CSCI 3104 Spring 2017 Problem Set 2

Samuel Cuthbertson (06/16)
Grant Baker (07/23)
Connor Hudson (05/07)

---

1. *Prove or disprove each of the following claims, where f(n) and g(n) are functions on positive values.*

   (a) $f(n) = O(g(n)) \implies g(n) = \Omega(f(n))$
   From the definitions of $O(n)$ and $\Omega(n)$, we have constants c_1, c_2, N_1, and N_2 such that

$$\begin{aligned} f(n) = O(g(n)) &\implies f(n) \leq c_1 g(n) \quad \forall n \geq N_1 \\ g(n) = \Omega(f(n)) &\implies g(n) \geq c_2 f(n) \quad \forall n \geq N_2 \end{aligned}$$

   We know that c_1 and N_1 exist, by assumption. Then, we must find c_2 and N_2. If we choose $c_2 = \frac{1}{c_1}$, and $N_2 = N_1$, we get

$$\boxed{c_1 g(n) \geq f(n) \implies g(n) \geq \frac{1}{c_1} f(n)}$$

   Therefore, we have

$$f(n) = O(g(n)) \implies g(n) = \Omega(f(n))$$

   (b) $f(n) = O(g(n)) \implies 2^{f(n)} = O(2^{g(n)})$
   This is false. We take the simple counterexample[1]

$$\begin{aligned} f(n) &= 2n^2 \\ g(n) &= n^2 \end{aligned}$$

   We see clearly that it is the case that $f(n) = O(g(n))$. If we let $c = 2$ and $N = 1$ we find that

$$2n^2 \leq 2n^2 \quad \forall n \geq 1$$

[1]Adapted from Stackoverflow user *Mehrdad* at `http://stackoverflow.com/questions/12361448/`.

However, consider $2^{f(n)} = 2^{2n^2}$ and $2^{g(n)} = 2^{n^2}$. We find that

$$2^{2n^2} = \left(2^2\right)^{n^2} = (4)^{n^2} = 4^{n^2}$$

We want to find new c_2 and N_2 such that

$$4^{n^2} \leq c_2 2^{n^2} \quad \forall n \geq N_2$$

However, shown by way of contradiction, assume c_2 exists.

$$4^{n^2} \leq c_2 2^{n^2} \implies \frac{4^{n^2}}{2^{n^2}} \leq c_2 \implies \left(\frac{4}{2}\right)^{n^2} \leq c_2$$

By the Archimedean Property, for any number c_2 there is N' such that

$$N' > \log_2 c_2 \implies 2^{N'} > c_2$$

This implies that no such N_2 exists.

Therefore, in general $\boxed{2^{f(n)} \neq O(2^{g(n)})}$

(c) $f(n) = O((f(n))^2)$

This is false for $\lim_{n\to\infty} f(n) < 1$. We take the simple counterexample

$$f(n) = \frac{1}{n}$$

To disprove $f(n) = O((f(n))^2)$, we assume by way of contradiction that it is the case. Therefore, there exist positive constants c and N such that

$$\frac{1}{n} \leq c\frac{1}{n^2} \quad \forall n \geq N$$

However by manipulating the equation, we get

$$\frac{n^2}{n} \leq c \implies n \leq c$$

By the Archimedean Property, there is always N such that $N > c$. Therefore, no such N can exits.

Therefore, in general $\boxed{f(n) \neq O((f(n))^2)}$

However, if we add the additional restriction that $f(n) \geq 1$, then the property holds true.

$$\begin{gathered} f(n) \geq 1 \\ (f(n))^2 \geq f(n) \\ \implies f(n) \leq (f(n))^2 \quad \forall n \geq 1 \end{gathered}$$

2. *Consider the following Python function:*

```
0 def find_max (L):
1     max = 0
2     for x in L:
3          if x > max:
4              max = x
5     return max
```

Suppose list L has n elements.

