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.

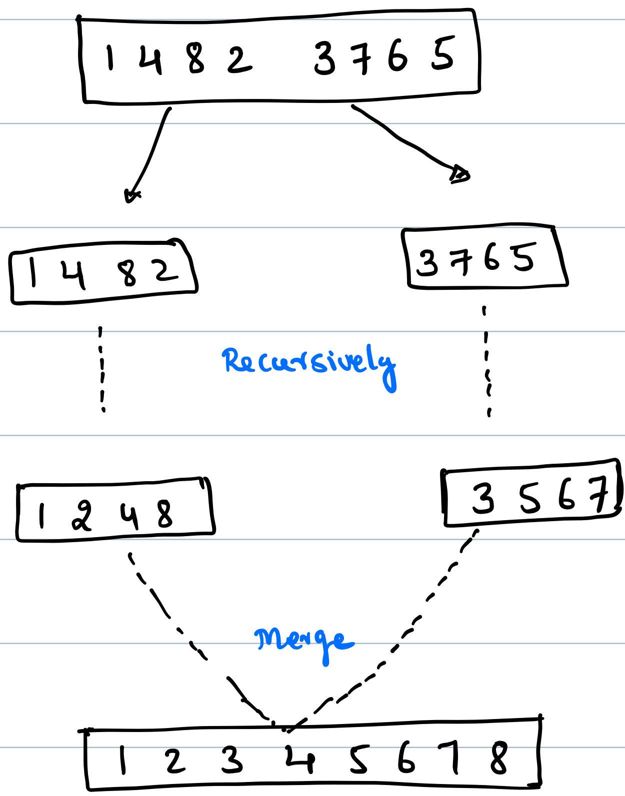
② Sort the two subsequences recursively using merge sort.

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

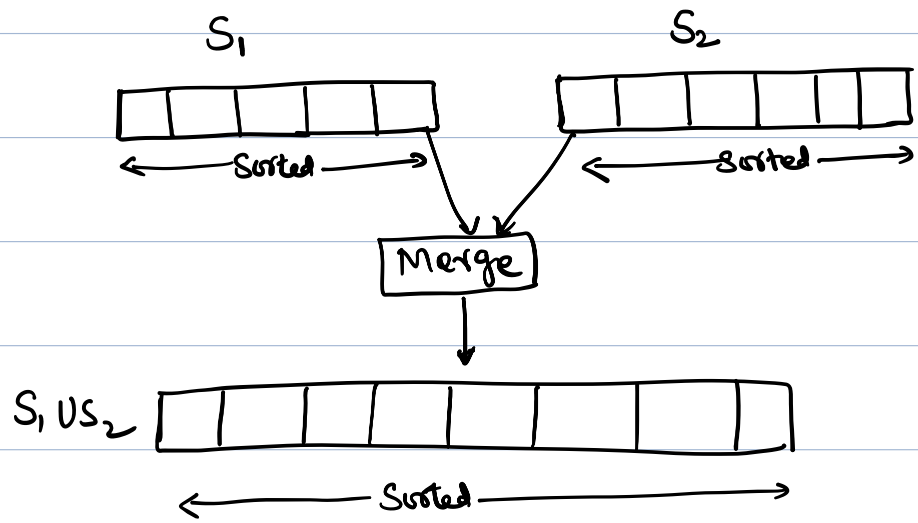
(Merging two sorted arrays)

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



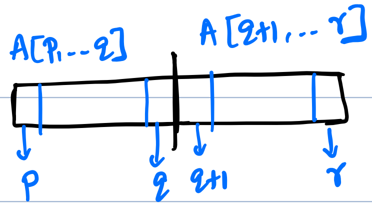
Merging two sorted arrays to form a single array.



