Exercise Sheet #1 Solutions, Computer Science, 308-251A

M. Hallett, K. Smith 2002

Question 1.1: Prove $(n+a)^b = \Theta(n^b)$.

Binomial Expansion:

$$(n+a)^b = \sum_{k=0}^{b} \binom{b}{k} a^k n^{b-k} = n^b + \sum_{k=1}^{b} \binom{b}{k} a^k n^{b-k}$$

Show $O(n^b)$:

$$\sum_{k=0}^{b} \binom{b}{k} a^k n^{b-k} \leq c_1 n^b$$

$$\sum_{k=0}^{b} \binom{b}{k} a^k n^{b-k} \leq \sum_{k=0}^{b} \binom{b}{k} a^k n^b$$

$$n^b \sum_{k=0}^{b} \binom{b}{k} k a^k \leq c_1 n^b$$

For any $c_1 > \sum_{k=0}^{b} \binom{b}{k}\ a^k$, $n_0 > 1$ and $a, b > 0$ this holds. So, $(n+a)^b$ $= O(n^b)$.

Show $\Omega(n^b)$:

$$n^b + \sum_{k=1}^{b} \binom{b}{k} a^k n^{b-k} \geq c_2 n^b$$

$\sum_{k=1}^{b} \binom{b}{k} a^k n^{b-k}$ is positive (assume $a \geq 0$ even though it wasn't stated in question).

For any $c_2 \geq 1$, $n_0 > 1$ the above statement holds. So, $(n+a)^b = \Omega(n^b)$.

Kaleigh.

Question 1.2: Prove or Disprove $2^{n+1} = O(2^n)$.

$$2^{n+1} = 2 \cdot 2^{n+1} \leq c \cdot 2^n$$

For any $c \geq 2$ this holds. So, $2^{n+1} = O(2^n)$.

Question 1.3: Prove or Disprove $2^{2n} = O(2^n)$.

$$2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n$$
$$2^n \leq c$$

There is no such constant c. Therefore, $2^{n+1} \neq O(2^n)$.

Kaleigh.

Question 1.4: Prove or disprove $2^{log_b\ n} \neq \Theta(2^{log_a\ n})$.

Let's prove that $2^{log_b\ n} \neq O(2^{log_a\ n})$ (thereby disproving it).

(Page 34 of CLR ed. 1 for logarithm properties. Appendix in CLRS ed. 2)

Suppose $2^{log_b\ n} \leq c2^{log_a\ n}$ for positive constant c. Then $n^{log_a\ 2} \leq c \cdot n^{log_b\ 2}$. Take the $log_2 (= lg)$ of both sides and rearrange. We get

$$log_a\ 2 \leq \frac{lg\ c}{lg\ n} + log_b\ 2$$

or, more succintly,

$$(log_a\ 2 - log_b\ 2) \cdot lg\ n \leq lgc.$$

If $a < b$, then $log_a\ 2 - log_b\ 2 > 0$ and no such constant c exists.

Now for the case when $a > b$ (when $a = b$, it's clearly true), we repeat the above algebra but this time we show that $n^{log_a\ 2} \geq c \cdot n^{log_b\ 2}$. We are left with

$$(log_a\ 2 - log_b\ 2) \cdot lg\ n \geq lgc.$$

Now $log_a\ 2 - log_b\ 2$ is negative, so there can exist no postive constant c for which this inequality holds.

Mike.

Question 1.5: Is $\lceil lgn \rceil!$ polynomially bounded?

The answer is no. We want to show that Using "splitting of products", we can underestimate $(lg\ n)!$:

$$\Pi_{i=1}^{(lg\ n)/2} 1 \cdot \Pi_{i=lg\ (n/2)+1}^{lg\ n} (lg\ n)/2.$$

This is equal to

$$((lg\ n)/2)^{(lg\ n)/2}.$$

Now taking the lg of both sides, we receive

$$\frac{lg\ n}{2} \cdot lg(\frac{lg\ n}{2}) \leq lg\ c \cdot d \cdot lg\ n.$$

Now

$$\frac{lg\ n \cdot lg\ lg\ n}{2} \cdot \frac{lg\ n}{2} - d \cdot lg\ n \leq lg\ c.$$

Finally,

$$\frac{1}{2} \cdot lg\ n \cdot (lg\ lg\ n - (1 + 2d)) \leq lg\ c.$$

But clearly $lg\ lg\ n - (1+2d)$ tends to infinity since d is a constant. Therefore, no c exists.

Kaleigh.

Question 1.6: Is $\lceil lglgn \rceil!$ polynomially bounded?

Let $n = 2^{2^k}$.

$$k! \leq k^k \leq c \cdot 2^{2^{kd}}$$

$$klog(k) \leq log(c) + 2^{kd}$$

This inequality holds for $d \geq 1$. So, $\lceil lglgn \rceil!$ is polynomially bounded.