(a) *In asymptotic notation, determine the best case running time as function of n.*

The best case for this algorithm is the max value being the first item, as lines 3 and 4 will only run once. However, note that regardless of where in the list the max is, the loop on line 2 will iterate over the entire list. Also note that the both operations inside and outside the loop are $O(1)$, as they don't scale with variable input size. This means that we run a $O(1)$ loop n times, if L is of length n. Thus we have

$$O(1) + O(n) \times O(1) = \boxed{O(n)}$$

(b) *In asymptotic notation, determine the worst case running time as function of n.*

In the worst case, the list is ordered from smallest to largest. In this case, we set the `max` variable each iteration. However, we are still only iterating through the length of the list once, with each iteration taking $O(1)$, and with n iterations. So, this is the same situation as above, and the function has running time $\boxed{O(n)}$

(c) *Now, assume L is sorted. Give an algorithm that takes asymptotically less time than the above algorithm, but performs the same function. Prove that your algorithm is correct, and explain why this algorithm is useless.*

```
0 def find_max_sorted(L):
1     max = L[1]
2     if L[-1] > max:
3         max = L[-1]
4     return max
```

This algorithm is correct because of the base assumption that L is sorted. That means the maximum value in L is either the first or last element, so our algorithm only has to compare these two elements. This algorithm takes $O(1)$, since the function is only looking at the first and last elements of the sorted list, and completes no function which depends on the length of L. This is asymptotically much better than $O(n)$.

However, sorted arrays are rarely encountered and adding code to verify that an array is sorted would run in either a similar or larger big-O bound, as it has more work to do. Moreover, sorting an array takes much longer with quicksort[2] coming in at $O(n \log n)$, which is clearly larger than $O(n)$. Having an algorithm that only works on sorted arrays is almost useless, due to both of the above reasons.

[2]Quicksort running time from `https://www.cise.ufl.edu/class/cot3100fa07/quicksort_analysis.pdf`

3. *Solve the following recurrence relations using the method specified.*

   (a) $T(n) = T(n-2) + n$ *by "unrolling" (tail recursion).*[3]

$$\begin{aligned}
T(n) &= T(n-2) + n \\
&= T(n-4) + 2n \\
&= T(n-6) + 3n \\
&= T(n-8) + 4n \\
&\vdots \\
&= T(0) + (\frac{n}{2})n \\
&= \frac{1}{2}n^2 + c
\end{aligned}$$

   Thus, we see that $\boxed{T(n) = \Theta(n^2)}$

   (b) $T(n) = 2T(\frac{n}{2}) + n^3$ *by the master method.*

   Note that here we have $a = 2$, $b = 2$, and $f(n) = n^3$, and thus $d = 3$. We then have $b^d = 2^3 = 8$, and $a = 2 < b^d$. Therefore, by the Master Theorem, $\boxed{T(n) = \Theta(n^3)}$

   (c) $T(n) = 2T(\frac{n}{2}) + n^3$ *by the recurrence tree method.*

$$\begin{tikzcd}
 & & & n \arrow[dll] \arrow[drr] & & & & n^3 \\
 & \frac{n}{2} \arrow[dl] \arrow[dr] & & & & \frac{n}{2} \arrow[dl] \arrow[dr] & & 2\left(\frac{n}{2}\right)^3 \\
\frac{n}{4} & & \frac{n}{4} & & \frac{n}{4} & & \frac{n}{4} & 4\left(\frac{n}{4}\right)^3 \\
\vdots & & \vdots & & \vdots & & \vdots & \vdots
\end{tikzcd}$$

   This will result in $\log_2 n$ levels, with the ith level requiring $\frac{n^3}{4^i}$ operations. So, we get the sum

$$\sum_{i=1}^{\log_2 n} \frac{n^3}{4^i} = n^3 \sum_{i=1}^{\log_2 n} \frac{1}{4^i} \leq n^3 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i = n^3 \frac{1}{1 - \frac{1}{4}} = n^3 \frac{4}{3} = \Theta(n^3)$$

[3]Reference used: `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf`

The sum was computed by the geometric series formula.

