Buffalo Hird
CS124 Problem Set 1
Problem 1:

A	B	O	o	Ω	ω	Θ
$\log n$	$\log(\frac{n}{\log n})$	*yes*	*no*	*yes*	*no*	*yes*
$\log(n!)$	$\log(n^n)$	*yes*	*yes*	*no*	*no*	*no*
$\sqrt[7]{n}$	$(\log n)^2$	*no*	*no*	*yes*	*yes*	*no*
$n^2 2^n$	3^n	*yes*	*yes*	*no*	*no*	*no*
$(n^2)!$	n^n	*no*	*no*	*yes*	*yes*	*no*
$\frac{n^2}{\log n}$	$n\log(n^2)$	*no*	*no*	*yes*	*no*	*no*
$(\log n)^{\log n}$	$4^{(\log n)(\log\log n)}$	*no*	*no*	*yes*	*yes*	*no*
$n + \log n$	$100n + \sqrt{n}$	*yes*	*no*	*yes*	*no*	*yes*

Problem 2:

- Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$.

We want to find an f_1 such that $f_1(2n)$ is $O(f_1(n))$

We can note this means we want:

$$f(n) \leq c * f(2n), n > N$$

We can take the example $f(n) = n$ and plug into the equation to get

$$2n \leq c * n$$

which is true for any $c \geq 3$.

We have therefore proven by the identity of $O()$ notation that $f(2n)$ is $O(f(n))$.

- Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

We want to find an f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

We can note that this means:

$$\frac{lim(f(2n))}{lim(f(n))} = \infty, n \to \infty$$

We can try the example $f(n) = 3^n$.

$$\frac{lim(3^{2n})}{lim(3^n)} = lim(3^n) \to \infty,\text{n} \to \infty$$

Since the limit goes to infinity, we have shown that $f(2n)$ is not $O(f(n))$ by the definition of $O()$.

- Prove that if $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

We can note by the definition of $O()$ we can rewrite these relationships as

$$f(n) \leq c * g(n), n > N_1$$

$$g(n) \leq d * h(n), n > N_2$$

We can therefore rearrange the equations to say

$$\frac{f(n)}{c} \leq g(n), n > N_1$$

Since this fraction is never more than $g(n)$, we can safely plug it in for $g(n)$ without breaking $O()$ relations:

$$\frac{f(n)}{c} \leq d * h(n), n > N_2$$

We can then rearrange to get:

$$f(n) \leq c_1 * h(n), n > N_2, c_1 = c * d$$

Since it doesn't matter what constant we apply for this equation, we can simply combine c and d. We must only alter our equation such that we rely on the $max(N_1, N_2)$ because it is not until this N that we expect $f(n)$ to grow less than or equal to $h(n)$'s asymptotic growth (with constants). So we have a final proved equation of:

$$f(n) \leq c_1 * h(n), n > max(N_1, N_2), c_1 = c * d$$

- Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$

We can simply prove this by creation a function which alters its $O()$ runtime each iteration. We define

$$f(n) = \left\{ \begin{array}{cc} even & n \\ odd & n^3 \end{array} \right\}$$

$$g(n) = n^2$$

We can take the ratio of these two functions' limits to get their aymptotic $O()$ relation. For even n:

$$\frac{lim(n)}{lim(n^2)} = lim(\frac{1}{n}) = 0, n \to \infty$$

So $f(n)$ is $O(g(n))$ in the even case, but $g(n)$ is not $O(f(n))$. Let's do odd n:

$$\frac{lim(n^3)}{lim(n^2)} = lim(n) = \infty, n \to \infty$$

So here $g(n)$ is $O(f(n))$ in the odd case but the opposite does not hold. We can note that in this case that even though there are piecewise $O()$ relations, when we combine them we are left with none. So even though f is not $O(g)$, g is not $O(f)$ because it does not hold in the even case.

