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

Solution: We can simplify the $8^{\log_2 n}$ term: $8^{\log_2 n} = (2^3)^{\log_2 n} = 2^{3\log_2 n} = (2^{\log_2 n})^3 = n^3$. Therefore both f and g are polynomials; the degree of f is 3 while the degree of g is 2.75. Therefore $f \in \omega(g)$.

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

Solution: In general, $n^n \gg n! \gg b^n \gg n^a$ for any constants $b > 1$ and $a > 0$, where we take $x \gg y$ to mean "x grows asymptotically faster than y". Therefore $f \in \omega(g)$.

To prove this formally, given any constants $c > 0$ and $n_0 \geq 0$, we will find $n > n_0$ such that $f(n) > cg(n)$.

$$\begin{array}{rcll}
n^n & > & cn! & \text{by definition of } f \text{ and } g \\
\prod_{i=1}^{n} n & > & c\prod_{i=1}^{n} i & \text{using product notation} \\
(\prod_{i=1}^{\lfloor n/2\rfloor} n)(\prod_{i=\lfloor n/2\rfloor+1}^{n} n) & > & c(\prod_{i=1}^{\lfloor n/2\rfloor} i)(\prod_{i=\lfloor n/2\rfloor+1}^{n} i) & \text{split both products at } \lfloor n/2\rfloor\text{, if } n \geq 2 \\
(\prod_{i=1}^{\lfloor n/2\rfloor} n)(\prod_{i=\lfloor n/2\rfloor+1}^{n} n) & > & c(\prod_{i=1}^{\lfloor n/2\rfloor} n/2)(\prod_{i=\lfloor n/2\rfloor+1}^{n} n) & \text{tighter bound for the right-hand side} \\
\prod_{i=1}^{\lfloor n/2\rfloor} n & > & c\prod_{i=1}^{\lfloor n/2\rfloor} n/2 & \text{divide by common factor} \\
\prod_{i=1}^{\lfloor n/2\rfloor} n & > & \frac{c}{2^{\lfloor n/2\rfloor}}\prod_{i=1}^{\lfloor n/2\rfloor} n & \text{pull constant factors out of product} \\
1 & > & \frac{c}{2^{\lfloor n/2\rfloor}} & \text{divide by common factor} \\
2^{\lfloor n/2\rfloor} & > & c & \text{multiply by } 2^{\lfloor n/2\rfloor} \\
\lfloor n/2\rfloor & > & \lg c & \text{logarithm base 2} \\
\frac{n-1}{2} & > & \lg c & \text{tighter bound for the left-hand side} \\
n & > & 2\lg c + 1 & \text{multiply both sides by 2, then add 1}
\end{array}$$

We thus establish $f \in \omega(g)$ by choosing $n = \max(2, n_0, 2\lg c + 2)$.

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

Solution: In general, $k^{p(n)} \gg k^{q(n)}$ for any constant k if $p(n) > q(n)$ for all sufficiently large n, where we take $x \gg y$ to mean "x grows asymptotically faster than y". Since $f(n) = 2^{n^2}$ and $g(n) = 2^{2n}$, and $n^2 > 2n$ for all $n > 2$, we therefore know that $f \in \omega(g)$.

To prove this formally, given any constants $c > 0$ and $n_0 \geq 0$, we will find $n > n_0$ such that $f(n) > cg(n)$.

$$\begin{array}{rcll}
2^{n^2} & > & c(2^n)^2 & \text{by definition of } f \text{ and } g \\
2^{n^2} & > & c2^{2n} & \text{by properties of exponentiation} \\
n^2 & > & 2n + \lg c & \text{logarithm base 2} \\
4n & > & 2n + \lg c & \text{tighter bound for the left-hand side, if } n \geq 4 \\
n & > & \frac{1}{2}\lg c & \text{subtract } 2n \text{ from both sides, then divide by 2}
\end{array}$$

We thus establish $f \in \omega(g)$ by choosing $n = \max(4, n_0, \frac{1}{2}\lg c)$.

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

Solution: We can simplify $f(n)$ to $2^{n+1} - 1$, which is between $g(n)$ and $2g(n)$. Therefore $f \in \Theta(g)$.

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

Solution: We can simplify $f(n)$ to $n(\log n)^{\frac{1}{2}}$ and $g(n)$ to $\frac{1}{2}n(\log n)$. The constant factor $\frac{1}{2}$ does not matter, and both f and g have the same polynomial degree of 1. Therefore the deciding factor is the "polylogarithmic" degree: f has a factor of $(\log n)^{\frac{1}{2}}$, whereas g has a factor of $(\log n)^1$. Therefore $f \in o(g)$.

