set min = Arr[i] & set Loc = I
4. Repeat step 5 for J = i+1 to n-1
5. If min > Arr[j], then
   (a) set min = Arr[j]
   (b) Set Loc = j
   (end of if)
   (end of step 4 loop)
6. Interchange Arr[i] & Arr(loc) Using temp variable.
7. end of step 2 outer loop

Analysis

In this sorting there are n-1 comparisons during pass 1 Next for Pass 2 again n-2 Comparisons are done.

Similarly →

$f(n) = (n-1)+(n-2)+\cdots(n-i) + \cdots 2+1$

It is form of A.P (Arithmetic Progression).

Sum = $n/2\,[2a+(n-1)d]$

$n = n-1,\ a=1,\ d=1$

$O(n^2)$.

Insertion Sorting: Strategy: In Pass 1 we suppose elements are already sorted. In next pass second element A[1] is compared with first one & placed at its appropriate location, similarly the process is repeated.

Ex. 7 3 4 1 8
A[0] A[1] [2] [3] [4].

Pass 1. A[0] is already sorted.

Pass 2. A[1] = 3, is compared with first element.

(a) Comp. A[1] < A[0]. 3 < 7 Yes, interchange.

3 7 4 1 8

Pass 3. A[2] i.e. 4 is compared with both 3 & 7.
A[2] = 4.,

(a) Compare A[2] < A[1], 4 < 7, A[2] = 7.
(b) Compare. A[2] < A[0], 4 < 3, no change.

3 4 7 1 8.
A - (0) (1) (2) (3) (4)

Pass 4. A[3] = 1, compared with - 3, 4 & 7.

(a) 1 < 7, ~~interchange~~ Yes A[3] = 7.
(b) 1 < 4 ~~interchange~~ Yes A[2] = 4
(c) 1 < 3 ~~interchange~~ Yes A[1] = 3.

A[0] = 1.

1 3 4 7 8.

Pass 5: In this pass all conditions become false. So no change.

In this sorting, we have 5 elements & no. of passes are also 5. i.e, for n elements there are n passes.

Algorithm

1. for j = 2 to A.length // i.e. n
2. key = A[j]
3. // Insert A[j] into sorted seq.
4. i = j - 1
5. while i > 0 and A[i] > key.
6. A [i+1] = A[i]
7. i = i-1
8. A[i+1] = key.

Cost	Times
c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j-1)$
c_7	$\sum_{j=2}^{n} (t_j-1)$
c_8	$n-1$.

Analysis

There is 1 comparison during pass 1.
2 comparisons for pass 2.
3 for pass 3 & so on.

$$f(n) = 1+2+3+\ldots+(n-1)$$
$$\Rightarrow \frac{n(n-1)}{2}.$$

1. Worst case: when elements are in reverse order & inner loop must use the max. no. (n-1) comparisons. then:

$$\text{Complexity} = \frac{n(n-1)}{2} = O(n^2)$$

2. Average case: when there are $\frac{(n-1)}{2}$ Comparisons. in inner loop. for average case:

$$f(n) = \frac{1}{2} + \frac{2}{2} + \ldots + \frac{(n-1)}{2}$$
$$\Rightarrow \frac{n(n-1)}{4} \Rightarrow O(n^2).$$

13.8 MERGE SORT

Merge sort is a sorting algorithm that uses the idea of divide and conquer approach.

```
MERGE_SORT (A, p, r)
    1. if p < r
    2.       then q ← ⌊(p + r)/2⌋
    3.             MERGE-SORT (A, p, q)
    4.             MERGE-SORT (A, q + 1, r)
    5.             MERGE (A, p, q, r)
```

```
MERGE (A, p, q, r)
n1 ← q - p + 1
n2 ← r - q
create arrays L[1..n1+1] and R[1..n2+1]
for i ← 1 to n1
     do L[i] ← A[p+i-1]
for j ← 1 to n2
     do R[j] ← A[q+j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
                j ← j + 1
```

The procedure MERGE-SORT (A, p, r) sorts the elements in the sub-array A[p..r]. If $p \le r$, the sub array has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions A[p..r] into two sub-arrays : A[p..q], containing [n/2] elements, and A[q+1..r], containing [n/2] elements.

To sort the entire sequence A = (A[1], A[2], ... A[n]) we call MERGE-SORT (A, 1, length[A]) where once again *length*[A] = n. If we look at the operation of the procedure bottom-up when n is a power of two, the algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length n/2 are merged to form the final sorted sequence of length n.