- Give a proof or a counterexample: if f is not $o(g)$, then f is $O(g)$

By definition f is $O(g)$ is:

$$f \leq c * g, N > n, \exists c$$

While f is $o(g)$ is:

$$f \leq c * g, N > n, \forall c$$

We can therefore prove this simply by these definitions. We note that $o(g)$ implies that this relationship holds for all constants c. While $O(g)$ holds for some constants c with sufficiently large n. Since $o(g)$ is for all constants, it must also hold for only some c by logic, therefore if f is $o(g)$ it must also be $O(g)$.

Problem 3:

We can note that most of the computation of insertion sort occurs in the while-loop in which each individual element is sorted. We can note that an element x_j is swapped a minimum of 0 times if the ordering is correct s.t. $T(n) = n$ as no swaps were required. In the worst case, each x_j must be swapped t_j times, which is some value $[1, n-1]$. Therefore if each item needed swapping we would have $T(n) = n^2$.

We must now show there eists a sequence of arrays $T(n) = n^{1+\epsilon}$ for any n and $0 \leq \epsilon \leq 1$. We can note this is logical because this bounds $T(n)$ between n and n^2 as just shown. We can note that $T(n) = n * n^{\epsilon}$ with which we can cleverly build an arithmetic sequence.

Given an array that requires sorting, we can note that the first $n - n^{\epsilon}$ elements would not require sorting. There are then n^{ϵ} terms which do require sorting. If we define these terms as x_i through x_k then we could note that each would require $\frac{k*n}{n^{\epsilon}}$ swaps s.t. $\sum_{i=1}^{k} x_k = \frac{k^{\epsilon}(k+1)}{2}$.

Putting these two terms together, we get:

$$\frac{n^{\epsilon}(n+1)}{2} + (n - n^{\epsilon})$$

We have created a sequence where the largest term is $n * n^{\epsilon}$ s.t we have shown that there is an infinite sequence of arrays that for $T(n) = \Omega(n)$ and $T(n) = O(n^2)$ for every $T(n)$. This was intuitive for insertion sort and proved a good launchpad from which to prove this.

Problem 4:

Note: this is a long and winded verbal proof. It is much easier to describe in person. A lot of mathematical list-based proofs were mentioned in office hours, but I didn't find them useful as they provided no intuition.

We can note that Stoogesort sorts correctly for the base cases of list length $n = 1$ and $n = 2$ because for 1 it can simply return the item and for 2 it can either return the 2 items or return the 2 items swapped. Since these base cases work, any non-divisble by 3 length wil eventually be split into an item of length 3 and one of these base cases, which can be easily solved. Therefore, this algorithm works for lists of any length.

We can prove it solves for all k by induction by assuming it works for list length up to k and showing it works for list length up to $k + 1$.

We can assume that each of the 3 $\frac{2(k+1)}{3}$ length sub-sorts are completed correctly since we are assuming that any list of length up to k sorts correctly. We can note that there are 3 actions

1. Sort first 2/3s
2. Sort second 2/3s
3. Sort first 2/3s

We therefore know that the first 2/3's must be properly sorted since they were just sorted last in step 3. We also know that the last 1/3 must be properly sorted since it was sorted in step 2 and is not touched in step 3. We must then simply show that these two sorted pieces are as a whole sorted.

Because step 1 moved the largest items of the first 2/3s to the middle 1/3 of the list, these big items should be properly sorted in the last 2/3rds in step 2. As a result these items will either be small enough to be moved correctly in the first 2/3rds in step 3 or be put in the last 1/3rd of the list in step 2. This must then thoroughly sort which we can show by contradiction.

Consider x_i and x_j such that $i < j$ but $x_i > x_j$. This means that a larger item occurs before a smaller item, invalidating our sort.

1. We can consider the example where x_i and x_j are both in the first 2/3rds. This means both have to be sorted by step 3 so this is impossible.
2. We can consider the example where x_i and x_j are both in the last 1/3rd. This means both have to be sorted by step 2 so this is impossible.
3. We can consider where x_i is in the first 2/3rds and x_j is in the last 1/3rd. This must mean that x_i was sorted in the first 2/3rds in step 3. This means that it was not sorted by step 2, otherwise it would have been placed after x_j since it is larger (and being in the last 1/3 means x_j had to have been sorted in step 2). This all implies x_i was not sorted in step

1, because if it were it would be placed near the end of the first 2/3rds, since it is larger than x_j which is in the largest 1/3**. As a result of this, it would be sorted in step 2 to the last 1/3 after x_j and after this it would not be touched in step 3 and would not be in the first 2/3rds. Therefore this case is also impossible.

