ex4.pdf Group 17 Lab 3

1. Derive the formula for worst-case complexity

The worst case for quick sort happens when the pivot chosen each time is the largest or smallest in the sub-array. For simplicity's sake, we will always choose the pivot to be the first element in the sub-array. This means that we can achieve the worst case is the array is either already sorted or sorted in reverse order (largest to smallest). When the array is in reverse order, when we choose the pivot as the first element we choose the largest element of the array. If this is the case, the algorithm will run the partition section about n times, because each time the chosen pivot is the smallest in the sub-array or the biggest in the sub-array. Switches are not plentiful and most of the time only a single switch happens per partition. The recursive calls are not helpful either, as one of them is always of size 0 and the other has the entire sub-array, meaning that the divide and conquer approach is not taken advantage of, but rather creates repetitive calls that don't divide the problem equally. Because each partition operates about n times (the pointers need to traverse the entire sub-array until they meet) we get the worst-case complexity of O(n * n) = O(n^2)

2. Come up with a vector of 16 elements which incurs worst-case complexity. Manually show the workings of the algorithm until the vector is sorted.

Brackets () are used to show sub-arrays. If an element is written outside of brackets, that means they are not part of the current working sub-array (i.e. they are already properly sorted, and the algorithm recognizes them as such). Note: this is the version of the algorithm that was presented in class.

Initial State:

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

After first partition run:

(1 15 14 13 12 11 10 9 8 7 6 5 4 3 2 16) ()

After second partition run:

(1) (15 14 13 12 11 10 9 8 7 6 5 4 3 2 16)

After third partition run:

1 (2 14 13 12 11 10 9 8 7 6 5 4 3 15) (16)

After fourth partition run:

1 (2) (14 13 12 11 10 9 8 7 6 5 4 3 15) 16

After fifth partition run:

1 2 (3 13 12 11 10 9 8 7 6 5 4 14) (15) 16

After sixth partition run:

1 2 (3) (13 12 11 10 9 8 7 6 5 4 14) 15 16

After seventh partition run:

1 2 3 (4 12 11 10 9 8 7 6 5 13) (14) 15 16

After eighth partition run:

1 2 3 (4) (12 11 10 9 8 7 6 5 13) 14 15 16

After ninth partition run:

1 2 3 4 (5 11 10 9 8 7 6 12) (13) 14 15 16

After tenth partition run:

1 2 3 4 (5) (11 10 9 8 7 6 12) 13 14 15 16

After eleventh partition run:

1 2 3 4 5 (6 10 9 8 7 11) (12) 13 14 15 16

After twelfth partition run:

1 2 3 4 5 (6) (10 9 8 7 11) 12 13 14 15 16

After thirteenth partition run:

1 2 3 4 5 6 (7 9 8 10) (11) 12 13 14 15 16

After fourteenth partition run:

1 2 3 4 5 6 (7) (9 8 10) 11 12 13 14 15 16

After fifteenth partition run:

1 2 3 4 5 6 7 (8 9) (10) 11 12 13 14 15 16

After sixteenth partition run:

1 2 3 4 5 6 7 (8) (9) 10 11 12 13 14 15 16

Note: Array was sorted after the fifteenth partition, but it partitioned one more time

(Part 3 can be found on ex4.py)

4. Plot the results, together with appropriate interpolating functions and discuss your results: do they match your complexity analysis?

Note: Plots can be found on ex4.py.

Yes, the results match our analysis, as we can see that the growth in time complexity for the quicksort algorithm when given its worst-case scenario is quadratic. The function used for fitting the curve is quadratic, in this case with parameters ‘a’ and ‘x’.

$$quadratic = a * x^2$$