Therefore, by counting the operations required at each level and the number of levels of the recurrence tree, we find that this recurrence relation is $\boxed{\Theta(n^3)}$

(d) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$ *by the master method.*

First note that $\sqrt{n} = n^{1/2}$. We then have $a = 2,\ b = 4,\ d = \frac{1}{2}$. $a = 2 = b^d = \sqrt{4} = 2$, so by the Master Theorem, $\boxed{T(n) = \Theta(\sqrt{n}\log n)}$

(e) $T(n) = 3T(\frac{n}{2}) + n$ *by the recurrence tree method.*[4]

$$\begin{tikzcd}
& & & & n \arrow[dlll] \arrow[d] \arrow[drrr] & & & & & n \\
& \frac{n}{2} \arrow[dl] \arrow[d] \arrow[dr] & & & \frac{n}{2} \arrow[dl] \arrow[d] \arrow[dr] & & & \frac{n}{2} \arrow[dl] \arrow[d] \arrow[dr] & & 3\frac{n}{2} \\
\frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & \frac{n}{4} & 3^2\frac{n}{4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{tikzcd}$$

This gives us a tree with height $\log_2 n$, and for level i we require $\left(\frac{3}{2}\right)^i n$ operations. So, we end up with the sum

$$\sum_{i=0}^{\lg n-1} \left(\frac{3}{2}\right)^i n = n \sum_{i=0}^{\lg n-1} \left(\frac{3}{2}\right)^i = n\frac{\left(\frac{3}{2}\right)^{\lg n} - 1}{\frac{3}{2} - 1}$$

which is evaluated using the geometric sum formula. Using a bit of algebraic manipulation,

[4]Collaboration with Luke Meszar.

$$
\begin{aligned}
\sum_{i=0}^{\lg n-1}\left(\frac{3}{2}\right)^i n &= n\frac{\left(\frac{3}{2}\right)^{\lg n}-1}{\frac{3}{2}-1} \\
&= n\frac{\left(\frac{3}{2}\right)^{\lg n}-1}{\frac{1}{2}} \\
&= n\left(2\left(\frac{3}{2}\right)^{\lg n}-2\right) \\
&= 2n\left(\frac{3^{\lg n}}{2^{\lg n}}\right)-2n \\
&= 2n\left(\frac{3^{\frac{\log_3 n}{\log_3 2}}}{n}\right)-2n \\
&= 2\left(3^{\log_3 n}\right)^{\frac{1}{\log_3 2}}-2n \\
&= 2n^{\lg 3}-2n \\
&= \Theta\left(n^{\lg 3}\right)
\end{aligned}
$$

We find that $\boxed{T(n)=\Theta\left(n^{\lg 3}\right)}$

4. *Can the master method be applied to the recurrence $T(n) = 4T(\frac{n}{2}) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.* [5]

   Using the template for the Master Method, we have $a = 4$, $b = 2$, and $f(n) = n^2 \log n$. We want to know if $f(n) = n^2 \log n$ fits one of the Master Theorem cases for $n^{\log_3 4}$.

   Since we have that $n^{\log_2 4} = O(n^2)$, and we also have that $n^2 \log n = \Omega(n^2)$. This means that $n^2 \log n \neq O(n^{\log_2 4 - \epsilon}) \neq \Theta(n^{\log_2 4}) \quad \forall \epsilon > 0$. Therefore, the first two cases for the Master Theorem do not apply.

   For the third case, if we pick $\epsilon \approx 0.74$, we see that $n^2 \log n$ is indeed $\Omega(n^2)$. However, we must also check if $4f(\frac{n}{2}) \leq cf(n)$ for $c < 1$ for sufficiently large n. This inequality evaluates to $\frac{4}{8}n^2(\log \frac{n}{2}) = \frac{4}{8}n^2(\log n - \log) \leq cn^2 \log n$, which is trivially false.

   Therefore, since none of the three cases for the Master Theorem hold, **we cannot use the Master Method on this recurrence**.

