Exam 2 - Comments

Asymptotic Operators

Let $f(n) = n^{0.5} \log n$ and $g(n) = n(\log(n))^2$, which of the following are true? Support your answer to each part with a convincing argument.

1. $f \in O(g)$

True. To show $f \in O(g)$, we need to specify $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that $\forall n > n_0$, $f(n) \leq c \cdot g(n)$. Choose $c = 1$ and $n_0 = 10$. Then, for any $n \geq 10$, $n^{0.5} \geq 1$ and $\log(n) \geq 1$. So, $f(n) = n^{0.5} \log n \leq n(\log(n))^2$ holds for any $n \geq 10$. This completes the proof since we have shwon the definition of $f \in O(g)$ holds for these functions.

2. $f \in \Omega(g)$

False. Intuitively, we should understand that g grows asymptotically faster than f, but we need to provide a more rigorous explanation to prove $f \notin \Omega(g)$. We need to prove that for any $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$, there always exists some $n \geq n_0$ such that $f(n) < c \cdot g(n)$, thus showing the definition of $f \in \Omega(g)$ cannot hold. To show that $f \notin \Omega(g)$ we need to show that for any value of c, we eventually have n where $n^{0.5} \log n < c \cdot n(\log n)^2$. Dividing out one of the $\log n$ terms (which must be positive, so can be safely divided out of the inequality), leaves $n^{0.5} < c \cdot n \log n$. We can also divide by $n^{0.5}$ since $n = (n^{0.5})^2$, leaving $1 < c \cdot n^{0.5} \log n$. Dividing by c, gives $\frac{1}{c} < n^{0.5} \log n$. The $\log n$ term is positive, so we know the inequality holds if $n^{0.5} > \frac{1}{c}$, so we can pick $n = (\frac{1}{c})^2 + 1$. Thus, for any choice of $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$, if we set $n = \max\{n_0, 1/c^2 + 1\}$, we know $f(n) < c \cdot g(n)$. Therefore, we prove that $f \notin \Omega(g)$.

3. $f \in \Theta(g)$

False. By Definition 3, we know $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$. Since we showed in question 2 that $f \notin \Omega(g)$, we can conclude $f \notin \Theta(g)$.

Counting Functions

Recall that $SIZE_n(s)$ is the set of all n-input, 1-output Boolean functions that can be implemented with s or fewer NAND gates.

The notation $\subsetneq$ means proper subset. If $A \subsetneq B$ it means that every element of A is an element of B, but that there is at least one element of B that is not in A.

Provide a **brief but convincing proof** for each of the statements below.

4. (Corrected!) If $2 < s \leq t$ then $SIZE_1(s) = SIZE_1(t)$. (This is corrected: the original question, with $1 < s \leq t$, is not true!)

The set $SIZE_1(s)$ is the set of all 1-input, 1-output Boolean functions that can be implemented with s of fewer NAND gates. There are $(2^1)^2 = 4$ 1-input, 1-output functions:

Input	*ZERO*	*IDENTITY*	*NOT*	*ONE*
0	0	0	1	1
1	0	1	0	1

We can implement each of these functions with no more than 3 NAND gates:
$IDENTITY(a) = a$,
$NOT(a) := NAND(NAND(a))$,
$ONE(a) := NAND(NAND(a,a),a)$,
$ZERO(a) := NAND(NAND(NAND(a,a),a), NAND(NAND(a,a),a))$.
Note that the last of these uses *three* gates, since although it looks like five gates, in a circuit we could reuse the $NAND(NAND(a,a),a)$ that appears twice (or store it in a variable in a *NAND-CIRC* program.

Thus, with 3 gates, we can implement all possible 1-input, 1-ouput Boolean functions. So, $SIZE_1(s) = 4$ for any $s \geq 3$. Thus, if $s > 2$ and $t \geq s$ as in the proposition, we known $SIZE_1(s) = SIZE_1(t)$.