**If this were not the case and there were $k/3$ larger items after x_i in step 1 and so there would be $k/3$ items place into the last 1/3, pushing x_j into the first 2/3rds in step 2, causing this to be case 1

By showing all these cases are impossible we have proved that stooge sort correctly sorts on all inputs.

We can now derive a recurrence and use this to bound the asymptotic running time of stoogesort. We can note that we call a recursive call on 3 lists of length $\frac{2}{3}n$ so that we have

$$T(n) = 3T(\frac{2}{3}n) + c$$

where c is some linear value of $\Theta(1)$ where the sort handles the actual swapping of values. We can then write

$$T(n) = 1 + 3T(\frac{2}{3}n) = 1 + 3 + 9T(\frac{4}{9}n) = 1 + 3 + 3^2 + ... + 3^{log_{\frac{3}{2}} n}$$

We can then note that in the limit this is simply:

$$\Theta(3^{log_{\frac{3}{2}} n}) = \Theta(3^{(log_3 n)/(log_3 (3/2))})$$

We can note that we have logs cancel to produce:

$$= \Theta(n^{\frac{1}{log_3(\frac{3}{2})}}) = \Theta(n^{2.71})$$

We have therefore found that stoogesort will perform asymptotically bounded to this value.

Problem 5:
$T(1) = 1, T(n) = T(n-1) + 3n - 3$

We can solve the recurrence exactly by solving a system of linear equations.

We begin by plotting some of the points to get:

$$\begin{aligned} T(1) = &\quad 1 \\ T(2) = &\quad 4 \\ T(3) = &\quad 10 \\ T(4) = &\quad 19 \end{aligned}$$

We can note that since this recurrence changes by polynomial degree 1, we can represent this as a function of degree $n + 1 = 2$. We are therefore looking for a function:

$$a_n = c_2 n^2 + c_1 n + c_0$$

where we can simply plug in our recurrence values as solutions to linear equations which represent our a_n at each n up to value $n = 2$ (degree + 1).

$$1 = c_0$$

$$4 = c_2 + c_1 + c_0$$

$$10 = 4c_2 + 2c_1 + c_0$$

Solving these we get $f(n) = 1.5n^2 - 1.5n + 1$. We can now prove its correctness by plugging it into the recurrence relation and seeing that it holds for case $n + 1$.

For our base case:

$$f(1) = 1.5(1)^2 - 1.5(1) + 1 = 1$$

For our inductive step:

$$T(n+1) = T(n) + 3n - 3$$

$$T(n+1) = 1.5n^2 - 1.5n + 1 + 3(n+1) - 3$$

$$T(n+1) = 1.5(n+1)^2 - 1.5(n+1) + 1$$

We have therefore shown by induction that this function correctly solves the recurrence equation.

$T(1) = 1, T(n) = 2T(n-1) + 2n - 1$

We can start by writing out some values for this recurrence, noting that it seems to grow with factor 2^n

$$\begin{aligned} T(1) &= \quad 1 \\ T(2) &= \quad 5 \\ T(3) &= \quad 15 \\ T(4) &= \quad 37 \end{aligned}$$

Doing some careful math we can note that this recurrence is equivalent to $f(n) = 3 * 2^n - 2n - 3$. We find this by noting that the recurrence will asymptotially more closely resmemby $c * 2^n$. We can find c by taking the limit of

$$lim(\frac{T(n)}{2^n})$$

To get that our $c = 3$. We can then essentially guess-and-check to get the rest of our equation. We can prove this equation by plugging the function into the original recurrence:

For our base case:

$$f(1) = 3 * 2^1 - 2(1) - 3 = 1$$

For our inductive step

$$T(n+1) = 2T(n) + 2n - 1$$

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

$$T(n+1) = 6 * 2^n - 4n - 6 + 2n + 2 - 1$$

$$T(n+1) = 3 * 2^{n+1} - 2(n+1) - 3$$

We have therefore shown by induction that this function correctly solves the recurrence equation.

