Name: ______________________________

Date: __________________________

CS4800: Algorithms and Data

Exam 1: Version A

Exam Policies. Your work on this exam must be your own. You *may not* copy solutions from other students. You *may not* communicate with other students during the exam. You *may not* communicate with anyone outside the class during the exam.

This exam is open book, open notes. You *may* refer to the text, prior homeworks, past exams, practice exams, your own notes, and anything else you brought with you on paper. You *may not* share notes or any other written materials with other students during the exam. You *may not* use electronic devices during the exam.

The contents of the exam are confidential. You *may not* discuss the contents of the exam with any student who has not yet taken the exam. Due to illness, travel, or other special circumstances, some students may take exams at different times than others.

You *must* complete the exam in the time provided. Your exam *will not* be accepted if you work past the deadline.

If you violate any of these rules, you will receive a zero as your grade for this exam and you will be reported to the Office of Student Conduct and Conflict Resolution (OSCCR).

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. $f(n) = n^2\sqrt{n} + 8^{\log_2 n},\ g(n) = n^{2.75}$

b. $f(n) = n^n,\ g(n) = n!$

c. $f(n) = 2^{(n^2)},\ g(n) = (2^n)^2$

d. $f(n) = \sum_{i=1}^{n} 2^i,\ g(n) = 2^n$

e. $f(n) = n\sqrt{\log n},\ g(n) = n \log \sqrt{n}$

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

a. $T(n) = 8T(n/2) + n^3$

b. $T(n) = 2T(n/8) + \sqrt{n}$

c. $T(n) = 4T(n/3) + n$

d. $T(n) = 3T(n/9) + 2^{\frac{\log_2 n}{2}}$

e. $T(n) = T(n/2) + T(n/3) + T(n/6) + 1$

f. $T(n) = T(n-10) + \frac{6}{100}n^2$

Problem 3. Selection sort determines the first element of a sorted list at each step. Here we consider a variant that determines both the first and last element of a sorted list at each step.

The `move-least-to-front` and `move-greatest-to-front` functions rearrange the elements of their input to contain its least or greatest element, respectively, before all other elements.

```
(define (move-least-to-front xs)
  (cond
    [(empty? (rest xs)) xs]
    [else
     (define least-first (move-least-to-front (rest xs)))
     (cond
       [(<= (first xs) (first least-first)) (cons (first xs) least-first)]
       [else (cons (first least-first) (cons (first xs) (rest least-first)))])]))

(define (move-greatest-to-front xs)
  (cond
    [(empty? (rest xs)) xs]
    [else
     (define greatest-first (move-greatest-to-front (rest xs)))
     (cond
       [(>= (first xs) (first greatest-first)) (cons (first xs) greatest-first)]
       [else (cons (first greatest-first) (cons (first xs) (rest greatest-first)))])]))
```

a. As `move-least-to-front` and `move-greatest-to-front` differ only in the use of `<=` versus `>=`, their running times are the same. State the running time of these functions as a recurrence.

b. Solve the recurrence using summations, recursion trees, or the master method.

The `bsort/tail` function takes a list of numbers `xs` as its first argument. Its second argument, `tail`, is a sorted list of numbers whose elements are all greater than the elements of `xs`. Its result is a sorted list containing all the elements of `xs` and `tail`. Finally, `bsort` sorts a list `xs` using `bsort/tail` and an empty tail.

```
(define (bsort/tail xs tail)
  (cond
    [(empty? xs) tail]
    [(empty? (rest xs)) (cons (first xs) tail)]
    [else
     (define least-first (move-least-to-front xs))
     (define greatest-first (move-greatest-to-front (rest least-first)))
     (cons (first least-first)
       (bsort/tail (rest greatest-first)
         (cons (first greatest-first) tail)))]))

(define (bsort xs)
  (bsort/tail xs empty))
```

c. State the running time of `bsort/tail` as a recurrence in terms of the length of `xs`.

d. Restate the running time of `bsort/tail` as a new recurrence in terms of *half* of the length of `xs`. Since the length of `xs` decreases by two at each step, this recurrence may be easier to solve.

e. Solve the second recurrence using summations, recursion trees, or the master method.

f. State the asymptotic running time of `bsort` in terms of the length of `xs`.

Problem 4. Not every divide-and-conquer algorithm starts with a large input and gradually splits it up into smaller ones. Sometimes, an implementation starts with many small pieces and gradually builds them up into one large result. In this problem, we examine a sorting algorithm that takes this approach.

The `make-singletons` function creates a singleton list for each element of its input.

```
(define (make-singletons xs)
  (cond
    [(empty? xs) empty]
    [else
     (define singleton (cons (first xs) empty))
     (cons singleton (make-singletons (rest xs)))]))
```

a. State the running time of `make-singletons` as a recurrence.

b. Solve the recurrence using summations, recursion trees, or the master method.

The `merge` function constructs a single sorted list out of the elements of two sorted lists.

```
(define (merge xs ys)
  (cond
    [(empty? xs) ys]
    [(empty? ys) xs]
    [else
     (define x (first xs))
     (define y (first ys))
     (cond
       [(<= x y) (cons x (merge (rest xs) ys))]
       [else (cons y (merge xs (rest ys)))])]))
```

c. State the running time of `merge` as a recurrence of one variable. In addition, state what the variable represents in terms of the function's two inputs `xs` and `ys`.

d. Solve the recurrence using summations, recursion trees, or the master method.

The `merge-pairs` function takes a list of lists as input. It returns a list of results of calling `merge` on each consecutive pair of lists.

```
(define (merge-pairs xss)
  (cond
    [(empty? xss) xss]
    [(empty? (rest xss)) xss]
    [else (cons (merge (first xss) (second xss))
                (merge-pairs (rest (rest xss))))]))
```

e. Argue that the length of `(merge-pairs xss)` is at most $\left\lceil \frac{\texttt{(length xss)}}{2} \right\rceil$.

f. Argue that the *total length* of the elements of `(merge-pairs xss)` is the same as the total length of the elements of `xss`.

g. State the running time of `merge-pairs` as a recurrence of one variable. In addition, state what the variable represents in terms of the function's input `xss`.

h. Solve the recurrence using summations, recursion trees, or the master method.

The function `merge-all` takes a list of sorted lists as inputs. It repeatedly merges all the pairs of lists until only one remains; it then returns that sorted list.

```
(define (merge-all xss)
  (cond
    [(empty? xss) empty]
    [(empty? (rest xss)) (first xss)]
    [else (merge-all (merge-pairs xss))]))
```

i. Argue that the length of `xss` always decreases in recursive calls to `merge-all`.

j. Argue that the *total length* of the elements of `xss` does not change in recursive calls to `merge-all`.

k. State the running time of `merge-all` as a recurrence of two variables. One variable must represent the total length of the elements of `xss`, and one variable must represent the length of `xss` itself.

l. Solve the recurrence using summations, recursion trees, or the master method. Note that because only one variable in the recurrence changes on recursive calls, you can treat the other variable as a constant when solving the recurrence.

Finally, the function `msort` uses `merge-all` and `make-singletons` to sort its input.

```
(define (msort xs)
  (merge-all (make-singletons xs)))
```

m. Argue that `msort` passes a list of sorted lists as the argument to `merge-all`.

n. State the asymptotic running time of `msort` in terms of one variable. In addition, state what the variable represents in terms of the function's input `xs`.