Kaleigh.

Question 1.7: Prove that $log(n!) = \Theta(n\ log(n))$.

Claim: $log(n!) = O(n\ log(n))$.
$log(n!) \leq c \cdot n \cdot log(n)$. Note that

$$log(n!) = log(n) + log(n-1) + \ldots + log(2) + log(1)$$

$$\leq log(n) + log(n) + \ldots + log(n) + log(n) = n \cdot log(n)$$

Therefore, for $c = 1$, this holds.

Claim: $log(n!) = \Omega(n\ log(n))$.
$log(n!) \geq c \cdot n \cdot log(n)$. Via Stirling's approximation, we get $log(n!) \geq log((n/e)^n) = n \cdot log(n) - n \cdot log(e)$. Now, we must show the following is true for a fixed positive constant c:

$$n \cdot log(n) - n \cdot log(e) \geq c \cdot n \cdot log(n).$$

Simplifying,

$$c \leq 1 - log(e)/log(n).$$

This is undefined for $n = 1$ (division by zero) and $n = 2$ (L.H.S. is negative). If we work with base e logarithms (the base of e is arbitrary), then we can find a positive constant $c < 1 - log(e)/log(3)$ for which this inequality is always true for $n \geq n_0 = 3$. Mike.

Question 1.8: Show $\sum_{i=1}^{n} \frac{1}{i} = \Theta(logn)$.

$H_n = \sum_{i=1}^{n} \frac{1}{i}$, where H_n is the harmonic series. We know $H_n = ln(n) + O(1)$. So, $ln(n) < H_n < ln(n) + 1$.

Show that $\sum_{i=1}^{n} \frac{1}{i} = O(log(n))$.

$$H_n < ln(n) + 1 \leq c_1 \cdot log(n)$$

$$\log_e(n) + 1 = \frac{log(n)}{log(e)} + 1 \leq c_1 \cdot \log(n)$$

For $c_1 \geq \frac{log(n)}{log(e)}$, and $n_0 \geq e$ this holds. So $\sum_{i=1}^{n} \frac{1}{i} isO(log(n))$.

Show that $\sum_{i=1}^{n} \frac{1}{i} = \Omega(log(n))$.

$$H_n > ln(n) = \frac{log(n)}{log(e)} \geq c_2 \cdot log(n)$$

For $c_2 \leq \frac{1}{log(e)}$ this holds. So $\sum_{i=1}^{n} \frac{1}{i} = \Omega(log(n))$.

Kaleigh.

Question 2: Prove that the i^{th} Fibonacci number satisfies the equality $F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5}$.

Basis:

$$F_0 = \frac{1}{\sqrt{5}} \cdot ((\frac{1+\sqrt{5}}{2})^0 - (\frac{1-\sqrt{5}}{2})^0)$$

$$= \frac{1-1}{\sqrt{5}} = 0.$$

$$F_1 = \frac{1}{\sqrt{5}} \cdot ((\frac{1+\sqrt{5}}{2})^1 - (\frac{1-\sqrt{5}}{2})^1)$$

$$= \frac{2\sqrt{5}}{2\sqrt{5}} = 1.$$

Inductive Hypothesis: Assume $F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5}$ and $F_{i-1} = (\phi^{i-1} - \hat{\phi}^{i-1})/\sqrt{5}$

Inductive Step:

$$\begin{aligned}
F_{i+1} &= F_i + F_{i-1} \\
&= \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} + \frac{\phi^{i-1} - \hat{\phi}^{i-1}}{\sqrt{5}} \\
&= \frac{\phi^i + \phi^{i-1} - (\hat{\phi}^{i-1} + \hat{\phi}^{i-1})}{\sqrt{5}} \\
&= \frac{\phi^{i-1}(\phi + 1) - \hat{\phi}^{i-1}(\hat{\phi} - 1)}{\sqrt{5}}
\end{aligned}$$

Now we are required to prove that $\phi + 1 = \phi^2$ and that $\hat{\phi} + 1 = \hat{\phi}^2$. We show the first equality only (same idea for second equality).

$$\begin{aligned}
\phi^2 &= \frac{1 + 2\sqrt{5} + 5}{4} \\
&= \frac{3 + \sqrt{5}}{2} = \frac{2 + 1 + \sqrt{5}}{2} \\
&= 1 + \frac{1 + \sqrt{5}}{2} = 1 + \phi
\end{aligned}$$

So now we can conclude that

$$\frac{\phi^{i-1}(\phi + 1) - \hat{\phi}^{i-1}(\hat{\phi} - 1)}{\sqrt{5}} = \frac{\phi^{i+1} - \hat{\phi}^{i+1}}{\sqrt{5}}$$

Mike.

