Algorithm Efficiency and Sorting

Burak Öztürk

21901841

Section 1

Assignment 1

# Question 1

## Part 1

* T(n) = 3T(n/3) + n, where T(1) = 1 and n is an exact power of 3.

Substitute T(n/3) for 3T(n/9) + n/3 by substituting n for n/3 in the original function.

T(n/3) = 3T(n/9) + n/3

T(n) = 3(3\*T(n/9) + n/3) + n

= 9T(n/9) + n + n

= 9T(n/9) + 2n

T(n/9) = 3T(n/27) + n/9

T(n) = 9(3T(n/27) + n/9) + 2n

= 27T(n/27) + n + 2n

= 27T(n/27) + 3n

Keep substituting until n/3^k becomes 1. (k is an integer as 0 < k <= log3(n))

T(n) = 3T(n/3) + n k = 1

= 9T(n/9) + 2n k = 2

= 27T(n/27) + 3n k = 3

… …

T(n) = (3^k)T(n/(3^k)) + kn k = log3(n)

Since 3^k = n and T(1) = 1 function becomes

T(n) = n + kn = n + nlog3(n)

Therefore O(T(n)) = O(nlog3n)

* T(n) = 2T(n−1) + n2, where T(1) = 1.

Substitute T(n-1) for 2T(n-2) + (n-1)2 by substituting n2 for (n-1)2 in the original function.

T(n-1) = 2T(n-2) + (n-1)2

T(n) = 2(2T(n-2) + (n-1)2) + n2

= 4T(n-2) + 2(n-1)2 + n2

T(n-2) = 2T(n-3) + (n-2)2

T(n) = 4T(n-2) + 2(n-1)2 + n2

= 4(2T(n-3) + (n-2)2) + 2(n-1)2 + n2

= 8T(n-3) + 4(n-2)2 + 2(n-1)2 + n2

Keep substituting until n-k becomes 1. (k is an integer as 0 < k < n)

T(n) = 2T(n−1) + n2 k = 1

= 4T(n-2) + 2(n-1)2 + n2 k = 2

= 8T(n-3) + 4(n-2)2 + 2(n-1)2 + n2 k = 3

… …

T(n) = 2n-1T(1) + k = n-1

= ((2n-4)(n-1)2)/2 + 2n-1

Therefore O(T(n)) = O(n22n)

* T(n) = 3T(n/4) + nlogn, where T(1) = 1 and n is an exact power of 4.

Substitute T(n/4) for 3T(n/16) + (n/4)log(n/4) by substituting n for n/4 in the original function.

T(n/4) = 3T(n/16) + (n/4)log(n/4)

T(n) = 3(3\*T(n/16) + (n/4)log(n/4)) + nlogn

= 9T(n/16) + (3n/4)log(n/4) + nlogn

T(n/16) = 3T(n/64) + (n/16)log(n/16)

T(n) = 9(3T(n/64) + (n/16)log(n/16)) + (3n/4)log(n/4) + nlogn

= 27T(n/64) + (9n/16)log(n/16) + (3n/4)log(n/4) + nlogn

Keep substituting until n/4k becomes 1. (k is an integer as 0 < k <= log4(n))

T(n) = 3T(n/4) + nlogn k = 1

= 9T(n/16) + (3n/4)log(n/4) + nlogn k = 2

= 27T(n/64) + (9n/16)log(n/16) + (3n/4)log(n/4) + nlogn k = 3

… …

T(n) = (3 k)T(n/4k) + k = log4(n)

Therefore O(T(n)) = O(nlog2n) since (3k)T(n/4k) is a number smaller than n = (4k)T(n/4k) and log4n terms are all at nlogn complexity.

* T(n) = 3T(n/2) + 1, where T(1) = 1 and n is an exact power of 2.

Substitute T(n/2) for 3T(n/4) + 1 by substituting n for n/2 in the original function.

T(n/2) = 3T(n/4) + 1

T(n) = 3(3T(n/4) + 1) + 1

= 9T(n/4) + 3 + 1

= 9T(n/4) + 4

T(n/4) = 3T(n/8) + 1

T(n) = 9T(n/4) + 4

= 9(3T(n/8) + 1) + 4

= 27T(n/8) + 13

Keep substituting until n/2k becomes 1. (k is an integer as 0 < k <= log2(n))

T(n) = 3T(n/2) + 1 k = 1

= 9T(n/2) + 3 + 1 k = 2

= 27T(n/64) + 9 + 3 + 1 k = 3

… …

T(n) = (3 k)T(n/2k) + k = log2(n)

= 3k + (3k-1)/2

Therefore O(T(n)) = O(3logn) = O(nlog3)

## Part 2

Array: [5, 6, 8, 4, 10, 2, 9, 1, 3, 7]

* Bubble Sort

Sorted and unsorted parts separated with | and swap candidates bolded and selected with parentheses.