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

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

Solution: Proceeds via the master method. Case 2 applies, as $n^{\log_b a} = n^{\log_2 8} = n^3$ and $f(n) = n^3 \in \Theta(n^3)$. Therefore $T(n) \in \Theta(n^3 \log n)$.

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

Solution: Proceeds via the master method. In this case, $n^{\log_b a} = n^{\log_8 2} = n^{1/3}$ and $f(n) = \sqrt{n} = n^{1/2}$. Using $\epsilon = 1/6$, we can show that $f(n) \in \Omega(n^{1/2}) = \Omega(n^{1/3+\epsilon})$. In order to use case 3, we must also show that $af(n/b) < cf(n)$ for $c < 1$ and sufficiently large n.

$$\begin{aligned}
2\sqrt{n/8} &< c\sqrt{n} && \text{by definition of } f \\
\tfrac{2}{\sqrt{8}}\sqrt{n} &< c\sqrt{n} && \text{split numerator and denominator of } \sqrt{\ } \\
\tfrac{1}{\sqrt{2}}\sqrt{n} &< c\sqrt{n} && \text{simplify constant factor} \\
\tfrac{1}{\sqrt{2}} &< c && \text{divide both sides by } \sqrt{n}
\end{aligned}$$

We can therefore choose $c = 0.8 > \frac{1}{\sqrt{2}} \approxeq 0.707$, and thus establish that $T(n) \in \Theta(n^{1/2})$.

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

Solution: Proceeds via the master method. Case 1 applies with $\epsilon = 1/4$, as $n^{\log_b a} = n^{\log_3 4} \approxeq n^{1.26}$ and $f(n) = n \in O(n^{(\log_3 4)-\epsilon}) \approxeq O(n^{1.01})$. Therefore $T(n) \in \Theta(n^{\log_3 4}) \approxeq \Theta(n^1.26)$.

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

Solution: We can simplify the $2^{\frac{\log_2 n}{2}}$ term: $2^{\frac{\log_2 n}{2}} = 2^{\frac{1}{2}\log_2 n} = (2^{\log_2 n})^{1/2} = n^{1/2} = \sqrt{n}$. Our solution proceeds from here via the master method. Case 2 applies, as $n^{\log_b a} = n^{\log_9 3} = n^{1/2} = \sqrt{n}$ and $f(n) = \sqrt{n} \in \Theta(\sqrt{n})$. Therefore, $T(n) \in \Theta(\sqrt{n}\log n)$.

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

Note: We do not have a tight Θ-bound using techniques taught in class. This was discovered after returning graded exams, so everyone will get **1 point** back on this question, as there is no correct final answer. The points for intermediate work are unchanged, as everything but the final answer can be worked out using recursion trees and summations.

Solution: Proceeds via recursion trees. Each node contributes 1 toward the total work, and each has 3 children: one taking 1/2 of the input, one taking 1/3 of the input, and one taking 1/6 of the input. Note that since the work done at each node does not vary with input size, the input size at each node only matters in that it determines the tree's height. The shallowest leaf of the tree is reached by taking 1/6 of the input at each step, while the deepest leaf is reached by taking 1/2 of the input at each step. The tree's height is therefore between $\log_6 n$ and $\log_2 n$. Each layer of the tree has at most 3 times as many nodes as the last. We can therefore place lower and upper bounds on the work in this recurrence:

$$\sum_{i=0}^{\log_6 n} 3^i \leq T(n) \leq \sum_{i=0}^{\log_2 n} 3^i$$

We can simplify both:

$$\begin{array}{lll l}
& \sum_{i=0}^{\log_6 n} 3^i & \sum_{i=0}^{\log_2 n} 3^i & \\
= & \frac{3^{(\log_6 n)+1}-1}{3-1} & = \frac{3^{(\log_2 n)+1}-1}{3-1} & \text{by properties of geometric series} \\
= & \frac{3}{2}(3^{\log_6 n}) - \frac{1}{2} & = \frac{3}{2}(3^{\log_2 n}) - \frac{1}{2} & \\
= & \frac{3}{2}(3^{\frac{\log_3 n}{\log_3 6}}) - \frac{1}{2} & = \frac{3}{2}(3^{\frac{\log_3 n}{\log_3 2}}) - \frac{1}{2} & \text{by properties of logarithms} \\
= & \frac{3}{2}(n^{\frac{1}{\log_3 6}}) - \frac{1}{2} & = \frac{3}{2}(n^{\frac{1}{\log_3 2}}) - \frac{1}{2} & \text{by properties of logarithms}
\end{array}$$

Therefore we know that $T(n) \in \Omega(n^{\frac{1}{\log_3 6}}) \approxeq \Omega(n^{0.613})$ and $T(n) \in O(n^{\frac{1}{\log_3 2}}) \approxeq O(n^{1.58})$.

