Merge Sort

→ Merge Sort is a sorting algorithm the use the divide, conquer and combine algorithmic paradigm.

→ Divide → means → partitioning the n-elements array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted.
However if there are more elements in the array, divide A into two sub-arrays A_1 and A_2, each containing about half of the elements of A.

→ Conquer → means Sorting the two sub-arrays recursively using MERGE SORT.

→ Combine → means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge Sort algo. focuses on two main concept to improve its Performance (running time).
→ smaller lists takes fewer steps and thus less time to sort than a large list.
→ ↓ no. of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The Basic steps of a Merge Sort algo. are as follows →
→ If the array is of length 0 or 1, then it is already sorted.
→ Otherwise, divide the unsorted array into two sub-arrays of about half the size.
→ Use merge sort algorithm recursively to sort each sub-array.
→ Merge the two sub-arrays to form a single sorted list.

The merge sort algorithm uses a function merge which combines the sub-arrays to form a sorted array. While the merge sort algo. recursively divides the lists into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one lists.

```
MERGE-SORT(ARR, BEG, END)

Step1: IF BEG< END
            SET MID = (BEG+END)/2
            CALL MERGE-SORT (ARR, BEG, MID)
            CALL MERGE-SORT ( ARR, MID+1, END)
           ·MERGE ( ARR, BEG, MID, END)
        [END OF IF]

Step 2: END
```

COMPLEXITY (Running Time).

The running time of merge sort in the average case and the worst case can be given as $O(n \log n)$. Although merge sort has an Optimal Time complexity, it needs an additional space of $O(n)$ for the temporary array TEMP.

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J <= END)

IF ARR[I] < ARR[J]

SET TEMP[INDEX] = ARR[I]

SET I = I + 1

ELSE

SET TEMP[INDEX] = ARR[J]

SET J = J + 1

[END OF IF]

SET INDEX = INDEX + 1

[END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

IF I > MID

Repeat while J <= END

SET TEMP[INDEX] = ARR[J]

SET INDEX = INDEX + 1

SET J = J + 1

[END OF LOOP]

[Copy the remaining elements of left sub-array, if any]

ELSE

Repeat while I <= MID

SET TEMP[INDEX] = ARR[I]

SET INDEX = INDEX + 1

SET I = I + 1

[END OF LOOP]

[END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K = 0

Step 5: Repeat while K < INDEX

SET ARR[K] = TEMP[K]

SET K = K + 1

[END OF LOOP]

Step 6: END

9	39	45	81	18	27	72	90
BEG I			MID	J			END

Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX & subsequently the value I or J is INCREMENTED.

~~TEMP~~

9	39	45	81	18	27	72	90
~~INDEX~~ BEG	I		MID	J			END

TEMP

9	18	
	INDEX	

9	39	45	81	18	27	72	90
BEG	I		MID		J		END

9	18	27	
		INDEX	

9	39	45	81	18	27	72	90
BEG	I		MID			J	END

9	18	27	39
			INDEX

9	39	45	81	18	27	72	90
BEG		I	MID			J	END

9	18	27	39	45
				INDEX

9	39	45	81	18	27	72	90
BEG			I MID			J	END

9	18	27	39	45	72		
					INDEX		

9	39	45	81	18	27	72	90
BEG			I MID				J END

9	18	27	39	45	72	81	
						INDEX	

When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.

9	39	45	81	18	27	72	90
BEG			MID	I			END

9	18	27	39	45	72	81	90
							INDEX.

SORTING BY MERGING

→ A class of sorting algo. based on the principle of 'merging' or 'collating' is discussed in this section.

→ Suppose A and B are two lists with n_1 and n_2 elements, respectively. Also assume that both the lists are arranged in ASCENDING ORDER.

→ Merging is an operation that combines the elements of A and B into another list C with $n_1 + n_2 = n$ elements in it and elements is C are also in ASCENDING ORDER.

→ Merging operation is called Sorting by merging & is important for the following two reasons:→

① Priniple is easily amenable to divide and conquer techniques.

② Suitable for sorting very large list, even the entire list is not Necessary to be residing in the main memory.

SIMPLE MERGING

ALGORITHM SIMPLE MERGE_Iterative.

Input: Two list A and B of size n_1 and n_2 respectively.
Elements in both the list A and B are sorted in ascending order.

