University of California, Los Angeles
CS 281 Computability and Complexity

Nakul Khambhati

Midterm Exam

Problem 1

We will first prove a more lenient lower bound of n steps. This is clear as if we have to place n containers then, for at least one of them, we need at least $2n$ steps. Let's see if we can modify this argument to prove the lower bound of $n \log n$.

Ideas:

1. To figure out where this $\log n$ factor comes in, recall that in order to keep track of how many containers have already been placed we need $\log n$ digits. I could not make much progress here as:
   (a) I'm unable to relate keeping track of the count to the actual number of steps involved.
   (b) Also, I'm not sure how tracking the number of containers already placed is relevant to placing the containers.
2. Another place I've seen the $n \log n$ factor show up is the mergesort algorithm which is used to recursively sort a list. Again, I could not make much progress here.
3. Finally, I tried to use the fact that $n \log n$ is the solution to the recursive algorithm $T(n) = 2T(\frac{n}{2}) + O(n)$.

Solution attempt: The optimal method for placing containers, which does not depend on n, should be the same regardless of how many containers are to be placed. Therefore, the only difference between placing the first $\frac{n}{2}$

containers and the last $\frac{n}{2}$ containers is the $O(n)$ steps taken to move from the closer half of the dock to the farther half. This argument implies that $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$ which is what we were required to show.

REMARK 1. I was able to make some progress here but I still don't think that my reasoning is perfectly sound as I've used the fact that n is not known a priori but it turns out that this is optimal even with the knowledge of n.

Problem 2

First we show that the copy function can be implemented in $O(n^2)$. Move the head of the tape to the end of the input and mark it. Then, go back to the start, copy the first symbol, mark it as visited, move to the right until the first blank is encountered, and write the copied symbol. Then move back to the left till the first unvisited symbol is hit. Repeat this step until each symbol before the end (previously marked) of the input has been visited. Copying each symbol involves $2n$ steps and this is done n times so this algorithm runs in $O(n^2)$ time.

Ideas:

1. My first idea was to use `COPY` to solve `PALINDROME`. Assume, in search of a contradiction, that we can simulate $f(x)$ on a TM in less than $O(n^2)$ steps using some TM M. I initially thought that computing `PALINDROME` would now be trivial and then this would give us a way of computing `PALINDROME` in a single tape TM in subquadratic time which is a contradiction. However it turns out that my intuition was wrong as even after copying the input, we would still need to make $O(n)$ steps for every character comparison, requiring $O(n^2)$ steps in total.

2. Next, I tried to use a diagonalization argument to prove that this is a lower bound. This approach was so ineffective that it is not worth writing down.

Intuition: From the algorithm provided above to compute `COPY` in $O(n^2)$ steps, it seems clear that asking for any better is unreasonable. The TM can only remember a constant amount of information so it must go back and forth every time to copy a new character. So, for each character, it must take $O(n)$ time. As a result, the total time must be $O(n^2)$.

Problem 3

This problem is undecidable, which we will prove by reducing the halting problem to this one which we shall call CONST. Assume that some TM C accepts a machine's encoding M_α and determines whether M_α halts in constant time. Now, given a turing machine M and some input string w, we construct a machine M_w such that on input x, M_w rejects if and only if $x \neq w$ or M does not halt in constant time.

If we use C, we can determine whether M_w halts i.e. whether M halts on input w. Therefore, we have found a way to construct a machine that computes whether an arbitrary machine halts on arbitrary input i.e. we have found a way to compute HALT. But this is a contradiction, so it must be the case that CONST is undecidable as well.

Problem 4

Let $L = \{\langle M \rangle : M \text{ is a Turing Machine with HALT oracle that does not accept } \langle M \rangle\}$ be a language. We will show, by contradiction, that this language cannot be decided with an oracle for HALT. Assume that this language can be decided by some TM M with an oracle for HALT. If we feed as input $\langle M \rangle$ into M:

1. If M accepts $\langle M \rangle$ as input then this means that M is a machine that does not accept $\langle M \rangle$. This is a contradiction.
2. If M does not accept $\langle M \rangle$, then M is a machine that does accept $\langle M \rangle$ as input, which is again a contradiction.

Therefore, no such M can exist. As a result, we have described a language L that cannot be computed by a Turing Machine even with an oracle for HALT.

