Merge - Sort

[Recap]

Sorting Problem

Input: A sequence of n numbers $a_1, a_2, \ldots a_n$

Output: A permutation $a_1', a_2', \ldots a_n'$ of input sequence

such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

We have looked at Selection Sort and Insertion Sort

in Previous lectures.

Divide and Conquer Paradigm

① Divide the Problem into a number of subproblems that are Smaller instances of the same Problem.

② Conquer the Subproblems by solving them recursively. Solve the Subproblems directly if their Sizes are Small

③ Combine the solutions of Subproblems into the solution for the original Problem.

For Merge Sort [Assume that size of n is even]

① Divide the input sequence into two subsequences of $\frac{n}{2}$ elements each.

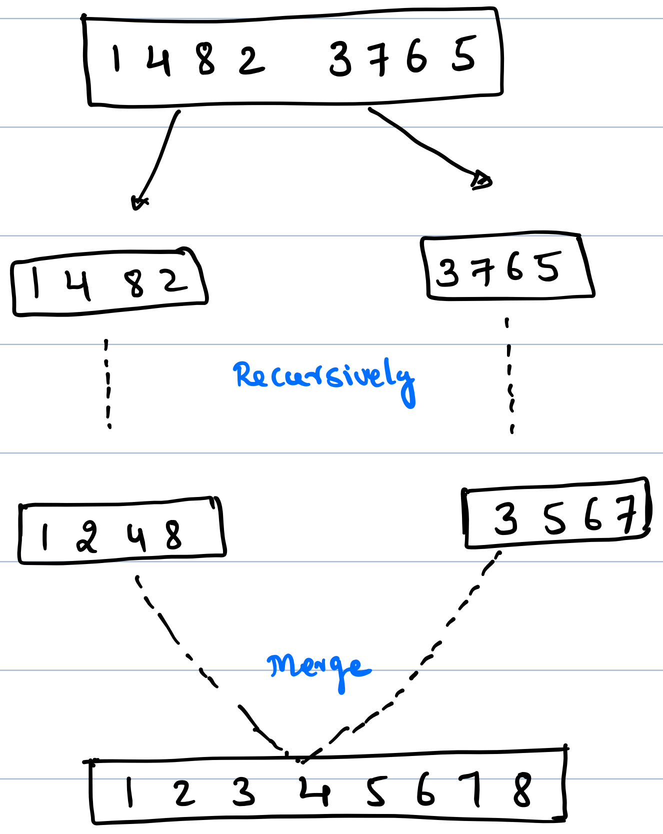
② Sort the two subsequences recursively using merge sort.

③ Combine the two sorted subsequences to produce the sorted answer.

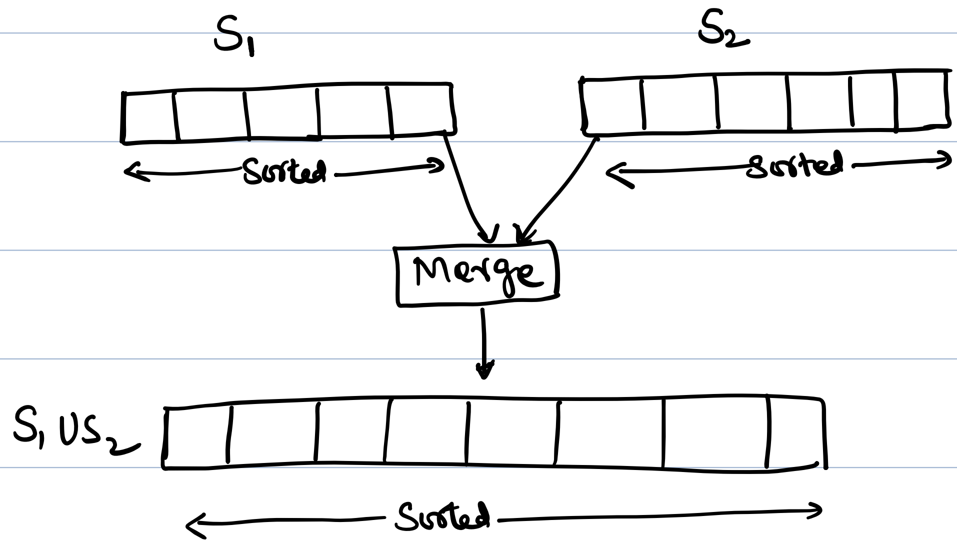
(Merging two sorted arrays)

Note: The base case of the recursion is when the sequence to be sorted has length 1, as every sequence of length 1 is already in sorted order.