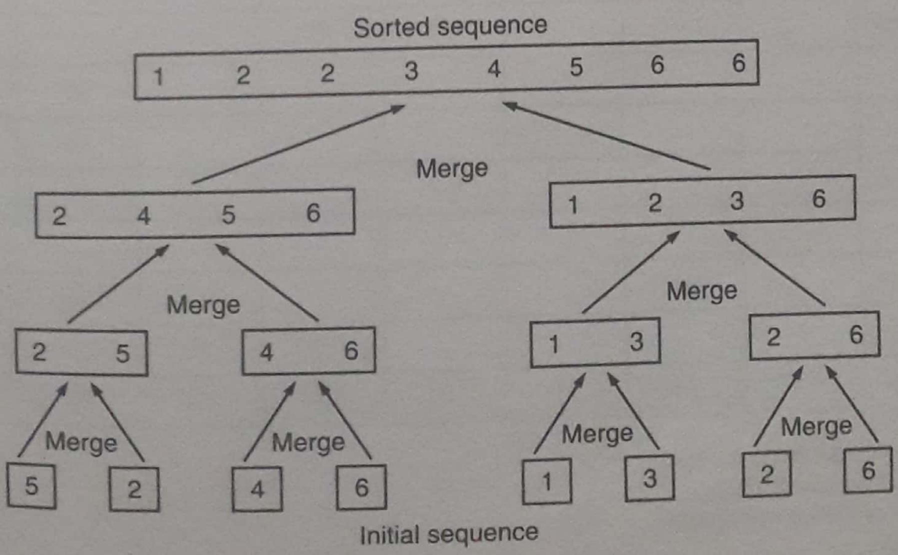


Fig. 13.1. Initial sequences.

Solution: Suppose A[ ] :

1	2	3	4	5	6	7	8
15	10	5	20	25	30	40	35
p			q				r

Here $\quad p = 1$
$\quad r = 8$

$1 < 8$ then $q = \lfloor(1+8)/2\rfloor = \lfloor 4.5\rfloor = 4$

$\quad q = 4$

$p + r/2 \rightarrow \quad \frac{1+8}{2} \rightarrow 4.5 \quad q = 4$

$p = 1, r = 8, q = 4$

i.e.,

Call $\boxed{\text{MERGE-SORT (A, 1, 4)}}$
MERGE-SORT (A, 5, 8)
MERGE (A, 1, 4, 8)

First call **MERGE-SORT (A, 1, 4)**

Here $\quad p = 1$
$\quad r = 4$

$1 < 4$, so $q = \left\lfloor \frac{1+4}{2} \right\rfloor = \lfloor 2.5\rfloor = 2$

$\therefore \quad q = 2$

$p = 1, r = 4 \quad \frac{1+4}{2} \rightarrow 2 = q$

then call $\boxed{\text{MERGE-SORT (A, 1, 2)}}$
MERGE-SORT (A, 3, 4)
MERGE (A, 1, 2, 4)

Now call **MERGE-SORT (A, 1, 2)** firstly

Here $\quad p = 1$
$\quad r = 2$

$1 < 2$, then $q = \left\lfloor \frac{1+2}{2} \right\rfloor = 1$

$p = 1 \quad q = 2 \rightarrow \quad \frac{2+1}{2} \quad q = 1$

So, call $\boxed{\text{MERGE-SORT (A, 1, 1)}}$
MERGE-SORT (A, 2, 2)
MERGE (A, 1, 1, 2)

Call MERGE-SORT (A, 1, 1)

Here $1 < 1$ No change

then call MERGE-SORT (A, 2, 2)

Here $2 < 1$ so, no change

then call MERGE-SORT (A, 1, 1, 2)

Here $\quad p = 1, q = 1, \ r = 2$

So, $\quad n_1 = q - p + 1 = 1 - 1 + 1$
$n_1 = 1$
$n_2 = 2 - 1$
$n_2 = 1$

Scanned by CamScanner

Create arrays L[1..2] and R[1..2]

A[ ]	15	10

	1
L[ ]	15

	1
R[ ]	10

For $i = 1$ to 1

$L[1] = A[1 + 1 - 1]$

$L[1] = A[1]$

For $j = 1$ to 1

$R[1] = A[1 + 1]$

$R[1] = A[2]$

Now $L[2] = \infty$ and $i = 1$

$R[2] = \infty$ $j = 1$

For $k = 1$ to 2

$k = 1$

if $L[1] \leq R[1]$

$15 \leq 10$ (False)

So, $A[1] = R[1]$ *i.e.*, $A[1] = 10$

and $j = 1 + 1 = 2$

Now $k = 2$ and $j = 2$

$L[1] \leq R[2]$

$15 \leq \infty$ (True)

So $i = i + 1 = 2$ and $A[2] = 15$

Now A[ ] :

	1	2	
A[ ] :	10	15	

Now MERGE-SORT (A, 1, 2) is completed.

then call $\boxed{\text{MERGE-SORT (A, 3, 4)}}$

Here $p = 3$

$r = 4$

$$3 < 4, \text{ then } q = \left\lfloor \frac{p+r}{2} \right\rfloor = \left\lfloor \frac{3+4}{2} \right\rfloor = 3$$

Call
- MERGE-SORT (A, 3, 3) (No change)
- MERGE-SORT (A, 4, 4) (No change)
- MERGE-SORT (A, 3, 3, 4)

