1. [worth 3 points] *Counting exact numbers of primitive operations.*

The algorithm selection sort is described below:

```
procedure SELECTION-SORT(A, n) is
    for k ← 1 to n − 1 do
        SELECT(A, k, n)
    end do
end SELECTION-SORT

procedure SELECT(A, k, n) is
▷ Precondition: A[1] ≤ ... ≤ A[k − 1] and, for k ≤ j ≤ n, A[k − 1] ≤ A[j]
(1) min ← A[k]; minPosition ← k
    ▷ Invariant: for k ≤ j ≤ i − 1, min = A[minPosition] ≤ A[j]
(2) for i ← k + 1 to n do
(3)     if A[i] < min then
(4)         min ← A[i]; minPosition ← i
        endif
    end do
(5) A[k] ↔ A[minPosition]
▷ Post-condition: A[1] ≤ ... ≤ A[k] and, for k + 1 ≤ j ≤ n, A[k − 1] ≤ A[j]
end SELECT
```

Give exact values for each of the following measures related to selection sort (worst case, as a function of n, the number of items being sorted): $C(n)$ = number of *key* comparisons;[1] $oc(n)$ = number of other comparisons[2]; and $A(n)$ = number of assignments (not counting the implicit ones during procedure calls).

Solution: The for loop in SELECT is executed $n-k$ times regardless of the contents of the array. Thus, the body of the loop is executed $\sum_{k=1}^{n-1}(n-k) = (n^2-n)/2$ times. The rest follows: The only key comparison is in line (3), so $C(n) = (n^2-n)/2$. The other comparisons are ones that control the loop bodies, once per iteration of each loop, so $oc(n) = (n^2+n)/2-1$. There are $2(n-1)$ assignments in the for loop of SELECTION-SORT, 2 in line (2), $2(n-k)$ in line (3), $2A'(n)$ overall in line (4), where $A'(n)$ = the number of times $A[i] < min$, and 3 in line (5). Adding all of this up, we get $A(n) = 2(n-1) + 2(n-1) + (n^2-n) + 3(n-1) + 2A'(n)$, which is $n^2 + 6n - 7 + 2A'(n)$. Determining the worst case for $A'(n)$ is complicated, so I don't expect you to have done that. Also, your answer may differ slightly from mine – it's possible that I counted wrong – you will get full credit if you justified your answer in detail and it's in the right ballpark.

2. [worth 6 points] *Practice with recursion, i.e., the concept of reducing (an instance of) a problem to a smaller instance of the same problem.*

For each of the following problems describe a process by which an instance of the problem can be reduced to a smaller one. Your description can be a recursive function (in pseudocode with easily understandable primitive operations), a recursive mathematical definition, or a sufficiently precise English description, as long as the smaller instance and how it is manipulated to obtain a solution to the original are clear. An example of an English description for insertion sort might be:

> When sorting a list L, create a smaller list $L' = L - x$, i.e., L' is L with some element x removed. Sort L' and insert x into the result.

This would be followed up with a description of the insertion operation. Note that in order for insertion sort to be correct, x can be any element. The implementations of insertion sort choose either the first or the last for convenience and efficiency.

(a) sorting a list using selection sort (see above).

[1] A key comparison compares (the keys of) two of the items being sorted.

[2] "**for** i ← 1 **to** n **do** ⟨*body*⟩ **end do**" is the same as "i ← 1; **while** i ≤ n **do** ⟨*body*⟩; i ← i+1 **end do**"

(b) determining whether an item x appears in a list L (items may be anything, not necessarily numbers or something else that can be sorted).

(c) finding the smallest number in a list of numbers L; by definition the smallest number in an empty list is ∞.

Solutions:

(a)

SELECTION-SORT(L) is

- Let x = the smallest element in L
- Let $L' = L - x$
- Then return x + SELECTION-SORT(L')

Note that the work is done before the recursive call (finding the smallest element).