Overview with an example



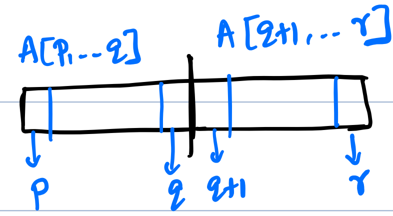
Merging two Sorted arrays to form a Single array.



Choose the Smaller of two arrays then delete it and Place it at first Place in $S_1 \cup S_2$.

Repeat this step until one of S_1 or S_2 is empty, at which time we just take the remaining input file and Place it at the end.

Pseudocode: [Ref: Coremen Page: 31]




```
MERGE(A, p, q, r)
1   n₁ = q − p + 1
2   n₂ = r − q
3   let L[1..n₁ + 1] and R[1..n₂ + 1] be new arrays
4   for i = 1 to n₁
5       L[i] = A[p + i − 1]
6   for j = 1 to n₂
7       R[j] = A[q + j]
8   L[n₁ + 1] = ∞
9   R[n₂ + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Loop-invariant

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p \, . . k-1]$ contains the $k - p$ smallest elements of $L[1 \, . . n_1 + 1]$ and $R[1 \, . . n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Running time of MERGE Procedure

Lines 1 - 3 $\rightarrow$ Constant time

Lines 4 - 7 $\rightarrow$ $\Theta(n_1 + n_2) = \Theta(n)$

Lines 8 - 11 $\rightarrow$ Constant time

Lines 12 - 17 $\rightarrow$ $\Theta(n)$

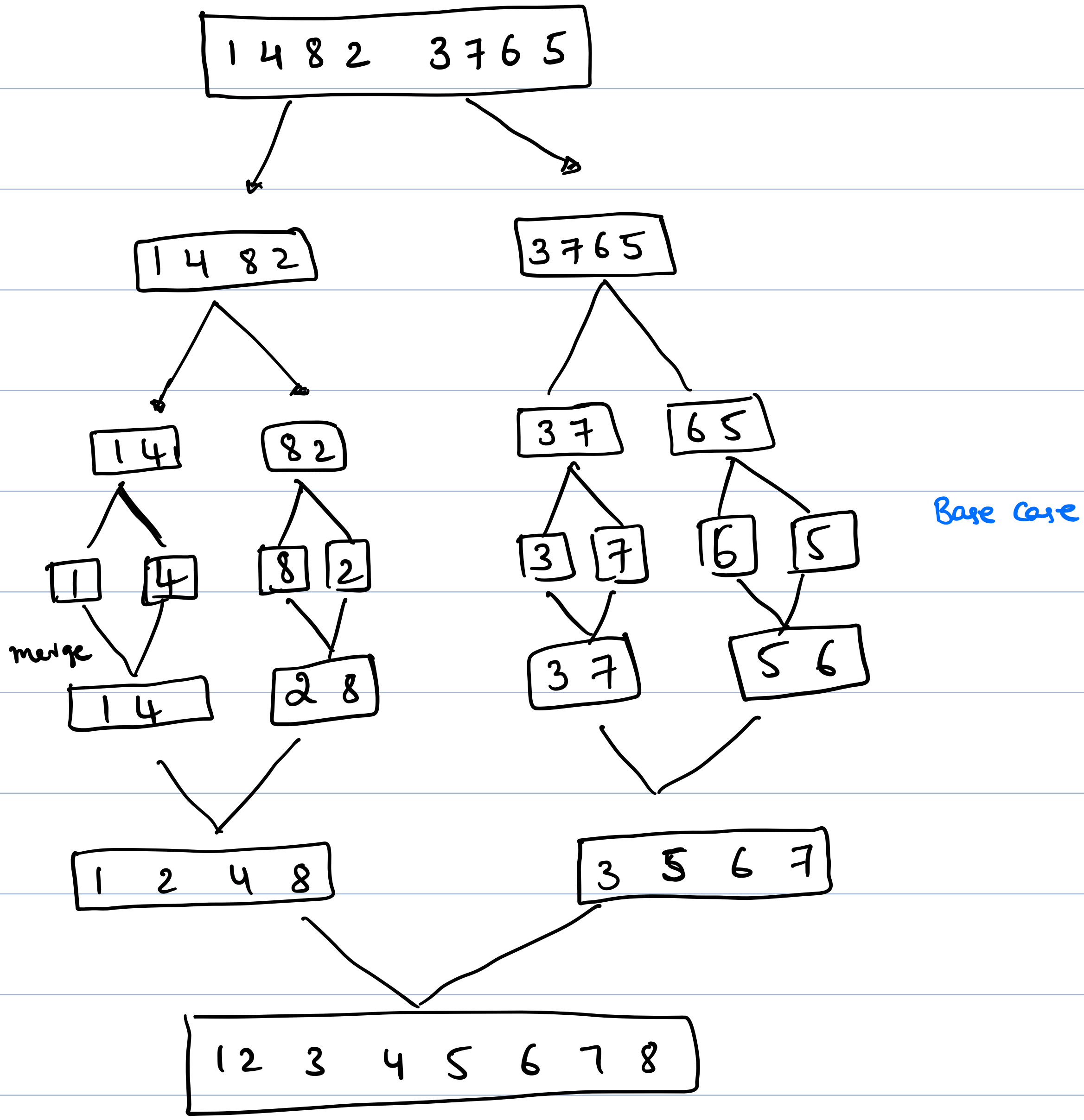
$\therefore$ MERGE Procedure runs in $\Theta(n)$ time.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

To sort the sequence $A = [A[1], A[2], \ldots A[n]]$,

we make the initial call MERGE-SORT$(A, 1, \underset{\downarrow\, n}{A.\text{length}})$.

Example:



1 4 8 2 3 7 6 5
1 4 8 2
3 7 6 5
1 4
8 2
3 7
6 5
Base case
1
4
8
2
3
7
6
5
merge
1 4
2 8
3 7
5 6
1 2 4 8
3 5 6 7
1 2 3 4 5 6 7 8

Analysis of merge sort :