Question 3 (a):

Show that $\Sigma_{k=1}^{n} k^r \leq c \cdot n^{r+1}$.
So

$$\Sigma_{k=1}^{n} k^r \leq \Sigma_{k=1}^{n} n^r = n \cdot n^r = n^{r+1}.$$

Therefore, for $c \geq 1$,

$$n^{r+1} \leq c \cdot n^{r+1}.$$

We can also show that $\Sigma_{k=1}^{n} k^r \geq c \cdot n^{r+1}$. Now, we use "splitting of sums".

$$\Sigma_{k=1}^{n} k^r \quad \geq \quad \Sigma_{k=1}^{n/2} 0 + \Sigma_{k=n/2+1}^{n} (n/2)^r$$

This is then equal to

$$(n/2) \cdot (n/2)^r = (n/2)^{r+1}.$$

Finally,

$$\begin{aligned} (n/2)^{r+1} &\geq c \cdot n^{r+1} \\ c &\leq \frac{1}{2^{r+1}} \end{aligned}$$

Question 3 (b):

First we show that $\Sigma_{k=1}^{n} lg^s\ k \leq c \cdot n \cdot lg^s\ n$, by bounding each term in the sum by $log^s\ n$.

$$n \cdot lg^s\ n \leq c \cdot n \cdot lg^s\ n$$

so $O(n \cdot log^s\ n)$.

Next we show that $\Sigma_{k=1}^{n} lg^s\ k \geq c \cdot n \cdot lg^s\ n$, by "splitting the sum".

$$\Sigma_{k=1}^{1} lg^s\ k + \Sigma_{k=2}^{n/2} lg^s\ k + \Sigma_{k=n/2+1}^{n} lg^s\ k \geq c \cdot n \cdot lg^s\ n$$

Set the first term to 0, all $n/2 - 1$ "lower terms" to 1 and all $n/2$ "higher terms" to $lg^s\ n/2$. We get

$$\Sigma_{k=1}^{1} lg^s\ k + \Sigma_{k=2}^{n/2} lg^s\ k + \Sigma_{k=n/2+1}^{n} lg^s\ k \geq n/2 - 1 + (n/2) \cdot lg^s\ n/2 \geq c \cdot n \cdot lg^s\ n.$$

$$= (n/2) - 1 + (n/2) \cdot (lg^s\ n - lg^s\ 2) = (n/2) \cdot (lg^s\ n) - 1$$

Now for $c \leq 1/4$ and $n_0 = 3$, it is the case that

$$1/2 \cdot n \cdot lg^s\ n - 1 \geq c \cdot n \cdot lg^s\ n.$$

Consider $c = 1/2 - 1/4$,

$$c \cdot n \cdot lg^s\ n = 1/2 \cdot n \cdot lg^s\ n - 1/4 \cdot n \cdot lg^s\ n$$

When $n \geq 3$, $1/4 \cdot n \cdot lg^s\ n \geq 1$.

This establishes that it is $\Omega(n \cdot lg^s\ n)$.
Therefore this sum is $\Theta(n \cdot lg^s\ n)$.
Kaleigh.

Question 3 (c):

We begin by showing that

$$\Sigma_{k=1}^{n} lg^s\ k \cdot k^r = O(lg^s\ n \cdot n^{r+1})$$

We bound each term in the sum by the maximum:

$$\Sigma_{k=1}^{n} lg^s\ k \cdot k^r \leq n \cdot (lg^s\ n \cdot n^r)$$
$$= n^{r+1} \cdot lg^s\ n \leq c \cdot lg^s\ n \cdot n^{r+1}$$

and this is true for $c \geq 1$.
Now we show the Ω bound via "splitting of sums".

$$\begin{array}{rcl}
\Sigma_{k=1}^{n} lg^s\ k \cdot k^r & \geq & c \cdot lg^s\ n \cdot n^{r+1} \\
\Sigma_{k=1}^{n/2} 0 + \Sigma_{k=n/2+1}^{n} lg^s\ k \cdot k^r & \geq & c \cdot lg^s\ n \cdot n^{r+1} \\
0 + (n/2) \cdot lg^s (n/2) \cdot (n/2)^r & \geq & c \cdot lg^s\ n \cdot n^{r+1} \\
\frac{n^{r+1}}{2^{r+1}} \cdot lg^s (n/2) = \frac{1}{2^{r+1}} \cdot n^{r+1} \cdot lg^s\ (n/2) & \geq & c \cdot lg^s\ n \cdot n^{r+1} \\
\frac{1}{2^{r+1}} \cdot lg^s (n/2) & \geq & c \cdot lg^s\ n \\
lg^s\ (n/2) & \geq & 2^{r+1} \cdot c \cdot lg^s\ n \\
lg^s\ n - 1 & \geq & 2^{r+1} \cdot c \cdot lg^s\ n \\
lg^s\ n & \geq & 2^{r+1} \cdot c \cdot lg^s\ n + 1
\end{array}$$

For what value of k is it true that $x \geq kx + 1$ where $x = lg^s\ n$ and $k = 2^{r+1} \cdot c$. The answer is $k \leq 1 - 1/x$.