(b)

APPEARS(x, L) is

- if L is empty return ***false***
- if x = the first element in L, return ***true***
- otherwise let L' be L with the first element removed and return APPEARS(x, L')

(c)

SMALLEST(L) is

- if L is empty, then return ∞
- let x be the first element in L and L' be the rest of L
- let y be SMALLEST(L')
- if $x \leq y$, return x, otherwise return y

2 points for each part; ***no credit if an answer has a loop of any kind***

3. [worth 7 points] *Understanding asymptotic notation*

For each of the following statements, prove that it is true or give a counterexample to prove that it is false. If you give a counterexample, you still have to prove that your example is, indeed, a counterexample.

(a). If $f(n)$ is $O(g(n))$ then $g(n)$ is $O(f(n))$.

False. Let $f(n) = n$ and $g(n) = n^2$, for example.

(b). If $f(n)$ is $\Theta(g(n))$, then $\lg f(n)$ is $\Theta(\lg g(n))$.

True if we assume both $f(n)$ and $g(n)$ are monotonically nondecreasing and asymptotically positive, both of which are the case for all functions related to time bounds.

First the big-Oh bound. By definition there exist $c > 0$ and $n_0 > 0$ so that $f(n) \leq cg(n)$ for $n \geq n_0$. Taking the log of both sides (lg is monotonically non-decreasing and we also need $g(n)$ to be monotonically non-decreasing), we get $\lg(f(n)) \leq \lg c + \lg(g(n))$ for $n \geq n_0$ and $\lg(f(n)) \leq (c+1)\lg(g(n))$. Since it is always the case that $c \geq \lg c$ the argument works unless $\lg g(n) = 0$ for arbitrarily large n, i.e., not asymptotically positive.

Then the big-Omega bound. By definition there exist $c > 0$ and $n_0 > 0$ so that $f(n) \geq cg(n)$ for $n \geq n_0$. Again we take the log of both sides to get $\lg(f(n)) \geq \lg c + \lg(g(n))$. If $c \geq 1$ then $\lg(f(n)) \geq \lg(g(n))$ and we're done. Otherwise we get $\lg(f(n)) \geq \lg(g(n)) - d$ for some constant $d > 0$. As long as $\lg(g(n))$ is asymptotically positive we can find $0 < c' < 1$ such that $c' \lg(g(n)) \geq d$ and then we have $\lg(f(n)) \geq (1 - c')\lg(g(n))$.

Some counterexamples that are not monotonically nondecreasing are posted on Piazza.

(c). If $f(n)$ is $\Theta(g(n))$, then $2^{f(n)}$ is $\Theta(2^{g(n)})$.

False. Let $f(n) = n/2$ and $g(n) = n$. But then $\lim_{n \to \infty} 2^{f(n)}/2^{g(n)} = \lim_{n \to \infty} 2^{-n/2} = 0$ and $2^{f(n)}$ is $o(2^{g(n)})$ instead of $\Theta(2^{g(n)})$.

(d). If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

True. By definition of O, there exist $c > 0$ and n_0 so that $f(n) \leq cg(n)$ for all $n \geq n_0$. This implies that for all $n \geq n_0$, $g(n) \geq (1/c)f(n)$ so, by definition, $g(n)$ is $\Omega(f(n))$.

(e). $f(n)$ is $\Theta(f(n/2))$.

False. Let $f(n) = 2^n$. Then $f(n/2) = 2^{n/2}$. But $\lim_{n\longrightarrow\infty} 2^n/2^{n/2} = \lim_{n\longrightarrow\infty} 2^{-n/2} = 0$ and $2^{f(n)}$ is $o(2^{g(n)})$ instead of $\Theta(2^{g(n)})$.

(f). If $g(n)$ is $o(f(n))$ then $f(n) + g(n)$ is $\Theta(f(n))$

True. $\lim_{n\to\infty}(f(n) + g(n))/f(n)$ will be 1.

(g). $f(n) + g(n)$ is $\Theta(\max(f(n), g(n)))$.

