1. Consider the following recurrence, originally used by the Sanksrit prosodist Piṅgala in the second century BCE, to compute the number 2^n:

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ (2^{n/2})^2 & \text{if } n > 0 \text{ is even} \\ 2 \cdot (2^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

We can use this algorithm to compute the decimal representation of 2^n, by representing all numbers using arrays of decimal digits, and implementing squaring and doubling using decimal arithmetic. Suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm?

Solution: Let's write the algorithm in pseudocode. Our input is an integer n; the desired output is a an array of decimal digits containing the decimal representation of n. We use two subroutines for decimal arithmetic:

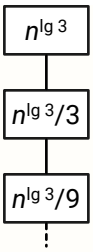
- DECIMALSQUARE(Z) takes a decimal string Z as input, representing some integer z, and returns the decimal representation of the integer z^2. For example, DECIMALSQUARE("374") returns the string "139876". If we implement this subroutine using Karatsuba's algorithm, then it runs in $O(|Z|^{\lg 3})$ time.
- DECIMALDOUBLE(Z) takes a decimal string Z as input, representing some integer z, and returns the decimal representation of the integer $2z$. For example, DECIMALDOUBLE("374") returns the string "748". This subroutine uses a simple for-loop and thus runs in $O(|Z|)$ time.

```
DECIMALTWOTOTHE(n):
  if n = 0
      return "1"
  m ← ⌊n/2⌋
  Z ← DECIMALTWOTOTHE(m)      ⟨⟨recurse!⟩⟩
  Z ← DECIMALSQUARE(Z)        ⟨⟨Karatsuba⟩⟩
  if n is odd
      Z ← DECIMALDOUBLE(Z)    ⟨⟨brute force⟩⟩
  return Z
```

Because $\log_{10}(2^m) = \Theta(n)$, the output string has length $\Theta(n)$. It follows that the call to DECIMALSQUARE takes $O(n^{\lg 3})$ time, and the call to DECIMALDOUBLE (if n is odd) takes $O(n)$ time. Thus, the running time of this algorithm satisfies the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + O(n^{\lg 3}).$$

We can safely ignore the floor in the recursive argument. The recursion "tree" for this algorithm is just a path; the value of the single node at depth i is $O((n/2^i)^{\lg 3}) = O(n^{\lg 3}/3^i)$.



The level "sums" form a descending geometric series, which is dominated by its largest term, at level 0. We conclude that our algorithm runs in $O(n^{\lg 3})$ *time*. ■

2. We can use a similar algorithm to convert the binary representation of any integer into its decimal representation. Suppose we are given an integer x as an array of n bits (binary digits). Write $x = a \cdot 2^{n/2} + b$, where a is represented by the top $n/2$ bits of x, and b is represented by the bottom $n/2$ bits of x. Then we can convert x into decimal as follows:

   (a) Recursively convert a into decimal.

   (b) Recursively convert $2^{n/2}$ into decimal.

   (c) Recursively convert b into decimal.

   (d) Compute $x = a \cdot 2^{n/2} + b$ using decimal multiplication and addition.

   Now suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm? (For simplicity, you can assume n is a power of 2.)

Solution: Here is the algorithm in pseudocode. (The lines in red are what we will change in problem 3.)

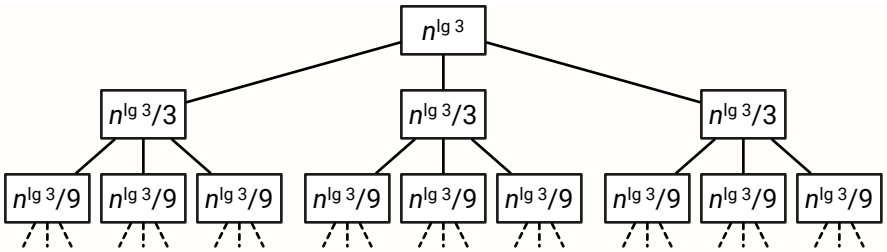
```
DECIMAL(X[0..n-1]):
  if n < 100
      use brute force
  m ← ⌈n/2⌉
  T[m] ← 1                          ⟨⟨build binary rep. of 2^m⟩⟩
  for i ← 0 to m-1
      T[i] ← 0
  M ← DECIMAL(T[0..m])              ⟨⟨Recurse!⟩⟩
  A ← DECIMAL(X[m..n-1])            ⟨⟨Recurse!⟩⟩
  B ← DECIMAL(X[0..m-1])            ⟨⟨Recurse!⟩⟩
  return DECIMALADD(DECIMALMULTIPLY(A, M), B)
```

The subroutines DECIMALADD and DECIMALMULTIPLY do exactly what they say. DECIMALADD runs in $O(n)$ time, and because we are using Karatsuba's algorithm, DECIMALMULTIPLY runs in $O(n^{\lg 3})$ time.