Problem 5

The parameters provided are reminiscent of the universal turing machine so let's try that approach. Consider the universal turing machine $\mathcal{U}$ that we discussed in lecture. We let $T(n)$ be the maximum time that $\mathcal{U}$ takes to halt on any input x of length n (such that $\mathcal{U}$ halts). Note that $\mathcal{U}$ itself has some representation as a binary string so the above $T(n)$ is a valid function $\mathbb{N} \to \mathbb{N}$. Now, given any turing computable function f we can construct a TM M that computes f. We then use $\mathcal{U}$ to simulate M running on input x. By construction, $\mathcal{U}$ can simulate any turing machine M in time $T(n)$ so the overall running time of M is $T(n) + O(1)$.

To compute $T(n)$, one approach would be to simply simulating $\mathcal{U}$ on all possible turing machines that have a binary representation of length n and output the maximum of the running times.

Problem 6

It is clear that $\lfloor n^{\frac{1001}{1000}} \rfloor \geq n$. We need to exhibit an algorithm that runs in $O(\lfloor n^{\frac{1001}{1000}} \rfloor)$ time that computes $\lfloor n^{\frac{1001}{1000}} \rfloor$. We will first compute $n^{1/1000}$ and then raise the result to the power of 1001.

1. Taking the k^{th} root of an integer: We can do iteratively use binary search to guess a k^{th} root as the midpoint between 0 and n. At every step, we set a midpoint M, compute M^k and based on the result, adjust the bounds for the next step of the binary search. There are a total of $\log(n)$ steps. At each step we compute some M^k, whose runtime we will analze in the next point.

2. Taking the k^{th} power of an integer: We iteratively multiply the result with M k times. Each mutliplication step can be done in $n \log n$ time and there are a constant number of such steps.

As a result, when we raise an integer to a fractional power, our first step of computation is to take the root in the denominator which takes a total of $n \log n \log n$ steps and then raising it to the k^{th} power takes another $O(n \log n)$ steps. Our computation halts in $O(n \log n \log n) < O(n^p)$ steps for $p > 1$. Therefore, the function is time constructible.

Problem 7

We are given languages $L_1, L_2, L_3 \in \mathbf{NP}$ and we are asked to prove $L = L_1 \cup (L_2 \cap L_3) \in \mathbf{NP}$. In other words, we are have polytime TMs M_1, M_2, M_3 such that $x \in L_i \iff \exists u \in \{0,1\}^* : M_i(x,u) = 1$ where $i \in \{1,2,3\}$. We need to construct a polytime TM M such that $x \in L \iff \exists u \in \{0,1\}^* : M(x,u) = 1$.

By definition, $x \in L \iff x \in L_1 \vee (x \in L_2 \wedge x \in L_3)$. We can construct M accordingly so that it accepts x if and only if there exists a certificate of the form $u_1u_2u_3$ such that $(M_1(x,u_1) = 1) \vee (M_2(x,u_2) = 1 \wedge M_3(x,u_3) = 1)$. Explicitly, M accepts an input and 3 certificates (where the partition is demarcated by a set of predefined symbols). M then runs $M_i(x,u_i)$ and outputs 1 based on the above logical formula. M still runs in polynomial time as each of M_1, M_2, M_3 run in polynomial time and the boolean formula can be evaluated in constant time.

Problem 8

In problem 10, we see that the time hierarchy theorem shows that $\mathbf{P} \subsetneq \mathbf{EXP}$. In the same way, it follows from the non-deterministic time hierarchy theorem that $\mathbf{NP} \subsetneq \mathbf{NEXP}$. This means that $\exists L \in \mathbf{NEXP} \setminus \mathbf{NP}$. If this language is **NP** hard, then we are done. Therefore, it suffices to show that there exists a language L that is **NEXP** complete. At this point, I got stuck and looked up https://en.wikipedia.org/wiki/NEXPTIME to find examples of **NEXP** complete problems. I encountered the concept of a succinct circuit which are a way of encoding graphs in exponentially less space. From there, it immediately became clear how to find an example of a problem that is **NEXP** complete. We consider problems that are **NP** complete when a graph is represented in the usual way (such as an adjacency matrix). An example here is finding a Hamiltonian path for a graph. If we now consider the language where the input is encoded as a succinct circuit, then the same problem is **NEXP** complete. This concludes our proof.

