1.a.

**SELECTION-SORT**

n=A.length

for i = 0 to n

min = i // current element as min

for j = i + 1 to n // find the min element in the unsorted array

if A[i] < A[min]

min = j

temp = A[min] // swap the found the min & first element

A[min] = A[i]

A[i] = temp

1.b.

**Loop Invariant:**

🡪 At the begining of the each iteration of outer for loop, we see that the subarray A[1..i-1] has i-1 elements in sorted order. (we will be sorting in the inner for loop.)

**Initialization:**

🡪 We start with inner for loop => j = i+1. So, we have j =2. Then, subarray A[1..j-1] becomes A[1]. Array with single element is always sorted. Therefore it holds for initialization.

**Maintainance:**

🡪 For particular j, we assume that min is smallest index. For inner for loop, we have 2 possibilities:

1. *Possibility 1:*

A[j] >= A[min] 🡺 in this case, nothing will be executed.

1. *Possibility 2:*

A[j] < A[min] 🡺 min becomes j (min=j). Since the conditon is correct A[j] must be less than A[1..j-1]. So, we do swapping for maintance to sort array in increasing order.

**Termination:**

For termination of inner loop min elements becomes less than A[i..n], since j=n+1 which means we do not go into inner loop for finding the smallest elements in subarray. We observe that the subarray A[1..j-1] is in sorted order, we conclude that entire array is sorder.

**Hence, we can say that algorithm is correct.**

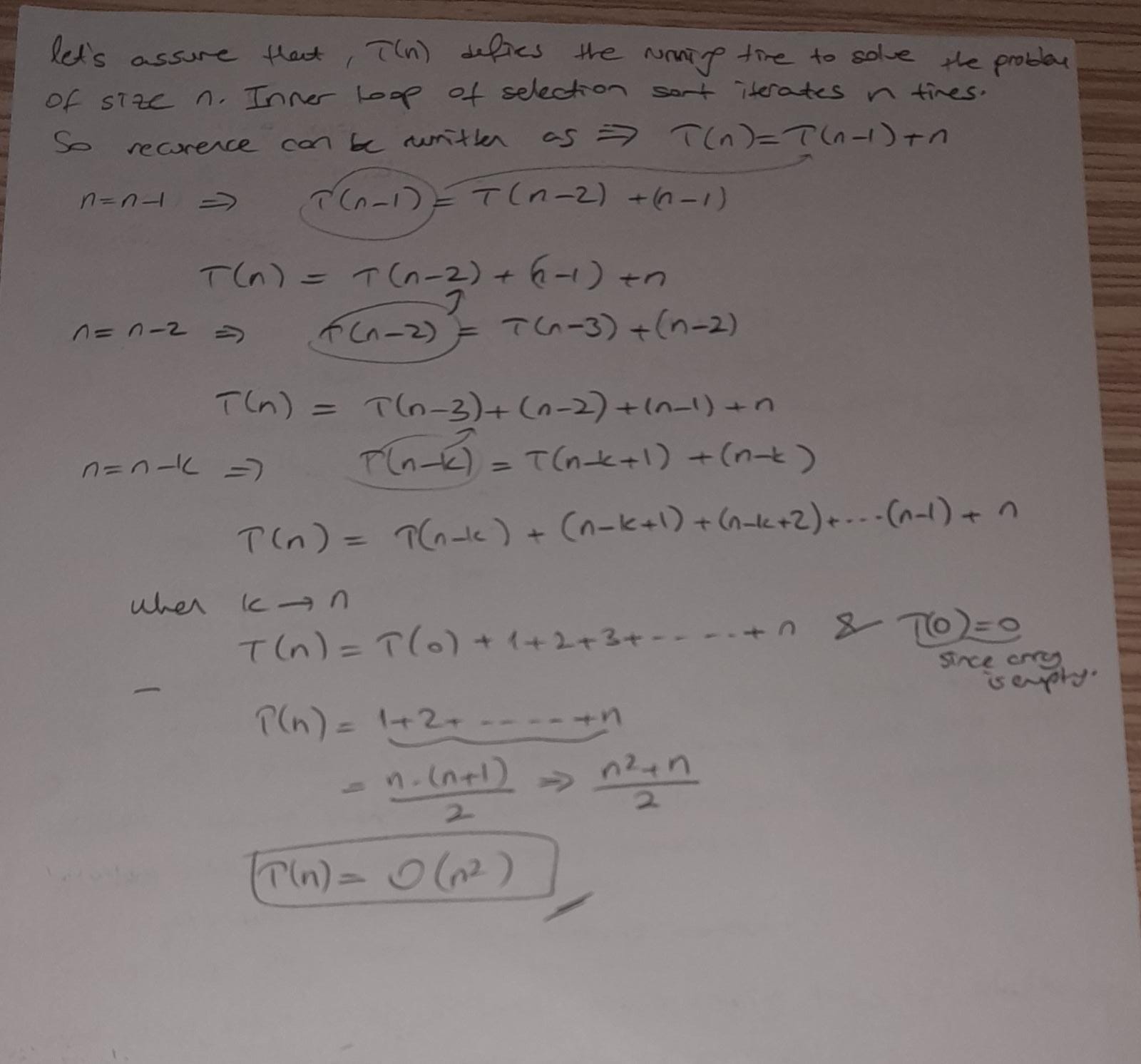
1.c.

