Problem 1: Asymptotics [20 points].

(a) Prove or disprove: If \( f(n) \) is \( O(g(n)) \) then \( g(n) \) is \( \Omega(f(n)) \). [5 points]

(b) Prove or disprove: \( f(n) \) is \( O(g(n)) \) or \( g(n) \) is \( O(f(n)) \). [5 points]

(c) Prove or disprove: For every \( k \geq 2 \), \( \log_k(n) \) is \( O(\log_2(n)) \). [5 points]

(d) Prove or disprove: If \( f(n) = O(g(n)) \), then \( g(n) - f(n) = \Theta(g(n)) \). [5 points]

Solution 1.

(a) This is true. Suppose that a function \( f(n) \) is \( O(g(n)) \). Then by definition, there exist constants \( c > 0 \), \( n_0 > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n \geq n_0 \). Then it follows that \( 0 \leq \frac{1}{c}f(n) \leq g(n) \) for all \( n \geq n_0 \), and \( \frac{1}{c} \) is a constant greater than 0 since \( c > 0 \). Then, by definition, \( g(n) \) is \( \Omega(f(n)) \).

(b) This is false in general. Consider, as a counterexample, \( f(n) = 1 \) if \( n \) is even and \( n \) if \( n \) is odd, and \( g(n) = \sqrt{n} \). Then \( f(n) \) is not \( O(g(n)) \), as can be seen by looking at the odd \( n \), and \( g(n) \) is not \( O(f(n)) \), as can be seen by looking at the even \( n \).

(c) This is true. Let \( k \geq 2 \). Then \( \log_k(n) = \log_2(n)/\log_2(k) \). So choosing \( c \) to be \( 1/\log_2(k) \), we can see that for all \( n \), \( \log_k(n) \leq c\log_2(n) \), so by definition \( \log_k(n) \) is \( O(\log_2(n)) \). Crucially note that while our choice of \( c \) depends on \( k \), it does not depend on \( n \).

(d) This is false. Consider, as a counterexample, \( f(n) = n - 1 \) and \( g(n) = n \). Then \( f(n) = O(g(n)) \), but \( g(n) - f(n) = 1 \neq \Theta(n) \).
Problem 2: Cryptography (Taken from DPV 1.45) [15 points]. Recall that in the RSA public-key cryptosystem, each user has a public key $P = (N,e)$ and a secret key $d$. In a digital signature scheme, there are two algorithms: sign and verify. The sign procedure takes a message and a secret key, then outputs a signature $\sigma$. The verify procedure takes a public key $(N,e)$, a signature $\sigma$, and a message $M$, then returns “true” if $\sigma$ could have been created by sign (when called with message $M$ and the secret key corresponding to the public key $(N,e)$); “false” otherwise.

(a) Assume that Alice is trying to send a message to Bob, but the malicious Mallory may be trying to interfere with their communications. Explain in a sentence or two why Alice and Bob might want to use a digital signature scheme. [5 points]

(b) In an RSA digital signature scheme, we simply implement $\text{sign}(M,d) = \sigma = M^d \mod N$, where $d$ is the secret key of the party doing the signing and $N$ is part of the public key of the party doing the signing. Give a corresponding verify procedure with the behavior outlined above. Argue that your verify procedure is correct. [10 points]

Solution 2.

(a) A digital signature scheme allows Bob to confirm that Alice (or someone with Alice’s private key, to be precise) actually sent the message he receives. That is, it allows Bob to be sure that Mallory has not altered the message in any way.

(b) The verify procedure should output “true” if $\sigma^e \equiv M \mod N$, otherwise output “false.”

To see the correctness of this procedure, suppose that $\sigma^e \equiv M \mod N$. We want to show that $\sigma^e \equiv M \mod N$. Note that

$$\sigma^e \equiv M^d \mod N$$

Recall that the component of the public key $N = pq$ for some primes $p$ and $q$, and $e$ is relatively prime to $(p-1)(q-1)$. Furthermore, $d$ is the inverse of $e$ modulo $(p-1)(q-1)$, meaning by definition that $de \equiv 1 \mod (p-1)(q-1)$. Therefore, we can write $de$ as $1 + k(p-1)(q-1)$ for some integer $k$ to get

$$\sigma^e \equiv M^{1+k(p-1)(q-1)} \mod N$$

We want to prove $\sigma^e \equiv M \mod N$, so we consider

$$\sigma^e - M \equiv M \left( M^{k(p-1)(q-1)} - 1 \right) \mod N$$

But since $p$ and $q$ are prime, by known from Fermat’s Little Theorem that $M^{k(p-1)(q-1)} - 1$ is divisible by $p$ and by $q$. To see this, note that $(M^{k(p-1)(q-1)}) - 1 \equiv (M^{p-1} \mod p)^{k(q-1)} - 1 \mod p$ is just 0 since by Fermat’s Little Theorem, $M^{p-1} \equiv 1 \mod p$. The argument is the same for $q$. Finally, since $M^{k(p-1)(q-1)} - 1$ is divisible by $p$ and by $q$, and $p$ and $q$ are both prime, it must be divisible by $pq = N$. Therefore, $M^{k(p-1)(q-1)} - 1 \equiv 0 \mod N$, so $\sigma^e - M \equiv 0 \mod M$, implying that $\sigma^e \equiv M \mod N$, as we wanted to show.

