Chieh Lee
Partner: You Zhang
Problem Set 3
CS4800

Part I

1) $n^2 + 5 \in \Omega(n^{\log_2 2+\epsilon})$
   According to the Master Theorem, $T(n) \in \Theta(n^2)$
2) $n + 12 \in \Theta(n^{\log_4 4})$
   According to the Master Theorem, $T(n) \in \Theta(n\log n)$
3) $T(n) = \log(n) + 2 + \log\left(\frac{n}{2}\right) + 2 + \cdots + \log(1) + 2$

   $T(n) = \log(n) + (\log(n) - 1) + \cdots + 1 + 2\log n$

   $T(n) = \log^2 n - \frac{(\log n + 1)\log n}{2} + 2\log n$

   $T(n) = \frac{\log^2 n}{2} + \frac{3\log n}{2}$

   $T(n) \in \Theta(\log^2 n)$
4) $n + 3 \in O(n^{\log_2 4-\epsilon})$
   According to the Master Theorem, $T(n) \in \Theta(n^2)$
5) $5 \in \Theta(n^{\log_{10} 1})$
   According to the Master Theorem, $T(n) \in \Theta(\log n)$

Part II

1.

Majority(A)
if A.size = 1
  return A[0]
end if
half ← A.size/2
left ← Majority(A[0 … half])
right ← Majority(A[half + 1 … a.size − 1])
if left = right *then* *Return* left
lcount ← 0
rcount ← 0
for i = 0 *to* A.size − 1
  if A[i] = left *then* lcount ← lcount + 1
  if A[i] = right *then* rcount ← rcount + 1

$end\ for$
if lcount $\geq$ half $then$ $Return$ left
if rcount $\geq$ half $then$ $Return$ right
$Return\ NULL$

The function can be represented as $T(n) = 2T\left(\frac{n}{2}\right) + n$ which according to the Master Theorem is Θ(N log N)

2.
Divide the series into half the process the both left and right.
Then divide left and right into another half until there's only <= 2 point.
And calculate lost and gain for all the divided portions.

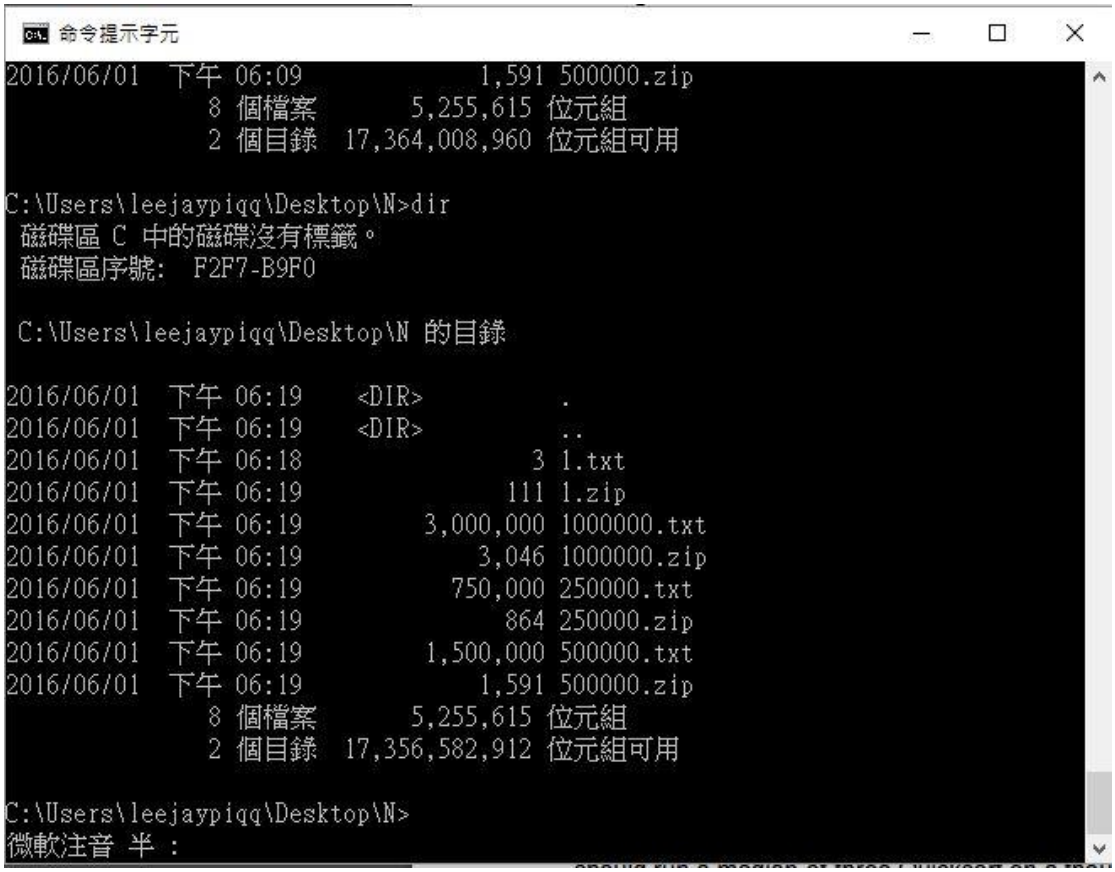
Combine all the result to get the final lost and gain.

