CSE 321 Homework 3 Report

1-) The solution approach starts with the second black box and swap it with the last second box which is white. It skips the third black box because we want to such a pattern “black-white-black...”. After skipping the third one, algorithm swaps the forth box with the last forth box. The process progress until the last black box (in initial order) is swapped.

When analyze the algorithm, we can say best case, worst case and average case is same for the algorithm because the input is always given in the same format (blacks first and whites after them) so there is always one case. The algorithm decrease the size of the problem four on each step: Skip two boxes from head and tail and swap two boxes. There is one swap process for each four box. Therefore, the recurrence relation becomes:

$$T(n) = T(n-4) + 1$$

To solve the relation we can do backward substitution:

$T(n) = T(n - 4) + 1$

$T(n) = T(n – 8) + 1 + 1$

$T(n) = T(n - 12) + 1 +1 +1$ } n/4 times

…

$T(n) = T (n - n) + n/4$

So solution of the reccurrence relation becomes:

$T(n) = n / 4$ and so:

$T(n) = \theta (n/4) = \theta(n)$

2-) We can assume that the fake coin is heavier than the others. Now we can design an decrease-and-conquer algorithm to solve the problem as follows: First divide the coins into two sets with equal size and use the weighbridge to detect which set is heavier. If there is odd number of coins; pick one single coin from the remaining coins and divide rest of them into two sets. In this case; if the both set have same weight then the single coin is obviously the fake coin. Otherwise, we can ignore the coins in the light set and continue with the heavier set because k-1 coin have same weight in both set and fake coin is heavier than one regular coin. The algorithm finds the fake coin after compare the last two sets with one item.

It is obvious that the algorithm finds the solution in O(1) time if there are odd number coins and the fake coin is on the middle. There are 2^k coins

and the fake coin is the first coin in the worst case. The algorithm ignores half of the coins on each step. Therefore, algorithm does $\log(2^k)$ times sum & compare processes. Therefore, the worst-case of the algorithm is O(log(n)). The average case is mostly better than the worst case but not much to determine a better boundary then log(n) so the average case is also θ (log(n)).

3-) As we know average-case complexity of quick sort is θ(nlogn) (It is in your notes). In insertion sort, the algorithm loops for n index of the array and on each item on index k, the algorithm compares and swaps with the array items that are on from k-1 to 0 until the current item is greater than any item on index t where $0 \leq t \leq k-1$. The best case of the algorithm is O(n) obviously because the algorithm loops n times independent from the input. The number of swaps for the algorithm is strongly connected with reversed item pairs because the algorithm must be swap each item if they are not truly ordered. To analyze the average case we need to compute the numbers of swaps of all permutations of inputs. It is impossible especially if the number of input is not constant. So we need to specify expected number of swaps to calculate average-case complexity. Expected value is equal to sum of probability of occurrence for each sub-situation. In this case, the sub-situations can be item k > item k-1 and item k < item k-1. In a normal distribution, both situation occurs with ½ probabilities. Therefore, In average case we need n . E swap operations. E is equal to Σ ½ which is also equal to n/2 so the average-case becomes $\theta(n*n/2) = \theta(n^2)$.

We did 100 fold cross-validation with an array that consists 10 items and we get the average swap counts of quick sort and insertion sort are 17.9 and 23.1 respectively. The experimental results supports the theoretical results. On the other hand, insertion sort shows so much better performance than it is theoretical expectation. That means the $\theta(n^2)$ is not a tight boundary for the algorithm and maybe any better boundary can be found for the average-case.

4-) We can find the k^{th} smaller item of an array by modify to quick sort algorithm. In partition, assign an item as pivot and replace the items if they are smaller than the pivot by starting on the start index. Do the partition until last replaced item location is equal to median-1. That means, first n/2 smallest item has been replaced and the last replace item is the median item. The worst-case of the algorithm is definitely equals to quick sort because we don't change the strategy but we made a little trick to find the median. So the worst-case is $O(n^2)$.

5-) The supposed algorithm generates all subsets of the given array recursively. If a subset satisfy the sum rule than the branch stops there and returns the optimality score of the current subset. Otherwise the branch continue to branching with next item in the array until there is no item to add the current branch. If any branch cannot satisfy the rule, it returns an infinity optimality score. On each recursion level, the algorithm compares all sub solutions' optimality score and returns the best optimality score and set that has the best optimality score.

Subset count of an array is equal to 2^n so, the algorithm should generates 2^n different subset to check and compare on the worst case. On the other hand, all optimality score of subsets should be compared on each sub-level. In other words, all subsets should be compared to find minimum optimality score. The process can be considered as linear search so it can be done O(n) in the worst case. Total worst-case complexity of the solution is $O(2^n + n) = O(2^n)$.