**COMP 301 Analysis of Algorithms HW2**

**Question 1**

1. Selection\_Sort(A):
2. **for** i=1 **to** A.length-1
3. min\_index=i
4. **for** j=i+1 **to** A.length
5. **If** A[j] < A[min\_index]:
6. min\_index=j
7. **If** i !=min\_index:
8. temp=A[i]
9. A[i]=A[min\_index]
10. A[min\_index]=temp
11. **return** A
12. There are two loop invariants
13. Inner Loop:

**Theorem**: After iteration j of the inner loop, min\_index is the index of the minimum element of A(ai,ai+1,….aj)

**Proof:**

**Initialization**: Loop invariant holds before the first loop iteration. Loop begins at j=i+1 so j=i is prior to the loop. The subarray A[a1…aj] is A[i]=A[1] in this case. Min\_index is 1 and it is the index of the minimum element of a single-element list A[a1], which shows that the loop invariant holds prior to the first iteration.

**Maintenance**: Assume prior to iteration j+1 that min\_index is the index of minimum element of A(ai…aj). During iteration j+1, there are two cases:

-Either A[j+1] <A[min\_index]: update min\_index as j+1. So after iteration j+1, min\_index is the index of smallest element in subarray A[ai…aj+1]

-Or A[j+1] >=A[min\_index]: no need to swap, min\_index is already the index of smallest element.

**Termination**: At the end of the inner loop, min\_index is the index of an element less than or equal to all elements in A[ai..aj]. Therefore, min\_index is the index of smallest element in A[ai…an].

1. Outer Loop:

**Loop Invariant:** After iteration i, the ai minimum elements of A in ascending order in position a1 to ai. In each loop, A[min\_index] is less than or equal to A[ai …aj-1]

**Proof:**

**Initialization**: Loop invariant holds before the first outer loop iteration. i=0 prior to the loop, so the sub array is empty. Therefore, empty list is sorted, and loop invariant is holds.

**Maintenance**: Assume prior to iteration i+1, i minimum elements of A[a1…. ai] are in ascending sorted order.

After inner loop, min\_index is the index of smallest element in A[ai+1…an]. We swap A[i] with A[min\_index]. After the swap, at the end of the iteration A[a1….ai] consists of the i smallest elements in position 1 to i.

**Termination**: When outer loop is completed, i is the length of A. So, we can conclude A[a1…an] has all elements in ascending sorted order.

1. Even if the list is completely ascending sorted or sorted from largest to smallest, both loops have to continue from start to finish. Therefore:

Best Case: O(n2) Worst Case: O(n2)

**Question 2**

a) No, it does not change. In normal insertion sort **while** loop turns n times and the outer loop also run n times. Therefore, it is O(n2). In while loop instead of checking element by element, if we use binary search which has the O(logn) time complexity, it becomes faster to find correct place. However, even if logn times is used to find the right place, it will still need to shift n times to insert.=> The time complexity is O(n2).

procedure binarySearch(Array, N, key)

L = 0

R = N

while L < R:

mid = (L + R)/2

if Array[mid] <= key:

L = mid + 1

else:

R = mid

return L

end procedure

procedure binaryInsertionSort(Array)

for i = 1 to length(Array) do: n times

key = Array[i]

pos = binarySearch(Array, key, 0, i-1)

j = i

while j > pos search: log(n) times but insert: n times

Array[j] = Array[j-1]

j = j-1

Array[pos] = key

end for

end procedure

b) In the above algorithm, after finding the correct position to sort, all elements are shifted and placed in the desired space. That means 0(n) time complexity to insert. But if we use a doubly linked list, it will take O(1) time to insert. It will still take O(logn) time complexity (in binary insertion sort) to find the correct position to insert. Therefore, the overall time complexity will be n\*logn= O(nlogn).

**Question 3**

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

=T(n-2)+(n-1)+n

=T(n-3)+(n-2)+(n-1)+n

…

=

=n(n+1)/2

=O(n2)