Merge Sort

- Merge Sort is a divide and Conquer algorithm
- It divides input array in two halves, Calls itself for the two halves and then merge the two Sorted halves.

Divide Divide the n element Seq to be Sorted into 2 Sub Seq of $n/2$ element each.

Conquer Sort the two Sub Seq recursively using merge Sort

Combine Merge the two Sorted Sub Seq to produce the Sorted answer

Merge function

- It is used for merging two halves.
- Merge algo takes 2 Sorted Sub arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$
- It merges them to form a Single Sorted Subarray $A[p \ldots r]$

Merge (A, p, q, r) (∞ is a Sentinel element)

```
n1 = q - p + 1
n2 = r - q
Let L[1 .. n1+1] and R[1 .. n2+1]
be new arrays
for i = 1 to n1
    L[i] = A[p+i-1]
for j = 1 to n2
    R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Bottoms out - When the Seq to be Sorted has length 1, there is no work to be done as it is already Sorted.

Loop invariant

intialization - Prior to first Iteration we have $K = p$ So that Subarray $A[p..K-1]$ is empty, Since $i = j = 1$ both $L[i]$ and $R[j]$ are the Smallest elements of their arrays.

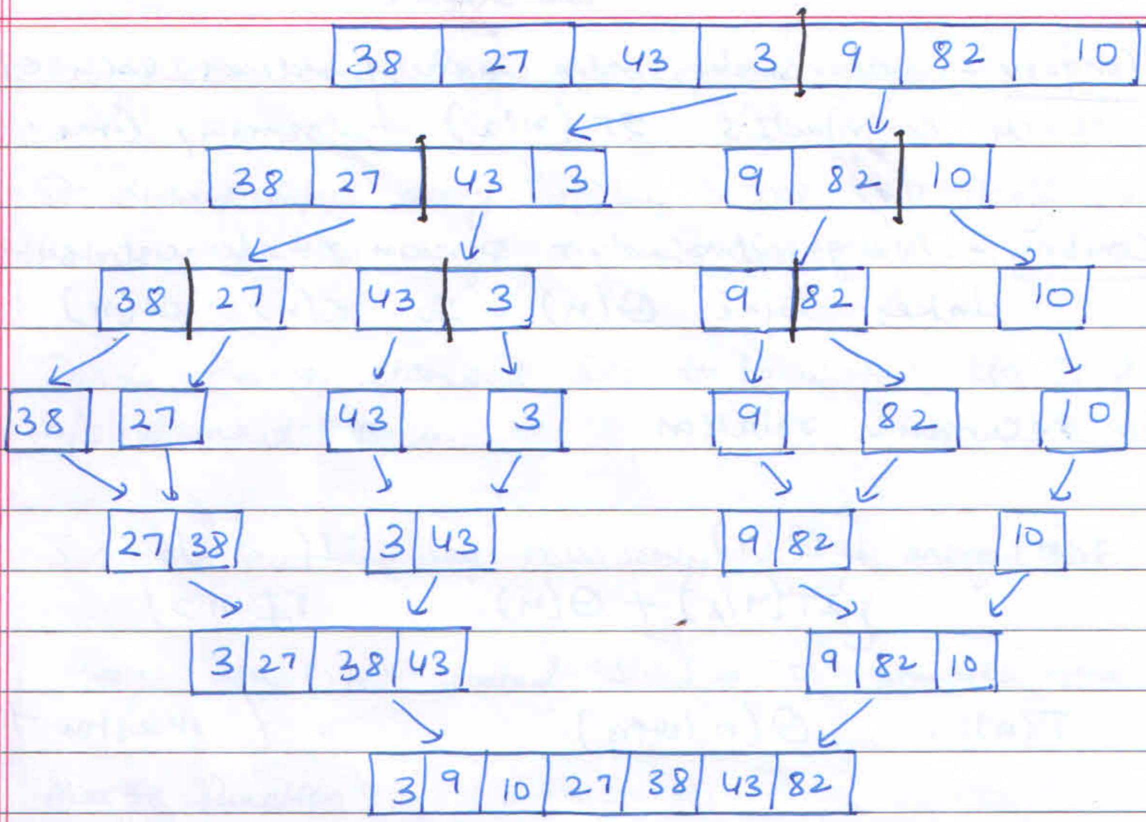
Maintenance - let $L[i] \le R[j]$, then $L[i]$ is the Smallest element not yet copied, $A[p...K-1]$ Contains the $K-p$ Smallest element

