Merge Sort

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

In this lecture, we will design the merge sort which sorts n elements in $O(n \log n)$ time. The algorithm illustrates a technique called divide and conquer, which is the most common and useful form of recursion in computer science.

Recall:

The Sorting Problem

Problem Input:

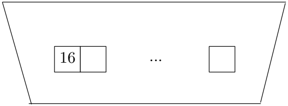
A set S of n integers is given in an array of length n. The value of n is inside the CPU (i.e., in a register).

Goal:

Design an algorithm to store S in an array where the elements have been arranged in ascending order.

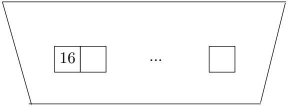
Example

Input:



38	28	88	17	26	41	72	83	69	47	12	68	5	52	35	9														

Output:



5	9	12	17	26	28	35	38	41	47	52	68	69	72	83	88														

Recall: The idea of recursion is to carry out two steps:

1. **[Base]**
   Solve the problem trivially where the input size n is a constant.
2. **[Reduce]**
   Argue that if we can solve the same problem with a size smaller than n, we can solve the original problem (with size n).

Merge Sort

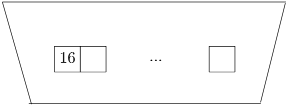
Base. If $n = 1$ (i.e., S has a single element), there is nothing to sort. Return directly.

Reduce. Otherwise:

1. Recursively sort the first half of the array S (i.e., same problem but with size $n/2$).
2. Recursively sort the second half of the array.
3. Merge the two halves of the array into the final sorted sequence (details later).

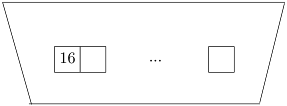
Example

Input:



38	28	88	17	26	41	72	83	69	47	12	68	5	52	35	9														

First step, sort the first half of the array by recursion.

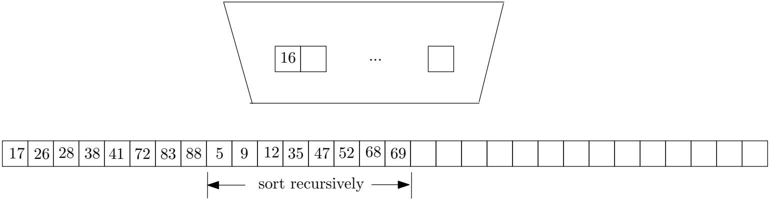


17	26	28	38	41	72	83	88	69	47	12	68	5	52	35	9														

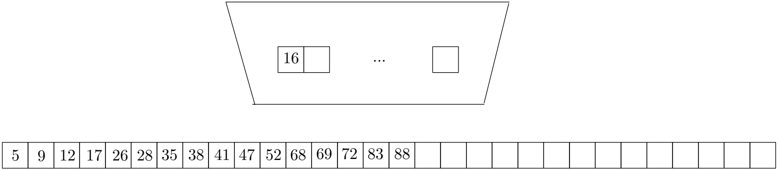
$\longleftarrow$ sort recursively $\longrightarrow$

Example

Second step, sort the second half of the array by recursion:



Third step, merge the two halves.

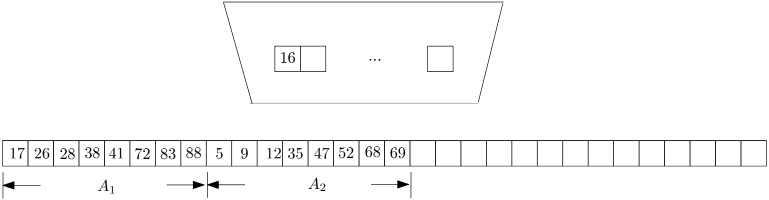


Merging

We are looking at the following (sub-)problem.

There are two arrays—denoted as A_1 and A_2—of integers. Each array has (at most) $n/2$ integers, which have been sorted in ascending order. The goal is to produce an array A with all the integers in A_1 and A_2, sorted in ascending order.

The following shows an example of the input:



Merging

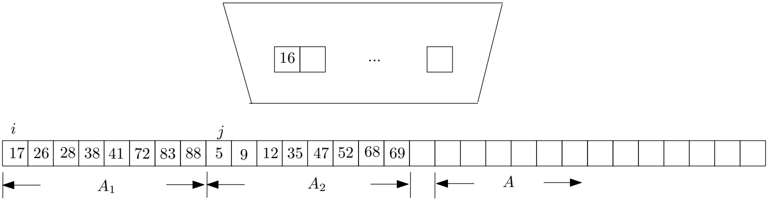
At the beginning, set i and j to 1.

