**Big Oh Workshop**

**Arrays**

1. Answer: O(1). No matter how many inputs there are, the function will always do one action. Space Complexity: O(1)
2. Answer: O(n). This function does 1 action for each item in the array.
3. Answer: O(n). This function prints out half of the array, so it does n/2 actions. (1/2) is a constant however and gets thrown out, so it ends up being O(n)
4. Answer: O(n). This one is different as it only breaks the loop if the item inside equals the length / 2. The answer is O(n) since worst case is O(n).
5. Answer: O(n). The first loop has worst O(n). The second loop has O(n) since it will always iterate through everything. The last line ‘puts arr.last’ is just a constant time action so it won’t count toward time complexity. So O(n) + O(n) is O(2n) which is just O(n) since 2 is a constant.
6. Answer: O(n). Worst case for this problem is O(n) since the target may not even be in the array at all.
7. Answer: O(n^2). For each item in the array, we are running the search function, which runs worst case at O(n) times. Therefore O(n) \* O(n) = O(n^2), a quadratic time.

**Iterations**

1. Answer: O(n \* m). We’re iterating through every number 0 to n, and for each one doing m actions, therefore it’s at quadratic time.
2. Answer: O(n^2). The outside loop runs n times, while the inside loop runs 0, then 1, then 2, then 3 times, which averages out to n/2 times. n \* (n/2) is (n^2)/2 which is just O(n^2)
3. Answer: O(n \* m). The outside function will always run either n or m times, and the inside function will always run around either n/2 or m/2. n \* m(/2) and m \* (n/2) will always just be O(n \* m) due to constants.

**Recursive**

1. Answer: O(n). The recursion will always run n / 5 times. (n / 5) is just O(n) time complexity.
2. Answer: O(log n). The # of iterations changes based on division of n, which means it’s reliant on the 10^x power, which mean it’s logarithmic time complexity.
3. Answer: O(2^n). This recursive call is creating a binary tree with n levels. It starts at n, and then makes two n-1 calls, then 4 n-2 calls. We want to compare n(the input) to the # of nodes on that line. 2^n will give us that since every new line is double the amount of nodes, 1 -> 2 -> 4 -> 8 -> 16 -> 32

**Grab** **Bag**

1. Answer: O(2^n + n). The recursive call runs at O(n) time, but produces an array that is of 2^n length. That 2^n length array is mapped over, so 2^n more actions are done. At this point we have n + 2^n. Now we take our original array an concat it with the new one, which is another 2^n process. Overall it’s 2(2^n) + n, which is 2^(n+1) + n, which is 2^n + n