```
MERGE-SORT(A, p, r)                → T(n)
if p < r                           → constant time = Θ(1)
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)            → T(n/2)
    MERGE-SORT(A, q + 1, r)        → T(n/2)
    MERGE(A, p, q, r)              → Θ(n)
```

$T(n)$: The worst case running time of merge sort on n numbers.

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

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

Where c represents the time needed to solve problems of size 1 as well as the time per array element of the divide & combine steps.

By expanding ①

$$T(n) = 2\left[2\,T(n/4) + c\,\frac{n}{2}\right] + cn$$

$$= 4\,T(n/4) + 2cn$$

$$= 4\left(2\,T(n/8) + c\,\frac{n}{4}\right) + 2cn$$

$$= 2^3\,T(n/2^3) + 3cn$$

$\approx \log_2^n$

$\vdots \qquad \vdots \qquad \vdots$

Q: How long we can go?

$$\frac{n}{2^i} \approx 1$$

$$i = \log_2^n$$

$$= 2^{\log_2^n}\,T(1) + (\log_2^n)\,cn$$

$$= cn + c.n\log n$$

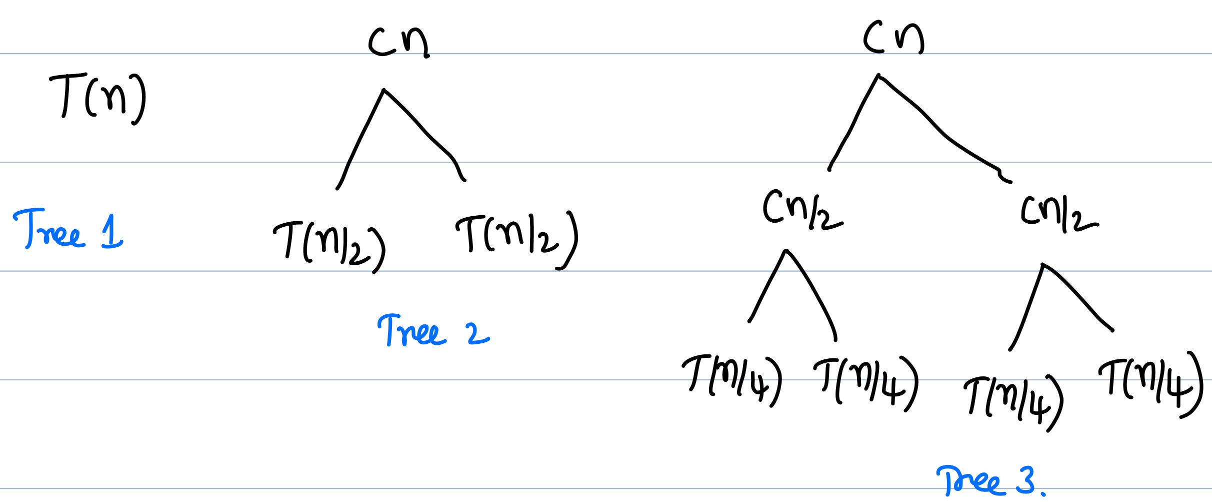
$$= \Theta(n\log n)$$

The above procedure can be represented using a recursion tree, where each node represent the cost of a single subproblem in the set of recursive function calls.

First, we sum the costs within each level of the tree and obtain a set of per-level costs.

Next, we sum all the per-level cost to obtain the total cost of the recursion.

$$T(n) = 2\,T(n/2) + cn$$

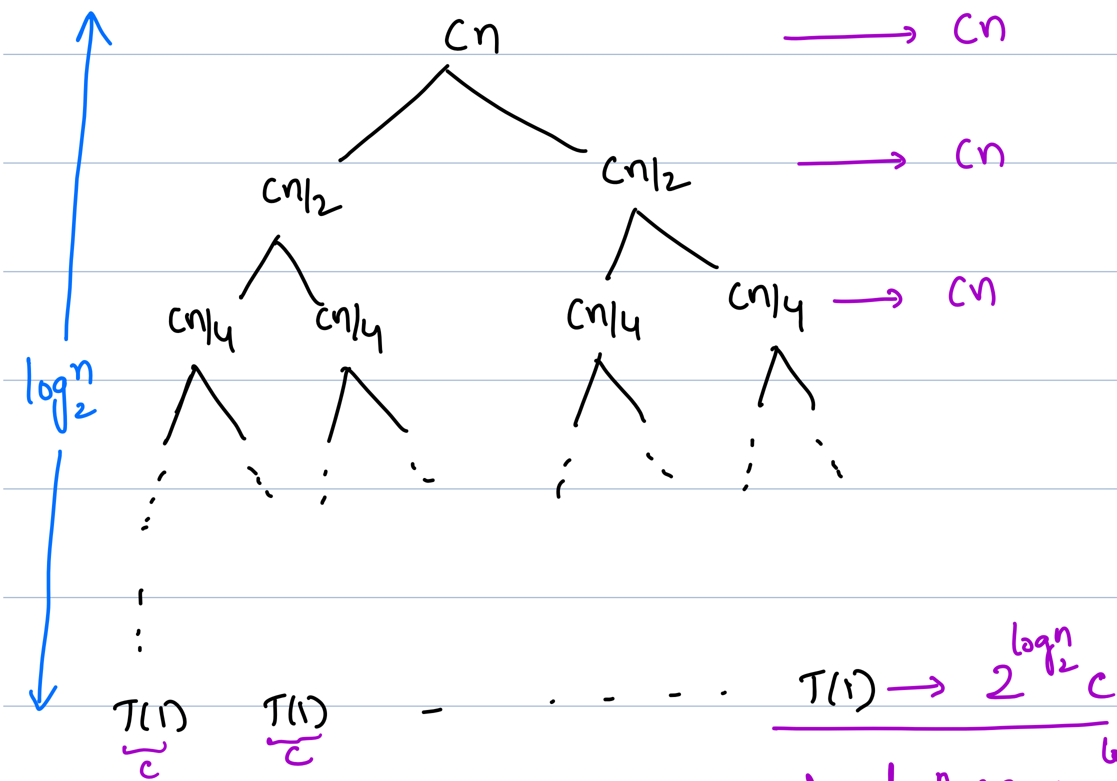


Tree 1

Tree 2

Tree 3.

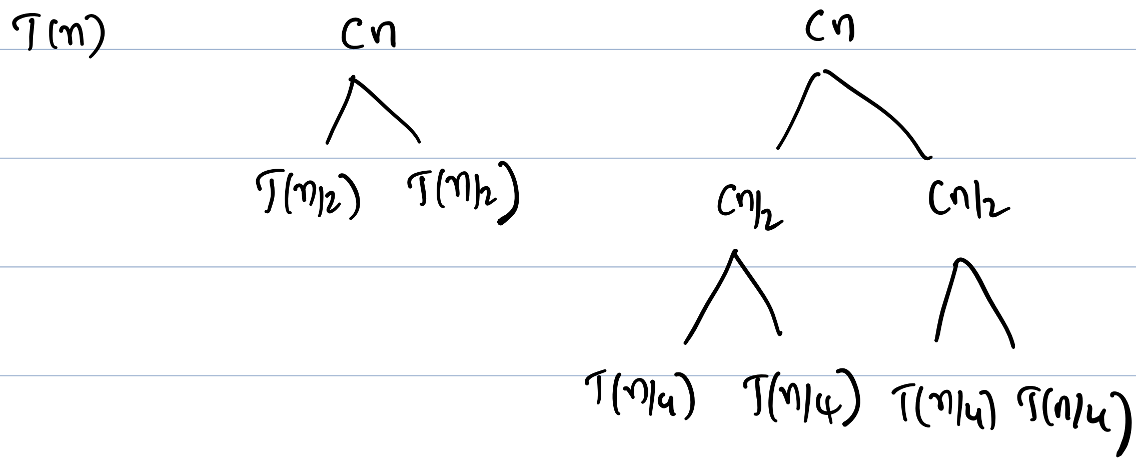
We continue expanding each node in the tree, by using recurrence, we get the following tree.

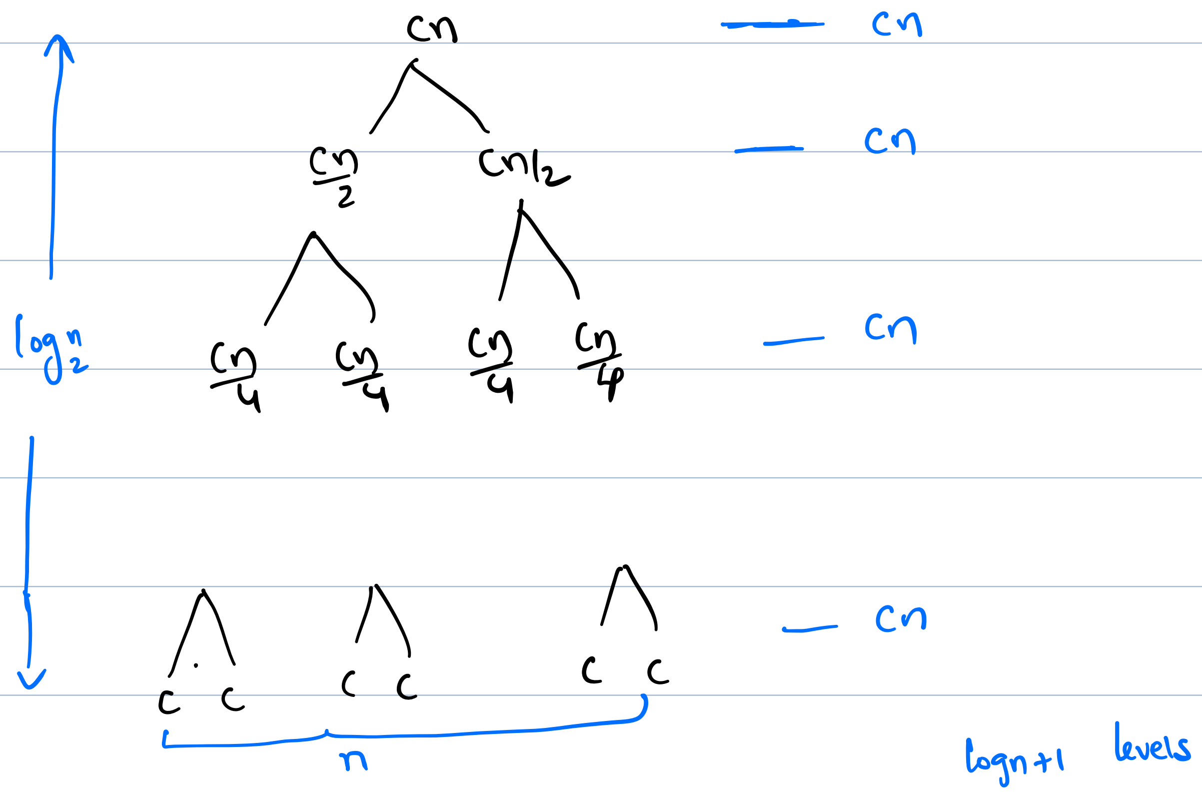


$$\text{Total}: \log_2^n cn + 2^{\log_2^n} c$$

$$= cn\log n + cn$$

$$= \Theta(n\log n)$$





Total : $cn \cdot \log n + cn$

$= \Theta(n \log n)$

Binary Search (Reference : Wikipedia)

It works on sorted arrays. Binary Search begins by comparing an element in the middle of the array with the target value.

if the target value matches the element, its position in the array is returned.

if the target value is less than the element the search continues in the lower half of the array.

if the target value is greater than the element the search continues in the upper half of the array.

This way, the algorithm eliminates the half in which the target value cannot lie in each iteration.

Binary search runs in logarithmic time in the worst case, making $O(\log n)$ Comparisons, where n is the number of elements in the array.