True. First, if $f(n)$ and $g(n)$ are both positive, it is always the case that $f(n) + g(n) \leq 2\max(f(n), g(n))$ for all $n \geq 0$. Thus, $f(n) + g(n)$ is $O(\max(f(n), g(n)))$.

To show that it's $\Omega(\max(f(n), g(n)))$ is easy: $f(n)$ and $g(n)$ are both positive, $f(n) + g(n) \geq f(n)$ and $f(n) + g(n) \geq g(n)$. So it's clearly $\geq \max(f(n), g(n))$.

1 point for each part

4. [worth 6 points] *Bounding summations.*

Give Θ bounds for each of the following summations.

(a). $$\sum_{i=2}^{n} i^2 \lg\lg i$$

Solution: This is $\leq n^3 \lg\lg n$ when $n \geq 2$ so $O(n^3 \lg\lg n)$. When $n \geq 2$ the sum $\geq \sum_{n/2\leq i\leq n}(n/2)^2 \lg\lg(n/2)$ which is $\geq (n/2)^3 \lg(\lg n - 1)$. If we choose $n_0 = 16$ (lots of other choices will work, but clearly c must be $< 1/8$), this is $\geq (n^3/8)(\lg\lg n/2) = (1/16)n^3 \lg\lg n$. So $c = 1/16$, $n_0 = 16$.

(b). $$\sum_{i=1}^{n} i^2 2^i$$

Solution: It's easy to show that this is $\Omega(n^2 2^n)$ – this is the last term in the summation so $c = n_0 = 1$. To get the O bound, note that the sum is $\leq n \sum_{i=1}^{n} i2^i$, and, using formula 6 in the list of useful formulae, this is $< n(2 + n2^{n+2})$. The rest is easy.

(c). $$\sum_{i=0}^{d} a^i n^i$$ Let a be any real number $\neq 0$, and $d \geq 0$ be an integer constant.

Your bound will involve d. Also, if d is odd, assume that $a > 0$.

Solution: Several cases need to be considered. In all of them the sum is $\Theta(n^d)$. Note: This can also be done more easily using limits.

If $a \geq 1$ the sum is $\leq da^d n^d$ and $\geq a^d n^d$, proving the big-oh bound with $c = da^d$ and $n_0 = 1$ and the Ω bound with $c = a^d$ and $n_0 = 1$

If $0 < a < 1$ the sum is $\leq dan^d$ and $\geq an^d$, so it's the big-oh bound with $c = da$ and $n_0 = 1$ and the Ω bound with $c = a$ and $n_0 = 1$.

If $a < 0$ the signs of the terms alternate. For the Ω bounds we pick a $c < a^d$ and adjust n_0 appropriately, ignoring the positive lower-order terms. For example, the sum $\geq a^d n^d/2$ if

$$an + a^3 n^3 + \cdots + a^{d-1} n^{d-1} \leq a^d n^d/2$$

if

$$n \geq a/n^{d-2} + a^3/n^{d-4} + \cdots + a^{d-1} \tag{1}$$

Again we break down into cases. The analysis of the big-oh bounds is identical to that for $a > 0$ since we can ignore the negative terms.

If $-1 < a < 0$ the inequality (1) can be satisfied if $n \geq da$ (clearly overkill but it gets the job done). So $c = a^d/2$ and $n_0 = da$

If $a \leq -1$ we can satisfy inequality (1) with $n \geq da^{d-1}$. So $c = a^d/2$ and $n_0 = da^{d-1}$.

2 points each part; in part (c), you get 1 point for identifying the three cases and recognizing what the issues are, 1 point for the remaining details.

5. [worth 10 points] *Solving recurrences.*

For each of the following recurrences, do one of two things. If the Master Theorem applies, give the Θ and the appropriate case. If not, solve the recurrence using any method you like, as long as it yields a rigorous proof that is easy for us to follow. In either case, assume that $T(n)$ is some constant c_0 when $n < s$ (you can choose s appropriately if the Master Theorem does not apply). The term "Master Theorem" here refers to the one in the textbook *not* the more general version found in, say, Wikipedia.