So let $t = 2^{r+1} \cdot c$.

$$\begin{aligned} lg^s\ n &\geq t \cdot lg^s\ n + 1 \\ \frac{lg^s\ n - 1}{lg^s\ n} &\geq t \\ 1 - \frac{1}{lg^s\ n} &\geq t \end{aligned}$$

For $s > 0$, $n > 2$, let $t = 1 - 1/lg^s\ 3$. Therefore, $c = (1 - 1/lg^s\ 3) \cdot (1/2^{r+1})$.

Now we have to check the special case of $s = 0$. (Is anybody still reading?)

$$\Sigma_{k=1}^{n} lg^s\ k \cdot k^r \geq c \cdot lg^s\ n \cdot n^{r+1}$$
$$\Sigma_{k=1}^{n} k^r \geq c \cdot n^{r+1}$$

(see proof in part (a))

Now we have to do the "base cases" of $n = 1, n = 2$.

For $n = 1$,

$$lg^s\ 1 \cdot 1^r \geq c \cdot lg^s\ 1 \cdot 1^{r+1},$$
$$0 \geq c \cdot 0.$$

For $n = 2$,

$$lg^s\ 1 \cdot 1^r + lg^s\ 2 \cdot 2^r \geq c \cdot lg^s\ 2 \cdot 2^{r+1}$$
$$\begin{aligned} 0 + 1 \cdot 2^r &\geq c \cdot 1 \cdot 2^{r+1} \\ 2^r &\geq c \cdot 2^{r+1} \\ 2^r &\geq \frac{1}{2^{r+1}}(1 - \frac{1}{lg^s\ 3}) \cdot 2^{r+1} \\ 2^r &\geq 1 - 1/lg^s\ 3 \end{aligned}$$

Ok ... finally, it's $\Omega(lg^s\ n \cdot n^{r+1})$ and we can all sleep a little better knowing it.

Mike.

Question 4: Let α be a constant, $0 < \alpha < 1$. Solve the following recurrence:

$$T(n) = T(\alpha \cdot n) + T((1 - \alpha) \cdot n) + n$$

Could use the recursion tree method to visualize what is happening in the recurrence. Let $\beta = \alpha - 1$. The following describes the recurrence tree:

$$\begin{aligned} T(n) &= T(\alpha \cdot n) + T((\beta) \cdot n) + n \\ &= T(\alpha^2 \cdot n) + T(\alpha \cdot \beta \cdot n) + T(\beta^2 n) + 2n \\ &\ldots \end{aligned}$$

Each level in the recursion tree contains n elements. The tree goes either to depth $log_{\frac{1}{\alpha}} n$ or $log_{\frac{1}{\beta}} n$, whichever is larger. When $\alpha < \beta$ $log_{\frac{1}{\alpha}} n < log_{\frac{1}{\beta}} n$. At each level of the tree, the problem size is n. When $\alpha < \beta$, tree has height $log_{\frac{1}{\beta}} n$, the problem size is bound by $n \cdot log_{\frac{1}{\beta}} n$. Otherwise, tree has height $log_{\frac{1}{\alpha}} n$ and the problem size is bound by $n \cdot log_{\frac{1}{\alpha}} n$. It follows that $T(n) = O(nlogn)$.

Kaleigh.

Question 5: (4-7 from CLR)

Part (a): The first observation is that any decision about whether a chip is good or bad must be based on what the good chips say. Consider a chip x that we do not know is good or bad. We ask all chips in the set whether x is good or bad. If there is a majority ($> n/2$) good chips in the set and x is bad, then a majority of the chips will say that x is bad. We may safely conclude that x is in fact bad. If there is a majority ($> n/2$) good chips in the set and x is good, then a majority of the chips will say that x is good. We may safely conclude that x is in fact good. (There is one special case here. Suppose that the number of good chips is exactly $n/2 + 1$ and x is good. Then $n/2$ bad chips will say x is bad and $n/2$ good chips will say x is good. It is easy to verify that, in case of such a tie, we may safely declare x to be good.)

Suppose there are $n/2$ good chips and $n/2$ bad chips. The good chips do not lie, so we may conclude that each good chip will say that the all other $n/2 - 1$ good chips are in fact good and they will say that all $n/2$ bad chips are in fact bad. The bad chips on the otherhand conspire to fool Professor Diogenes as follows: each bad chip says that all other bad chips are good and all good chips are bad.

Since we have complete symmetry between good and bad and there is no majority of good chips, Professor Diogenes can not identify any chip as being either good or bad.

