Merge Sort

Mergesort needs to implemented using recursion. First, divide the N-sized array in half. Then, two N/2 arrays are created when the original array is divided once. The next split produces four N/4 arrays. And it goes on like this when array size increases. This means that the segmentation process is reduced by half each time, so we have to repeat logN to divide into a one-size array.

In our case, merging left = T(2/N) and merging right = T(2/N). And, merge them together O(N). Basically, we split the array in half continuously with variables such as start, mid, and end. Using a while loop, we compare the values of the array of each position to swap using a[i] > a[j], and increase the value of the i and j according to the conditions. So, it is converted to

$T(n) = 2 \times T(n/2) + O(n) = 2(2T(n/4) + O(n/2)) + O(n) = 4T(n/4) + 2O(n) = \ldots.$

$= n \times T(1) + \log n \times O(n) = n \times O(1) + O(n \log n) = O(n) + O(n \log n) = O(n \log n)$

Therefore, it has a time complexity of O(NlogN).

Quicksort

We implemented quicksort recursively with partitioning. First, we first pick an element as pivot O(1) and classify small, equal, and large values compare with pivot value O(n). Then, quicksort each case.

If a left variable is smaller than the pivot value, then increase its index (i). If the right variable is larger than the pivot value, then decrease its index (j).

If left<=right, swap them using while loop. Then, increase the left variable and decrease the right variable. Repeat this process until sorting complete.

Hence, T(n) = T(left) + T(right) + O(n).

// left = number of elements less than or equal to the pivot

// right = number of elements larger than or equal to the pivot.

Suppose the pivot divides the number of elements in half. If so, it shows that T(n) = 2T(n/2) + O(n). Consequently, the average time complexity is converted to T(n) = O(nlogn) and the worst is O(n^2).

Gold’s Poresort

First, we compare all even-indexed cells to their next neighbor cell, and then compare all odd-indexed cells to their next neighbor cell. We implemented this algorithm using variable “status” to distinguish whether it is checking even-indexed cells or odd-indexed cells. When it’s swapping even-indexed cells, the status is equal to 0 O(1). When it’s over, the status changes to 1 O(1). Then, it starts to swap odd-indexed slots. And, when it’s over, the status changes to 0 again. Repeat this process until sorting is complete. We have for loop that increases the i=0 to array length N O(N). And, there are two for loops that comparing even-indexed cells or odd-indexed cells increasing by two. Therefore, there are (N/2) passes, so T(N) = N * (N/2) = N^2. Consequently, the time complexity is converted to O(N^2).