3.

4.
The assumption is each swap will make the list more like to be disorder. A sorted list will be ideal worst case and this assumption is very close to the real situation when p, k is not very large.

Insertion sort will finish sorting elements in k*p*n of swapping, due to the insertion sort has to compare all elements and swap if necessary. Therefore the comparison should be k*p*n + n, hence O((1+kp)n)

Part III



1) The compress file used deflate algorithm that uses Huffman Coding for compress a file. Huffman Coding is known as to reduce a bit for character base on how often it appears, rather than 8 bit for all character, in order to achieve compressing a file. In order to achieve Huffman Coding, the algorithm will build a Huffman Tree and give a different character code.
   In order to unzip the file, we have to rebuild the Hoffman Tree to revert those compress code to normal 8 bit code. Therefore we need to add those weighted value while compressing a file for rebuilding Hoffman Tree.
   Therefore, the extra byte might contain the information we need to rebuild the Huffman Tree and revert compressed value to its normal 8 bit form.
2) “foo” repeat 1,000,000 times.
   Assuming run-length encoding encodes repetitions of strings instead of repetitions of individual bits. And we use 8 bit representation so 255 (11111111) is the maximum number that a section can handle
   Original 1,000,000 repeated “foo” file is 3,000,000 byte
   1,000,000 / 255 = 3921.56, round up = 3922;
   Therefore, 3,922 byte (3,922 of (11111111), the last byte is rounded down)
   Plus 3,922 * 3 (3,922 section times string “foo” which is 3 byte)
   = 3,922 + (3,922 * 3)
   = 15,688

Which is the size is far greater than the file output from my zip utility, which is

3,046 byte

3)

Part IV

I run both QuickSort and Bucketsort 3 times each for comparison. The measure unit is represent in rounded milliseconds (although I use System.nanoTime() for measuring).

1st Try

	10	1,000	100,000	1,000,000
QuickSort	0.040	2.729	57.340	379.069
Bucketsort	1.089	4.384	54.147	330.615

2nd Try

	10	1,000	100,000	1,000,000
QuickSort	0.036	2.904	62.310	353.221
Bucketsort	1.056	4.289	42.417	328.368

3rd Try

	10	1,000	100,000	1,000,000
QuickSort	0.392	2.821	60.403	350.650
Bucketsort	1.030	4.422	41.231	327.113

Both sorting algorithm are very fast even N = 1 million, both algorithm can still sort the list within 1 second.
Bucketsort appears slower when the N size is smaller, almost 2-3 times slower than quicksort. However, when the N size come larger, N = 100,000 and 1,000,000, bucketsort works faster than quicksort.
The reason is because when the N size get larger, elements are more equally distributed to each bucket, so the running time will be more close to linear (N)