Output: An output list with n elements stored in ascending order.

Steps:-

```
1.  i=1 ,j=1, k=1          // Three pointers, initially point to first locations.
2.  While (i≤n1 and j≤n2) do
3.      If (A[i] ≤ B[j]) then
4.          C[K]=A[i]
5.          i=i+1 , k=k+1
6.    Else              // A[i]>B[j]
7.          C[K] = B[j]
8.          j=j+1 , k=k+1
9.    EndIf
10. EndWhile
11. If ( i>n1) then  ——→ // A is fully covered.
12.     For p=j to n2 do ——→ // Move rest of the elements in B to C.
13.         C[K] = B[p]
14.         K=K+1
15.     EndFor
16. Else
17.   If (j>n2) then ——→ // B is fully covered.
18.     For p=i to n1 do
19.         C[K]=A[p] ——→ // Move the rest of the elements in A to C.
20.         K=K+1
21.     Endfor
22.   EndIf
23. EndIf
24. Stop.
```

ALGORITHM SIMPLEMERGE_Recursive

Steps:

```
1.  If (i > n1) then
2.      For p=j to n2 do
3.          C[K] = B[p]
4.          K = K+1
5.      EndFor
6.      Return
7.  EndIf
8.  If (j > n2) then
9.      For p=i to n1 do
10.         C[K] = A[p]
11.         K = K+1
12.     EndFor
13.     Return
14. EndIf
15. If (A[i] ≤ B[j]) then
16.     C[K] = A[i]
17.     K = K+1 , i = i+1
18. Else
19.     C[K] = B[j]
20.     K = K+1 , j = j+1
21. EndIf
22. SimpleMerge_Recursive(A, B, i, j)
23. Stop.
```

ANALYSIS OF THE SIMPLE MERGE OPERATION

Time Complexity

Case 1: $n_1 = 1$ and $n_2 = n-1$ and A contains the smallest element
In this case we require only one comparison.

Case 2: $n_2 = 1$ and $n_1 = n-1$ and B contains the smallest element
In this case we require only one comparisons.

~~Case~~ Best case with only one comparison.

A | 1 |

B | 2 | 3 | ...... | 49 | 50 |

→ | 1 | 2 | 3 | ..... | 49 | 50 | C

Case (1)

A | 2 | 3 | .... | 49 | 50 |

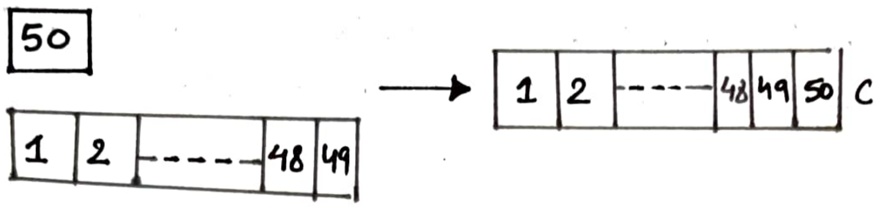
B | 1 |

→ | 1 | 2 | 3 | ... | 49 | 50 |

Case (2)

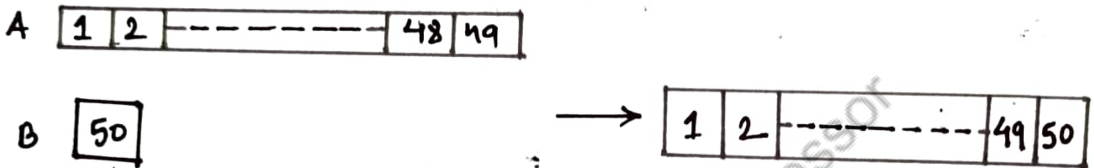
Case 3: $n_1 = 1$ and $n_2 = n-1$ and A contains the largest element.

In this case, we require $n-1$ comparisons.