S means swap is carried out.

Pass 1:

[(**5, 6)**, 8, 4, 10, 2, 9, 1, 3, 7]

[5, (**6, 8)**, 4, 10, 2, 9, 1, 3, 7]

[5, 6, (**8, 4)**, 10, 2, 9, 1, 3, 7] S

[5, 6, 4, (**8, 10)**, 2, 9, 1, 3, 7]

[5, 6, 4, 8, (**10, 2)**, 9, 1, 3, 7] S

[5, 6, 4, 8, 2, (**10, 9)**, 1, 3, 7] S

[5, 6, 4, 8, 2, 9, (**10, 1)**, 3, 7] S

[5, 6, 4, 8, 2, 9, 1, (**10, 3)**, 7] S

[5, 6, 4, 8, 2, 9, 1, 3, (**10, 7)**] S

[5, 6, 4, 8, 2, 9, 1, 3, 7 | 10]

Pass 2:

[(**5, 6)**, 4, 8, 2, 9, 1, 3, 7 | 10]

[5, (**6, 4)**, 8, 2, 9, 1, 3, 7 | 10] S

[5, 4, (**6, 8)**, 2, 9, 1, 3, 7 | 10]

[5, 4, 6, (**8, 2)**, 9, 1, 3, 7 | 10] S

[5, 4, 6, 2, (**8, 9)**, 1, 3, 7 | 10]

[5, 4, 6, 2, 8, (**9, 1)**, 3, 7 | 10] S

[5, 4, 6, 2, 8, 1, (**9, 3)**, 7 | 10] S

[5, 4, 6, 2, 8, 1, 3, (**9, 7)** | 10] S

[5, 4, 6, 2, 8, 1, 3, 7 | 9, 10]

Pass 3:

[**(5, 4)**, 6, 2, 8, 1, 3, 7 | 9, 10] S

[4, (**5, 6)**, 2, 8, 1, 3, 7 | 9, 10]

[4, 5, (**6, 2)**, 8, 1, 3, 7 | 9, 10] S

[4, 5, 2, (**6, 8)**, 1, 3, 7 | 9, 10]

[4, 5, 2, 6, (**8, 1)**, 3, 7 | 9, 10] S

[4, 5, 2, 6, 1, (**8, 3)**, 7 | 9, 10] S

[4, 5, 2, 6, 1, 3, (**8, 7)** | 9, 10] S

[4, 5, 2, 6, 1, 3, 7 | 8, 9, 10]

Pass 4:

[(**4, 5)**, 2, 6, 1, 3, 7 | 8, 9, 10]

[4, (**5, 2)**, 6, 1, 3, 7 | 8, 9, 10] S

[4, 2, (**5, 6)**, 1, 3, 7 | 8, 9, 10]

[4, 2, 5, (**6, 1)**, 3, 7 | 8, 9, 10] S

[4, 2, 5, 1, (**6, 3)**, 7 | 8, 9, 10] S

[4, 2, 5, 1, 3, (**6, 7)** | 8, 9, 10]

[4, 2, 5, 1, 3, 6 | 7, 8, 9, 10]

Pass 5:

[(**4, 2)**, 5, 1, 3, 6 | 7, 8, 9, 10] S

[2, (**4, 5)**, 1, 3, 6 | 7, 8, 9, 10]

[2, 4, (**5, 1)**, 3, 6 | 7, 8, 9, 10] S

[2, 4, 1, (**5, 3)**, 6 | 7, 8, 9, 10] S

[2, 4, 1, 3, (**5, 6)** | 7, 8, 9, 10]

[2, 4, 1, 3, 5 | 6, 7, 8, 9, 10]

Pass 6:

[**(2, 4)**, 1, 3, 5 | 6, 7, 8, 9, 10]

[2, **(4, 1)**, 3, 5 | 6, 7, 8, 9, 10] S

[2, 1, **(4, 3)**, 5 | 6, 7, 8, 9, 10] S

[2, 1, 3, **(4, 5)** | 6, 7, 8, 9, 10]

[2, 1, 3, 4 | 5, 6, 7, 8, 9, 10]

Pass 7:

[**(2, 1)**, 3, 4 | 5, 6, 7, 8, 9, 10] S

[1, **(2, 3)**, 4 | 5, 6, 7, 8, 9, 10]