So, call **MERGE (A, 3, 3, 4)**

Here $p = 3$

$q = 3$

$r = 4$

So, $n_1 = 1$

$n_2 = 1$

So create array L and R

For $i = 1$ to 1

$L[1] = A[3 + 1 - 1]$

$L[1] = A[3]$ i.e $L[1] = 5$

For $j = 1$ to 1

$R[1] = A[3 + 1]$

i.e., $R[1] = 20$

and $L[2] = \infty$

and $R[2] = \infty$

Now for $i = 1$ and $j = 1$

$k = 3$ to 4

$k = 3, \quad i = 1, \quad j = 1$

$L[1] = 5$ Here $5 \le 20$ (True)

$R[1] = 20$ then $A[3] = 5$ and $i = i + 1 = 2.$

$k = 4, \quad i = 2, \quad j = 1$

$L[2] = \infty$

$R[1] = 20$

$L[2] \nless R[1]$

So $A[4] = 20$

i.e., Now

1	2	3	4	
10	15	5	20	

Now call **MERGE (A, 1, 2, 4)**

Here $p = 1$

$q = 2$

$r = 4$

So, $n_1 = 2 - 1 + 1$

$n_1 = 2$ i.e. $n_1 = 2$

$n_2 = 2$

$n_2 = 4 - 2 = 2$

Create array L[1...3] and R[1...3]

For $i = 1$ to 2

$i = 1 \quad L[1] = A[1 + 1 - 1]$

$L[1] = A[1]$

i.e., $L[1] = 10$

For $i = 2 \quad L[2] = A[1 + 2 - 1]$

$= A[2]$

$L[2] = 15$

i.e.,

L[ ] =

1	2	3
10	15	∞

For $j = 1$ to 2

$R[1] = A[2 + 1] = 5$

$R[2] = A[2 + 2] = 20$

i.e., R[ ] =

1	2	3
5	20	∞

Now $i = 1$ and $j = 1$

For $k = 1$ to 4

$k = 1$ $L[1] \le R[1]$ *i.e.*, $10 \le 5$ (False)

So $A[1] = R[1]$ *i.e.*, $A[1] = 5$ and $j = 1 + 1 = 2$

$k = 2$ $L[1] \le R[2]$ *i.e.*, $10 \le 20$ (True)

So $A[2] = L[1]$ *i.e.*, $A[2] = 10$ and $i = 1 + 1 = 2$

$k = 3$ $L[2] \le R[2]$ *i.e.*, $15 \le 20$ (True)

So $A[3] = L[2]$ *i.e.*, $A[3] = 15$ and $i = 2 + 1 = 3$

$k = 4$ $L[3] \le R[2]$ *i.e.*, $\infty \le 20$ (False)

So $A[4] = R[2]$ *i.e.*,. $A[4] = 20$ and $j = 2 + 1 = 3$

i.e., Now array is

A[ ]

1	2	3	4	
5	10	15	20	

Now call **MERGE-SORT (A, 5, 8)**

Here $p = 5$

$r = 8$

$$5 < 8, \text{ so } q = \left\lfloor \frac{5+8}{2} \right\rfloor = 6 \quad \text{i.e.,} \quad q = 6.$$

then call

MERGE-SORT (A, 5, 6)
MERGE-SORT (A, 7, 8)
MERGE-SORT (A, 5, 6, 8)

First we call **MERGE-SORT (A, 5, 6)**

Here $p = 5$

$r = 6$

$$5 < 6, \text{ then } q = \left\lfloor \frac{5+6}{2} \right\rfloor = 5$$

then call

MERGE-SORT (A, 5, 5)
MERGE-SORT (A, 6, 6)
MERGE (A, 5, 5, 6)

Call **MERGE-SORT (A, 5, 5)**

Here $5 < 5$ (False) So no change.

then call **MERGE-SORT (A, 6, 6)**

Here $6 < 6$ (False) So no change.

Call **MERGE (A, 5, 5, 6)**

Here $p = 5 \quad q = 5 \quad r = 6.$

$n_1 = 1$

$n_2 = 1$

Create arrays L[1..2] and R[1..2]

L[1] = A[5] *i.e.,* L[1] = 25

R[1] = A[6] *i.e.,* R[1] = 30

L[2] = ∞ *i.e.,* L[ ]

1	2
25	∞

R[2] = ∞ R[ ]

1	2
30	∞

i = 1 and j = 1

k = 5 to 6

For

k = 5 L[1] ≤ R[1] i.e. 25 ≤ 30 (True)

So A[5] = 25 and i = 1 + 1 = 2.

k = 6 L[2] ≤ R[1] i.e. ∞ ≤ 30 (False)

So A[6] = 30 and j = j + 1 = 2.

Thus now array is

A[ ] =

1	2	3	4	5	6	
5	10	15	20	25	30	

Now call **MERGE-SORT (A, 7, 8)**

We get

A[ ] =

1	2	3	4	5	6	7	8
5	10	15	20	25	30	35	40

We call **MERGE (A, 5, 6, 8)**

We get the array

A[ ] =

1	2	3	4	5	6	7	8	
5	10	15	20	25	30	35	40	

Now call **MERGE (A, 1, 4, 8)**

We get the sorted array as:

A[ ] =

1	2	3	4	5	6	7	8
5	10	15	20	25	30	35	40

This is final sorted array.