1. In each situation, write a recurrence relation, including base case(s), that describes the recursive structure of the problem. You do not need to solve the recurrence.

(a) (2 points) When you cut a pizza, you cut along a diameter of the pizza. Let \( P(n) \) be the number of slices of pizza that exist after you have made \( n \) cuts, where \( n \geq 1 \). Write a recurrence for \( P(n) \).

**Solution:** Since each cut creates two additional slices of pie, the recurrence is

\[
P(n) = P(n - 1) + 2, \text{ with } P(1) = 2.
\]

(b) (2 points) A bunch of motorcycles and SUVs want to parallel park on a street. The street can fit \( n \) motorcycles, but SUVs take up three motorcycle spaces. Let \( A(n) \) be the number of arrangements of cars and motorcycles on a street that fits \( n \) motorcycles. For example, \( A(5) = 4 \) because there are four ways to park vehicles on a street with five motorcycle spaces. If \( M \) stands for motorcycle, and \( C \) stands for car, then the four arrangements are: MMC, MCM, CMM, and MMMMM. Write a recurrence for \( A(n) \).

**Solution:** If the first vehicle parked is a motorcycle, then the other vehicles form an arrangement that fills \( n - 1 \) spots. If the first vehicle parked is a car, then the other vehicles form an arrangement that fills \( n - 3 \) spots. Therefore, the recurrence is

\[
a(n) = a(n - 1) + a(n - 3), \text{ with } a(0) = 1, a(1) = 1, a(2) = 1
\]

or equivalently,

\[
a(n) = a(n - 1) + a(n - 3), \text{ with } a(1) = 1, a(2) = 1, a(3) = 2.
\]

(c) (2 points) Let \( B(n) \) be the number of length \( n \) bit sequences that have no three consecutive 0’s. Write a recurrence for \( B(n) \).

**Solution:** Any bit string that has no 000 must have a 1 in at least one of the first three positions. Break up all bit strings avoiding 000 by when the first 1 occurs. That is, each bit string of length \( n \) avoiding 000 falls into exactly one of these cases:

(i) 1 followed by any bit string of length \( n - 1 \) avoiding 000.
(ii) 01 followed by any bit string of length \( n - 2 \) avoiding 000.
(iii) 001 followed by any bit string of length \( n - 3 \) avoiding 000.

Therefore, the recurrence is

\[
B(n) = B(n - 1) + B(n - 2) + B(n - 3), \text{ with } B(0) = 1, B(1) = 2, B(2) = 4
\]

or equivalently,

\[
B(n) = B(n - 1) + B(n - 2) + B(n - 3), \text{ with } B(1) = 2, B(2) = 4, B(3) = 7.
\]

(d) (2 points) Let \( C(n) \) be the number of \( 1 \times 1 \) cells in an \( n \times n \) grid. Write a recurrence for \( C(n) \).

**Solution:** We can think of each \( n \times n \) grid as containing an \( (n - 1) \times (n - 1) \) grid in the lower left corner. The larger \( n \times n \) grid contains all the cells of the \( (n - 1) \times (n - 1) \) grid plus an extra strip around the outside which contains \( (n - 1) + 1 + (n - 1) = 2n - 1 \) additional cells.
Therefore, the recurrence is

\[ C(n) = C(n-1) + 2n - 1, \text{ with } C(1) = 1 \]

or equivalently,

\[ C(n) = C(n-1) + 2n - 1, \text{ with } C(0) = 0. \]

(e) (2 points) A ternary string is like a binary string except it uses three symbols, 0, 1, and 2. For example, 12210021 is a ternary string of length 8. Let \( T(n) \) be the number of ternary strings of length \( n \) with the property that there is never a 2 appearing anywhere after a 0. For example, 12120012 has this property but 10120012 does not. Write a recurrence for \( T(n) \).