Termination - $K = r+1$, Subarray $A[p..K-1]$ which is $A[p..r]$ Contains $K-p$ Smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$ in Sorted order.

- merge procedure take $\Theta(n)$ time where $n = r-p+1$ (total no of elements)

Mergesort (A, p, r)

1. If $p < r$
2. $q = \lfloor (p+r)/2 \rfloor$
3. MergeSort(A, p, q)
4. Mergesort(A, q+1, r)
5. Merge (A, p, q, r)



Recurrence relation (merge sort)

- When an algo contains a recursive call to itself the running time is describe by recurrence equation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \quad (\text{Some Constant } c) \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

$T(n)$ = running time on a problem of size n.
a = no. of subproblem yields during division
b = size of each subproblem (merge sort $a = b$)
$D(n)$ = time to divide the problem into subproblems
$C(n)$ = time to combine the solutions to the subproblems into the solution to the original problem.

- for small size $n \le c$ for some constant c $\Theta(1)$.

Analysis

Divide - compute middle of the subarray, $D(n) = \Theta(1)$

Conquer - recursively solve 2 subproblems each of size $n/2$ which contributes $2T(n/2)$ to running time.

Combine - Merge procedure on an n-element subarray takes time $\Theta(n)$ so $C(n) = \Theta(n)$

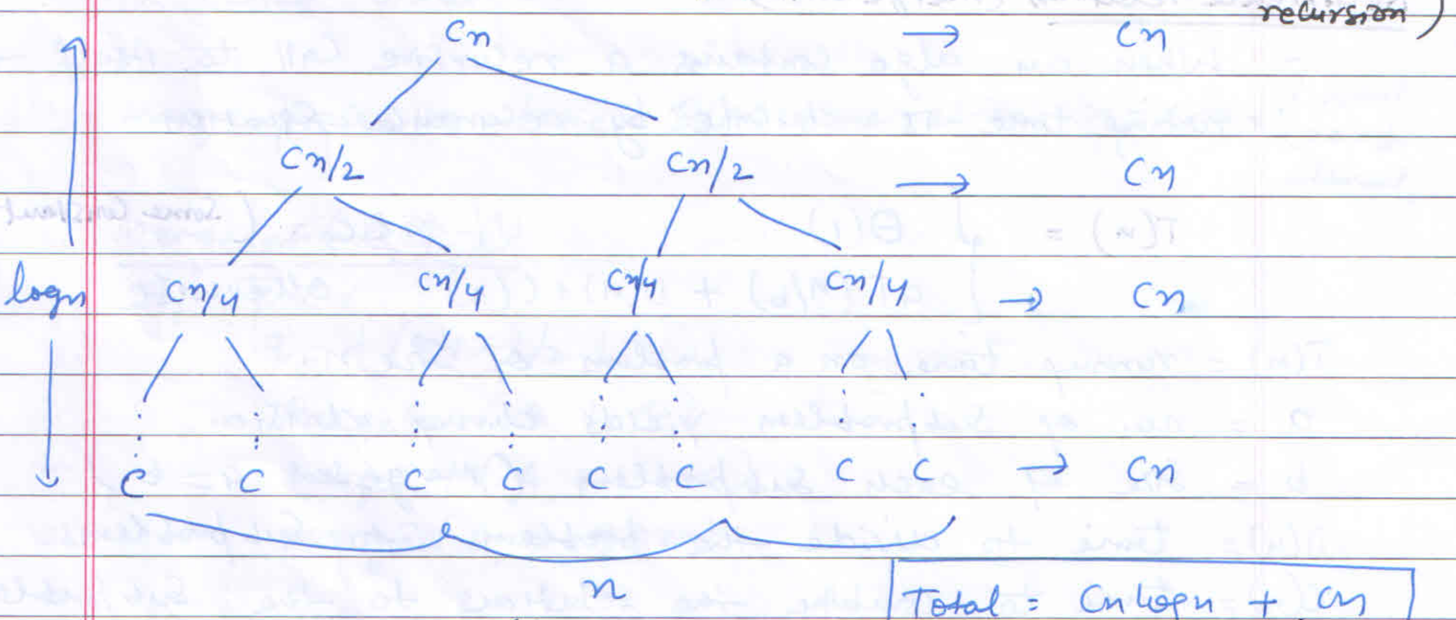
So recurrence relation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$T(n) = \Theta(n \log n)$ (master theorm)