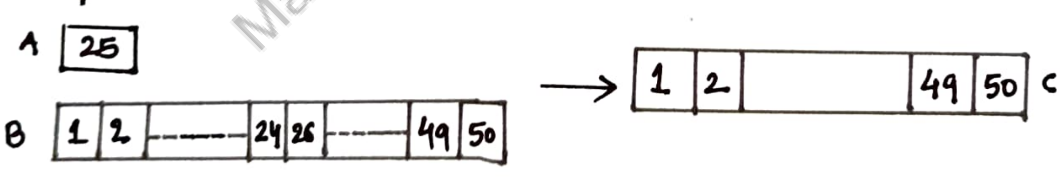
Case 4: $n_2 = 1$ and $n_1 = n-1$ and B contains the largest element

In this case we require $n-1$ comparisons.



Case 5: Either A or B contains a single element, which is neither smallest nor largest

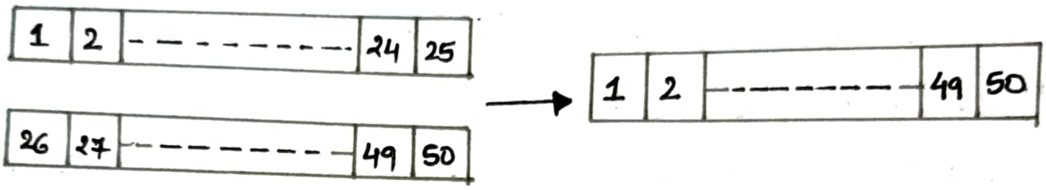
In this case, the number of comparison is decided by the single element itself. With out the loss of generality, we can say that on the average, $n/2$ no. of comparison required



Avg case with $n/2$ comparison

Case 6. The largest element in A is smaller than the smallest element in B and vice verso.

In this case, the no. of comparisons in Best case is $\min(n_1, n_2)$ that is, the min of n_1 and n_2.



(d) Another Average case with $n/2$ comparisons.

Case 7. Otherwise, that is, any trivial ordering

In this case, we need as many as $n-1$ comparisons.

TIME COMPLEXITY

Best Case : $T(n) = \min(n_1, n_2)$

$= 1$ if $n_1 = 1$ or $n_2 = 1$

Worst Case : $T(n) = n-1$

Average Case : $T(n) = n/2$

We seen that in the best and worst case, the merge operation require 1 and $(n-1)$ comparisons. These two cases are corresponding to the merging of two lists, one which is smallest and one is the largest.

Now, in a case when two input lists are an arbitrary sizes, say $n_1 < n$ and $n_2 < n$, the number of comparisons is minimum of n_1 and n_2, which can be taken on an average $n/2$ ~~elements~~ comparisons.

There is also another WORST-CASE situation with $n-1$ comparisons, where the last elements in the two lists are the largests and next to the largest.

→ Space Complexity

The output list is stored in a separate storage space c and size of this should be $n = (n_1 + n_2)$. Hence, the storage space complexity of merging n elements is

$$\boxed{S(n) = n_1 + n_2 = n \text{ (say)}}$$

MERGESORT

→ All sorting method based on merging can be divided into two broad categories:

- → INTERNAL MERGE SORT
- → EXTERNAL MERGE SORT
  - ↳ deals with very large lists of elements such that size of memory (primary) not adequate to accomodate the entire list.

→ **Internal Merge Sort**

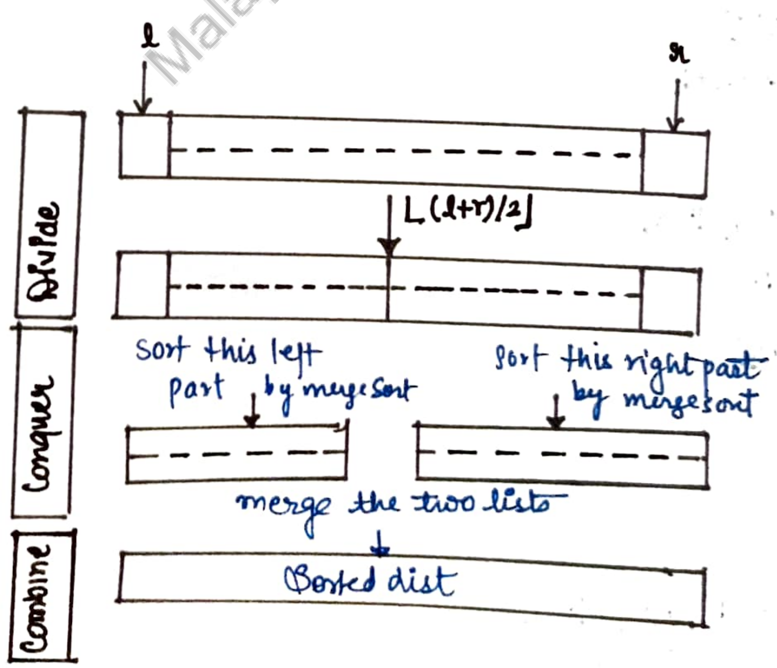
→ The lists under sorting are small and assumed to be stored in the high speed primary memory.