Problem 3: Recurrence Relations [10 points].

(a) Solve the following recurrence relation. Assume $T(1) = 1$. Express your answer in big-\(\Theta\) notation. $T(n) = 2T(n-1) + 2^n$. [5 points]

(b) $T(n) = 2T(\sqrt{n}) + 1$ with base $T(2) = 1$. (To simplify the solution, suppose you can always take the square root without a remainder in each step of the recurrence). [5 points]
Solution 3.

(a) We solve the recurrence by back substitution.

\[ T(n) = 2T(n - 1) + 2^n \]
\[ = 2(2T(n - 2) + 2^{n-1}) + 2^n = 4T(n - 2) + 2^n + 2^n \]
\[ = 4(2T(n - 3) + 2^{n-2}) + 2^n + 2^n = 8T(n - 3) + (3)2^n \]
\[ \vdots \]
\[ = 2^{n-1}T(1) + (n - 1)2^n \]
\[ = \Theta(n2^n) \]

(b) We cannot apply the master theorem because of the square root, so we draw the recursion tree:

First, we determine the height of the tree; denote this by \( h \). By observing the powers of the arguments in the tree, we can see that in order to get to the base case of \( n = 2 \), it must be that

\[ n^{\left(\frac{1}{2}\right)} = 2 \]
\[ \left(\frac{1}{2}\right)^h = \log_{n}(2) = \frac{\log 2}{\log n} = \frac{1}{\log n} \]
\[ 2^h = \log n \]
\[ h = \log(\log(n)) \]

Now, since we do a constant amount of work (1, specifically) at each node in the recursion tree, we note that at depth \( d \) in the recursion tree we do \( 2^d \) total work. We can therefore write the sum over the tree as a geometric series as follows:

\[ T(n) = \sum_{d=0}^{\log(\log(n))} 2^d = \frac{1 - 2^{\log(\log(n))+1}}{1 - 2} = 2^{\log(\log(n))+1} - 1 \]

\[ T(n) = 2\log(n) - 1 \]

Therefore, we conclude that \( T(n) = \Theta(\log n) \)

Problem 4: Divide and Conquer [20 points]. Consider the following problem of covering a chess board board: You are given a \( 2^n \times 2^n \) size chess board with 1 arbitrary tile already covered and an unlimited number of “trominoes” which cover 3 tiles of the board in an L shape. Write a recursive algorithm in pseudocode to cover the rest of the board. Argue for the correctness of your algorithm. Infer a recurrence relation for your algorithm, solve that recurrence relation, and give the asymptotic running time of your algorithm.
1: procedure RecursiveTile($B_n$)
2:   if $n=1$ then
3:     Place a single tromino on the 3 empty tiles and return the covered board
4:   end if
5:   Let $B_{++}, B_{+-}, B_{-+}, B_{--}$ denote the 4 ($2^{n-1} \times 2^{n-1}$) quadrants of the board.
6:   Place a tromino onto $T$ covering one tile from each of $B_{++}, B_{+-}, B_{-+}, B_{--}$ except the quadrant
7:      which already has one tile covered. In each quadrant, treat the covered tile as missing.
8:   return (RecursiveTile($B_{++}$), RecursiveTile($B_{+-}$), RecursiveTile($B_{-+}$), RecursiveTile($B_{--}$))
9: end procedure

Solution 4. Let $T$ represent the four center tiles on the board. Let $B_n$ be a $2^n \times 2^n$ board with exactly
one tile covered. Our algorithm will be called RecursiveTile($B_n$).

Now we argue for the correctness of this algorithm. In the base case of $n = 1$, we have a $2^1 \times 2^1$ board
with exactly one tile covered, so the remaining three tiles of the board form exactly the shape of a tromino,
and we correctly place a single tromino and return. For the inductive hypothesis, suppose that for all $B_n,
RecursiveTile(B_n)$ returns a correctly tiled $2^n \times 2^n$ board when exactly one tile is missing. Consider some
$B_{n+1}$. We want to show that it follows that RecursiveTile($B_{n+1}$) returns a correctly tiled $2^{n+1} \times 2^{n+1}$
board, again when exactly one tile is missing.

The missing tile in $B_{n+1}$ is in exactly one of the four quadrants. By placing a single tromino at the
center of $B_{n+1}$ as we do and, we cover exactly one tile from the three other quadrants and then treat these
as missing, so in the recursive call, each of $B_{++}, B_{+-}, B_{-+}, B_{--}$ satisfy the requirement of being a $2^n \times 2^n$
board with exactly one tile missing. By the inductive hypothesis, we get correct tilings of each quadrant.
Combined with the tromino we place in the center, this produces a correct tiling on $B_{n+1}$.

