Rob Lorch

Write-up of Complexity Analyses (Assignment 6)

1. The operation I will cout is the number of ‘<’ comparisons, subtractions, steps in the ‘for’ loop, and calls to ‘Math.abs()’, and the input to the model is an array of doubles with length ‘n’. The worst-case scenario is that every element of ‘arr’ has to be traversed (because all but the last element fail the comparison). So, all of the above would happen ‘n’ times. Additionally, the function would have to make an iterator, which takes 3 steps (making ‘next()’, ‘hasNext()’, and ‘remove()’), and there is also a return statement, which is a step. This is a total of T(n) = 4n + 4. So, an upper bound would be O(n) with c = 73 and x\_0 = 73.
2. The operation I will first count is the number of calls to ‘fastModExp’, and the input to the model is an integer ‘n’ (the exponent to which x is called, mod m). Assuming ‘n’ is a power of 2, the recurrence relation for the time complexity is T(1) = 0 and T(n) = 1 + T(n/2). After ‘k’ repetitions, we have T(n) = k + T(n/[2^k]). We stop when n/(2^k) = 1, or when k = log2n. So, we get T(n) = log2n. Also, we can count the number of conditional comparisons, and there are twice as many conditional comparisons as there are calls to ‘fastModExp’, plus one (for each of the cases before the base case, you check ‘y <= 1’ and ‘y % 2 == 0’, and at the base case, you check ‘y <= 1’ an additional time. Also, there is a ‘return’ call. So, ‘fastModExp’ is T(3\*log2*n +* 2), and so ‘fastModExp’ is in O(log2n). We can prove this with c = 73 and x\_0 = 73.
3. I will count the number of array assignments, and the input to the model is an array of integers with ‘n’ elements. The function iterates through the array of ‘n’ elements ‘n’ times. Also, there is a conditional check for ‘arr == null’, which is 1 operation. Additionally, you increment index every time you do an array assignment. So, the total is T(n) = 2\*n^2 + 1. Therefore, an upper bound would be O(n^2) with c=73 and x\_0 = 73.
4. I will count the number of string concatenations, and the inputs to the model are an array of strings of length ‘x’ and an integer ‘y’ (that describes the number of times each string is concatenated). Not taking the cost of concatenation into account, the function runs in O(x\*y) because the function has to do ‘x’ number of concatenations, each one ‘y’ times. If each concatenation of two strings with length ‘n’ and ‘m’ is O(n+m), then the runtime analysis changes. Suppose the longest string in the array of strings is of length ‘n’. Then, the number of operations is at most ‘n \* [x\*y(x\*y-1)/2)]’, which is in O(n\*[x\*y]^2). I came to this conclusion because the concatenations would be of cost ‘n + 2n + 3n + … + (x\*y)n’ (because there are x\*y total concatenations). Factor out the ‘n’ to get ‘n[1 + 2 + 3 + … + (x\*y)]’ which is the same as ‘n \* [x\*y(x\*y-1)/2)]’.
5. I will count the number of array assignments, and the inputs to the model are two arrays of integers of length ‘n’ and ‘m’. The function runs in O(n+m) because there is one array assignment for each index of each array, so we can just count the total number of indices with ‘n+m’. The space allocation is in O(n+m) as well because we make an array of integers ‘ret’ of length ‘n+m’, and the integers ‘i’ and ‘j’ only increase the space allocation by a constant number. After reviewing the previous problems’ space complexity, I do not think one can put a bound on a function’s time complexity given its space complexity. This is because, for example, even if we are only working with a single array of length ‘n’, we do not know from the space complexity how many operations we do that relate to that array. We may need to only traverse the array once, which would be O(n), but we might also need to traverse the array ‘n’ times, which would be O(n^2).