→ closely follow the DIVIDE-AND-CONQUER paradigm.
Let the list of n elements to be sorted with l and r being the position of leftmost and rightmost element on the lists.

→ Divide → Partition the list midway, that is, at $\left\lfloor \frac{l+r}{2} \right\rfloor$ into sublists with $\frac{n}{2}$ elements in each, if n is even $= \left\lceil \frac{n}{2} \right\rceil$ and $\left\lceil \frac{n}{2} \right\rceil - 1$ elements if n is odd.

→ Conquer → Sort the two list recursively using the merge sort.

→ Merge → Merge the sorted sublist to obtain the sorted output.



→ **ALGORITHM MERGESORT**

Steps:

1. If $(r \leq l)$ then
2. Return ———— // Termination of Recursion
3. Else
4. $mid = \lfloor \frac{l+r}{2} \rfloor$ ———— // Divide : Find the index of the middle of the list
5. MergeSort (A[l.....mid]) — // conquer for the left part
6. MergeSort (A[mid+1...r]) — // Conquer for the right part
7. Merge (A, l, mid, r) — // Combine : Merging the sorted left and right part.
8. EndIf
9. Stop

→ **ALGORITHM MERGE**

Steps:

1. $i=l$, $j=mid+1$, $K=1$
2. While $((i \leq mid)$ and $(j \leq r))$ do
3. If $(A[i] \leq A[j])$ then
4. $C[K] = A[i]$
5. $i=i+1$, $K=K+1$
6. Else
7. $C[K] = A[j]$
8. $j=j+1$, $K=K+1$.
9. EndIf
10. EndWhile
11. If $(i>mid)$ and $(j \leq r)$ then
12. For $m=j$ to r do
13. $C[K] = A[m]$
14. $K=K+1$
15. Endfor
16. Else
17. if $(i<mid)$ and $(j>r)$ then
18. For $m=i$ to mid do

```
        C[K] = A[m]
        K = K+1
      Endfor
   EndIf
EndIf
For m=1 to K-1 do
   A[m] = C[m]
     m = m+1
End for
Stop.
```

* float a = 0.5

if a == 0.5

R

else 2

a=4.28

ANALYSIS OF THE MERGESORT

Time Complexity

- Basic operation in the Merge Sort is KEY COMPARISON.
- It is evident that merge sort on just one element requires only one comparisons. → Let C_1 be the time for this.
- When $n>1$, we break down the RUNNING TIME for the THREE TASKS involved in the merge sort procedure.
- Let $T(n)$ denote the time to sort n elements. We can express $T(n)$ as below :→

$$
\boxed{\begin{aligned} T(n) &= C_1 && \text{if } n=1 \\ &= D(n) + T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + C(n) && \text{if } n>1 \end{aligned}}
$$

step 1. $D(n)$ denotes the time for divide task. The divide step just computes the middle of the sub-array, which takes constant time. Thus

$D(n) = C_2$

Step 2. Each divide steps yields two subarrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Since we "RECURSIVELY" solve the problem, the running time for this task is $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$.

Step 3. Combine where the algorithm Mergesort merges two lists to an output list of size n.
Let $C(n)$ denote the time to merge 2 lists of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ so that the output list is of size n.

In order to simplify the calculation

1. No. of elements in the list is a power of 2, let $n = 2^k$ (with this assumption, therefore each divide step yields two subsequences of exactly $n/2$). Thus,

$$\boxed{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) = 2T(n/2)}$$

2. We consider the merge operation whose worst case TIME COMPLEXITY is $C(n) = n-1$. With these assumptions, we express the recurrence relation

$$T(n) = c_1 \quad \text{if } n=1$$
$$= c_2 + 2T\left(\frac{n}{2}\right) + (n-1) \quad \text{if } n>1.$$

Let us solve the Recurrence Relation

(1). No. of elements in the list is a power of 2, let $n = 2^K$ (with this assumption, therefore, each divide step yields two sub sequences of size exactly $n/2$).