**Solution:** This is similar to an example from class with binary strings. Any such ternary string of length \( n \) starts with 0, 1, or 2:

\[
\begin{align*}
0 & \ldots \\
1 & \ldots \\
2 & \ldots 
\end{align*}
\]

In the first case, since we cannot have 2 anywhere after a 0, the dots represent a binary string, that is a string of length \( n - 1 \) containing all 0s and 1s. There are \( 2^{n-1} \) binary strings of length \( n - 1 \).

In the last two cases, the dots represent any ternary string of length \( n - 1 \) having the property that there is never a 2 anywhere after a 0.

For the base case, note that all three ternary strings of length one satisfy the required property. Equivalently, the empty string of length 0 also satisfies the required property.

Therefore, the recurrence is

\[ T(n) = 2T(n-1) + 2^{n-1}, \text{ with } T(1) = 3 \]

or equivalently,

\[ T(n) = 2T(n-1) + 2^{n-1}, \text{ with } T(0) = 1. \]

2. (a) (5 points) Suppose a function \( g \) is defined by the following recursive formula, where \( n \) is a positive integer. \( g(n) = 4g(n/2) + n^2 \). Use the Master theorem to solve for \( g \) up to \( O \).

We can directly apply the master theorem, with \( a = 4, b = 2 \) and \( d = 2 \) (since \( n^2 \in O(n^2) \)). Since \( a = 4 = 2^2 \), we are in the “steady-state” case, and \( T(n) \in O(n^d \log n) = O(n^2 \log n) \).

(b) (5 points) Suppose a function \( f \) is defined by the following recursive formula, where \( n \) is a positive integer. \( f(n) = 2f(n/3) + O(n) \). Use the Master theorem to solve for \( f \) up to \( O \).

In this case, \( a = 2, b = 3 \) and \( d = 1 \). Since \( a = 2 < 3^1 = b^d \), we are in the “top-heavy” case, and the function \( f(n) \in O(n^d) = O(n) \).

3. Present the predecessor algorithm from homework 2 as a recursive algorithm, and prove it is correct.

Give a recurrence for the time complexity of this algorithm, and use the master theorem to solve it.

(3 points algorithm description, 4 points proof of correctness, 3 points recurrence.)

We are looking for the maximum position \( p \) so that \( A[p] \leq x \); if \( x \) is smaller than all elements in the array, we return 0. Because we are interested in the position, I’ll assume that arrays can have any interval of integers as array positions indices. If not, we have to pass through appropriate offsets in the recursive call. In terms of an arbitrary interval \( A[I..J] \), we can redefine the predecessor to be \( I - 1 \) if \( x < A[I] \), i.e., \( x \) is smaller than all elements in the array. When we are searching the entire array, this is consistent with the original definition, so if we show that our algorithm meets the new definition on any shifted array, it also shows that it meets the original definition on any array starting with index 1.
We compare the element searched for to the middle of the array, \( A[m] \). If it is larger, we search in \( A[m..n] \), if smaller in \( A[1..m] \). This translates to the following recursive algorithm:

\[ \text{Pred}(x, A[I..J]) \]

1. \( n := J - I + 1 \)
2. IF \( x < A[I] \) return \( I - 1 \)
3. IF \( n = 1 \) return \( I \)
4. \( m \leftarrow \lfloor (I + J)/2 \rfloor \)
5. IF \( A[m] \leq x \) THEN return \( \text{Pred}(x, A[m+1..J]) \)
6. ELSE return \( \text{Pred}(x, A[I..m]) \)

We prove this algorithm is correct by strong induction on \( n \).

If \( n = 1 \), we return \( I - 1 \) if \( x < A[I] \), and \( 1 \) if \( x \geq A[I] \). Since \( A[I] \) is the only element of the array, if \( x < A[I] \), there are no smaller elements than \( x \), and so the predecessor is defined to be \( 0 \). If \( A[1] \leq x \), \( J = 1 \) is the only position in the array, and hence the largest one with \( A[J] \leq x \). Thus, the algorithm is correct when \( n = 1 \).