   To find an upper asymptotic bound on the running time, we use the loop unrolling method as follows.[6]

[5] Sanity checks from Jeffrey "Matt" Maierhofer

[6] The following site was incorrect, but helped our thinking:
`http://web.cs.ucdavis.edu/ gusfield/cs222f07/mastermethod.pdf`

$$
\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + n^2 \log n \\
&= 4\left(4T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \log\frac{n}{2}\right) + n^2 \log n \\
&= 4\left(4\left(4T\left(\frac{n}{16}\right) + \left(\frac{n}{4}\right)^2 \log\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \log\frac{n}{2}\right) + n^2 \log n \\
&\vdots \\
&= T(0) + \sum_{i=0}^{\log_2 n} 4^i \frac{n^2}{2^{2i}} \log\frac{n}{2^i} \\
&= T(0) + \sum_{i=0}^{\log_2 n} n^2 \log\frac{n}{2^i} \\
&= T(0) + n^2 \log\left(\prod_{i=0}^{\log_2 n} \frac{n}{2^i}\right) \\
&= O\left(T(0) + n^2 \log\left(n^{\frac{\log_2(2n)}{2}}\right)\right) \\
&= O\left(n^2 \left(\frac{\log_2(2n)}{2}\right) \log_2 n\right) \\
&= O\left(n^2 \log_2 n \log_2 n\right) \\
&= \boxed{O\left((n \lg n)^2\right)}
\end{aligned}
$$

This is an asymptotic upper bound for the recurrence relation for $T(n)$.

5. *Professor Snape has n magical widgets that are supposedly both identical and capable of testing each other's correctness. Snape's test apparatus can hold two widgets at a time. When it is loaded, each widget tests the other and reports whether it is good or bad. A good widget always reports accurately whether the other widget is good or bad, but the answer of a bad widget cannot be trusted. Thus, the four possible outcomes of a test are as follows:*

Widget A says	Widget B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

(a) *Prove that if $\frac{n}{2}$ or more widgets are bad, Snape cannot necessarily determine which widgets are good using any strategy based on this kind of pairwise test. Assume a worst-case scenario in which the bad widgets are intelligent and conspire to fool Snape.*[7]

Let $l \geq k$, where k is the number of good widgets and l is the number of bad widgets. We know that $k + l = n$. Assign every good widget $g_i, i = 1, ..., k$ a corresponding bad widget $b_i, i = 1, ..., k$. For leftover bad widgets $b_j, j = k + 1, ..., l$, label them $\hat{b_i}$ if they exist.

Then, we have three groups of widgets. Let $G = \{g_i | i = 1, ..., k\}$ be the set of all good widgets, and let $B = \{b_i | i = 1, ..., k\}$ be the set of all the corresponding bad widgets. Let $\hat{B} = \{\hat{b_j} | j = k+1, ..., l\}$ be the set of remaining bad widgets.

For each good widget $g_i \in G$, let it output:

- "Good", if examining any other $g_i \in G$.
- "Bad", if examining any of the $b_i \in B \cup \hat{B}$, including the $\hat{b_i}$.

This is exactly the expected behavior of a good widget.

[7]Collaboration with Luke Meszar and Jeffery 'Matt' Maierhofer

For each bad widget $b_i \in B$, let it output:

- "Good", if examining any other $b_i \in B$.
- "Bad", if examining any $g_i \in G$ or any $\hat{b_j} \in \hat{B}$.

For each bad widget $\hat{b_j} \in \hat{B}$, let it output:

- "Bad", if examining any other widget.

By construction, widgets in G will claim that all the bad widgets are in $B \cup \hat{B}$. Widgets in B will claim that all the bad widgets are in $G \cup \hat{B}$. Since we know that there are equal or more bad widgets than good widgets, there is no way of distinguishing B and G, since $|B| = |G|$ by construction and they are each claiming themselves as the set of good widgets.

Therefore, if there are $\frac{n}{2}$ or more bad widgets in a set of n widgets, Snape will be unable to determine the identity of all the widgets.

