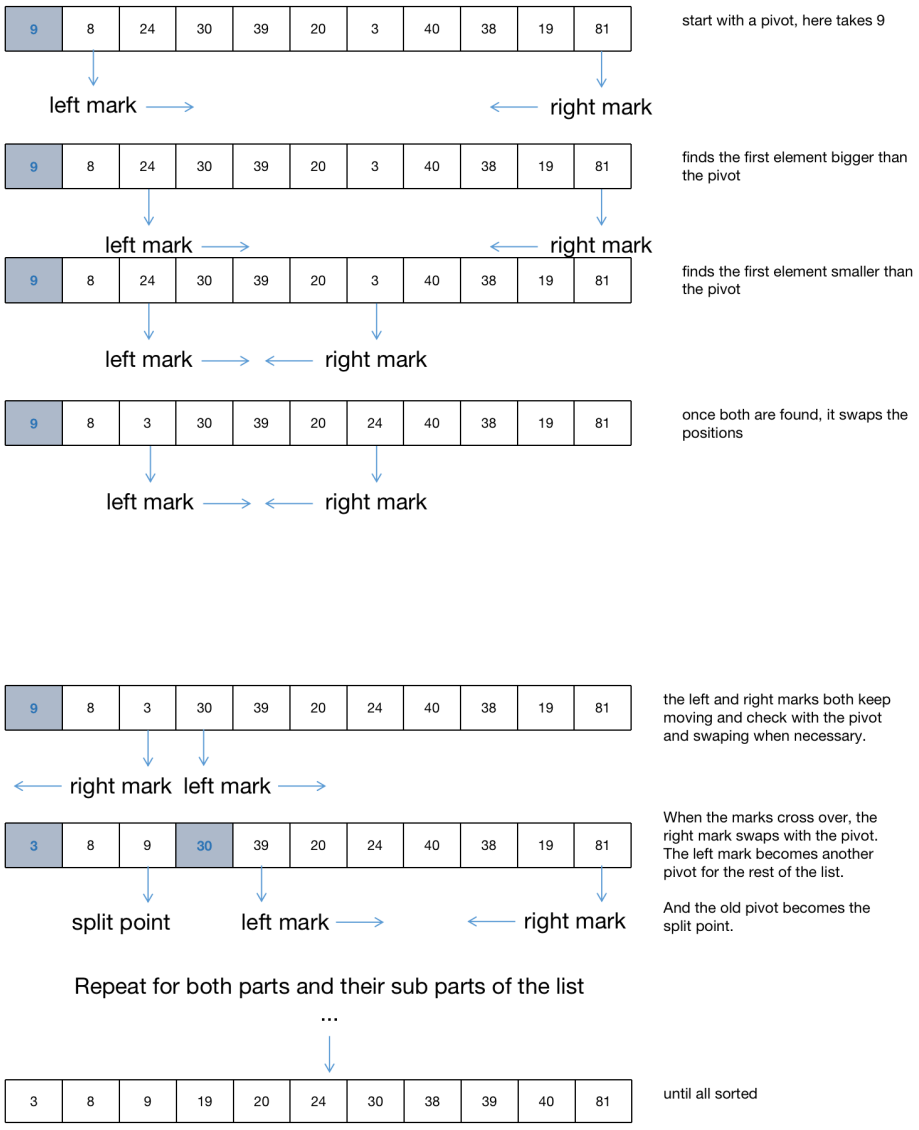
Assignment 2

Maya K. Nachesa, Yini Chen

February 4, 2022

1 Quick Sort



2 Time complexity

The following discussion covers time complexity in the worst-case scenario. The time complexity for linear search is O(n), because in the worst case scenario (the target is at the end of the list or not in the list at all), the algorithm has to run through the entire list to find the target. I.e., if the list size = n, it takes O(n) steps. Binary search is faster at $O(\log_2(n))$. This is because binary search keeps dividing the list into two, and thus dividing the search space by a factor of 2 at each step. Dividing a list of size n until you are left over with 1 element is equivalent to taking the $\log_2(n)$.

Both bubble sort and quick sort have a time complexity of $O(n^2)$, making them slower. Bubble sort works by comparing each two consecutive elements with each other, and if the second element is smaller than the first, it swaps them around. Even if the list is completely sorted. No matter to what degree the list is sorted, it always goes through the list entirely. This leads to any iteration of the list taking at least n-1 time. In the worst case scenario, the list is completely reversed. This means that it has to run through the list n times: n-1 to “bubble" each element up, and 1 last time to verify that it is sorted. This leads to a complexity of $(n-1) \times n$. When the data grows, this -1 becomes irrelevant, yielding a time complexity of $O(n^2)$. Quick sort is also $O(n^2)$ in the worst case scenario, when the list is reversed. First, it has to go through the list n-1 times, because it has to move every of the elements to its correct place. Once it gets to the last one, it is done, hence it does not need to check it. Then, for each of the pivots, it tries to find an element that should be to the left of the pivot (which will always be the last yet unsorted number), and a number that should be to the right. It will never find this number, leading to it checking whether every number leading up to this right number should be on the left, which is n-1 comparisons. Last, the right and left edge coincide, and it swaps them. As the data grows thus, this also approaches time complexity $O(n^2)$.

List size	Linear Search (s)	Binary Search (s)
10	1.690^{-6}	7.607^{-7}
100	8.975^{-6}	1.907^{-6}
1000	8.975^{-5}	1.582^{-5}

Table 1: Average running times over 10000 trials for linear and binary search in seconds

In table 1 we see the average running times for linear and binary search. Since linear search has complexity O(n), we expected the search for list size 100 to be 10 times as slow as for 10, and the search for list size 1000 to be 10 times as slow as for 100. However, searching through a list of 100 was 5.31 times as fast as for 10, and searching through a list of 1000 was 9.45 times as fast as for 100, which is not what we would expect. Our expectation that linear search would take 10 times as long on a list that is 10 times as long was based on the worst case scenario (the item is not in the list or at the end). This expectation explains why our results did not show this pattern, since our results are based on an average, and not the worst case scenario.

For binary search, we expected that searching through a list of 1000 would be 1.5 times as slow as searching through a list of 100, and that searching through a list of 100 would take 2 times as long as through a list of 10. Thus, the longer the list, the smaller the differences in search time. However, we found the opposite effect: in the former case, the search was 2.51 times as slow, and in the latter it was 8.29 times as slow. This is surprising, because one would expect that average case scenarios would show smaller differences, and not bigger, and certainly not reverse.