Problem 6:

We can solve these problems using Master's Theorem, using it to give an asymptotic bound for $T(n)$ in each recurrence

Master's Theorem: $T(n) = aT(\frac{n}{b}) + cn^k$

•$T(n) = 5T(\frac{n}{3}) + n^3$
Since $5 < 3^2$, we have $T(n) = \Theta(n^3)$

•$T(n) = 25T(\frac{n}{4}) + n^2$
Since $25 > 4^2$, we have $T(n) = \Theta(n^{log_4 25})$

•$T(n) = 8T(\frac{n}{2}) + n^3$
Since $8 = 2^3$, we have $T(n) = \Theta(n^3 log(n))$

•$T(n) = T(n^{\frac{1}{4}}) + 1$
Here we have to use a change of variables to solve the problem.

We can define $U(n)$ such that $U(n) = T(log(n))$.

We can write the problem as $T(log(n)) = T(\frac{1}{4}log(n)) + 1 = U(n) = U(\frac{1}{4}n) + 1$

So we can note that by the Master Theorem $U(n) = \Theta(log(n))$ and so:

$$U(n) = T(log(n)) = U(log(n))$$

This is actually very useful, noting that for any n we plug into $U()$ we will get the log of that value.

We therefore have that

$$T(log(n)) = U(log(n)) \rightarrow T(n) = \Theta(log(log(n))$$

Problem 7

We can begin to solve for this more general form by plotting values on the first value n values:

n	x	dx
1	0	
2	1	1
3	3	2
4	5	2
5	8	3
6	11	3
7	14	3
8	17	3
9	21	4

We can quickly see that these values are changing by 1 every 2^n terms, suggesting we have an exponential growth problem. We can note that this is equivalent to:

$$T(n) - T(n-1) = \lceil log_2 n \rceil$$

We can then note that these differences $T(n) - T(n-1)$ produces a telo-scoping sum such that:

$$\sum_{k=2}^{n} T(k) - T(k-1) = T(n) - T(1) = \sum_{k=2}^{n} \lceil log_2 k \rceil$$

This is found by noticing that in subtracting these $k-1$ terms we end up subtracing out all middle terms.

We can divide this into two sums: a sum for all terms leading up to the largest 2^a less than n and a sum for all terms following this number.

$$= \sum_{k=2}^{2^a} \lceil log_2 k \rceil + \sum_{k=2^a+1}^{n} \lceil log_2 k \rceil$$

We can notice that the second term is simply $n - 2^a$ instances of $a+1$ and compute the first sum to get:

$$\sum_{l=1}^{a} l * 2^{l-1} * (a+1)(n-2^a)$$

This first sum can be reduced to:

$$a * 2^a - 2^a + 1$$

Returning back to terms of $T(n)$ we can rewrite this simplified as:

$$T(n) = a * 2^a - 2^a + 1 + (n - 2^a) * (a+1)$$

$$T(n) = n(a+1) - 2^{a+1} + 1$$

where, as defined, $a(n) = \lfloor log_2 n \rfloor$.

We can now prove the correctnes of this:

We have our base cases $T(1) = 0$ and $T(2) = 1$.

Now for our inductive step we consider the $n+1$ case. We can break this up into subcases based on if the input to the function is even or odd

Even case:

$$T(n+1) = T(\lceil \frac{n+1}{2} \rceil) + T(\lfloor \frac{n+1}{2} \rfloor) + (n+1) - 1 = 2(T(\frac{n+1}{2})) + n$$

We can then plug in our inductive hypothesis:

$$T(n+1) = 2(-2^{\lfloor log_2 \frac{n+1}{2} \rfloor + 1} + \frac{n+1}{2}(\lfloor log_2 \frac{n+1}{2} \rfloor + 1) + 1 + n$$

$$T(n+1) = 2(-2^{\lfloor log_2 (n+1) - 1 \rfloor + 1} + \frac{1}{2}(n+1)(\lfloor log_2 (n+1) - 1 \rfloor + 1) + 1) + n$$