Now we draw the recursion tree and infer a recurrence relation.

\[ T(n) \]
\[ \vdots \]
\[ T(n-1) \]
\[ \vdots \]
\[ T(n-1) \]
\[ \vdots \]
\[ T(n-1) \]
\[ \vdots \]

The recurrence structure is clear, $n$ decreases by 1 at each level and there is a branching factor of 4 from the
division into four quadrants. Assuming that a single tromino can be placed in a constant amount of time
with respect to $n$, this gives the following recurrence: $T(n) = 4T(n-1) + \Theta(1)$. The master theorem does
not apply, but we note that we do a constant amount of work ($\Theta(1)$) at every node in the recursion tree.
Thus, all we need to do is count the number of nodes. The height of the tree is clearly $n$, and if we begin
indexing at 0, we do $4^d\Theta(1)$ work at each depth $d$. Thus, we can write that:

$$ T(n) = \sum_{d=0}^{n-1} \Theta(1)4^d $$

Since $\Theta(1)$ is a constant, this is a geometric series and we can apply the formula to get:

$$ T(n) = \Theta(1) \frac{1 - 4^n}{1 - 4} = \Theta(1)3(4^n - 1) = \Theta(4^n) $$

Problem 5: More Divide and Conquer [20 points]. Given two sorted arrays of integers $A$ and $B$,
each of size $n > k$, write an algorithm that finds the $k$th smallest element of the union of the two arrays (for
example, if $A = [0, 1, 5]$ and $B = [2, 3, 6]$ then the 2nd smallest element of their union would be 1). Argue
for the correctness of your algorithm. Infer a recurrence relation for your algorithm, solve that recurrence
Finally, we note that because, in this case, we have removed sorted, this means that $B$ than the $A$ case without loss of generality (the opposite argument is symmetric). Since it was not a base case, neither $A[i]$ nor $B[j]$ are the correct $k$th smallest element. Because $A[i] \leq B[j]$, $A[i]$ must be smaller than the $k$th smallest; since $A$ is sorted, this means that $A[0], ..., A[i]$ are all smaller than the $k$th smallest, and can be disregarded. Similarly, since $B[j] \geq A[i]$, $B[j]$ must be larger than the $k$th smallest; since $B$ is sorted, this means that $B[j], ..., B[size(B) - 1]$ are all larger than the $k$th smallest, and can be disregarded. Finally, we note that because, in this case, we have removed $i + 1$ items smaller than $k$, we decrease $k$ by $i + 1$ in the recursive call.

Note that we start with truncated arrays of size $k$, and half of each array is thrown away at each iteration. Each recursive call can be completed in constant time, so the running time of the algorithm is described by the recurrence $T(k) = T(k/2) + \Theta(1)$. Thus, it is immediate from the master theorem that the running time is $O(\log k)$.

We have already argued that the base case conditions are correct. In the recursive case, consider the $A[i] \leq B[j]$ case without loss of generality (the opposite argument is symmetric). Since it was not a base case, neither $A[i]$ nor $B[j]$ are the correct $k$th smallest element. Because $A[i] \leq B[j]$, $A[i]$ must be smaller than the $k$th smallest; since $A$ is sorted, this means that $A[0], ..., A[i]$ are all smaller than the $k$th smallest, and can be disregarded. Similarly, since $B[j] \geq A[i]$, $B[j]$ must be larger than the $k$th smallest; since $B$ is sorted, this means that $B[j], ..., B[size(B) - 1]$ are all larger than the $k$th smallest, and can be disregarded. Finally, we note that because, in this case, we have removed $i + 1$ items smaller than $k$, we decrease $k$ by $i + 1$ in the recursive call.

We write the algorithm recursively. In the base case, note that if $i + j = k - 1$ and $B[j - 1] \leq A[i] \leq B[j]$ then $A[i]$ must be the $k$th smallest element of the union. This is because all $j$ items $B[0], ..., B[j - 1]$ and all $i$ items $A[0], ..., A[i - 1]$ must be less equal to $A[i]$, and $j + i = k - 1$. By the same argument, if $A[i - 1] \leq B[j] \leq A[i]$ then $B[j]$ must be the $k$th smallest. In the recursive case, if neither of these conditions hold, then if $A[i] \leq B[j]$ we can safely disregard the left half of $A$ (including $A[i]$) and the right half of $B$ (including $B[j]$). The opposite holds if $B[j] \leq A[i]$. When we do this, we update $k$ by decreasing it by the number of left elements we disregard (since they must have been among the elements smaller than the $k$th smallest). To make this procedure precise:

```
1: procedure K-SMALLEST-UNION(A, B, k)
2:   i ← \lfloor \frac{k}{2} \rfloor
3:   j ← (k - 1) - i
4:   if $B[j - 1] \leq A[i] \leq B[j]$ then
5:     return $A[i]$
6:   end if
7:   if $A[i - 1] \leq B[j] \leq A[i]$ then
8:     return $B[j]$
9:   end if
10:  if $A[i] \leq B[j]$ then
11:     return K-SMALLEST-UNION($A[i + 1, ..., size(A) - 1], B[0, ..., j - 1], k - (i + 1)$)
12: end if
13:  if $B[j] \leq A[i]$ then
14:     return K-SMALLEST-UNION($A[0, ..., i - 1], B[j + 1, ..., size(B) - 1], k - (j + 1)$)
15: end if
16: end procedure
```