Recursion Tree

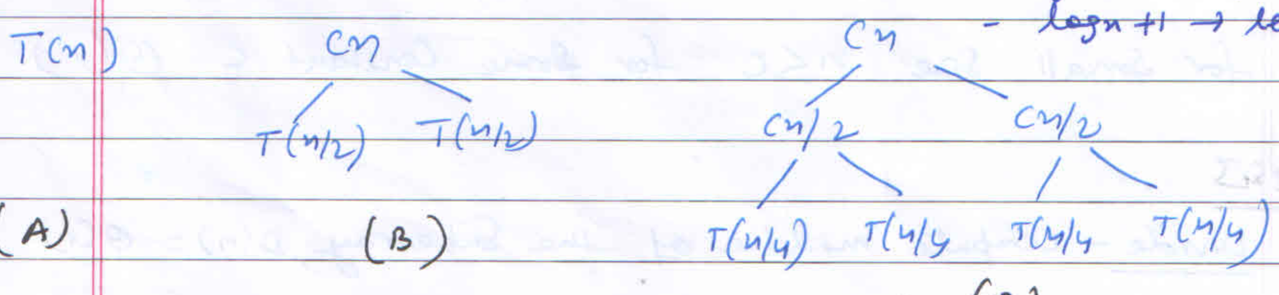
(cn = cost incurred at the top level of recursion)



(D)

Total = $cn\log n + cn$

- height of $\log n$
- $\log n + 1 \rightarrow$ levels



(A) (B) (C)

classmate Date ____ Page ____

Merge Sort

- Level i below top has 2^i nodes
- Each Contributing a cost of $c(n/2^i)$
- So ith level below the top has total cost $= 2^i \times c(n/2^i) = cn$ total cost
- Bottom level has n nodes, each Contributing Cost of c So cn.
- Total no. of levels of recursion tree is $\log n + 1$. (n = no. of leaf)

Properties

- Time Complexity $= \Theta(n \log n)$ in all 3 Cases. (worst, Avg, best)
- Auxiliary Space $= \Theta(n)$
- Algorithmic Paradigm – Divide and Conquer
- Sorting in place – No in a typical implementation [- inplace merge is Complicated and expensive]
- Stable – Yes
- Not Adaptive
- Does not require random access to data.

Application

1. Useful for Sorting linked List ($\Theta(n \log n)$ time). Other algo $\Theta(n\log n)$ Heapsort, Quicksort (Avg Case) Cannot be applied to linked list [- $\Theta(\log(n))$ extra space is needed for linked list in merge sort]
2. Inversion Counting problem
3. Used in External Sorting.

Note – If extra space $\Theta(n)$ is of no concern then Merge Sort is an excellent choice.

- **Adaptive Sort** if it takes advantage of existing order in its input.