Foreword to Part (b): We may assume that a majority of the chips are good. Consider the following scheme:

```
Pick a chip $x$ at random from the set.
Ask all other chips whether $x$ is good or bad.
If a majority of the chips say that $x$ is good, then $x$ is in fact
    good.
Otherwise, $x$ is in fact bad.
```

Claim 1 *We can classify a single chip as being good or bad in $\Theta(n)$ time by the above algorithm.*

Proof. Since we have a majority of good chips, we can be believe the majority answer we receive from all $n-1$ chips. ∎

Claim 2 *If chip x is good, we can classify all chips as being good or bad in $\Theta(n)$ time.*

Proof. If x is good, we can believe it. Therefore we need only ask x what it thinks of each of the remaining $n-1$ chips. Finished. ∎

If the arbitrarily choosen chip x is bad, we are stuck since we can not believe x and we still have not located a good chip. **This whole problem really boils down to locating a single good chip.**

Part (b): We may assume a majority of good chips. Consider a pair of chips x and y. We say that this x, y pair is *mutually agreeing* if x says that y is good, and y says that x is good. Otherwise, we say that they are *disagreeing*. Consider the following algorithm:

```
1. Let $S = \{s_1, \ldots, s_n\}$ be our set of chips.
2. If $|S| = 2$, then return one of the two elements.
3. If $|S| = 3$, then do all three pairwise comparisons. Since there
     must be at least one mutually good pair, choose arbitrarily
     either of the elements of this pair.
4. Arbitrarily pair all of the $|S|$ chips (so that we have $|S|/2$
     pairs).
5. (If $|S|$ is odd, arbitrarily choose three elements in $S$, remove
     them from $S$, and treat this threesome in the same manner
     as step 3 above.)
6. For each pair $x, y$, compare $x$ and $y$.
7.        If $x, y$ are mutually good, then let $S' \leftarrow S' \cup \{x\}$.
8. Let $S \leftarrow S'$.
9. Go to step 2.
```

The output from this algorithm is one chip, call it x. We claim that x is good. If so, we can trust x's opinon of the remaining $n-1$ chips so we are finished. We also claim that the algorithm runs in time $\Theta(n)$.

First, we show that x must be a good chip.

Consider our set S of n chips of which a majority are good. Consider an arbitrary pairing of the chips and our set S' as in steps 4 and 5 of the algorithm. Let b be the number of chips in mutually agreeing pairs but where both members of this pair x and y are bad chips (they lie and say that they are both good). Let g be the number of of chips in mutually agreeing pairs but where both members of this pair x and y are good chips (they tell the truth and both say "good"). Let b' be the number of bad chips that are in disagreeing pairs (For a pair x, y, one says the other is "good" and the other calls the second one "bad".). Let g' be the number of good chips that are in disagreeing pairs (For a pair x, y, one says the other is "good" and the other calls the second one "bad".).

Claim 3 *The number of mutually agreeing pairs of truly good chips g is always strictly greater than the number of mutually agreeing pairs of lying bad chips $b/2$.*

Proof. I will show this by a method called *contradiction.* For simplicity, let us assume that n is even (it is easy to show this when n is odd). We know that

$$b + g + b' + g' = n \tag{1}$$
$$g + g' > n/2; b + b' < n/2 \tag{2}$$
$$b + b' < g + g' \tag{3}$$
$$b' \geq g'. \tag{4}$$

Statement 4 follows from the fact that no disagreeing pair is comprised of two good chips. Therefore at least half must be bad chips.

Now suppose that $g < b$ (the number of truly good chips in mutually agreeing pairs g is strictly less than the number of bad chips in mutually agreeing pairs b). Now since b' is also $\geq g'$, it must be the case that

$$b + b' > g + g'.$$

This contradicts (3) above. Therefore we may conclude that $g \geq b$. ▮

This means that when we are finished lines $6, 7, 8$ in the algorithm, our set S' has a majority of good chips in it and S has size $n/2$. We now return to step 2 and repeat until the size of S is either 2 or 3. We have performed $\lfloor n/2 \rfloor$ comparisons at each stage of the algorithm

Part (c): The recurrence that describes the running time is

$$T(n) = T(n/2) + n/2.$$

We show by induction that this is $O(n)$ (it is equally easy to show that it is $\Omega(n)$).

Basis: $n = 2$. $T(2) = 1 \leq c \cdot 2$
$n = 3$. $T(3) = 3 \leq c \cdot 3$ when $c \geq 1$.

Inductive Hypothesis: $T(i) \leq c \cdot i$, for all $i < n$.
Inductive Step:

$$T(n) = c \cdot (n/2) + n/2 = (n/2)(c + 1) \leq c \cdot n \tag{5}$$

This is true when $c \geq 1$. Mike.