The original version of the question asked for a proof that if $1 < s \leq t$ that $SIZE_1(s) = SIZE_1(t)$. For this to be true, we would also need $SIZE_1(2) = SIZE_1(3)$. But, this is not the case since there is no way to implement *ZERO* with only two NAND gates!

Sorry for the unintentionally impossible problem — it is easy to forget the constant functions, which we did when developing the exam, but they are indeed necessary (and often useful).

5. $SIZE_{10}(9) \subsetneq SIZE_{10}(15)$.

To show that $SIZE_{10}(9) \subsetneq SIZE_{10}(15)$ we need to show $SIZE_{10}(9) \subset SIZE_{10}(15)$ (every function in $SIZE_{10}(9)$ is also in $SIZE_{10}(15)$), and that there is at least one function in $SIZE_{10}(15)$ that is not in $SIZE_{10}(9)$.

The first part is easy — since the definition of $SIZE_n(s)$ is the set of all functions that can be implemented with s or fewer NAND gates, increasing s can never result in fewer functions.

For the second part, observe that we have fewer NAND gates than inputs. This means we can implement $NAND(x_0, NAND(x_1, NAND(x_2, NAND(x_3, NAND(x_4, NAND(x_5, NAND(x_6, NAND(x_7, NAND(x_8, x_9)))))))))$. The circuit represented by this outputs 0 for all inputs other than when all of $x_0, ..., x_9 = 0$.

Suppose we want to compute a function that outputs 0 for all inputs other than $x_0 = 1, x_1, ..., x_9 = 0$ and $x_0 = 0, x_1 = 1, x_2, ..., x_9 = 0$. With 9 NAND gates we cannot do this — we need to NAND all of the inputs x_2 through x_9 to check they are 0, which requires 7 NAND gates, and then check the XOR of x_0 and x_1, which requires 2 NAND gates (and can't be made more efficient by combining in any other way), and then need to NAND that output with the output from the other 7 inputs, so need at least 10 NAND gates. But, with 15 gates, we can compute this function. So, we know there is a function in $SIZE_{10}(15)$ that is not in $SIZE_{10}(9)$.

6. $SIZE_{10}(3102) \subsetneq SIZE_{11}(3102)$.

This is the second (unintentionally) impossible question! You shouldn't have been able to prove this, since it is not true: $SIZE_n(s)$ is the set of all n-input, 1-output Boolean functions, so $SIZE_{10}(3102)$ and $SIZE_{11}(3102)$ are disjoint sets. A function with 10 inputs is always a different function than a function with 11 inputs.

No one actually noticed this (at least not in a way that brought it to our attention), which is a bit disappointing (although not as embarrassing as it is for us to have three impossible questions on this exam). If you interpreted the question as $SIZE_{10}(3102) \not\subset SIZE_{11}(3102)$ (that is $SIZE_{10}(3102)$ is not a subset of $SIZE_{11}(3102)$), then the answer that they are disjoint is correct.

Turing Machines

7. A Turing Machine's configuration contains all the information needed to describe the current status of its computation (i.e., if I paused my computation then wrote the configuration down, I could resume the computation using what I had written). List three necessary components of a Turing Machine's configuration.

The things we need to record the configuration of a Turing Machine are:

1. The current state (s in the definition of the execution model of the Turing Machine).
2. The current tape index (i).
3. The tape contents ($T[0], \ldots, T[k]$), where k is the index of the rightmost non-blank element.

8. A *Linear-Tape Turing Machine* is a variant of a Turing Machine where instead of having access to an unbounded tape, the machine can only use as many cells as the length of the machine's input (i.e., if it received a k-bit input, it is able to use k cells).

Is a Linear-Tape Turing Machine less powerful than a standard Turing Machine? Support your answer with a convincing argument.

