1. Order the following functions by their asymptotic behavior. Circle the groups of functions that are Big-$\Theta$ to each other.

\[
\begin{align*}
    a(x) &= x & b(x) &= \sum_{i=0}^{x} 2^i & c(x) &= 2^x \\
    d(x) &= 3^x & e(x) &= x \log x & f(x) &= \sum_{i=1}^{x} i \\
    g(x) &= x^{1.00001} & h(x) &= x^2 & i(x) &= \sqrt{x} \\
    j(x) &= \log_2 3^x & k(x) &= x^{1/\log x} & \ell(x) &= \log x 2^x
\end{align*}
\]

**Answer:** First we should simplify some of the functions.

\[
\begin{align*}
    b(x) &= 2^{x+1} - 1 \\
    f(x) &= \frac{x(x+1)}{2} \\
    j(x) &= (\log_2 3)x \\
    k(x) &= (\log x)^{1/\log x} = 2^1 = 2 \\
    \ell(x) &= \log x + \log 2^x = \log x + x
\end{align*}
\]

Second we should apply our heuristics: logarithms are smaller than polynomials, polynomials are smaller than exponentials, order polynomials by their exponent, and order exponentials by their base.

Polynomials:
\[
i(x) - a(x) \quad j(x) \quad \ell(x) - g(x) \quad h(x) \quad f(x)
\]

Exponentials:
\[
b(x) \quad c(x) - d(x)
\]

Leftover are $e(x)$ and $k(x)$. The function $k(x)$ is a constant. So $\lim_{x \to \infty} \frac{k(x)}{i(x)} = 0$ and we should put $k(x)$ at the very beginning. The function $e(x) = x \log x$ and we can compare it to $a(x)$ and $g(x)$.
\[
\lim_{x \to \infty} \frac{a(x)}{e(x)} = \lim_{x \to \infty} \frac{1}{\log x} = 0
\]

\[
\lim_{x \to \infty} \frac{e(x)}{g(x)} = \lim_{x \to \infty} \frac{\log x}{x^{0.00001}} = \lim_{x \to \infty} \frac{1/x}{0.00001x^{0.00001} - 1} = \lim_{x \to \infty} \frac{1}{0.00001x^{0.00001}} = 0
\]

Therefore the correct ordering and grouping of the functions is:

\[k(x) - i(x) - a(x) - j(x) - \ell(x) - e(x) - g(x) - h(x) - f(x) - b(x) - c(x) - d(x)\]

2. For each sorting algorithm below what order of the numbers from 1 to 8 minimizes the number of comparisons performed? What order maximizes the number of comparisons performed? How many comparisons are performed on each list?

- Insertion sort
- Merge sort

Answer:

- Insertion, Minimum: 1,2,3,4,5,6,7,8 causes 7 comparisons
- Insertion, Maximum: 8,7,6,5,4,3,2,1 causes 28 comparisons
- Merge, Minimum: 1,2,3,4,5,6,7,8 causes 12 comparisons
- Merge, Maximum: 1,3,2,8,4,6,5,7 causes 17 comparisons

There may be multiple answers for the merge sort inputs.

3. For each of the following problems decide whether or not the problem can be solved faster on sorted rather than unsorted data (i.e. does sorting the data speed up the time to solve the problem):

- Searching for a specific value in the array
- Finding the mean of an array of ints
- Finding the median of an array of ints
- Finding the mode of an array of ints

Answer:

- Searching for a specific value in the array: On sorted data we can perform binary search which takes \(O(\log n)\) time, but on unsorted data we need to scan through every element taking \(O(n)\) time. Therefore sorting speeds up the solution.
- Finding the mean of an array of ints: To compute the mean of an array, we need to sum up every entry. Whether or not the data is sorted doesn’t help.
Finding the median of an array of ints: If the data is sorted, the median element is simply at the middle index. On unsorted data the median can be found using an algorithm called quick select in $O(n)$ time. So finding the median is much faster for sorted data.

Finding the mode of an array of ints: For sorted data, all equal elements appear consecutively. Therefore the mode of the data can be found by counting the longest run during one scan through the list which takes $O(n)$ time. Naively it seems difficult to find the mode for unsorted data. However a data structure known as a hash map that we use later in this course will enable an $O(n)$ time solution that works for unsorted data. Asymptotically sorting the data does not help for speeding up finding the mode.

4. Recall an inversion is a pair of elements whose current order is opposite of sorted. As discussed in class, the number of inversions in an array can be counted using insertion sort in $O(n^2)$ time. Modify merge sort to count the number of inversions in an array in $O(n \log n)$ time.

Answer:
Merge sort fixes inversions inside the merge function. Whenever an element from the right half of the list is smaller than the front of the left half of the list, that right half element was out of order with every element remaining in the left hand list. So we can create a global inversion counter and during merge whenever the right half has the smaller element, add the remaining size of the left half to the counter. In code:

```java
merge(T[] left, T[] right):
    T merged[length(left) + length(right)]
    i=0, j=0, k=0
    while i < length(left) and j < length(right):
        if left[i] < right[j]:
            merged[k] = left[i]
            i++, k++
        else:
            merged[k] = right[j]
            j++, k++
            inversion_counter+=(length(left) - i)
    while i < length(left):
        merged[k] = left[i]
        i++, k++
    while j < length(right):
        merged[k] = right[j]
        j++, k++
    return merged
```