Repeat the following until $i > n/2$ or $j > n/2$:

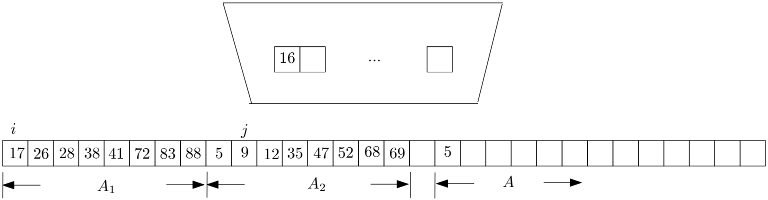
1. If $A_1[i]$ (i.e., the i-th integer of A_1) is smaller than $A_2[j]$, append $A_1[i]$ to A, and increase i by 1.
2. Otherwise, append $A_2[j]$ to A, and increase j by 1.

Example

At the beginning of merging:

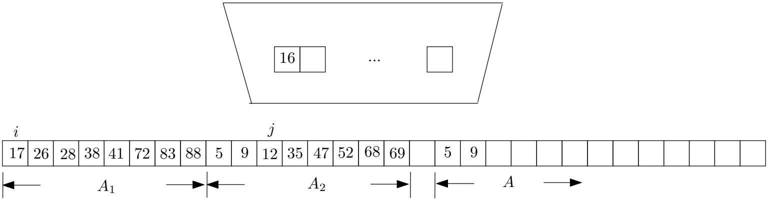


Appending 5 to A:

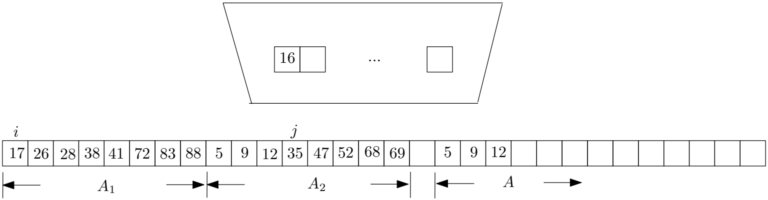


Example

Appending 9 to A:

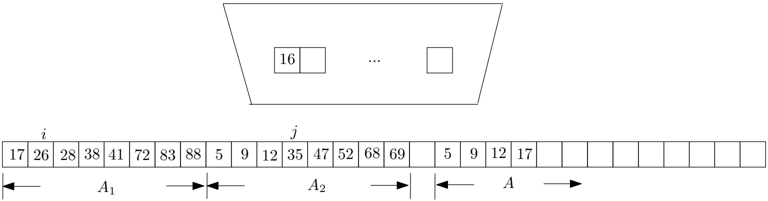


Appending 12 to A:



Example

Appending 17 to A:



And so on.

Running Time of Merge Sort

Let $f(n)$ denote the worst-case running time of merge sort when executed on an array of size n.

From the base of recursion, we have:

$$f(n) = O(1)$$

From the reduce part, we know:

$$f(n) \leq 2f(n/2) + O(n)$$

where the term $2f(n/2)$ is because the recursion sorts two arrays each of size $n/2$, and the term $O(n)$ is the time of merging (convince yourself this is true).

Running Time of Merge Sort

So it remains to solve the following recurrence:

$$\begin{aligned} f(n) &\leq c_1 \\ f(n) &\leq 2f(n/2) + c_2 n \end{aligned}$$

where c_1, c_2 are constants (whose values we do not care). Using the expansion method, we have:

$$\begin{aligned} f(n) &\leq 2f(n/2) + c_2 n \\ &\leq 2(2f(n/4) + c_2 n/2) + c_2 n = 4f(n/4) + 2c_2 n \\ &\leq 4(2f(n/8) + c_2 n/4) + 2c_2 n = 8f(n/8) + 3c_2 n \\ &\ldots \\ &\leq 2^i f(n/2^i) + i \cdot c_2 n \\ &\ldots \\ (h = \log_2 n) &\leq 2^h f(1) + h \cdot c_2 n \\ &\leq n \cdot c_1 + c_2 n \cdot \log_2 n = O(n \log n). \end{aligned}$$

Running Time of Merge Sort

The previous discussion assumed n to be a power of 2. How do we remove the assumption?

Hint: The rounding approach discussed in a previous lecture.

The form of recursion we used in merge sort is also called divide and conquer. The name is fairly intuitive: we "divided" the input array into two halves, "conquered" them separately (i.e., sorting them), and derived the overall result. This form of recursion is frequently applied in computer science—it can be utilized to solve numerous problems elegantly.

Recall that selection sort performs sorting in $O(n^2)$ time. Today, we have significantly improved the running time to $O(n \log n)$. Interestingly, this can no longer been improved asymptotically using the so-called "comparison-based" approach—we will prove later in this course that any comparison-based algorithm must incur $\Omega(n \log n)$ time!