Problem 1. For the following functions, state either $f(n) \in o(g(n))$, $f(n) \in \omega(g(n))$, or $f(n) \in \Theta(g(n))$.

a. $f(n) = (n + \log n)(\sqrt{n} + 5)$, $g(n) = n \log n$

b. $f(n) = 2^{n \lg n}$, $g(n) = 3^n$

c. $f(n) = \frac{n}{\lg n}$, $g(n) = \sqrt{n}$

d. $f(n) = \Sigma_{i=1}^{n}(3i^2 \log i + 2i(\log i)^2)$, $g(n) = n^3(\log n)^3$

e. $f(n) = \frac{n}{n^{1/2}}$, $g(n) = \sqrt[4]{n^2}$

Problem 2. Solve the following recurrences using summations, recursion trees, and/or the master method.

a. $T(n) = 3T(\frac{1}{4}n) + 1.5^n$

b. $T(n) = 2T(\frac{1}{2}n) + \log n$

c. $T(n) = T(\frac{3}{5}n) + \sqrt[3]{n}$

d. $T(n) = T(n-1) + n^2 + 3n + 2$

e. $T(n) = 2T(\frac{1}{4}n) + \sqrt{n}$

f. $T(n) = T(\frac{1}{3}n) + T(\frac{2}{3}n) + n$

Problem 3. Here we examine a sorting algorithm for vectors based on swapping elements that are out of order. The first function, naturally enough, swaps elements that are out of order.

```
(define (sort!-swap-if-needed v i j)
  (define v.i (vector-ref v i))
  (define v.j (vector-ref v j))
  (cond
    [(<= v.i v.j) (void)]
    [else
     (vector-set! v i v.j)
     (vector-set! v j v.i)]))
```

a. State the running time of `sort!-swap-if-needed` as a recurrence. Solve the recurrence using summations, recursion trees, or the master method.

The `sort!-from/to` function sorts elements that are swapped from position `i` to position `j`, incrementing `j` until the end of the vector.

```
(define (sort!-from/to v i j)
  (cond
    [(>= j (vector-length v)) (void)]
    [else
     (sort!-swap-if-needed v i j)
     (sort!-from/to v i (add1 j))]))
```

b. State the running time of `sort!-from/to` as a recurrence. Solve the recurrence using summations, recursion trees, or the master method.

The `sort!-from` function sorts elements that are swapped from position `i` to each position at a greater index; `i` increments until the end of the vector. Finally, `sort!` calls `sort!-from` starting at index 0.

```
(define (sort! v)
  (sort!-from v 0))

(define (sort!-from v i)
  (cond
    [(>= i (vector-length v)) (void)]
    [else
     (sort!-from/to v i (add1 i))
     (sort!-from v (add1 i))]))
```

c. State the running time of `sort!` as a recurrence. Solve the recurrence using summations, recursion trees, or the master method.

Problem 4. The *median of medians* algorithm selects the ith smallest element of a list xs, much like quickselect. The implementation starts with partitioning functions.

```
(define (all-less-than pivot xs)
  (cond
    [(empty? xs) empty]
    [(< (first xs) pivot) (cons (first xs) (all-less-than pivot (rest xs)))]
    [else (all-less-than pivot (rest xs))]))

(define (all-greater-than pivot xs)
  (cond
    [(empty? xs) empty]
    [(> (first xs) pivot) (cons (first xs) (all-greater-than pivot (rest xs)))]
    [else (all-greater-than pivot (rest xs))]))
```

a. State the running time of all-less-than and all-greater-than as recurrences. Solve the recurrences using summations, recursion trees, or the master method.

The groups-of-five function splits a list into groups of five (or fewer) elements.

```
(define (groups-of-five xs)
  (cond
    [(empty? xs) empty]
    [(empty? (rest xs)) (list (list (first xs)))]
    [(empty? (rest (rest xs))) (list (list (first xs) (second xs)))]
    [(empty? (rest (rest (rest xs))))
     (list (list (first xs) (second xs) (third xs)))]
    [(empty? (rest (rest (rest (rest xs)))))
     (list (list (first xs) (second xs) (third xs) (fourth xs)))]
    [else (cons (list (first xs) (second xs) (third xs) (fourth xs) (fifth xs))
            (groups-of-five (rest (rest (rest (rest (rest xs)))))))]))
```

b. State the running time of groups-of-five as a recurrences. Solve the recurrence using summations, recursion trees, or the master method.

The median function uses slow-select to find the median element of a list. Assume that slow-select runs in $n \log n$ time. The medians-of-five function takes a list of lists, where each inner list has at most five elements. It produces a list of numbers, where each number is the median of the corresponding list in the input.

```
(define (median five-or-less)
  (slow-select (quotient (length five-or-less) 2) five-or-less))

(define (medians-of-five fives)
  (cond
    [(empty? fives) empty]
    [else (cons (median (first fives))
                (medians-of-five (rest fives)))]))
```

c. State the running time of medians-of-five as a recurrence. Solve the recurrence using summations, recursion trees, or the master method.

Finally, the select function itself operates like quickselect, subdividing the input by partitioning. In order to guarantee a good pivot, the algorithm splits the input into groups of five elements, finds the median of each group of five, then uses a recursive call to select to find the median of all of those medians. That *median of medians* serves as the pivot for partitioning the input.

```
(define (select i xs)
  (define n (length xs))
  (cond
    [(<= n 5) (slow-select i xs)]
    [else
     (define fives (groups-of-five xs))
     (define medians (medians-of-five fives))
     (define pivot (select (quotient n 10) medians))
     (define left (all-less-than pivot xs))
     (define right (all-greater-than pivot xs))
     (define n1 (length left))
     (define n2 (length right))
     (cond
       [(< i n1) (select i left)]
       [(>= i (- n n2)) (select (- i (- n n2)) right)]
       [else pivot])]))
```

d. Argue that the length of (medians-of-five (groups-of-five xs)) is at most $\lceil \frac{1}{5}n \rceil$.

e. Argue that there are no more than $\frac{7}{10}n$ elements in `left`.

f. Argue that there are no more than $\frac{7}{10}n$ elements in `right`.

g. State the running time of `select` as a recurrence. **Note:** this recurrence may have an unusual form, as not every recursion in `select` has the same worst-case input size.

h. Solve the recurrence for the running time of `select` using summations, recursion trees, or the master method.