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

Solution: Proceeds by summation, using a rewritten recurrence based on $n/10$.

$$T'(m) = T'(m-1) + \frac{6}{100}(10m)^2 = T'(m-1) + 6m^2$$

We can state and solve this recurrence as a summation.

$$T'(m) = \sum_{i=1}^{m} 6i^2 = 6\sum_{i=1}^{m} i^2 = m(m+1)(2m+1) = 2m^3 + 3m^2 + m \in \Theta(m^3)$$

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.

Solution: $T(n) = T(n-1) + 1$

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

Solution: $T(n) = \sum_{i=1}^{n} 1 = n \in \Theta(n)$

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.

Solution: $T(n) = T(n-2) + n$

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.

Solution: $T'(m) = T(m-1) + 2m$

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

Solution: $T'(m) = \sum_{i=1}^{m} 2i = 2\sum_{i=1}^{m} i = m^2 + m \in \Theta(m^2)$

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

Solution: $T(n) \in \Theta(n^2)$

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.

Solution: $T(n) = T(n-1)+1$

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

Solution: $T(n) = \sum_{i=1}^{n} 1 = n \in \Theta(n)$

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`.

Solution: Here, n represents the sum of the lengths of `xs` and `ys`. At any given step, either one may decrease, and thus the total must decrease. If we used just `xs` or just `ys`, or if we chose the larger or smaller of `xs` and `ys`, there would be some steps where n would not decrease.
$T(n) = T(n-1)+1$

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

Solution: $T(n) \in \Theta(n)$, as before.

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 $\left\lceil \frac{\texttt{(length xss)}}{2} \right\rceil$.

Solution: For every two consecutive lists in xss, merge-pairs produces one list in its output, thus halving the length of the list except for any odd-numbered remainder.

Formally, we could prove this by induction, with base cases when xss has 0 or 1 elements and an inductive step for $n+2$ elements. The clauses of the induction would therefore mirror the clauses of the definition exactly, and the inductive hypothesis for n when reasoning about $n+2$ would correspond to the recursive call.

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.

Solution: First, we must note that the length of (merge xs ys) is the same as the sum of the lengths of xs and ys; each time merge takes an element from either xs or ys, it adds that element to its output. Second, we can see that each time merge-pairs takes a first and second element off of its input, it adds the result of merge-ing those elements to its output. Therefore the total length of the elements of the input is preserved as the total length of the elements of the output.

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.

Note: This problem is flawed, and was not graded. The precise running time of merge-pairs is not best stated as a recurrence of one variable. If we take the length of xss as our variable, the amount of work at each step varies arbitrarily. If we take the total length of each element of xss as our variable, the size of the input decreases by arbitrary amounts at each step. The running time is best stated in closed form in terms of the total length of each element of xss.

Solution: If take k to be the total length of the elements of xss, the running time of merge-pairs is simply:

$$T(k) = k$$

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

Note: This problem is based on 4g, and so was not graded.

Solution: Using the second "recurrence" above, based on k, $T(k) \in \Theta(k)$.

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`.

Solution: As argued in 4e, the length of `(merge-pairs xss)` cuts in half, rounded up. For any length of two or more, this is a decrease. Since `merge-all` only recurs when its input has two or more elements, the length of its input always decreases in recursive calls.

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

Solution: As argued in 4f, the total length of `(merge-pairs xss)` is the same as the total length of `xss`. Therefore the total length 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.

Note: This problem is based on 4g, and so was not graded.

Solution: Here, we take n to be the length of `xss` and k to be the total length of its elements. $T(n,k) = T(n/2,k)+k$

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.

Note: This problem is based on 4g, and so was not graded.

Solution: Proceeds by recursion trees. The "tree" is a straight line with $\log_2 n$ nodes and k work at each. Therefore the total work is $T(n,k) \in \Theta(k \log n)$.

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`.

Solution: The `make-singleton` functions produces a list of single-element lists. Any single-element list is guaranteed to be sorted.

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`.

Note: This problem is based on 4g, and so was not graded.

Solution: The running time of `msort` is just the time for `make-singletons`, which is linear, plus the time for `merge-all`, using the length of `xs` as both n and k, which is $\Theta(n \log n)$.

Using the length of `xs` as n, the running time for `msort` is therefore $T(n) \in \Theta(n \log n)$.