**MergeSort**

Divide array into two equal halves, “divide and conquer”.

[5 8 6 4 3 7 1 2]

[5 8 6 4] [3 7 1 2]

[5 8] [6 4] [3 7] [1 2]

Sort them, then piece them together in a sorted fashion.

[5 8] [4 6] [3 7] [1 2]

[4 5 6 8] [1 2 3 7]

[1 2 3 4 5 6 7 8]

T(1) = 1

T(N) = 2T(N/2) + N

Adding all the equations on the left and right, most terms in the telescoping sum and the final result is T(N)/N = T(1)/1 + log(N). The log(N) is a result of adding all +1 but since it’s T(N) = T(N/2), the steps are halved.

T(N)/N = T(1)/1 + log(N) = T(N) = N + Nlog(N) = O(log(N)).

Java: comparison is expensive, movement is not.

C++: Comparison is cheap, movement is not.

Java prefers MergeSort because it has the least amount of comparisons, but high movement.

Java also uses QuickSort for primitive types.

C++ prefers QuickSort because it has low amount of movement, but high comparison.

**QuickSort**

Uses a pivot, best pivot if the array is sorted and uses the median.

Bad pivots are the smallest or largest value.

Left: smaller than pivot. Right: bigger than pivot.

5 8 6 4 9 3 7 1 2

2 5 9

2 8 6 4 1 3 7 5 9

^ ^

2 3 6 4 1 8 7 5 9

Splits array into 2 groups. 1 group has values less than pivot, the other has values greater than or equal to pivot. This partitioning takes O(N).

[\_\_\_\_\_] P [\_\_\_\_\_]

^ ^

i N – i – 1

T(N) = T(i) + T(N – i – 1) + CN 🡨 number of QuickSorts left.

Worst case: pivot is smallest element.

[ ] P [\_\_\_\_\_\_\_\_\_\_\_\_\_\_]

0 N – 1

T(N) = T(N – 1) + CN

T(N – 1) = T(N – 2) + C(N – 1)

T(N – 2) = T(N – 3) + C(N – 2)

…

T(2 (N = 4)) = T(1) + C(2)

T(1) = C

Simplify:

T(N) = T(0) + CN + C(N – 1) + … C(2) + C 🡪 C(N + (N – 1) + … + 2 + 1) 🡪 N(N + 1)/2

N(N + 1)/2 🡪 N2/2 + N/2 🡪 O(N2).

Best case: the pivot is the median (after partitioning).

[\_\_\_\_\_] P [\_\_\_\_\_]

N/2 N/2

T(N) = T(i) + T(N – i – 1) + CN

T(N) = T(N/2) + T(N/2) + CN 🡪 2T(N/2) + CN

T(N)/N = T(N/2)/(N/2) + C

T(N/2)/(N/2) = T(N/4)/(N/4) + C

…

T(2 (N = 4))/2 = T(1)/1 + C

T(1)/1 = C

T(N)/N = 1 + C\*log(N) 🡪 T(N) = N + CNlog(N) 🡪 O(Nlog(N)).

Average case: assume size of partition is equally likely.

[ ] P [\_\_\_\_\_\_\_\_\_\_\_]

0 N - 1

[\_] P [\_\_\_\_\_\_\_\_\_\_]

1 N - 2

…

[\_\_\_\_\_\_\_\_\_] P [ ]

N – 1 0

T(N) = T(i) + T(N – i – 1) + CN

T(N) = 2/N \* Summation(i = 0; to N – 1; T(i)) + CN

NT(N) = 2 \* Summation(i = 0; to N – 1; T(i)) + CN2

(N – 1)T(N – 1) = 2 \* Summation(i = 0; to N – 2; T(i)) + C(N – 1)2 🡨 Only telescoping sum remaining. Thus, T(N – 1) remains.

**RadixSort**

Get the last digit of element.

[123 238 210 019 528 003 513 129 220 294]

Last digits: 3, 8, 0, 9, 8, 3, 3, 9, 0, 4. Sort based on last digit, from left to right.

210, 220, 123, 003, 513, 294, 238, 528, 019, 129

Then next digit: 1, 2, 2, 0, 1, 9, 3, 2, 1, 2. Sort.

003, 210, 513, 019, 220, 123, 528, 129, 238, 294

Then next digit: 0, 2, 5, 0, 2, 1, 5, 1, 2, 2. Sort.

003, 019, 123, 129, 210, 220, 238, 294, 513, 528

Good: very fast.

Bad: Only works for numbers.