(b) *Consider the problem of finding a single good widget from among the n widgets, assuming that more than $\frac{n}{2}$ of the widgets are good. Prove that $\lfloor \frac{n}{2} \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

Consider the set of widgets $\{w_1, w_2, ..., w_{n-1}, w_n\}$, where it is known that more than $\frac{n}{2}$ widgets are good.

Group the widgets into pairs of the form $p_i = \{w_{2i-1}, w_{2i}\}$, for $i = 1, ..., \lfloor \frac{n}{2} \rfloor$. Note that if n is odd, there will be a leftover widget w_n.

Run a pairwise test within each pair p_i for $i = 1, ..., \lfloor \frac{n}{2} \rfloor$. Since there are $\lfloor \frac{n}{2} \rfloor$ pairs, we know that there are $\lfloor \frac{n}{2} \rfloor$ pairwise tests.

- If both widgets report "Good", then we know they are the same—they are both good or both bad. Discard w_{2i}.
- If at least one widget reports "Bad", then we know that at least one is bad. Discard both w_{2i-1} and w_{2i}.

This takes $\lfloor \frac{n}{2} \rfloor$ comparisons, and leaves us with a set of widgets of size at most $\lceil \frac{n}{2} \rceil$.

To prove that this algorithm is correct, we use an invariant. We know that there are strictly more good widgets than bad. This is true in the new set, by construction:

- Since each pair of 1 good and 1 bad is discarded, the difference in the number of good widgets and the number of bad widgets does not change for this operation.
- Suppose all pairs that are both bad report two "Good". In this case, we keep all pairs of widgets that are the same. There are two cases for dealing with these pairs:
  - If there are more good pairs than bad pairs, then discarding one widget from each pair still results in more good widgets than bad ones.
  - If there are an equal number of good pairs and bad pairs, then for the original condition of more good widgets than bad widgets, it must be the case that n is odd and the final, unobserved widget w_n is good. So, keeping one widget from each pair as well as w_n still guarantees more good widgets than bad ones.

This produces a new set in which there are strictly more good widgets than bad widgets.

If we repeat this until we are left with a set of size 1, then we are left with a set with still strictly more good widgets than bad widgets. Since there is only one widget in this set, it must be good. Label it w_G.

This algorithm takes $\Theta(n)$ steps to find w_G, since for each set n we are dividing the problem into a set of size at most $\lceil \frac{n}{2} \rceil$. Let $T(n)$ be the number of comparisons:

$$T(n) = \sum_{i=1}^{\lg n} \frac{n}{2^i} < \sum_{i=1}^{\infty} \frac{n}{2^i} = n \implies T(n) = \Theta(n)$$

(c) *Prove that the good widgets can be identified with $\Theta(n)$ pairwise tests, assuming that more than $\frac{n}{2}$ of the widgets are good. Give and solve the recurrence that describes the number of tests.*

Using part (b), we find w_G with $\Theta(n)$ pairwise tests. Then with the one good widget w_G, compare all widgets w_i, $i = 1, ..., n$ with w_G, and conclusively determine the identity of each widget w_i. This will require $\Theta(n)$ pairwise tests.

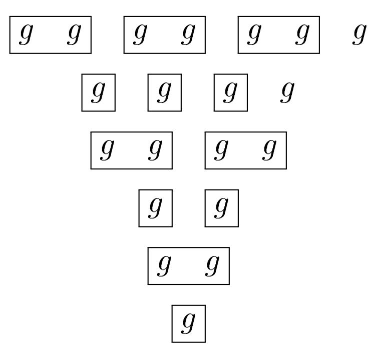
Combining the two operations, $\Theta(n) + \Theta(n) = \Theta(n)$ pairwise tests are needed to determine the identity of every widget.