Question 6:

```
for $p, q$ where $1 \leq p < q \leq k$ do
    $x \leftarrow min\{X[p], X[q]\}$
    $y \leftarrow max\{X[p], X[q]\}$
    $j \leftarrow n + 1; i \leftarrow 0$
    while true do
        repeat $j \leftarrow j - 1$
        until $A[j] = x$ or $j < 1$
        repeat $i \leftarrow i + 1$
        until $A[i] = y$ or $i > n$
        if $i < j$ then
            exchange $A[i] \leftrightarrow A[j]$
        else break
        end do
```

This requires time $k^2 \cdot n$.

(b): The lower bound for comparison sort is $\Omega(n\ log(n))$. If k is a function of n, then as long as $k = O(\sqrt{(log\ n)})$ our algorithm does not worse than the lower bound.

(c): The lower bound for comparison does hold. Consider the case where $X_i \cap X_j = \emptyset$, for all $1 \leq i, j \leq n$. In such a case, the fact that each a_i is restricted to a small set of k values X_i does not help in any way and the search tree is at least as big as the search tree for standard comparison sort. Mike.

Question 7: The following answer assumes (correctly) that we can merge $\sqrt(n)$ lists in time $\Theta(n^{2/3})$. We will come back to this question at a later point in the course. The analysis can be modified accordingly if you assumed you could do this in $\Theta(n)$ time.
Question 7 (a): $T(n) = k \cdot T(n/k) + \Theta(n)$.

Question 7 (b): The depth of the tree is $log_k(n)$ and the work done at each level is $\Theta(n)$. Therefore, the overall worst case upper bound is $O(n \cdot logn)$.

Question 7 (c): $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + \Theta(n^{2/3})$

Question 7 (d):
Question 7 (e): First, we determine the height of the tree. We can write this as a tower of $\frac{1}{2}$'s as follows:

$$n^{1/2^{1/2^{\ldots i \ times \ \ldots 1/2}}} = c$$

Let c be the base condition in the algorithm (the point where we stop the recursion). In order to simplify the algebra let $c = 2$. (Regardless, it is not necessary to stop at $c = 1$. We could stop at $c = 2$ or $c = 3$ and use a simpler sorting method.)

Simplifying, we get

$$n^{1/2^i} = c \tag{6}$$
$$1/2^i \cdot log(n) = 1 \text{ when } c = 2 \text{ and using log base 2} \tag{7}$$
$$log(n) = 2^i \tag{8}$$
$$log\ log(n) = i \tag{9}$$

Therefore, the height of the tree is $O(log\ log(n))$.

Now we answer how much work is done at each level of the tree. At height i, $1 \leq i \leq c \cdot log\ log(n)$, the exponent on n is described by the following series (you should verify this; it isn't difficult):

$$(\Sigma_{j=0}^{i-1} \frac{2^j}{2^i}) + \frac{2}{2^i}$$

$$= 1/2^i \cdot (\Sigma_{j=0}^{i-1} 2^j) + \frac{2}{2^i}$$
$$= 1/2^i \cdot 2^i - 1 + 2/2^i$$
$$= 2^i + 1/2^i$$

Now, the total number of steps is

$$\Sigma_{i=1}^{log\ log(n)} n^{\frac{2^i+1}{2^i}}$$

Bounding each term in the sum by its maximum (this was sufficient for the purposes of this course), we receive $n^{3/2} \cdot log\ log(n)$. Mike.

Question 8:

Claim 4 *$o(g(n)) \cap \omega(g(n)) = \emptyset$.*

Proof. Suppose $o(g(n)) \cap \omega(g(n)) \neq \emptyset$. Let $f(n)$ be an element of $o(g(n)) \cap \omega(g(n))$. From the definition of $o(g(n))$, for any positive constant $c > 0$, there exists n_0 s.t. $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$. From the definition of $\omega(g(n))$, for any positive constant $c' > 0$, there exists $n'_0 > 0$ s.t. $0 \leq c' \cdot g(n) < f(n)$ for all $n \geq n'_0$. Therefore, if $c = c'$,

$$f(n) < c \cdot g(n) < f(n),$$

for all $n \geq max\{n_0, n'_0\}$. Clearly, no such $f(n)$ exists. Contradiction. ∎

Mike.

Question 9: If the running time of $\Phi(n)$ is $O(n^d)$, where d is a constant, then it is the case that the running time of $\Phi \leq c \cdot n^d$ for constant c and all $n \geq n_0$. If d is large (how useful is a $n^{10^5=d}$ algorithm?) or if c is too large ($c = 183, 234, 293, 192, 872, 102$). Mike.

Question 10:

(a) Construct B_4.

Construction Start:
Start with an array S of 16 integers. We are going to assign each of the $(0 \ldots 15)$ integers to one position in the array. Start with

$$S = (0, 1, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 1).$$

Binary view of S,

$$SB = 00001??????????1.$$

Note: you can easily get SB from S by setting $SB[i]$ = first bit of binary S.

