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 $0 < k <= \log_3(n)$)

$T(n) = 3T(n/3) + n$	$k = 1$
$= 9T(n/9) + 2n$	$k = 2$
$= 27T(n/27) + 3n$	$k = 3$
…	…
$T(n) = (3^k)T(n/(3^k)) + kn$	$k = \log_3(n)$

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

  $T(n) = n + kn = n + n\log_3(n)$

  Therefore $O(T(n)) = O(n\log_3 n)$

- $T(n) = 2T(n-1) + n^2$, where $T(1) = 1$.

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

  $T(n) = 2(2T(n-2) + (n-1)^2) + n^2$
  $= 4T(n-2) + 2(n-1)^2 + n^2$

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

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

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

$T(n) = 2T(n-1) + n^2$	k = 1
$= 4T(n-2) + 2(n-1)^2 + n^2$	k = 2
$= 8T(n-3) + 4(n-2)^2 + 2(n-1)^2 + n^2$	k = 3
…	…
$T(n) = 2^{n-1}T(1) + \sum_{i=1}^{k-1}(2\hat{}i)(n-i)\hat{}2$	k = n-1
$= ((2^n-4)(n-1)^2)/2 + 2^{n-1}$	

Therefore $O(T(n)) = O(n^2 2^n)$

- $T(n) = 3T(n/4) + n\log n$, 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.
  $T(n/4) = 3T(n/16) + (n/4)\log(n/4)$

  $T(n) = 3(3*T(n/16) + (n/4)\log(n/4)) + n\log n$
  $= 9T(n/16) + (3n/4)\log(n/4) + n\log n$

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

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

  Keep substituting until $n/4^k$ becomes 1. (k is an integer as $0 < k <= \log_4(n)$)

$T(n) = 3T(n/4) + n\log n$	k = 1
$= 9T(n/16) + (3n/4)\log(n/4) + n\log n$	k = 2
$= 27T(n/64) + (9n/16)\log(n/16) + (3n/4)\log(n/4) + n\log n$	k = 3
…	…
$T(n) = (3^k)T(n/4^k) + \sum_{i=1}^{k-1}(3^i * n/4\hat{}i)log(n/4\hat{}i)$	$k = \log_4(n)$

  Therefore $O(T(n)) = O(n\log^2 n)$ since $(3^k)T(n/4^k)$ is a number smaller than $n = (4^k)T(n/4^k)$ and $\log_4 n$ 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 $n/2^k$ becomes 1. (k is an integer as $0 < k <= \log_2(n)$)

$T(n) = 3T(n/2) + 1$ k = 1
$= 9T(n/2) + 3 + 1$ k = 2
$= 27T(n/64) + 9 + 3 + 1$ k = 3
… …
$T(n) = (3^k)T(n/2^k) + \sum_{i=0}^{k-1} 3^i$ $k = \log_2(n)$
$= 3^k + (3^k-1)/2$

Therefore $O(T(n)) = O(3^{\log n}) = O(n^{\log 3})$

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)

$$
\begin{aligned}
T(n) &= T(n-1) + 2n \\
&= T(n-2) + 2(n-1) + 2n \\
&\ldots \\
T(n) &= T(1) + 4 + 6 + \ldots + 2n \\
&= 1 + 4 + 6 + \ldots + 2n \\
&= (n(n+1)) - 1
\end{aligned}
$$

Therefore $O(T(n)) = O(n^2)$ for worst case of Quicksort.