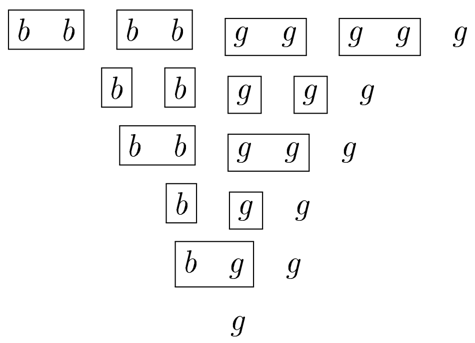
The recurrence relation of the number of pairwise comparisons for each step is

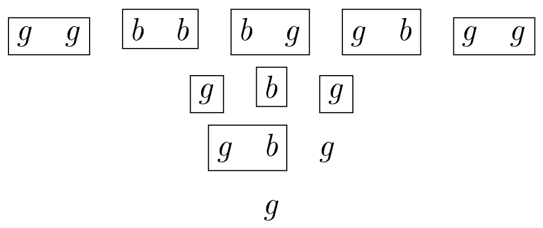
$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left\lfloor \frac{n}{2} \right\rfloor$$

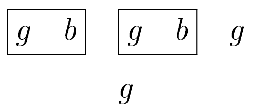
By applying the Master Theorem, with $a = 1$, $b = 2$ and $f(n) = \left\lfloor \frac{n}{2} \right\rfloor$, we find that the third case is true, in which the driving term dominates. This implies that $\boxed{T(n) = \Theta(n)}$, which is the same as what is obtained by counting.

Some pictorial demonstrations of finding w_G:









6. *Sort the following functions by order of asymptotic growth such that the final arrangement of functions $g_1, g_2, ..., g_{12}$ satisfies the ordering constraint $g_1 = \Omega(g_2), g_2 = \Omega(g_3), ..., g_{11} = \Omega(g_{12})$.*

n	n^2	$(\sqrt{2})^{\lg n}$	$2^{\lg^* n}$	$n!$	$(\lg n)!$	$(\frac{3}{2})^n$	$n^{\frac{1}{\lg n}}$	$n \lg n$	$\lg(n!)$	e^n	1

Give the final sorted list and identify which pair(s) functions $f(n)$, $g(n)$, if any, are in the same equivalence class, i.e., $f(n) = \Theta(g(n))$.

We used the following Wolfram Language code, executed in *Mathematica* to determine the order of the list. It implements bubble sort with the symbolic evaluation of the limit

$$\lim_{n\to\infty} \frac{g_i(n)}{g_{i+1}(n)}$$

as the method of comparison. $2^{\lg^* n}$ required different analysis, but its placement was determined by comparison with the smallest non-constant result.

```
functionList = {# &, #^2 &, (Sqrt[2])^Log2[#] &, #! &, (3/2)^# &, #^(1/Log2[#]) &,
    #*Log2[#] &, Log2[#!] &, Exp[#] &, 1 &, Log2[#]! &};
For[i = 2, i <= Length[functionList], i++,
  For[j = 1, j < Length[functionList] - i + 2, j++,
    Print[Limit[functionList[[j]][x]/functionList[[j + 1]][x], x -> Infinity]];
    If[Limit[functionList[[j]][x]/functionList[[j + 1]][x], x -> Infinity] == 0,
      tmp = functionList[[j + 1]];
      functionList[[j + 1]] = functionList[[j]];
      functionList[[j]] = tmp;
    ,
      If[Limit[functionList[[j]][x]/functionList[[j + 1]][x], x -> Infinity]
          < Infinity,
        Print["Equal"];
        Print[functionList[[j]]];
        Print[functionList[[j + 1]]];
      ];
    ];
  ];
];
TableForm[functionList[[All]] /. {# -> n}]
```

The result is below. It was also determined that $n \lg n = \Theta(\lg(n!))$ and $n^{\frac{1}{\lg n}} = \Theta(1)$.

$n!$	e^n	$(\frac{3}{2})^n$	$(\lg n)!$	n^2	$n \lg n$	$\lg(n!)$	n	$(\sqrt{2})^{\lg n}$	$2^{\lg^* n}$	$n^{\frac{1}{\lg n}}$	1