Assume the algorithm works correctly on all inputs of size \( 1 \leq n \leq k - 1 \), where \( k \geq 2 \). We will show that it works correctly on an input array of size \( n = k \). Note that we set \( m = \lfloor (I + J)/2 \rfloor \), and since \( k \leq 2 \), \( I \leq m < m + 1 \leq J \). So both arrays we might call recursively are of size strictly less than \( k \), so by the induction assumption, each recursive call returns the predecessor of \( x \) within the sub-array.

If \( A[m] \leq x \) our algorithm returns \( \text{Pred}(x, A[m+1..J]) \), which is the largest \( m + 1 \leq p \leq J \) so that \( A[p] \leq x \), or \( m \) if no such \( p \) exists. If the \( p \) returned is bigger than \( m \), then since \( p > m \), \( p \) is larger than any index \( I \leq p \leq m \), so \( p \) is still the largest index in the range \( I \leq p \leq J \) with \( A[p] \leq x \), hence \( p \) is the predecessor within the array \( A[I..J] \). If the \( p \) returned is \( m \), \( x < A[p] \) for all \( m + 1 \leq p \leq J \), and \( A[m] \leq x \), so \( m \) is the correct predecessor for \( x \) in the array \( A[I..J] \).

If \( A[m] > x \), our algorithm returns \( \text{Pred}(x, A[I..m]) \), which by the induction hypothesis is the largest \( I \leq p \leq m \) so that \( A[p] \leq x \), or \( I - 1 \) if none exists. In the first case, note that there is no \( p \geq m + 1 \) with \( A[p] < x \), since for all such \( p x < A[m] \leq A[p] \), since the array is sorted. So \( p \) is also the predecessor in the entire array. In the second case, no such \( p \) can exist in the entire array for the same reason, so \( I - 1 \) is the predecessor for the entire array.

Thus, by strong induction on the size of the array, the algorithm returns the predecessor for all inputs.

On an input of size \( n \geq 1 \), this algorithm takes constant time and makes a single recursive call to an instance of size at most \( n/2 \). So the total time is given by \( T(1) = c \), \( T(n) = T(n/2) + c' \) for some constants \( c \) and \( c' \). This is of the form \( T(n) = 1T(n/2) + O(n^0) \), so we can apply the Master theorem with \( a = 1, b = 2, d = 0 \). Since \( a = 1 = 2^0 \), we use the “steady state” formula, \( T(n) \in O(n^{d} \log n) = O(\log n) \).

4. Let \( n \) be a nonnegative integer. In this problem, we are given an array of integers \( A[1, \ldots, n] \) and an integer \( x \). We wish to compute the successor of \( x \) in \( A \), which we define as the smallest element in \( A \) which is greater than \( x \). For example, if \( A = [8, 4, 2, -7, -5, 6, 2] \) and \( x = 2 \), then the successor of \( x \) in \( A \) is 4. Similarly, the successor of \( -6 \) in \( A \) is \( -5 \). We define the successor of \( x \) in \( A \) to be \( \infty \) if there is no integer in \( A \) which is greater than \( x \).

Here is a recursive algorithm which takes as input \( A[1, \ldots, n] \) and an integer \( x \), and returns the successor of \( x \) in \( A \), as defined above.

\textbf{procedure} Successor\((A[1, \ldots, n], x)\)

1. \textbf{if} \( n = 0 \) \textbf{then return} \( \infty \)
2. \( s := \text{Successor}(A[1, \ldots, n - 1], x) \)
3. \textbf{if} \( (A[n] > x \text{ and } A[n] < s) \) \textbf{then} \( s := A[n] \)

4
4. \textbf{return} \(s\)

(a) (6 points) Prove that this algorithm correctly returns the successor of \(x\) in \(A\).

\textbf{Solution:} We will prove this by induction on \(n\), the length of the array. For the base case, when \(n = 0\), the algorithm returns \(\infty\), which is correct because for any integer \(x\), there is no integer in \(A\) which is greater than \(x\), since there is nothing in \(A\) at all.