Yes, the Linear-Tape Turing Machine is less powerful than a standard Turing Machine because it has finite memory. With k tape cells, there are $|\Sigma|^k$ possible contents of the tape. This means the number of configurations is finite, so we know that any L-T Turing Machine that runs for more than $|\Sigma|^k \cdot k \cdot |S|$ steps (since the configuration also includes the tape index and the current state) must be in a repeating cycle and will never halt. Since we can solve *HALTS* for Linear-Tape Turing Machines (but just simulating the machine for that number of steps), they must be less powerful than standard Turing Machines (for which we know *HALTS* is undecidable).

Another way to see this is that a standard TM can produce an output longer than its input, but the Linear-Tape Turing Machine cannot. So, there are obvious functions like "write down two copies of the input" that cannot be done by a Linear-Tape Turing Machine that can be computed by a standard Turing Machine.

9. Grace created a brand new programming language, Hopper, with a useful property: for any program written in this language, the compiler will warn you when there is some input which would cause the program to run forever.

Prove that there is no way to write a Hopper program that behaves as a Universal Turing Machine (i.e., that can simulate any given Turing Machine).

This is impossible! Suppose the language Hopper contains just one program, `U`, which is interpreted as a Universal Turing Machine. The Hopper compiler should output a warning for this program, since we know there are some inputs for which a Universal Turing Machine runs forever (namely, when the simulated TM would run forever on the given input). So, we have a counter-example: an imaginable programming language which can implement a Universal Turing Machine (and no other programs), and warns that it can run forever.

Another silly interpretation of this question would be a compiler that outputs the warning for all input programs. Actual compilers often do output false warnings, so this isn't so hard to imagine.

But, if you assume Hopper is a sensible programming language, and that the warnings about a program having an input that causes it to run forever are precise (that is, it always issues a warning for such a

program, and never issues a false warning), then you could prove that it is impossible to implement a Universal Turing Machine in Hopper since it would enable you to also implement a program that decides $HALTS_{TM}$.

Assume (towards a contradiction) it is possible to implement a Universal Turing Machine in Hopper. This means there is some Hopper program, *U(m, x)* that takes as its input the description of a Turing Machine (*m*), and outputs the result of executing *m* on input *x*. There also exists a compiler, *HOPPER*(*p*), that takes as input a Hopper program *p*. We consider the output of the compiler just the warning, so *HOPPER*(*p*) = 1 if the Hopper program *p* runs forever on some input, and otherwise *HOPPER*(*p*) = 0.

Now, we show how to define $HALTS_{TM}(w, x)$ using *U* and *HOPPER*. Recall that $HALTS_{TM}(w, x)$ outputs 1 if the Turing Machine represented by w halts on input x, and otherwise outputs 0:

$$M_{HALTS}(w, x) = HOPPER(Z)$$

where Z is a Hopper program that takes no input and calls the program *U* with *w* and *x* (converted to the appropriate form for the Hopper UTM) as its inputs. If the machine represented by w would halt on input x, then evaluating *U*(*w*, *x*) will also halt, so $Z()$ will never run forever and $HOPPER(Z)$ outputs 0 which is the correct output for $HALTS_{TM}(w, x)$. If w would run forever on input x, then evaluating *U*(*w*, *x*) will also run forever, so $Z()$ will run forever and $HOPPER(Z)$ outputs 1 which is the correct output for $HALTS_{TM}(w, x)$. Thus, we have shown that it is possible to implement $HALTS_{TM}$ if both $HOPPER$ and *U* exist, and it is possible in Hopper to write a program that calls another program with given inputs (e.g., it has string literals that can represent w and x). Since we know $HALTS_{TM}$ is undecidable, there cannot exist a program that decides it (Z). Thus, we know either $HOPPER$ does not exist, or Z does not exist. Since the question stipulated that $HOPPER$ does exist, this means Z must not exist. But, if we have *U*, and other things we expect in a sensible programming language, we can build Z, so *U* must not exist.

Note that the counterxample doesn't contradict this — it allows both $HOPPER$ and *U* to exist, but says that something else we used to build Z does not exist. We assumed lots of things can be done in Hopper (the language) to build Z from U, including the ability to call another program, and to create literals that represent the inputs.