(a). $T(n) = 2T(n/2) + n \lg \lg n$

Solution: Not Master Theorem.

level	**number of instances**	**size of each**	**cost of each**	**total cost**
0	1	n	$n \lg \lg n$	$n \lg \lg n$
1	2	$n/2$	$(n/2) \lg \lg (n/2)$	$n \lg \lg (n/2)$
...				
i	2^i	$n/2^i$	$(n/2^i) \lg (\lg n - i)$	$n \lg (\lg n - i)$
k	2^k	1	c_0	$c_0 2^k$

Assuming $n = 2^k$ the total is $c_0 n + n \sum_{i=0}^{k-1} \lg(k-i)$.

Letting $j = k - i$, we get $\sum_{i=0}^{k-1} \lg(k-i) = \sum_{i=1}^{k} \lg j$ which is $\Theta(k \lg k)$, using rough over- and underestimates. So the solution is $\Theta(nk \lg k) = \Theta(n \lg n \lg \lg n)$.

(b). $T(n) = 5T(n/4) + n \lg n$

Solution: Master Theorem Case 1: $\Theta(n^{\log_4 5})$

(c). $T(n) = 2T(n/2) + n \lg^2 n$

Solution: Not Master Theorem.

level	**number of instances**	**size of each**	**cost of each**	**total cost**
0	1	n	$n \lg^2 n$	$n \lg^2 n$
1	2	$n/2$	$(n/2) \lg^2 (n/2)$	$n \lg^2 (n/2)$
...				
i	2^i	$n/2^i$	$(n/2^i)(\lg n - i)^2$	$n(\lg n - i)^2$
k	2^k	1	c_0	$c_0 2^k$

Assuming $n = 2^k$ the total is $c_0 n + n\sum_{i=0}^{k-1}(k-i)^2$. The sum can be rewritten as $\sum_{j=1}^{k} j^2$ which is $\Theta(k^3)$. Therefore the solution is $\Theta(n \lg^3 n)$.

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

Solution: Master Theorem Case 3: $\Theta(\frac{n^2}{\lg^2 n})$.

Some of you did this the hard way; if you did, and you bounded your summation correctly, you will get the 1 point.

(e). $T(n) = 4T(n/2) + \frac{n^2}{\lg^2 n}$

Solution: Not Master Theorem.

level	**number of instances**	**size of each**	**cost of each**	**total cost**
0	1	n	$n^2/\lg^2 n$	$n^2/\lg^2 n$
1	4	$n/2$	$(n/2)^2/\lg^2(n/2)$	$n^2/\lg^2(n/2)$
...				
i	4^i	$n/2^i$	$(n^2/4^i)/(\lg n - i)^2$	$n^2/(\lg n - i)^2$
k	4^k	1	c_0	$c_0 4^k$

Assuming $n = 2^k$ the total is $c_0 n^2 + n^2 \sum_{i=0}^{k-1} /(k-i)^2$. The sum can be rewritten as $\sum_{j=1}^{k} 1/j^2$ which is constant. Therefore the solution is $\Theta(n^2)$.

(f). $T(n) = 27T(n/3) + n^3$

Solution: Master Theorem Case 2: $\Theta(n^3 \lg n)$

(g). $T(n) = 6T(n/2) + n^3$

Solution: Master Theorem Case 3: $\Theta(n^3)$

10 points total, not evenly distributed. 1 point where the Master Theorem applies, 2 points where it doesn't – 1 point for the correct table, 1 point for the remaining details.

6. [worth 5 points] *Applying analysis of divide and conquer to a concrete problem.*

The problem statement has been modified as explained on Piazza.