We then distribute values and multiply by distributing the 2:

$$T(n+1) = -2^{\lfloor log_2 (n+1) \rfloor + 1} + (n+1)(\lfloor log_2 (n+1) \rfloor) + 2 + n$$

$$T(n+1) = -2^{\lfloor log_2 (n+1) \rfloor + 1} + (n+1)(\lfloor log_2 (n+1) \rfloor + 1) + 1 + n - n$$

$$T(n+1) = -2^{\lfloor log_2 (n+1) \rfloor + 1} + (n+1)(\lfloor log_2 (n+1) \rfloor + 1) + 1$$

Odd case:

$$T(n+1) = T(\lceil \frac{n+1}{2} \rceil) + T(\lfloor \frac{n+1}{2} \rfloor) + (n+1) - 1 = T(\frac{n+2}{2}) + T(\frac{n}{2}) + n$$

Plugging in our nductive hypothesis:

$$T(n+1) = -2^{\lfloor log_2 \frac{n+2}{2} \rfloor + 1} + \frac{n+2}{2}(\lfloor log_2 \frac{n+2}{2} \rfloor + 1) + 1 - 2^{\lfloor log_2 \frac{n}{2} \rfloor + 1} + \frac{n}{2}(\lfloor log_2 \frac{n}{2} \rfloor + 1) + 1 + n$$

We can simplify our log_2 values:

$$T(n+1) = -2^{\lfloor log_2 (n+2) \rfloor} + \frac{n+2}{2}(\lfloor log_2 (n+2) \rfloor) - 2^{\lfloor log_2 (n) \rfloor} + \frac{n}{2}(\lfloor log_2 n \rfloor) + 2 + n$$

We can not proceed generally from here, because we are not sure of the equality of

$$\lfloor log(n) \rfloor and \lfloor log(n+2) \rfloor$$

We can break this into a case where they are equal first:

$$T(n+1) = -2^{\lfloor log_2(n) \rfloor} + \frac{n+2}{2}(\lfloor log_2(n) \rfloor) - 2^{\lfloor log_2(n) \rfloor} + \frac{n}{2}(\lfloor log_2 n \rfloor) + 2 + n$$

$$T(n+1) = -2^{\lfloor log_2(n) \rfloor + 1} + (n+1)(\lfloor log_2(n) \rfloor) + 2 + n$$

Rearranging:

$$T(n+1) = -2^{\lfloor log_2(n) \rfloor + 1} + (n+1)(\lfloor log_2(n) \rfloor + 1) + 2 + n - (n-1) = T(n+1) = -2^{\lfloor log_2(n) \rfloor + 1} + (n+1)(\lfloor log_2(n) \rfloor + 1) +$$

We can rewrite this in our inductive form, noting that $\lfloor log_2(n+1) \rfloor = \lfloor log_2(n) \rfloor$:

$$T(n+1) = -2^{\lfloor log_2(n+1) \rfloor + 1} + (n+1)(\lfloor log_2(n+1) \rfloor + 1) + 1$$

Now we consider the case $\lfloor log_2(n+2) \rfloor = \lfloor log_2(n) \rfloor + 1$

$$T(n+1) = -2^{\lfloor log_2(n) + 1 \rfloor} + \frac{n+2}{2}(\lfloor log_2(n) \rfloor + 1) - 2^{\lfloor log_2(n) \rfloor} + \frac{n}{2}(\lfloor log_2 n \rfloor) + 2 + n$$

$$T(n+1) = -2^{\lfloor log_2(n) \rfloor + 2} + \frac{n}{2}(\lfloor log_2(n) \rfloor + \frac{n}{2} + \lfloor log_2(n) \rfloor + 1 + (n+1)(\lfloor log_2(n) \rfloor) + 2 + n$$

Multiplying through we can convert this messy equation (noting $\lfloor log_2(n+1) \rfloor = \lfloor log_2(n) \rfloor$ from case 2 to get:

$$T(n+1) = -2^{\lfloor log_2(n+1) \rfloor + 1} + (n+1)(\lfloor log_2(n+1) \rfloor + 1) + 1$$