[1, 2, **(3, 4)** | 5, 6, 7, 8, 9, 10]

[1, 2, 3 | 4, 5, 6, 7, 8, 9, 10]

Pass 8:

[**(1, 2)**, 3 | 4, 5, 6, 7, 8, 9, 10]

[1, **(2, 3)** | 4, 5, 6, 7, 8, 9, 10]

[1, 2 | 3, 4, 5, 6, 7, 8, 9, 10]

Pass 9:

[**(1, 2)** | 3, 4, 5, 6, 7, 8, 9, 10]

Finalized array:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

* Selection sort:

Sorted and unsorted parts separated with | values that will be swapped with last element of unsorted array are bolded and selected with parentheses.

[5, 6, 8, 4, **(10)**, 2, 9, 1, 3, 7]

[5, 6, 8, 4, 7, 2, **(9)**, 1, 3 | 10]

[5, 6, **(8)**, 4, 7, 2, 3, 1 | 9, 10]

[5, 6, 1, 4, **(7)**, 2, 3 | 8, 9, 10]

[5, **(6)**, 1, 4, 3, 2 | 7, 8, 9, 10]

[**(5)**, 2, 1, 4, 3 | 6, 7, 8, 9, 10]

[3, 2, 1, **(4)** | 5, 6, 7, 8, 9, 10]

[**(3)**, 2, 1 | 4, 5, 6, 7, 8, 9, 10]

[1, **(2)** | 3, 4, 5, 6, 7, 8, 9, 10]

[**(1)** | 2, 3, 4, 5, 6, 7, 8, 9, 10]

Finalized array:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

## Part 3

Worst case for Quicksort algorithm is the situation where the pivot (which is selected as first element of the array) is the biggest or smallest of the list. In that case, Quicksort algorithm will have time function as T(n) = T(n-1) + 2n.

(T(n-1) is for recursion with one less item and 2n is for comparison and swaps)