There are useful programming languages in which all programs are guaranteed to terminate (so, as long as they provide other sensible things, there is indeed no way to implement a Universal Turing Machine in those language). Such languages are useful in many contexts where we need to ensure programs run to completion without consuming too many resources (e.g., programs that run inside your operating system kernel to decide what to do with network packets like Berkeley Packet Filter).

Halting Beavers

10. We defined the the Busy Beaver Problem as:

Definition 1 (Busy Beaver Problem) *For any $n \in \mathbb{N}$, define $BB_2(n)$ as the maximum number of steps for which a Turing Machine with n states and 2 symbols can execute and halt, starting from a blank tape.*

In class we demonstrated that $BB_2(n)$ was not computable by showing that we could use a decider for $BB_2(n)$ to decide $HALTS$, concluding that $BB_2(n)$ is "at least as hard as" $HALTS$.

Complete the proof that the two problems are "equivalent" in difficulty by showing how one could use a decider for $HALTS$ to decide $BB_2(n)$.

Towards a contradiction, assume there exists a machine M_H that decides $HALTS_T M$. We will use this to construct a machine that computes BB_2, thus proving that $HALTS$ and BB_2 are "equivalent" since we already showed the other direction.

This is a tricky reduction since it is in the opposite direction of most of the reductions you have seen (they prove X is undecidable by showing how to use a machine that decides X to decide $HALTS$), and because the input and output of BB_2 are numbers (which means we can't just output a Boolean result directly).

To compute the value of $BB_2(n)$ we need to consider all possible n-state, 2 symbol Turing Machines. For any $n \in \mathbb{N}$, the number of n-state, 2-symbol Turing Machines is *finite*. We know we can enumerate the Turing Machines, so this means we can enumerate all the n-state, 2-symbol Turing Machines in a way that will cover them all and eventually finish. So, as long as the amount of work we do for each TM is finite (that is, it halts!), we can just simulate all the halting TMs and find the one that takes the most steps.

More explicitly, here is a proof using Python-like notation to describe out BB_2 machine. We assume `M_H` is the machine that decides $HALTS$, `U` is a routine the implements a Universal Turing Machine, `None` represents the blank input, and there is a function `add_step_counter` that takes and TM description as input and outputs the description of a machine that behaves like that machine but counts the number of steps it executes. Here is an implementation of a machine that computes BB_2:

```
def BB_2(n):
    bb = 0 # maximum number of steps seen so far
    for w in all n-state, 2-symbol Turing Machines:
       if M_H(w, None): # does w halt on the blank input tape
           z = add_step_counter(w) # create a machine that behaves like w, but counts and outputs
           steps = U(z, None)
           if steps > bb: bb = steps
    return bb
```

This is guaranteed to terminate (assuming `M_H` that decides *HALTS* exists) since the number of TMs to enumerate is finite, and we only run `U` on the machines that halt, so it must finish eventually (note, that in practice, of course, it doesn't finish - I tried simulating just one of the 6-state TMs for about two weeks, and it didn't finish by then, but I had to reboot my computer so it won't be finishing this semester...)

Beyond Turing Machines

11. We have discussed several models of computation so far this semester (e.g., *NAND-CIRC* programs, Finite State Automata, Turing Machines). Each of these models only allows for finite-length representations, and for each we have demonstrated functions not computable by that model.

Prove that *any* model of computation which only allows finite-length representations cannot compute all infinite boolean functions (i.e., all functions of the form $\{0,1\}^* \to \{0,1\}$).

If the programs for our model of computation are all finite binary strings, we know that the number of finite binary strings is countably infinite. But, the number of functions on unbounded binary strings, $\{0,1\}^* \to \{0,1\}$, is *uncountable*. So, there are more functions than programs, and each program just computes (at most) one function, so there must be some functions that have no corresponding program.

