**2-1 Insertion sort on small arrays in merge sort**

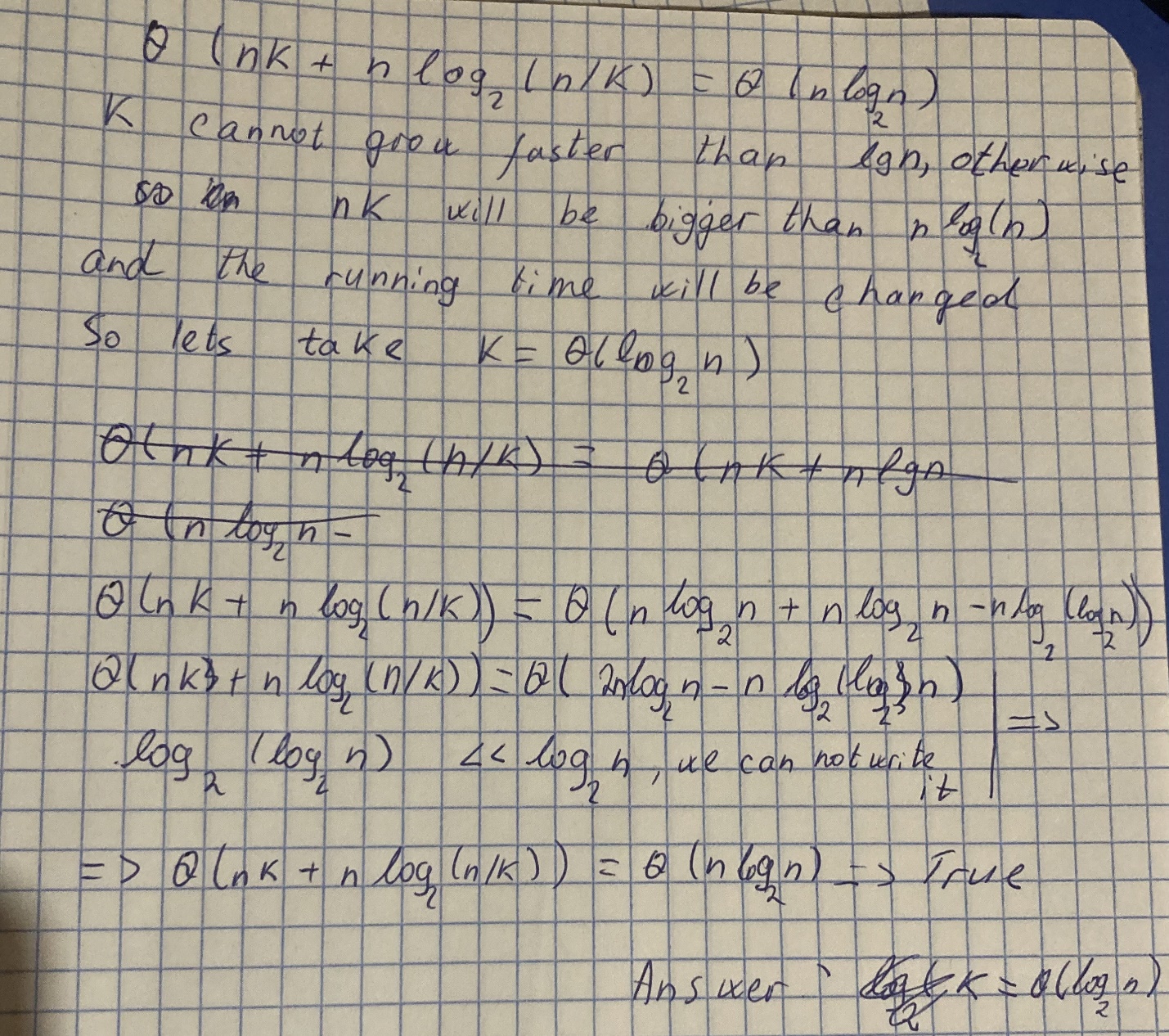
**a**. Show that insertion sort can sort the n/k sublists, each of length k, in O(nk) worst-case time.

Insertion sort can sort an array of length k in O(k2) worst-case time. So n/k sublists can sort in O(k2\*) worst-case time. Which is equal to O(nk).

**b**. Show how to merge the sublists in O(nlog2(n/k)) worst-case time.

There are n/k sublists and each time we are taking 2 of them, in order to merge. So we need to merge log2n times to get one single array. And in every merge we are comparing together n elements. That why the running time is O(nlog2(n/k)).

**c.** Given that the modified algorithm runs in O(nk + n log2(n/k)) worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of ‚O notation?