The decimal strings A and B and M each have length $\Theta(n)$. Thus, the calls to DECIMALTWOTOTHE and DECIMALMULTIPLY both take $O(n^{\lg 3})$ time, and the call to DECIMALADD takes $O(n)$ time. Thus, the running time of the overall algorithm satisfies the recurrence

$$T(n) = 3T(n/2) + O(n^{\lg 3}).$$

The recursion tree for $T(n)$ is a ternary tree, with 3^i nodes at recursion depth i. Each node at depth i corresponds to a subproblem with input size $n/2^i$; the non-recursive time spent on that subproblem is $O((n/2^i)^{\lg 3}) = O(n^{\lg 3}/3^i)$.



Thus, the total work at level i is $3^i \cdot O(n^{\lg 3}/3^i) = O(n^{\lg 3})$, so the level sums are all equal! The depth of the recursion tree is $\log_2 n = O(\log n)$. We conclude that our algorithm runs in $\boldsymbol{O(n^{\lg 3} \log n)}$ ***time***. ■

3. Now suppose instead of converting $2^{n/2}$ to decimal by recursively calling the algorithm from problem 2, we use the specialized algorithm for powers of 2 from problem 1. Now what is the running time of the resulting algorithm (assuming we use Karatsuba's multiplication algorithm as before)?

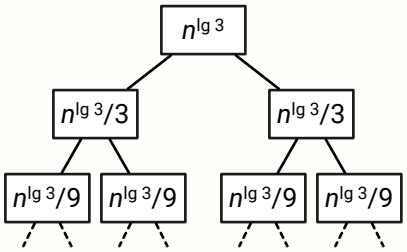
Solution: Here is the modified algorithm in pseudocode.

```
DECIMAL(X[0..n-1]):
  if n < 100
      use brute force
  m ← ⌈n/2⌉
  M ← DECIMALTWOTOTHE(m)
  A ← DECIMAL(X[m..n-1])      ⟨⟨recurse!⟩⟩
  B ← DECIMAL(X[0..m-1])      ⟨⟨recurse!⟩⟩
  return DECIMALADD(DECIMALMULTIPLY(A, M), B)
```

Because we already know that DECIMALTWOTOTHE runs in $O(n^{\lg 3})$ time, the running time of this modified algorithm satisfies the recurrence

$$T(n) = 2T(n/2) + O(n^{\lg 3}).$$

The recursion tree for $T(n)$ is a binary tree, with $\mathbf{2^i}$ nodes at recursion depth i. Each recursive call at depth i converts an $n/2^i$-bit binary number to decimal; the non-recursive work at the corresponding node of the recursion tree is $O((n/2^i)^{\lg 3}) = O(n^{\lg 3}/3^i)$.



Thus, the total work at depth i is $2^i \cdot O(n^{\lg 3}/3^i) = O((2/3)^i n^{\lg 3})$. The level sums form a descending geometric series, which is dominated by its largest term, at level 0. We conclude that our algorithm runs in $\boldsymbol{O(n^{\lg 3})}$ ***time***. ■

Harder problem to think about later:

4. In fact, it is possible to multiply two n-digit decimal numbers in $O(n \log n)$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n \log^2 n)$ time.

Solution: We modify the solutions of problems 2 and 3 to use the faster multiplication algorithm instead of Karatsuba's algorithm. Let $T_2(n)$ and $T_3(n)$ denote the running times of DECIMALTWOTOTHE and DECIMAL, respectively. We need to solve two recurrences:

$$T_2(n) = T_2(n/2) + O(n \log n)$$
$$T_3(n) = 2T_3(n/2) + T_2(n) + O(n \log n).$$

We can solve the first recurrence $T_2(n) = T_2(n/2) + O(n \log n)$ using the recursion tree method, by conservatively bounding log factors. Each node at level i of the recursion tree has value $(n/2^i)\log(n/2^i) \le (n/2^i)\log n$, so the levels define a descending geometric series with an extra factor of $\log n$. We conclude that $T_2(n) \le O(n \log n)$.

Now our recurrence for $T_3(n)$ simplifies to $T_3(n) = 2T_3(n/2) + O(n \log n)$, so essentially the same recursion tree argument implies that $T_3(n) \le O(n \log^2 n)$.

Alternatively, we can modify the DECIMAL algorithm to compute decimal representations of both x and 2^n, as follows:

```
⟨⟨Compute decimal representations of both x (given in binary) and 2^n⟩⟩
DECIMAL(X[0..n-1]):
  if n < 100
      use brute force
  m ← ⌈n/2⌉
  DA, MA ← DECIMAL(X[m..n-1])                    ⟨⟨Recurse!⟩⟩
  DB, MB ← DECIMAL(X[0..m-1])                    ⟨⟨Recurse!⟩⟩
  ⟨⟨DA and DB are decimal reps of a and b, where x = a·2^m + b⟩⟩
  ⟨⟨MA and MB are decimal reps of 2^(n-m) and 2^m⟩⟩
  DX ← DECIMALADD(DECIMALMULTIPLY(DA, MB), DB)   ⟨⟨O(n log n)⟩⟩
  MX ← DECIMALMULTIPLY(MA, MB)                   ⟨⟨O(n log n)⟩⟩
  return DX, MX
```

The running time of this modified algorithm satisfies the recurrence $T(n) = 2T(n/2) + O(n \log n)$, which implies $T(n) = O(n \log^2 n)$, as required. ■