CSE 321 Introduction to Algorithm Design Homework #3 Report

Gökçe Nur Erer 171044079

Question 1:

The algorithm designed for this question gets an input box list where the elements of the list are characters ('B' for black and 'W' for white boxes) and also gets a boolean value if the first and last element of the list should be swapped. Initially the algorithm checks if the first and last values should be swapped or not. Then depending on the check, the first and last elements of the list swaps. The value for swapping is reversed (false if true, true if false) and by getting the sublist of the current list by cutting first and last elements and sending it recursively to the algorithm sorts all the boxes alternatingly.

For the best-case n is 1 and there are 2 boxes, so no swap is required. For the average case and worst case, you need $\left\lfloor\frac{n}{2}\right\rfloor$ swaps. Which means linear time complexity for all best, worst and average cases.

PS: For this question numpy library is used because normal lists in Python gets copied when their sublists are taken so to keep the swaps changed on the original list numpy arrays are used so no copying happens during the sublist creation.

Question 2:

The algorithm designed for this question gets a input list of coin weights. (which are greater than 0). If the list has only one element, then the only coin is the fake one. If the list has two elements the lighter coin is send to the algorithm recursively. If the list has more than two elements the coin list is divided into 3 piles first 2 piles having $\left\lfloor\frac{n}{3}\right\rfloor$ coins and the third one having the rest of the coins. Then if the first two piles weight the same that means the third pile has the fake coin, so the third pile is sent to the algorithm recursively to find the fake coin in it. If the first two piles don't weight the same that means the lighter one has the fake coin in it. So, depending on which one has the least weight pile 1 or 2 is sent to the algorithm recursively.

The best case, average case and the worst case of this algorithm results $\log_3 n$ weightings.

Question 3:

<u>Average case of the insertion sort is:</u> O(n^2) since on the average case there will be (n-1)/2 comparisons in the while loop. So the total comparisons will be (n-1)/2+...+2/2+1/2 which is n(n-1)/4 which results in O(n^2) complexity.

<u>Average case of the quick sort is:</u> O(n*logn) since every list on every step can only be divided to 2. So overall the quicksort will be called $\log_2$n times and since n elements are compared in partitioning function. Overall complexity is n*logn for average case.

Swap comparisons:

For array 1,3,9,8,2,7,5 quick sort does 8 swaps and insertion sort does 9 swaps.

For array 9,8,7,6,5,4,3,2,1 quick sort does 24 swaps and insertion sort does 36 swaps.

For array 1,2,3,4,5,6,7,8,9 quick sort does 44 swaps and insertion sort does 0 swaps.

Which means for a sorted or almost sorted array quick sort is worse than insertion sort. But for average cases and reversely sorted cases quick sort performs better. So overall it would be okay to say quick sort is better than insertion sort.

From the experimental results it was decided that quick sort performed better than insertion sort in most of the cases. And theoretical analysis showed that quick sort has n*logn average time complexity and insertion sort having n^2. Which also shows quick sort is better.

Question 4:

The worst case occurs if the pivot value ends up dividing the list into very unbalanced sublists to call the partitioning (sublists only decreasing by 1) which becomes n calls for quick select and since partitioning has n operations the overall worst case complexity is $O(n^2)$.

Question 5:

The algorithm designed for this question creates all subarrays of the given array and checks the conditions and returns the subarrays that satisfies this condition. After that by traversing the satisfying subarrays, the multiplication of the subarrays are calculated and appended to a parallel array. By finding the index of the minimum multiplication result, the algorithm also finds the optimal subarray's index since the arrays are parallel and returns the index.

The worst case occurs if all the subarrays of the given list satisfies the condition. Which results in $O(2^n * n)$. Since for all subarrays the loop will execute 2^n times also to find all the multiplications of each subarray to find the optimal one we need to loop every subarray. A subarray can have at most n elements so overall the complexity would be $O(2^n * n)$.