T(n) = T(n-1) + 2n

= T(n-2) + 2(n-1) + 2n

…

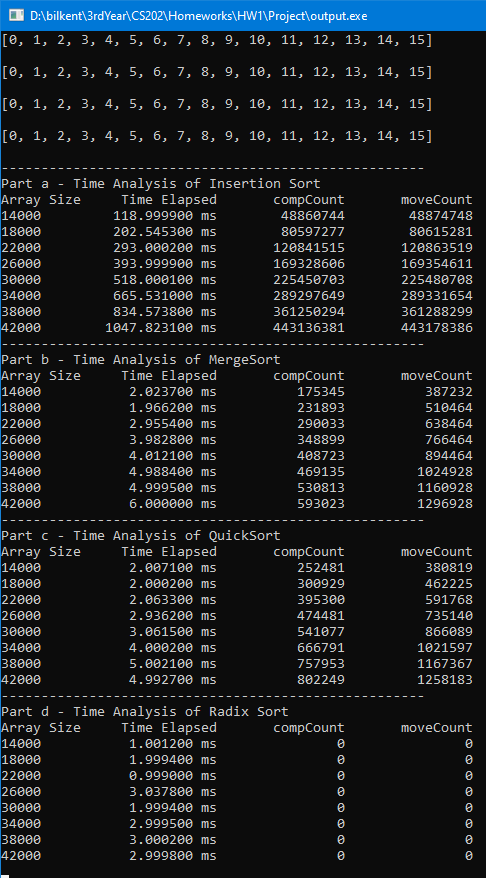
T(n) = T(1) + 4 + 6 + … + 2n

= 1 + 4 + 6 + … + 2n

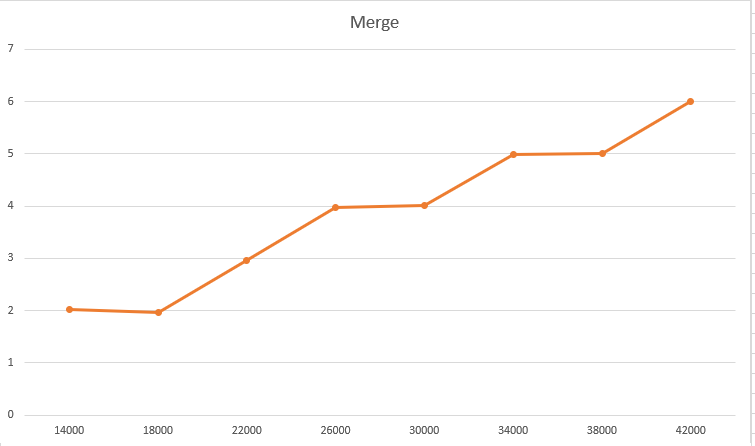
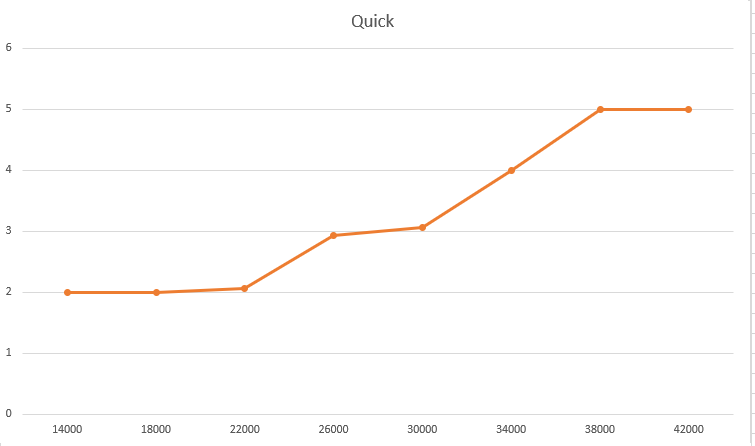
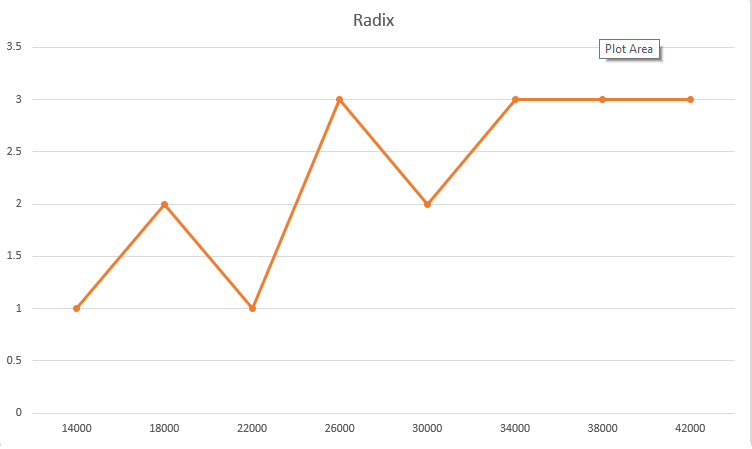
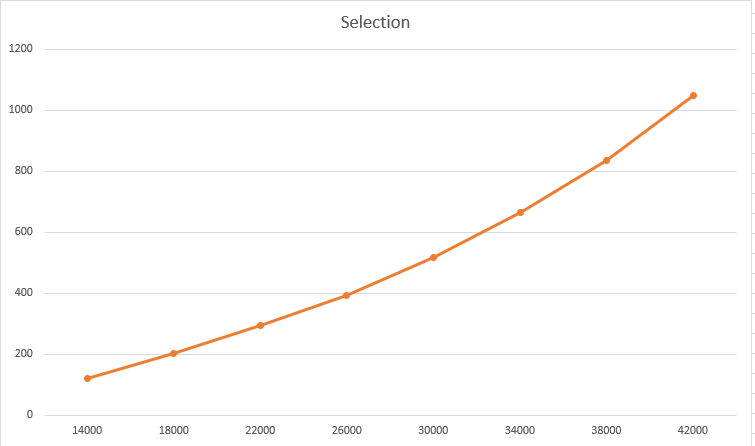
= (n(n+1)) - 1

Therefore O(T(n)) = O(n2) for worst case of Quicksort.

# Question 2

Array sizes are increased 12000 each to make time measurement possible.

# Question 3

 Graphs:

There are four separate graphs because in all-in-one graph, Selection Sort’s massive times were dominating all other algorithms.

First of all, in all measurements, results are always very close to an integer like 0,1,2. This happens because the library I used to measure time is not very precise but still the increase trend can be seen as expected.

Selection Sort has O(n2) complexity, therefore it has an upwards curve as expected. But not as sharp as y = x2, because it’s actual time function is T(n) = n2/2-n/2 therefore curve is softer than x2.

MergeSort and QuickSort are similar but QuickSort seems faster. This has to be a measuring mistake because two reasons:

1. QuickSort’s worst case is O(n2) and MergeSort’s is O(nlogn), therefore TQuickSort >= TMergeSort has to be true for all situations.
2. For all array sizes, QuickSort always have more key comparisons and moves.

But they both have close to nlogn curves.

Radix sort is pretty inaccurate unfortunately but it has a O(n) curve.