Construction Step:
Assume array indexed from 1 to 16. Grow S from both ends. Initialize left and right indexes, $l = 6$ and $r = 15$.

Until all of the positions of S have been filled, do the following:

Let a candidate $c = \text{dec}(\ SB[l-3]SB[l-2]SB[l-1]0\ )$. If c has not already been assigned a location in S, then set $S[l] = c$. Otherwise, set $S[l] = c + 1$. Increment l.

Check to see if $S[r]$ must be set. $S[r]$ must be set when it has only one possible value that hasn't been assigned to S. $S[r]$ can be either $c0 = 0S[r+1]S[r+2]S[r+3]$ or $c1 = 1S[r+1]S[r+2]S[r+3]$. When $r + i > 16$, cycle around.

Example

$S =$ 0 1 X X X X X X X X X X X X X 8
$SB =$ 0 0 0 0 , 1 X X X , X X X X , X X X 1
$S =$ 0 1 2 X X X X X X X X X X X X 8
$SB =$ 0 0 0 0 , 1 0 X X , X X X X , X X X 1
$S =$ 0 1 2 4 X X X X X X X X X X 12 8
$SB =$ 0 0 0 0 , 1 0 0 X , X X X X , X X 1 1
$S =$ 0 1 2 4 9 X X X X X X X X X 12 8
$SB =$ 0 0 0 0 , 1 0 0 1 , X X X X , X X 1 1
$S =$ 0 1 2 4 9 3 X X X X X X X X 12 8
$SB =$ 0 0 0 0 , 1 0 0 1 , 1 X X X , X X 1 1
$S =$ 0 1 2 4 9 3 6 X X X X X X 14 12 8
$SB =$ 0 0 0 0 , 1 0 0 1 , 1 0 X X , X 1 1 1
$S =$ 0 1 2 4 9 3 6 13 X X X X X 14 12 8
$SB =$ 0 0 0 0 , 1 0 0 1 , 1 0 1 X , X 1 1 1
$S =$ 0 1 2 4 9 3 6 13 10 X X X X 14 12 8
$SB =$ 0 0 0 0 , 1 0 0 1 , 1 0 1 0 , X 1 1 1

$S = 0\ 1\ 2\ 4\ 9\ 3\ 6\ 13\ 10\ 5\ \mathrm{X}\ \mathrm{X}\ \mathrm{X}\ 14\ 12\ 8$
$SB = 0\ 0\ 0\ 0\ ,\ 1\ 0\ 0\ 1\ ,\ 1\ 0\ 1\ 0\ ,\ 1\ 1\ 1\ 1$
$S = 0\ 1\ 2\ 4\ 9\ 3\ 6\ 13\ 10\ 5\ 11\ 7\ 15\ 14\ 12\ 8$
$SB = 0\ 0\ 0\ 0\ ,\ 1\ 0\ 0\ 1\ ,\ 1\ 0\ 1\ 0\ ,\ 1\ 1\ 1\ 1$

So, $SB = 0000\ 1001\ 1010\ 1111$ and $S = 0, 1, 2, 4, 9, 3, 6, 13, 10, 5, 11, 7, 15, 14, 12, 8$.

(b) Algorithm is described above. The following is a pseudocode implementation.

```
integers $k$, $r$, $l$, $c$, $l_0$, $l_1$ $size = 2^k$
integer array $S[0..size-1]$
integer array $SB[0..size-1] = 0$
$S[0] = 0$ $S[1] = 1$ $S[size-1] = 8$
$r = 2$ $l = size - 2$
$l_0 = binShiftRight(S[l+1])$
$l_1 = l_0 + 8$
while $(r < l)$
        $c = binaryshiftleftofS[r-k]$
        found=false i=0
        while $(i < r)$ and (not found)
                if $(S[i] = c)$ found:=true
                else $i++$ fi
                end while
        $i = l + 1$
        while $(i < size)$ and (not found)
                if $(S[i] = c)$ found:=true
                else $i++$ fi
                end while
        if found then $S[r] := c + 1$
        else $S[r] := c + 1$ fi
        if $S[r] = l_0$ then                    $S[l] := l_1$
                $l--$
                $l_0 = binshiftrightofS[l+1]$
                $l_1 = l_0 + 8$
        else if $S[r] == l_1$ $S[l] = l_0$
                $l--$
                $l_0 = binaryshiftrightofS[l+1]$
                $l_1 = l_0 + 8$
```

```
            $r++$ fi
    end while
```

This is less than 2^{2k}.