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 $\rightarrow$ $\Theta(n)$

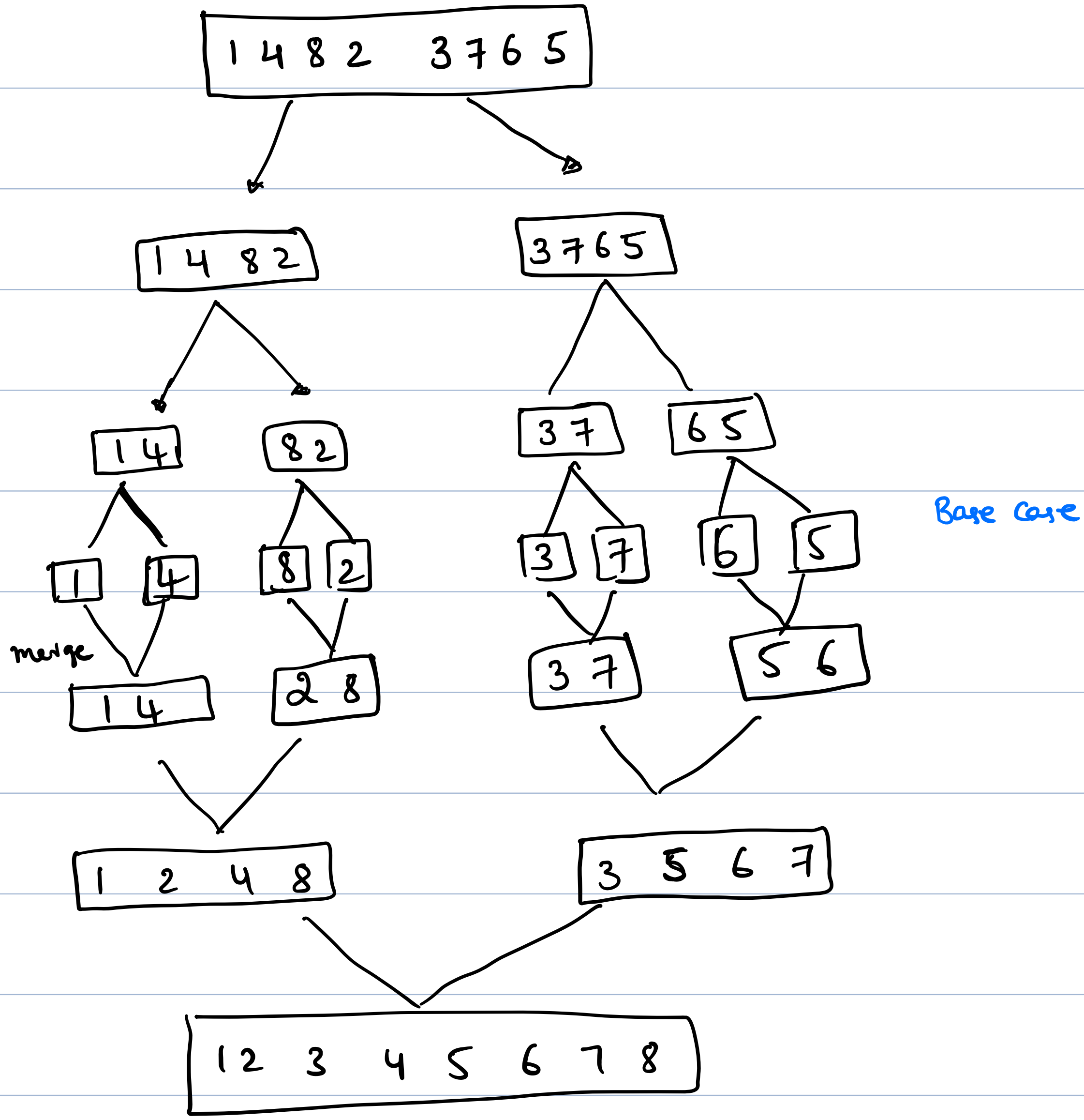
$\therefore$ MERGE Procedure runs in $\Theta(n)$ time.

```
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$(A, 1, \underset{n}{\underset{\downarrow}{A.length}})$.

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
1
4
8
2
3
7
6
5
Base case
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
      q = ⌊(p + r)/2⌋          → constant time = Θ(1)
      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.

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

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

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

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

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

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

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

$\approx \log_2^n$ ⋮ ⋮ ⋮

Q: How long we can go?

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

$$i = \log_2^n$$

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

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

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

Comparison Sort:

Comparison based sorting algorithms: Sorts the elements by comparing pairs of them.

We can also interpret this as follows.

Suppose we have some objects, which are hidden in a box, The goal is to sort the objects based on their weight without any information except that obtained by placing two weights on the scale and seeing which one is heavier.

All the Algorithms we have studied so far are Comparison sorts.

- Insertion Sort
- Merge sort
- Selection sort
- Heap sort

We know that Merge sort and Heap sort can sort n numbers in $O(n \log n)$ time.

In this Class, We Prove the following

Any comparison Sort must make $\Omega(n\log n)$ Comparisons in the worst case to Sort n elements.

ie., Merge sort and Heap sort are asymptotically optimal and no Comparison Sort exists that is faster by more than a constant factor.

<u>Main Result</u>

Any Comparison sort algorithm requires $\Omega(n\log n)$ comparisons in the worst case.

More details will be given in the Next course.