Recurrence Let $T(n)$ be the function given by the recursion: $T(n) = nT(\lfloor \sqrt{n} \rfloor)$ for $n > 1$ and $T(1) = 1$. Is $T(n) \in O(n^k)$ for some constant $k$, i.e. is $T$ bounded by a polynomial in $n$? Prove your answer either way. (Note: logic and definition of $O$ notation are more important than exact calculations for this problem.)

Yes, the function is polynomially bounded.

First proof method, change of variables. We give an exact calculation, when $n$ is a power of 2, that it is itself a power of 2. We pick this kind of number, because the square root of a power of a power of 2 is also of the same form. If $n = 2^d$, $T(n) = T(2^d) = 2^d T(2^d - 1) = 2^d (2^d - 1) T(2^d - 2) = \ldots = 2^d + 2^{d-1} + 2^{d-2} + \ldots + 1 = 2^{d+1} - 1 = 1/2n^2$. Since for every $n$, there is a power of a power of 2 at most $n^2$, $(n' = 2^{\lceil \log \log n \rceil})$ and $T(n)$ is non-decreasing, this gives an upper bound of $T(n) \leq T(n') \leq 1/2(n')^2 \leq 1/2n^4 \in O(n^4)$.

Second method: Induction. By trying a few values, we guess that $T(n)$ grows roughly as $n^2$. We then verify that guess: We prove by strong induction on $n$, that for every $n \geq 1$, $T(n) \leq n^2$. Since $T(1) = 1$, the base case holds. Let $k \geq 1$, and assume $T(n) \leq n^2$ for all $1 \leq n \leq k$. Then $T(k+1) = (k+1)T(\lfloor \sqrt{k+1} \rfloor)$. Applying the induction hypothesis to $\lfloor \sqrt{k+1} \rfloor$, $T(k+1) \leq (k+1)(\lfloor \sqrt{k+1} \rfloor)^2 \leq (k+1)(\sqrt{k+1})^2 = (k+1)^2$.

So by induction, $T(n) \leq n^2$ for all $n$.

Reasoning about order Let $f(n)$ be a positive, integer-valued function on the natural numbers that is non-decreasing. Show that if $f(2n) \in O(f(n))$, then $f(n) \in O(n^k)$ for some constant $k$. Is the converse also always true?

First, if $f(2n) \in O(f(n))$, by definition of order, there are constants $n_0 > 0, c > 0$ so that for all $n > n_0$, $f(2n) \leq cf(n)$. (Note this is similar to the recurrence: $T(n) = cT(n/2)$, and the proof that $f$ is polynomial is just another proof of the Master Theorem). We’ll first prove that $f$ is polynomial on powers of 2, then use monotonicity to conclude the same thing for other values. Without loss of generality, assume $c > 1$ and that $n_0$ is a power of 2, $n_0 = 2^i$. Let $c' = f(n_0)/c^i$. We’ll prove by induction that for $j \geq i$ for $n = 2^j$, $f(n) = f(2^j) \leq c'c^j = c'(2^j \log c) = c'n^j\log c$. The claim is true for $j = i$, by definition of $c'$. If $f(2^j) \leq c'c^j$, then $f(2^{j+1}) = f(2^{j+1}) \leq c'c^{j+1}$, so the claim holds inductively.

Then for any $n \geq n_0$, let $n' = 2^{\lceil \log n \rceil}$ be the next power of 2; $n \leq n' \leq 2n$, so $f(n) \leq f(n') \leq c'(n')^{\log c} \leq c'(2n)^{\log c} = c'n^{\log c}$. Picking $n_0$ and $c'$ in the definition of $O$, we have $f(n) \in O(n^{\log c})$.

The converse isn’t always true. Let’s use a counter-example that came up in problem 1, the first solution. Let $f(n) = 2^{\lceil \log \log n \rceil}$. Then $f(n) \leq
\[ 2^{\log \log n} = 2^{\log n} = n^2, \text{ so } f(n) \in O(n^2). \] However, if \( n = 2^i \), \( f(n) = n \) and \( f(2n) \geq n^2 \). Since a gap of \( n \) occurs for infinitely many \( n \), \( f(2n) \) is NOT in \( O(f(n)) \).

**Base Conversion** Present and analyze an \( O(n^2) \) time algorithm that inputs an array of \( n \) base 10 digits representing a positive integer in base 10 and outputs an array of base 2 bits representing the same integer in base 2. Count each operation on a single digit as a step, e.g., adding two \( n \) bit binary strings takes time \( O(n) \) since one addition involves \( O(n) \) bit operations.

The main strategy we’ll use is: We can tell whether a decimal number is even or odd by looking at the least significant digit. We can then record this as the least significant bit. By themselves the other bits in binary represent \( x \div 2 \), so we divide by 2 and repeat.

To use this strategy, we’ll need to see how long it takes to divide numbers. Fortunately we only need to divide by 2, not a general division algorithm. In fact, if we use the long division algorithm from grade school, we see that we only need one digit carry to divide by a single digit number like 2. Each step then becomes divide an at most 2 digit number by a one digit number, subtract the product of two one digit numbers from a two digit number, and use the resulting carry as the first digit for the next operation, bringing down one digit of the input string. So the total work of long division by 2 is \( O(n) \), where \( n \) is the number of digits in the number we are halving. We can think of the long division algorithm as given to us as a procedure \( \text{LDiv2} \) that takes an input \( X \) as its array of \( n \) single digits, and returns \( X \div 2 \) expressed in decimal as its array of at most \( n \) single digits. Then our binary conversion algorithm is:

1. Initialize \( BX[0..n] \) array of bits.
2. \( I \leftarrow 0 \) \{a pointer to which bit we are computing\}
3. Until \( I > 4n \) do:
   4. begin;
   5. \( BX[I] \leftarrow X[0] \div \text{div2} \);
   6. \( X \leftarrow \text{LDiv2}[X] \);
   7. \( I++ \);
   8. end;
9. Return \( BX \) (possibly removing initial 0’s, if you want).

A few things need explaining. First, why did we initialize \( 4n \) bits in \( BX \)? This is because \( 10 < 16 = 2^4 \), so an \( n \) digit number \( X \) is at most \( 10^n < 2^{4n} \), so the length of \( X \) in binary is at most 4 times its length in decimal. Thus,
the main loop executes $O(n)$ times, and each time it calls the $O(n)$ time operation $LDiv2$. Since $X[0]$ is just a single digit, we can take $X[0]div2$ in $O(1)$ time, and so the rest of the loop is $O(1)$. Thus, the inside of the loop is $O(n)$, and we repeat it $4n$ times, so the total time is $O(n^2)$. $4n$ is an overestimate on the number of bits required, so we could then go back and decrease the dimension of the array until we see a bit with value 1. This would be an additional $O(n)$ time, which would not change the $O(n^2)$ total complexity.

**Implementing Base Conversion**. Implement the above algorithm, and test it on many random $n$ bit strings for $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$, $n = 4096$, $n = 8192$, $n = 16384$, and $n = 32768$. Plot time vs. input size on a log vs. log curve. Does the algorithm’s observed time fit the analysis? Why or why not?

I can’t give a real model solution here, but the results for last year were that for most implementations, the observed times seem to fit the asymptotic analysis quite well. I thought there might be some cache misses causing overhead in some algorithms, but evidently cache sizes are large enough for this not to be an issue.