1. To find the total cost using aggregate analysis we need to look at all numbers 1 to n and add a cost for each of those numbers. The majority of those numbers are not powers of two (1, 2, 4, 8…), meaning more of the operations will have a cost of 1. However, the numbers that are powers of two will have a cost greater than 1 and increases exponentially. We can see that the number of powers of 2 between 1 and n is about log2(n) which will be much smaller than n. Adding up these larger costs and all the ones in between them will give us a number less than 2n every time. We can then take 2n / n, since n is the total number of operations and aggregate analysis is essentially the average, to get 2. This means each operations will cost 2 on average giving us O(1) time complexity.

2. We can say that each of these push and pop operations cost 1 each. Since we make a copy of the stack over k operations, the stack will cost at most k. However, copying the stack will take time so we need to build up credit leading to those operations. To do this we can just make each operation cost 2, then when we get to the kth operation and need to copy the stack, the built up credit will pay for it. This gives each operation a cost of 2 for n operations. This results in a time complexity of (n).

3. Min-Heap Amortized Cost
    a. For the potential function, we can use something like potential(n) = c*n*log(n). This will get the potential of the head based on its size, and we can use it to find the change before and after an insert or extract.
    b. Insert
        i. For inserting, we know the size of the heap will change from n to n+1. Using that we can get potential(n+1) – potential(n) to get the change in potential. This becomes (c*n+1*log(n+1)) – (c*n*log(n)). Using this, we know that the potential function will increase by about log(n) when inserting.
    c. Extract-Min
        i. Extracting works in a similar way, but we get the change in potential by taking potential(n-1) – potential(n) which is the same as (c*n-1*log(n-1)) – (c*n*log(n)) using our potential function. From this, we get that the change in potential is -log(n). Knowing that and the fact the removing actually costs log(n), we get the cost = log(n) – log(n) which simplifies to a time complexity of O(1).

4. Because the priority queue is implemented using a binary heap, we're really trying to analyze the amortized cost of inserting and removing from a binary heap. Looking at that, we know that some inserts will take longer than others like if the number inserted needs to be placed at the top or middle. This would require a lot of swapping. The same concept would apply to removing, but only from the top due to how heaps work. Inserting a number that goes to the bottom would be much quicker as it wouldn't require swapping any values around. Combined, these operations average out to be O(log(n)).

5. Binary Counter Amortized Cost
    a. Aggregate
        i. In this case, since we're working with 2-bit binary, the total cost for n operations will be O(2n). However, since were looking at the aggregate cost, essentially the average, we can divide this time by the number of operations, n. The results in an amortized cost of 2, meaning a time complexity of O(1).
    b. Accounting
        i. For this methos of finding the amortized cost, we can say that each operation has a cost of 1, but we store 1 extra to use later when it gets flipped back to a 0. The means that each bit that got turned into a 1 will be free when it's time to flip it back to a 0. Because of this, the amortized time complexity is O(1).
    c. Potential
        i. We can define the potential functions as $\Phi = 1 - t$ where t is the number of 1's from the bit being flipped to the end of the number. When we do an operation, the actual cost will be $t + 1$ and the change in potential is $1 - t$. Since the amortized cost for this methos comes from the actual cost + the change in potential, we get the amortized cost = $(t + 1) + (1 - t)$. this simplifies to 2 which is O(1) time complexity.

6. Using the aggregate method and that the tree is bounded by O(log (n)) we can get the total number of operations, num, to get the amortized cost. To get the total cost we add all the operations, each one having a max cost of log(n). This means the total cost will be at most num*log(n) and dividing by the number of operations, num, gives us a amortized cost of log(n). This means the average time complexity is O(log(n)).