**The best case is when array is sorted.**

**The worst case is when array is reverse-sorted.**

Selection sort compare one element with the rest of the array which means if we have n elements in the array, we will compare them with (n-1) elements. So, running time would be n\*(n-1) 🡺 Θ(n^2).

(I also included my calculations for finding the running time with recurrence.)



2.

With the while loop, we are able to make a linear search to scan in the sorted subarray to find the key and shift the elements which are bigger than key for finding the correct place for the key.

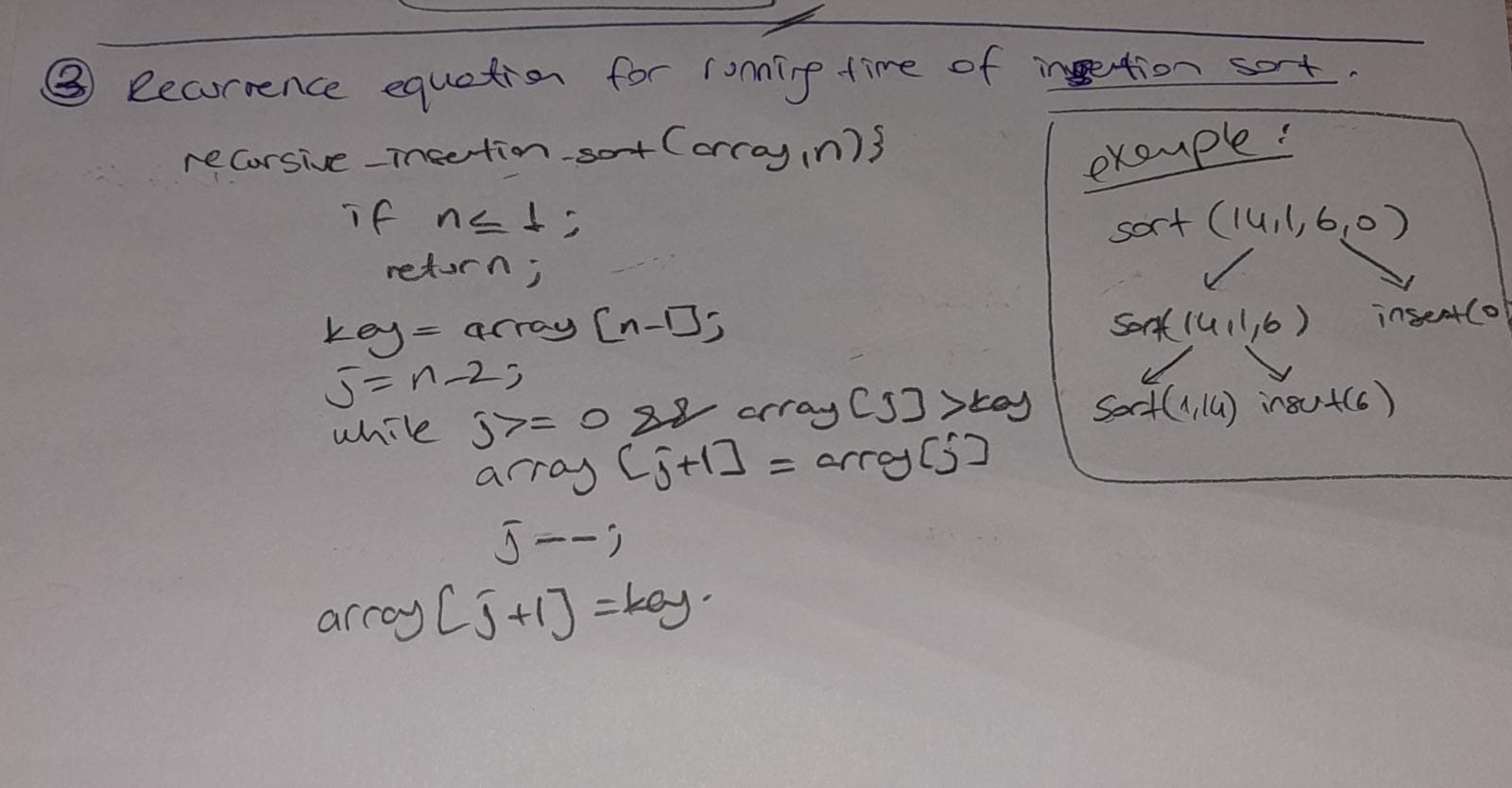
By adding this while loop, the comparisons will be less with the help of binary search. However, for finding the correct place for the key would be Θ(*n*) since we need iterate through the whole array. **So, for the insertion sort, the running time would not change from the original value of Θ(*n^2).***

**If we use doubly linked list, the running time for worst case would not change**. Algorithm of insertion with doubly linked list:

1. Create an empty sorted list.
2. Store every node in a temp variable and remove from the original list.
3. Add the current node with sort method
   1. To add node, do comparions every node with the current node. If the current node less than any node, current node will be added before that node. On the other hand, if the node is greater than every node, we will add it after all the nodes.
4. Increment the temp value
5. Change the head pointer to sorted list.

* **Since we compare each node(n) with the rest of the list(n-1), the running time for worst case would not change.**

3.



Text, letter

Description automatically generated