1. Using the technique of “iterated expansion,” derive big-$\Theta$ bounds in closed form for the following recurrences. Show your work. You should start by showing what summation it expands to. Then turn this into a big-$\Theta$ expression in closed form. For each, either verify that the master theorem gives the same bound or state that the master theorem doesn’t apply.

(a) $T(n) = 5T(n/5) + n$
(b) $T(n) = 25T(n/5) + n$
(c) $T(n) = 5T(n/5) + n^2$
(d) $T(n) = 5T(n/5) + n \log_5 n$
(e) $T(n) = 2T(n - 1) + 1$
(f) $T(n) = T(n - 1) + n$
(g) $T(n) = T(n/2) + \log_2 n$.
(h) $T(n) = T(n/2) + 1/\log_2 n$.

2. For each of the above, indicate whether the master theorem applies, and, if so, show that it gives the same time bound.

3. Give a recurrence for the running time of the following version of the solution to the Game problem from Homework 1, and give a big-$\Theta$ bound in closed form.

```python
# Find best score you can force in L[i..j], where sum is the sum of these elements
def Game(L, i, j, sum):
    if i == j:
        return L[i]
    else:
        return sum - min(Game(L, i, j-1, sum-L[j]), Game(L, i+1, j, sum-L[i]))
```

4. Give a recurrence for the running time of the solution to this game problem that I posted in the solutions for Homework 1.

*Hint: Show that when you expand it, each successive term is at least some constant times the previous term. Find the minimum constant such that this is true, then give the bound and cite one of the useful rules we’ve gone over in class.*

**Programming**

- We can represent a polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ in python as follows: $[a_0, a_1, a_2, \ldots, a_{n-1}]$. Let $n$ denote the length of the polynomial. For example, $[3, 5, 0, 7]$ represents the polynomial $3 + 5x + 7x^3$, and has length 4.

Fill in the method `plus` that takes two polynomials, $[a_0, a_1, a_2, \ldots, a_{n-1}]$ and $[b_0, b_1, b_2, \ldots, b_{m-1}]$ and returns the polynomial that gives their sum.
• Fill in the method **negative**(*A*), which returns the coefficients of the negative of polynomial *A*. This is the list whose elements are the negatives of the elements of list *A*. It shouldn’t modify *A*; it should return a new list.

• Recall that \( \log_2 n \) is the number of times in a row you can divide by two, starting at *n*, until you reach 1. This works when *n* is a power of two. When *n* is not a power of two, we can get \( \lfloor \log_2 n \rfloor \) by counting the number of times in a row you can take \( \lfloor k/2 \rfloor \), starting at *n*, until you reach 1, and we can get \( \lceil \log_2 n \rceil \) by counting the number of times you can take \( \lceil k/2 \rceil \), starting at *n*, until you reach 1.

Fill in the method **logCeil**(*n*), which finds \( \lceil \log_2 n \rceil \) using the latter method.

• Study the code in **main**. Fill in the method **nextPowerTwo**(*n*), which returns the smallest power of two greater than or equal to *n*. It should use a combination of the \( \ast \ast \) operator and a call to **logCeil**(*n*).

• Fill in the body of the method **pad**(*P*, *n*), which pads a polynomial to the next higher power of two greater than or equal to *n*.

• Fill in the body of **shift**, which multiplies a polynomial by \( x^k \) for \( k \geq 0 \).

• Fill in the body of **trim**, which trims off trailing zeros.

• Fill in the method **convolve1**(*X*, *Y*), which finds the coefficients of the product of *X* and *Y*, using the gradeschool method. For example, **convolve1**([3, 5, 2], [1, 2]) should return [3, 11, 12, 4]. This operation on two sequences is known as the convolution.

• Fill in the method, **convolve2Aux**(*X*, *Y*), which takes two polynomials *X* and *Y* whose lengths are equal and a power of two, and returns the polynomial that is equal to their product. For example, **convolve1**([3, 5, 2, 1], [1, 2, 4, 3]) could return [3, 11, 24, 34, 25, 10, 3, 0].

Proceed as follows. Let *X* and *Y* be the two polynomials. Let \( A = X[: \lfloor n/2 \rfloor] \), \( B = X[\lfloor n/2 \rfloor:] \), \( C = Y[: \lfloor n/2 \rfloor] \), and \( D = Z[\lfloor n/2 \rfloor:] \).

Note that \( X = A + B \ast x^{n/2} \) and \( Y = C + D \ast x^{n/2} \). Therefore, \( XY = AC + AD \ast x^{n/2} + BC \ast x^{n/2} + BD \ast x^n \).

Have the algorithm compute \( AC, AD, BC, \) and \( BD \) with recursive calls, and then assemble \( XY \) using the above expression.

For example, if \( X = [3, 5, 2, 1] \) and \( Y = [1, 2, 4, 3] \), \( A = [3, 5] \), \( B = [2, 1] \), \( C = [1, 2] \), \( D = [4, 3] \). The algorithm should compute \( XY \) as follows:

\[
\begin{align*}
[3, 11, 10] & \quad \text{(AC)} \\
[0, 0, 12, 29, 15] & \quad \text{(AD shifted by n/2)} \\
[0, 0, 2, 5, 2] & \quad \text{(BC shifted by n/2)} \\
[0, 0, 0, 0, 8, 10, 3] & \quad \text{(BD shifted by n)} \\
\hline
[3, 11, 24, 34, 25, 10, 3]
\end{align*}
\]

The running time of this algorithm has the recurrence \( T(n) = 4T(n/2) + n = \Theta(n^2) \) by the master theorem. This is no improvement over the grade-school method, but we can revise it so that it will be.
• Fill in the body of convolve2(X, Y), which is a front end for convolve2Aux(X, Y). The preconditions of convolve2Aux are more rigid, so you have to do some padding of trailing zeros before you can call it.

• Come up with a faster variant of convolve2Aux for convolve3Aux. It should use three recursive calls instead of four. The running time should be $T(n) = 3T(n/2) + n$, which is $\Theta(n \log_2 3)$ by the master theorem. \textit{Hint: Make one of the recursive calls calculate $(A + B)(C + D)$.}