Now, as our inductive hypothesis, suppose that for any input array \(A\) of length \(n - 1\) and any integer \(x\), \(\text{Successor}(A, x)\) correctly returns the successor of \(x\) in \(A\). If \(n > 0\), then in line 2, the algorithm recursively calls itself with an input array of length \(n - 1\), and by the inductive hypothesis, this recursive call returns the successor of \(x\) in \(A[1, \ldots, n - 1]\). That is, after line 2, \(s\) holds the value of the smallest element in \(A[1, \ldots, n - 1]\) which is greater than \(x\). Then in line 3, \(s\) is updated to be \(A[n]\) if and only if \(A[n]\) is also greater than \(x\) and even smaller than the current value of \(s\). Therefore, after line 3, \(s\) holds the value of the smallest element in \(A[1, \ldots, n]\) which is greater than \(x\). This is by definition the successor of \(x\) in \(A\), so the algorithm is correct.

(b) (2 points) Let \(T(n)\) be the running time of this algorithm. Write a recurrence relation that \(T(n)\) satisfies.

\textbf{Solution:} In the base case, we only return a value, which takes constant time. Otherwise, we make a recursive call with an array of size one smaller than before, then do some constant-time operations in lines 3 and 4. Therefore, the recurrence is

\[T(n) = T(n - 1) + c, \text{ with } T(0) = d, \text{ where } c \text{ and } d \text{ are constants.}\]

(c) (2 points) Solve the recurrence found in part (b) and write the solution in \(\Theta\) notation.

\textbf{Solution:} Unraveling the recurrence gives

\[
T(n) = T(n - 1) + c \\
= T(n - 2) + c + c \\
= T(n - 3) + c + c + c \\
\vdots \\
= T(n - k) + ck \\
\vdots \\
= T(0) + cn \\
= d + cn
\]

In \(\Theta\) notation, this is \(\Theta(n)\), or linear time in the length of the input array \(A\).

5. Say we are given two polynomials of degree \(n - 1\) as their arrays of co-efficients, \(a_0, a_{n-1}\) and \(b_0, b_{n-1}\), so that \(a(x) = \sum_{i=0}^{i=n-1} a_i x^i\) and \(b(x) = \sum_{i=0}^{i=n-1} b_i x^i\) and we want to compute the description of \(a(x)b(x)\), their product.

For all parts, you are not required to prove correctness of your algorithms, but are required to give a time analysis in \(O\) form.

(a) (5 points) Give the straight-forward, iterative algorithm for this problem.

In the iterative algorithm, we multiply each term of the first polynomial with each term of the second. In terms of arrays, the product might have degree \(2n - 2\), so we’ll use \(e_0, e_{2n-2}\) as the description of the product.

In pseudo-code, this looks like:
i. FOR \( I = 0 \) TO \( 2n - 2 \) do: initialize \( c_I \) to 0.

ii. FOR \( I = 0 \) TO \( n - 1 \) do:

iii. FOR \( J = 0 \) TO \( n - 1 \) do:

iv. \( c_{I+J} := c_{I+J} + a_I b_J \).

v. Return \( c_0...c_{2n-2} \).

Since the two nested loops each iterate exactly \( n \) times, the total time is \( \Theta(n^2) \).

(b) Give a simple divide-and-conquer algorithm for this problem.

(5 points)

Let \( a(x) = \sum_{i=0}^{n-1} a_i x^i \) be the first polynomial, and \( b(x) = \sum_{i=0}^{n-1} b_i x^i \) be the second.

We can break \( a \) and \( b \) up into two \( n/2 - 1 \) degree polynomials each, with \( ah = \sum_{i=0}^{n/2-1} a_{n/2+i} x^i \) and \( bl = \sum_{i=0}^{n/2-1} a_{n/2+i} x^i \), and similarly for \( bh, bl \). Then \( a(x) = ah(x)x^{n/2} + al(x) \) and \( b(x) = bh(x)x^{n/2} + bl(x) \). So \( a(x)b(x) = ah(x)bh(x)x^n + (al(x)bh(x) + ah(x)bl(x))x^{n/2} + al(x)bl(x) \).