Problem 9

We are required to give a deterministic polynomial-time algorithm for determining whether a given 2-CNF formula is satisfiable. Say we are given some 2-CNF formula which is a conjunction of clauses of the form $(x \vee y)$. The 2-CNF is true if and only if all such clauses evaluate to true. With a shift in perspective, we can view each clause as a pair of implications since if $x = 0$ then for $x \vee y$ to be true, we must have $y = 1$ and vice versa. In other words, $(x \vee y \iff \overline{x} \Rightarrow y \wedge \overline{y} \Rightarrow x)$. We will now consider each implication as an edge in an *implication graph*. The question of finding a satisfying assignment to the formula reduces to analyzing the graph. If, due to certain clauses, the graph has a directed path from x to $\overline{x}$, then we know that the formula cannot be satisfied. At this point, I got stuck and could not figure out how to proceed. I browsed the net and encountered $https://en.wikipedia.org/wiki/Strongly_connected_component$ the concept of a Strongly Connected Component (SCC) in a directed graph. An algorithm (Kosaraju-Sharir) can be used to determine if x and $\overline{x}$ lie in the same SCC, which runs in $O(V + E)$ time. Our TM can run this algorithm (in polynomial time) on the constructed graph and returns 1 if and only if for all variables x in the 2-CNF, x and $\overline{x}$ are not in the same SCC.

Problem 10

It suffices to show that $\mathbf{P} \subsetneq \mathbf{EXP}$ because then $\mathbf{P} \subsetneq \mathbf{PSPACE}$ or $\mathbf{PSPACE} \subsetneq \mathbf{EXP}$. We will prove this using the Time Hierarchy Theorem. It is clear that $\forall c > 0 : \frac{2^n}{n^c \log n^c} = \frac{2^n}{n^c c \log n} \to \infty$ when $n \to \infty$. Therefore, $\mathbf{DTIME}(n^c) \subsetneq \mathbf{DTIME}(2^n)$ for all $c > 0$ so $\mathbf{P} \subsetneq \mathbf{DTIME}(2^n) \subset \mathbf{EXP}$ therefore $\mathbf{P} \subsetneq \mathbf{EXP}$.

Problem 11

For this part, we will use the fact that $TQBF$ is **PSPACE** complete. It is clear that $\mathbf{P}^{TQBF} \subset \mathbf{NP}^{TQBF}$ since if $L \in \mathbf{P}^{TQBF}$ then we can run the same computation on non-deterministic polytime TM. Equivalently, we can use the same M with an empty certificate. The other direction is tricky and follows from Savitch's theorem as well as the completeness of $TQBF$. First, note that $\mathbf{NP}^{TQBF} \subset \mathbf{NPSPACE}$. This is because we can query each instance of the oracle for $TQBF$ in $\mathbf{PSPACE} \subset \mathbf{NPSPACE}$. Next, using Savitch's theorem, $\mathbf{NPSPACE} \subset \mathbf{PSPACE}$. The proof has been omitted as we discussed it in depth in lecture. Also, $\mathbf{PSPACE} \subset \mathbf{P}^{TQBF}$ since every language in **PSPACE** can be efficiently (in polynomial time) reduced to an instance of $TQBF$. Finally, we get $\mathbf{NP}^{TQBF} \subset \mathbf{NSPACE} \subset \mathbf{PSPACE} \subset \mathbf{P}^{TQBF}$ so $\mathbf{NP}^{TQBF} \subset \mathbf{P}^{TQBF}$. In conclusion, $\mathbf{P}^{TQBF} = \mathbf{NP}^{TQBF}$

Problem 12

In lecture, we argued that $\mathbf{NTIME}(S(n)) \subset \mathbf{SPACE}(S(n))$. In particular, this implies that $\mathbf{NTIME}(n^{1000}) \subset \mathbf{SPACE}(n^{1000})$. Also, by the space hierarchy theorem, $\mathbf{SPACE}(n^{1000}) \subsetneq \mathbf{SPACE}(n^{1001})$ and by definition $\mathbf{SPACE}(n^{1001}) \subset \mathbf{PSPACE}$. In conclusion, $\mathbf{NTIME}(n^{1000}) \subsetneq \mathbf{PSPACE}$.

Problem 13

TQBF is **PSPACE** complete. This means that any language in **PSPACE** is polynomial time reducible to TQBF. My first instinct here was that such an algorithm can't exist as then all of **PSPACE** would collapse to $O(n^{1/1000})$ space. However, I noticed an error in my reasoning here as we also allow polynomial time reductions. Therefore, such an algorithm would only imply that **PSPACE** $= O(n^{1/1000})$ space $+\mathbf{P}$. I don't know anything that contradicts this so it is worth looking for such an algorithm. The algorithm that we discussed in lecture certainly runs in polynomial time but is slower than linear so it seems unlikely that the same one can be modified to run in $O(n^{1/1000})$.