Evan Zhang | z5383657

Question 4

4.1

The absolute difference for an:

- Odd minus odd is always an even number
- Even minus even is always an even number
- Odd minus even (or even minus odd) is always an odd number

The last one, if carried out on array A, must come in pairs. This is due to the fact that there are an equal number of odds and evens in A and in n, since A contains every integer from 1 to n. So when an odd number is paired with an even number or vice versa, there will eventually need to be a pairing of an even with an odd number. Since the last absolute difference always comes in pairs and is the only source of a .5 in the equation, we can safely conclude that S will always be an integer.

e.g. If n = 10, there are 5 odds and 5 evens in both n and in the array A. If you pair an odd from n and an even from array A, there will be 4 odds and 5 evens in n and 5 odds and 4 evens in A. You can see that even if you try to only make odd – odd or even – even pairs, you will eventually need to make an even – odd pair.

4.2

S shows the total sum of the differences between where a number should be and where a number is, as the number represents it's rightful index and the index represents it's current position. As such, we will eventually need to pay the difference to move it to it's rightful position. However, since two indexes are being swapped, you can potentially get double the value for the positions changed, as long as you aren't shifting a number away from it's position. Therefore, if you only shift numbers towards their rightful position, it will eventually cost you half the total differences between all indexes swapped, i.e. S. If you make inefficient swaps (i.e. shift numbers away from their rightful position), it will cost more than S as you are getting less than double the value for each swap.

It is always possible to find a swap with maximum value.

Consider the array [x, x, x, x, x, x]. For there to be no swaps with max value for the first element, the first element must not be equal to the index it is in (i.e. it is in the wrong place). Therefore, it will need to be between 2 and 6. However, it must be swapped with a number that it is smaller than the index it is in or 1 and that number must be somewhere between itself and it's rightful index. Say for instance there is no number that is currently between itself and it's rightful index that is smaller than the index it is in or is 1 (i.e. no max value swap for the first element). If this condition holds true, they all must exist somewhere after the rightful index of the first index's current element, which must be between 2 and 5 (Since there is nothing to the right of 6, the first index must be between 2 and 5). If we repeat this process on the second element, we find that it must be between 3 and 5. Repeating on the third gives us 4 and 5 as options and the fourth element must equal 5. Working backwards, the array must look like this [2,3,4,5,x,x]. At the fifth element, it is impossible for the conditions to hold up where no max value swaps exist, as the fifth element must either be 1 or 6, giving either a max value swap for the fourth and fifth element if 1 is chosen or a max value swap for the fifth and sixth if the six is chosen. Using the same reasoning of an array of any size, you will always find a swap with maximum value in any unsorted array.

If this is the case, you can simply find the swap with max value at each step of sorting and sort an array of any size for ‘S’ dollars.

4.3

Consider the array [3,1,2]. The current S is 1 + 0.5 + 0.5 = 2. It will cost 1 dollar to shift 3 into the 2nd index and 1 into the 1st index and then 1 more dollar to shift 3 (now in index 2) into the 3rd index and 2 into the 2nd. This is completed with total 2 dollars = S. Please refer to 4.2 for why any array can be sorted for a total cost of S dollars.

4.4

Starting with the first element in the array, check it’s value to it’s current index. This is the starting index/element. If the index is matching, mark the index as completed in a separate array and go onto the next index. If the index is not matching, then check whether the element is greater than or less than the index.

If it is greater than, inspect every element after the current element up to and including the index where it is meant to be. If any element looked at is less than the index in which the starting index is in, perform a swap. If any element is equal to the index in which the starting index is in, perform a swap and mark any completed index as completed. Start from the start on any non-completed index.

If it is less than, inspect every element before the current element up to and including the index where it is meant to be. If any element looked at is greater than the index in which the starting index is in, perform a swap. If any element is equal to the index in which the starting index is in, perform a swap and mark any completed index as completed. Start from the start on any non-completed index.

When searching, skip any index marked as completed. Checking takes O(1) time as we know the index being checked in the array.

This algorithm will eventually sort the array with the minimum possible total cost, as no swaps are made that will move an index away from where it should be, for additional reasoning, see 4.2.

This algorithm will take at most O(n+S) time. This is because throughout the algorithm, we will have inspected every element of the array to see if the element matches the index taking at most O(n) time. Then for each element that is in the wrong place, we scan up to the difference between where it is and where it should be, taking at most O(|A[i] − i|) time per starting element.
Every time something is swapped, it will take less time to find the next swap.