Question 2

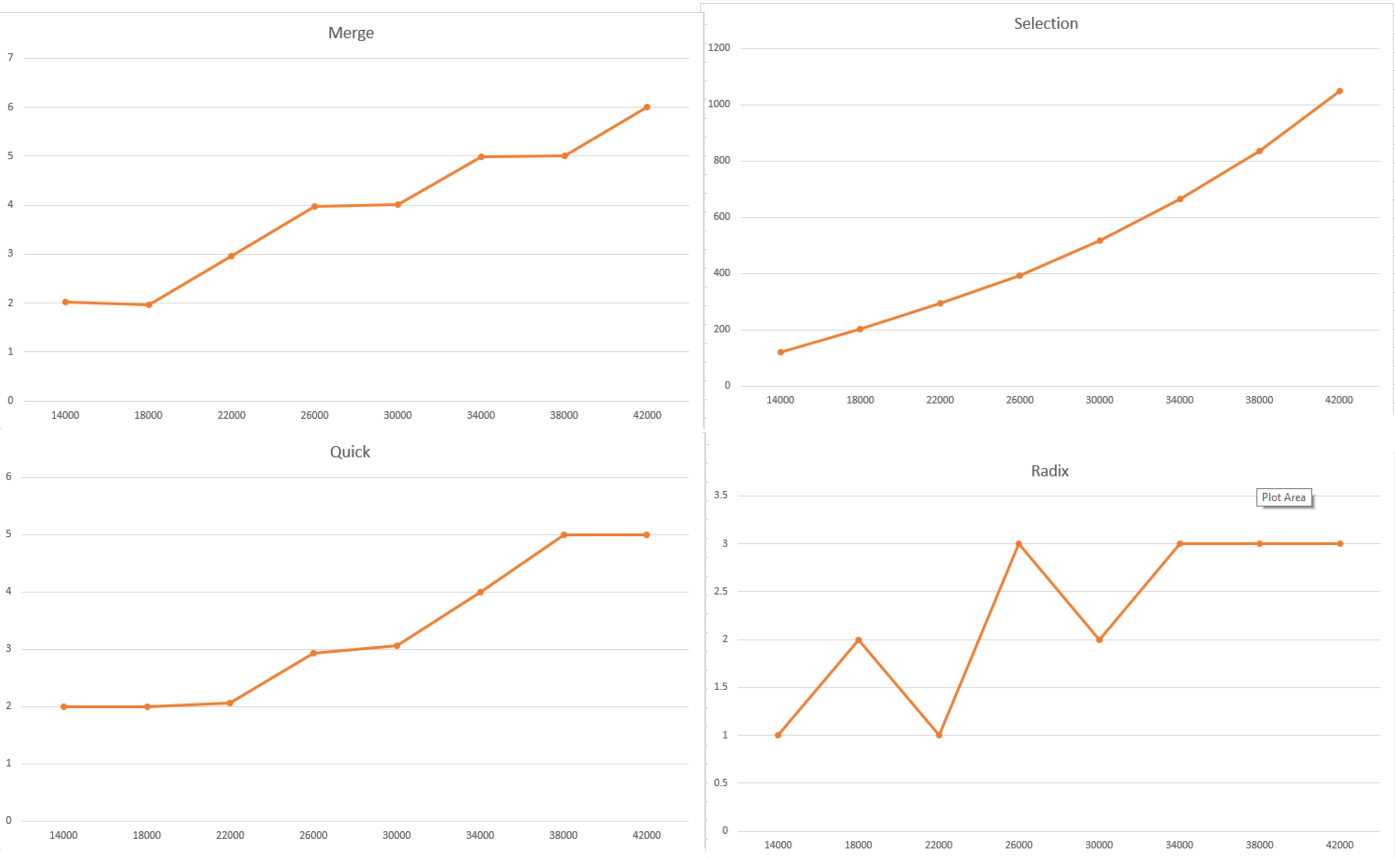
D:\bilkent\3rdYear\CS202\Homeworks\HW1\Project\output.exe

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

-----------------------------------------------------
Part a - Time Analysis of Insertion Sort
Array Size      Time Elapsed        compCount        moveCount
14000          118.999900 ms         48860744         48874748
18000          202.545300 ms         80597277         80615281
22000          293.000200 ms        120841515        120863519
26000          393.999900 ms        169328606        169354611
30000          518.000100 ms        225450703        225480708
34000          665.531000 ms        289297649        289331654
38000          834.573800 ms        361250294        361288299
42000         1047.823100 ms        443136381        443178386
-----------------------------------------------------
Part b - Time Analysis of MergeSort
Array Size      Time Elapsed        compCount        moveCount
14000            2.023700 ms           175345           387232
18000            1.966200 ms           231893           510464
22000            2.955400 ms           290033           638464
26000            3.982800 ms           348899           766464
30000            4.012100 ms           408723           894464
34000            4.988400 ms           469135          1024928
38000            4.999500 ms           530813          1160928
42000            6.000000 ms           593023          1296928
-----------------------------------------------------
Part c - Time Analysis of QuickSort
Array Size      Time Elapsed        compCount        moveCount
14000            2.007100 ms           252481           380819
18000            2.000200 ms           300929           462225
22000            2.063300 ms           395300           591768
26000            2.936200 ms           474481           735140
30000            3.061500 ms           541077           866089
34000            4.000200 ms           666791          1021597
38000            5.002100 ms           757953          1167367
42000            4.992700 ms           802249          1258183
-----------------------------------------------------
Part d - Time Analysis of Radix Sort
Array Size      Time Elapsed        compCount        moveCount
14000            1.001200 ms                0                0
18000            1.999400 ms                0                0
22000            0.999000 ms                0                0
26000            3.037800 ms                0                0
30000            1.999400 ms                0                0
34000            2.999500 ms                0                0
38000            3.000200 ms                0                0
42000            2.999800 ms                0                0
```

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 $O(n^2)$ complexity, therefore it has an upwards curve as expected. But not as sharp as $y = x^2$, because it's actual time function is $T(n) = n^2/2 - n/2$ therefore curve is softer than x^2.

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 $O(n^2)$ and MergeSort's is $O(n\log n)$, therefore $T_{QuickSort} >= T_{MergeSort}$ 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.