The dominant operations in Strassen's matrix multiplication algorithm, particularly if the numbers being manipulated are multiple precision (take up more than one machine word), are scalar multiplications and scalar additions. Assuming that multiplication takes twice as long as addition and using addition as a unit of time, a recurrence for time is

$$\begin{array}{lcll} T(n) & = & 7T(n/2) + 10(n/2)^2 & \text{note: no scalar multiplications are done before or after any recursive calls} \\ T(1) & = & 2 & \text{accounts for all scalar multiplications} \end{array}$$

Additional note: The above fails to account for the 8 additions of $n/2 \times n/2$ matrices done at the end to obtain the C_{ij}'s. The correct recurrence is therefore

$$\begin{array}{lcll} T(n) & = & 7T(n/2) + 18(n/2)^2 & \text{10 additions to prepare the recursive calls, 8 afterward} \\ T(1) & = & 2 & \text{accounts for all scalar multiplications} \end{array}$$

Strassen's algorithm, even if you ignore bookkeeping overhead, is not a good choice for small matrices.

Your job is to figure out the exact value of n at which it makes sense to stop recursing and use the ordinary matrix multiplication algorithm instead. The number of scalar operations for the ordinary algorithm are n^3 multiplications and n^3 additions. Once you've set up an equation to solve, you can use a sophisticated calculator, MatLab, or any program that works (I wrote a simple Python program.)

Solution: The easiest way to approach this is to find the break even point between the two algorithms. Let $M(n)$ be the time for the traditional algorithm. If we find the value of s where $T(s) = M(s)$ we know that $T(n) \leq M(n)$ for all $n \geq s$ – since $T(n)$ grows asymtotically more slowly than $M(n)$.

Solving the recurrence for $T(s)$ and assuming s is a power of 2, we get (for the incorrect recurrence in the problem statement)

level	**number of instances**	**size of each**	**cost of each**	**total cost**
0	1	n	$(5/2)n^2$	$(5/2)n^2$
1	7	$n/2$	$(5/2)(n/2)^2 = (5/2)n^2/4$	$7(5/2)n^2/4$
...				
i	7^i	$n/2^i$	$(5/2)n^2/4^i$	$7^i(5/2)n^2/4^i$
$k = \lg s$	7^k	1	2	$2 \cdot 7^k$

Adding everything up we get

$$\begin{aligned} T(n) &= 2 \cdot 7^k + (5/2) \textstyle\sum_{i=0}^{k-1} 7^i s^2/4^i \\ &= 2 \cdot 7^k + (5s^2/2) \textstyle\sum_{i=0}^{k-1} (7/4)^i \\ &= 2 \cdot 7^k + (5s^2/2) \tfrac{7^k - 4^k}{3 \cdot 4^{k-1}} && \text{summation formula} \\ &= 2s^{\lg 7} + (10/3)(s^{\lg 7} - s^2) && \text{after some algebraic manipulation} \\ &= (16/3)s^{\lg 7} - (10/3)s^2 \end{aligned}$$

Making this equal to the traditional method we get

$$\begin{aligned} (16/3)s^{\lg 7} - (10/3)s^2 &= 3s^3 && \text{one multiplication and one addition in the inner loop of the traditional algorithm} \\ 3s^3 - (16/3)s^{\lg 7} + (10/3)s^2 &= 0 && \text{put it in form of a polynomial whose root you want to find} \\ 3s - (16/3)s^{\lg 7 - 2} + 10/3 &= 0 && \text{to get rid of the trivial root } s = 0 \end{aligned}$$

Now we can use some brute force root finding method to get an actual number. I used a simple Python program that does binary search and came up with ≈ 13; round this up to the next power of two and get 16 (not that that means anything).

If I had used the correct recurrence (including the additions for the C_{ij}'s), there would be a (9/2) instead of (5/2) as coefficient in front of the sum. The final polynomial would then be (if I did this right):

$$3s - 8s^{\lg 7 - 2} + 6 = 0$$

which has a numerical solution of ≈ 152.

3 points for coming up with the right recurrences with a base case that is not 1 and evaluating them; 2 points for the remaining details.