(c) Prove that this construction always leaves a candidate for a position in S until all 2^k positions are filled. Each $k-1$ prefix of k-string occurs in proper S at most twice. Using this approach, if a $k-1$ prefix is completed into one of its two possible k-strings and the second occurance of that $k-1$ prefix already exists in S, the second occurance will be completed with the remaining k-string. We won't get the case of having a $k-1$ prefix that cannot be completed. Since all 2^{k-1} prefices can be completed in the length 2^k S sequence, it follows that S contains all 2^k possible completions.
Kaleigh.

Question 11: Part (a): Any of the k cards are equally likely to be chosen. Any of the $k!$ possible orderings of $e_1, \ldots, e_k$ is equally likely. However only one of these orderings will have the property $e_1 < e_2 < \ldots < e_k$. Therefore, the probability is $1/(k!)$.

Part (b): Let's do this for $k = 3$ to show you the idea. Try it for arbitrary k later. There are a total of n^3 ways to choose 3 elements from the list of n items (with repetition).

Case 1: There are

$$\binom{n}{3} = n(n-1)(n-2) = n^3 - 3n^2 + 2n$$

ways to choose three distinct elements. Therefore the probability that I choose three distinct elements is

$$\frac{n^3 - 3n^2 + 2n}{n^3}$$

.

Now, calculate the number of ways to choose 2 distinct element.

Case 2: The first possibility is that I choose a first element, then choose a second element distinct from the first, then choose a third element that is equal to either the first or second item. There are $2n(n-1)$ ways to do this. Therefore the probability that I choose one duplicate and this duplicate is the third choice is

$$\frac{2n(n-1)}{n^3}$$

.

Case 3: The second possibility is that I choose a first element, then choose a second element equal to the first, then choose a third distinct element. There are $n(n-1)$ ways to do this and the probability is therefore $n(n-1)/n^3 = (n-1)/n^3$.

Case 4: Finally, there are n ways to choose three equal elements. The probability of choosing three equal elements is $1/n^2$.

(You can verify that these 4 probabilities sum to 1.) Now we calculate the probability that the 3 choices are sorted.

Case 4 is the easiest. Since all elements are equal the probability is 1 that they are sorted. Therefore, in total, there is a

$$\frac{1}{n^2}$$

chance that we choose 3 equal elements and they are sorted.

For Case 1, there are 3! orders and only one is correct. This gives a probability of

$$\frac{n^2-3n+2}{3!\cdot n^2}.$$

For Case 2, there are 2 possible orders for each of the $2n(n-1)$ ways of choosing a duplicate element on the 3rd choice. Of these 4 possibilities, only 2 are sorted. This gives a probability of

$$\frac{n-1}{2n^2}.$$

For Case 3, there are $(n-1)/n^2$ possible orders of which half are correct. This gives a probability of

$$\frac{n-1}{2n^2}.$$

If we sum these 4 probabilities, we receive the probability of choosing 3 elements ordered. This sum is

$$\frac{n^2+3n+2}{3!\cdot n^2}.$$

I will leave it to you to try it for arbitrary k. If you are having trouble, then try it for $k=4$ first. Mike.

Question 12:

We need to find the median, the fourth smallest element in the list of 7 elements. The best way to solve this is to set up a tournament of comparisons between the elements. Consider building an upside down binary tree. There are 7 leaves, l_i is labeled by element a_i. Let consecutive pairs of elements be paired, and the seventh element is unpaired in the tree.

$w(a_i, a_j)$ is the winner of the comparison between a_i and a_j. In this case the contest is to find the smallest element. So consider the winner to be the minimum.

```
         w( w(w(a1, a2), w(a3, a4)), w(w(a5, a6), a7)) )
                     /            \
  w(w(a1, a2), w(a3, a4))      w(w(a5, a6), a7))
       /       \                /      \
w(a1, a2)  w(a3, a4)        w(a5, a6)   a7
  / \         / \              / \
a1  a2       a3  a4           a5  a6
```

At the top of the tree is the smallest element a_s. Remove a_s from the tournament tree and remove the internal vertex containing the bottom-most comparison with a_s. This means that we will have to redo the comparisons where a_s was a winner of the comparison to determine the second smallest element. Repeat this process to remove the second smallest element, redo the necessary comparisons to sift the third smallest element to the root of the tournament tree. For the last step, remove the third smallest element and redo the necessary comparisons to sift the fourth smallest element to the root. Done.

How many comparisons did this take? To find the smallest element, we did a comparison at each internal node of the tournament tree. There are 6 internal nodes, so 6 comparisons. Now are each subsequent ¡em¿remove smallest element and redo necessary comparisons¡/em¿ operations, we must redo at most 2 comparisons. As we do three operations with at most 2 comparisons each, we do a total of 12 comparisons.

Things to think about. Consider how many comparisons it would take to sort this list of 7 elements. Are we doing fewer comparisons by using a tournament scheme? What is the best case input for this tournament algorithm?

Kaleigh.

Question 13: Solution to come. Not on the quiz.

Question 14: Solution to come. Not on the quiz.