This gives the following recursive algorithm, where in terms of arrays, \( Add \) adds the values of corresponding elements in the two arrays (if the arrays have different dimensions, \( Add \) will treat the smaller dimension array as having 0's in the larger dimensions), and \( MultiplyPower(a, m) \) shifts an array over \( m \) positions, adding \( m \)'s at the beginning. \( DnCPM(a[0..n-1], b[0..n-1]) \)

i. IF \( n = 1 \) return the array with one element, \( c[0] = a[0]b[0] \)

ii. IF \( n \) is odd, define \( a_n = b_n = 0 \) and increment \( n \)

iii. \( al[0..n/2-1] := a[0..n/2-1], ah[0..n/2-1] := a[n/2..n-1] \)

iv. \( bl[0..n/2-1] := b[0..n/2-1], bh[0..n/2-1] := b[n/2..n-1] \)

v. \( H[0..n-2] := DnCPM(ah[0..n/2-1], bh[0..n/2-1]) \)

vi. \( H[0..n-2] := MultiplyPower(H[0..n-2], n) \)

vii. \( M[0..n-2] := Add(DnCPM(al[0..n/2-1], bl[0..n/2-1]), DnCPM(al[0..n/2-1], bl[0..n/2-1])) \)

viii. \( M[0..3/2n-2] := MultiplyPower(M[0..n-2], n/2) \)

ix. \( L[0..n-2] := DnCPM(al[0..n/2-1], bl[0..n/2-1]) \)

x. Return \( Add(Add(H[0..2n-2], M[0..3/2n-2]), L[0..n-2]) \)

Since both \( Add \) and \( MultiplyPower \) are linear time, as is copying over the subarrays, and we make 4 recursive calls to multiply arrays of dimension \( n/2 \), this gives us the recurrence \( T(n) = 4T(n/2) + O(n) \). Using the Master Theorem with \( a = 4, b = 2, d = 1 \), we are in the bottom-heavy case, and \( T(n) = O(n \log_2 n) = O(n \log_2 n) = O(n^2) \), just like the iterative algorithm.

(c) Use the Karatsuba method to give an improved divide-and-conquer algorithm for this problem.

(10 points)

Karatsuba's method is based on the polynomial identity:

\((ah(x) + al(x))(bh(x) + bl(x)) = ah(x)bh(x) + al(x)bh(x) + ah(x)bl(x) + al(x)bl(x)\).

So to get the middle two terms, we can compute this one product, and subtract off the first and last terms, which are also needed as the high and low terms of the product. I'm going to leave off the dimensions for arrays in calls to procedures for brevity. Let \( Subtract \) be a procedure that subtracts two arrays co-efficient by co-efficient, similar to \( Add \) from the previous part.

KaraPM \((a[0..n-1], b[0..n-1]) \)

i. IF \( n = 1 \) return the array with one element, \( c[0] = a[0]b[0] \)

ii. IF \( n \) is odd, define \( a_n = b_n = 0 \) and increment \( n \)

iii. \( al[0..n/2-1] := a[0..n/2-1], ah[0..n/2-1] := a[n/2..n-1] \)

iv. \( bl[0..n/2-1] := b[0..n/2-1], bh[0..n/2-1] := b[n/2..n-1] \)

v. \( H := KaraPM(ah, bh) \)

vi. \( L := KaraPM(al, bl) \)

vii. \( S := KaraPM(Add(ah, al), Add(bh, bl)) \)
viii. \( M := \text{Subtract}(S, \text{Add}(L, H)) \)

ix. \( H := \text{MultiplyPower}(H, n) \).

x. \( M := \text{MultiplyPower}[M, n/2] \)

xi. Return \( \text{Add}(\text{Add}(H, M), L) \)

Now, we make 3 recursive calls to arrays of size \( n/2 \), and add, subtract, and shift a fixed number of times. So the recurrence is \( T(n) = 3T(n/2) + O(n) \). Since \( 3 > 2^{1} \), we are still in the bottom-heavy case and the time is \( T(n) \in \Theta(n^{\log_2 3}) \subseteq O(n^{1.6}) \). So we’ve improved the running time substantially.