Thus $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) = T(n/2)$

(2) We consider the merge operation whose worst case time complexity is $C(n) = n-1$.

With these assumptions, we express the Recurrence Relation

$$T(n) = C_1 \quad \text{if } n=1 \quad \text{———} \; (4)$$

$$= C_2 + 2T\left(\frac{n}{2}\right) + (n-1) \quad \text{if } n>1. \; \text{—} \; (5)$$

Let us solve the Recurrence Relation as Below →

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1) + C_2$$

$$= 2\left[2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}-1\right) + C_2\right] + (n-1) + C_2 \quad \left[\text{Expanding } T(n/2)\right]$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + (n-2) + (n-1) + 2 \cdot C_2 + C_2 \quad \left[\text{After the first expansion}\right]$$

$$= 2^2\left[2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}-1\right) + C_2\right] + (n-2) + (n-1) + 2 \cdot C_2 + C_2 \quad \left[\text{Expanding } T\left(\frac{n}{2^2}\right)\right]$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + (n-4) + (n-2) + (n-1) + 2^2 \cdot C_2 + 2 \cdot C_2 + C_2$$

......

$$= 2^K T\left(\frac{n}{2^K}\right) + (n-2^{K-1}) + (n-2^{K-2}) + \cdots + (n-2) + (n-1)$$
$$+ C_2\left[2^{K-1} + 2^{K-2} + \cdots + 2^0\right]$$

[After the (K-1) expansions]

$$= 2^K T(1) + K \cdot n - \sum_{i=1}^{K} 2^{K-i} + c_2 \sum_{i=1}^{K} 2^{K-i}$$

$$= 2^K T(1) + K \cdot n + (c_2 - 1) \sum_{i=1}^{K} 2^{K-i}$$

$$= 2^K T(1) + K \cdot n + (c_2 - 1)(2^K - 1)$$

Since $n = 2^K$ and $T(1) = c_1$, finally, we get

$$\boxed{T(n) = n \cdot c_1 + n \log_2 n + (c_2 - 1)(n - 1)}$$

gives the time complexity of the Algorithm in Worst Case.

Best case Time Complexity

The Best case occurs when the list is almost sorted in order. In such a situation, the merge operation requires $n/2$ operation and Time Complexity can be obtained as:

$$\boxed{T(n) \approx O(n \log_2 n) + \Theta(n)}$$

Average Case Complexity

$$\boxed{T(n) \approx \Theta(n \log_2 n)}$$

Space Complexity

Algorithm uses the merge operation, which stores the output list into an auxiliary storage space.

The storage space in Merge Operation is $= n$ (when we merge two sublists of size $n/2$ elements in each).

Hence we need extra storage space n other than the input list itself.

Therefore the storage complexity of element is

$$\boxed{S(n) = n}$$

Write a program to Implement Merge Sort

```
# include <stdio.h>
# include <conio.h>
# define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[], int, int);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array:");
    scanf("%d", &n);
    printf("\n Enter the element of the array:");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    merge_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0; i<n ; i++)
    printf("%d \t", arr[i]);
    getch();
}

void merge(int arr[], int beg, int mid, int end)
{
    int i=beg, j=mid+1, index=beg, temp[size], k;
    while((i<=mid) && (j<=end))
    {
        if(arr[i] < arr[j])
        {
            temp[index] = arr[i];
            i++;
        }
        else
        {
            temp[index] = arr[j];
            j++;
        }
        index++;
    }
```

```
{
        while (j<=end)
        {
            temp[index]=arr[j];
            j++;
            index++;
        }
    }
    else
    {
        while (i<=mid)
        {
            temp[index]=arr[i];
            i++;
            index++;
        }
    }
    for (k=beg ;k<index;k++)
        arr[k]=temp[k];
}

void merge-sort (int arr[], int beg, int end)
{
    int mid;
    if (beg<end)
    {
        mid=(beg+end)/2;
        merge-sort (arr, beg, mid);
        merge-sort (arr, mid+1, end);
        merge (arr, beg, mid, end);
    }
}
```