For up to 10 points extra credit, do an experiment to determine the best value for n at which to stop the recursion. You can get up to 3 points for a detailed description of how such an experiment should be conducted. Submit your description, a writeup of your experiment, and your code with instructions for compiling and running it on a Unix-based system to the h1ex locker. It should be a zip archive with the name h1ex_uid.zip, where uid is your unity login id.

To get full credit on this, every part of your experimental study has to be thorough: the description of implementation issues such as machine architecture and how you handled parameter passing, documentation of the program and how to run it, tables and charts showing results, explanation of the results, accounting for cache effects by, for example, comparing with the recursive version that does 8 multiplications of $n/2 \times n/2$ sub-matrices.

The three points for the writeup requires an equal level of thoroughness. The only difference is that you may not have a working implementation.

7. [worth 5 points] *Demonstrating abstract problem solving ability needed in CSC 505; also an interesting example of divide-and-conquer.*

You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values – so there are $2n$ values in all – and you can assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here as the n-th smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the databases, and the chosen database will return the k-th smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\lg n)$ queries. Make your description as succint as possible while still being precise; as Einstein would say "as simple as possible but no simpler."

Solution: For convenience suppose that $a[1] \leq \cdots \leq a[n]$ are the elements in the first database and $b[1] \leq \cdots \leq b[n]$ are the elements in the second. Of course only the databases have this information.

Any instance of the problem, particularly any arrived at via divide-and-conquer, asks for the median of a set consisting of the same number of (consecutive) elements from each database. More formally, an instance asks for the median of

$$\{a[i_1], \cdots, a[j_1]\} \cup \{b[i_2], \cdots, b[j_2]\}$$

where $j_2 - i_2 = j_1 - i_1$. Initially $i_1 = i_2 = 1$ and $j_1 = j_2 = n$. Lest the solution is too focused on the messy details of manipulating an arbitrary instance, the intervals $[i_1, j_1]$ and $[i_2, j_2]$ will each be mapped to $[1, n']$, where $n' \leq n$. The details are easy to fix: for example, the k-th smallest element among $a[i_1], \ldots, a[j_1]$ is the $(i_i + k - 1)$-st smallest of $a[1], \ldots, a[n]$.

Define $n' = j_1 - i_1 + 1$ to be the *instance size*. To achieve an $O(\lg n)$ bound on the number queries, a constant number of queries are used to reduce the instance size to $\alpha n'$ for some constant $\alpha < 1$. Eventually, a base case, in which the instance size n' is some constant, is reached, and the problem can be solved by "brute force": use $2(n'-1)$ queries to establish the total order within each database and merge the two sorted lists to find the median.

In this case, two queries can be used to reduce the instance size by half. The two queries ask for the $n'/2$-th element of each database, i.e., $a[n'/2]$ and $b[n'/2]$ — use integer division, so $n'/2 = \lfloor n'/2 \rfloor$.

Suppose $a[n'/2] < b[n'/2]$; the case where $b[n'/2] > a[n'/2]$ is symmetric. Then the elements $a[1], \ldots, a[n'/2 - 1]$ are $< a[n'/2], \ldots, a[n']$ and $< b[n'/2 + 1], \ldots, b[n']$. Since they are less than at least half of the elements, none of them can be the median.

Based on the information already gathered, $b[3n'/4 + 1]$ is greater than
$b[1], \ldots, b[3n'/4]$ and $a[1], \ldots, a[n'2 - 1]$, so neither it nor the rest of the $b[i]$'s up to $b[n']$ can be the median.

Now $n'/4$ elements on the b side cannot be the median and $n'/2$ on the a side. To even things out, only $n'/4$ are chosen to be eliminated on the a side. This gives the recurrence $T(n) = T(\lceil 3n/4 \rceil) + 2$. The base case needs to be ≥ 3 (why?).

3 points for a correct algorithm (full credit only if description is simple); 2 points for correct analysis of your algorithm (if either your algorithm is correct or it has an analysis that is as interesting as the correct algorithm)