**d**. How should we choose k in practice?

k must be the largest possible size of the array, where insertion sort works faster than the merge sort.

We need to know the constant factors

**2-2 Correctness of bubblesort**

**a**. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

We need to show that the elements of A’ array are the same elements that are in the A array, and we haven’t changed their values.

**b**. State precisely a loop invariant for the for loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

Loop invariant-at the end of each jth iteration in the subarray A[j…n] the smallest element is in the first place.

In order to prove that the loop invariant is correct we need to prove these 3 properties- Initialization, maintenance, termination

Initialization- In the first step the subarray contains only the last element of the array, and it’s the smallest in the subarray (there is one element)

Maintenance-in jth step we compare jth element with its previous one, and the smaller gets the lower index(j-1), so after an iteration the first element of the subarray is the smallest one

Termination-the loop end when j=i+1, and again the first element is the smallest of the subarray

**c**. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter

The loop invariant for the for loop in lines 1–4 can be stated as follows:

At the start of each iteration of the for loop, the subarray A[1 ... i - 1] consists of the elements that are smaller than the elements in the subarray A[i … n] in sorted order.

And here is how the three necessary properties hold for the loop invariant:

**Initialization:** Initially the subarray A[1…i−1] is empty and trivially this is the smallest element of the subarray.

**Maintenance:** From part **(b)**, after the execution of the inner loop, A[i] will be the smallest element of the subarray A[i…n]. And in the beginning of the outer loop, A[1…i−1] consists of elements that are smaller than the elements of A[i…n], in sorted order. So, after the execution of the outer loop, subarray A[1…i] will consists of elements that are smaller than the elements of A[i+1…n], in sorted order.

**Termination:** The loop terminates when i=A.length. At that point the array A[1…n] will consists of all elements in sorted order.

**d**. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Worst-case of bubble sort is the reverse sorted array, it will go through the whole array for each element. Running time is O(n2). Insertion sorts running time is also O(n2) but the difference is the c constant factor, which is bigger in bubble sort, because in bubble sort we are doing much more swaps than in insertion sort.

**2.3 Horner’s rule**

**a.** In terms of O notation, what is the running time of this code fragment for Horner’s rule?

There is **only one loop**, which runs n+1 times. This means that the running time of this code is O(n)

**2.4 inversions**

If i<j the A[i]>A[j] => (i,j) is inversion of A

**a.** List the five inversions of the array [2,3,8,6,1]

2>1 - (1,5)

3>1 – (2,5)

8>6 – (3,4)

8>1 – (3,5)

6>1 – (4,5)

The answer is (1,5), (2,5), (3,5), (4,5), (3,4)

**b.** What array with elements from the set {1,2…n} has the most inversions? How many does it have?

The most inversions has the reverse sorted array ([n, n-1, …1]).

The amount of inversion that n makes is n-1. (n-1, n-2, …1)

The amount of inversion that n-1 makes is n-2. (n-2, n-3, …1)

…

The amount of inversions that 1 makes is 0.

If we add all these inversions, we will get (n-1) + (n-2) +...+2+1+0==

**c.** What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

In the array is divided into two subarrays (sorted and not sorted), and every time we take an element from the not sorted subarray, and try to find it’s right place in already sorted array with going through while loop. And the further away the element from its right position, the longer will the while loop run. And on the other hand the further away the element from its right position, the more inversion it makes with the elements that are both before it and greater that it.

So the more inversions there are in the array, the more time while loop will run. In conclusion the motre inversions there are, the longer insertion sort will last

**d**. Give an algorithm that determines the number of inversions in any permutation on n elements in O(n log2n)worst-case time. (Hint: Modify merge sort.)

COUNT-INVERSIONS(A,p,r)

if p<r

q= ⌊(p+r)/2⌋

return COUNT-INVERSIONS(A,p,q)+ COUNT-INVERSIONS(A,q+1,r)+INVERSIONS(A,p,q,r)

return 0

INVERSIONS(A,p,q,r)

n1=q-p+1

n2=r-q

let L[1.. n1] and R[1.. n2] be new arrays

for i=1 to n1

L[i]=A[p+i-1]

for j=1 to n2

R[j]=A[q+j]

L[n1+1]=∞

R[n2+1]=∞

i=1

j=1

inversions=0

for k=p to r

if L[i]<=R[j]

i=i+1

else

inversions=inversions+1

j=j+1

return inversions

As in merge sort we need to divide the array into smaller subarrays. Now we need to count inversions in each subarray than add one to another. So we can just modify the merge sort. We can change the name of the MERGE-SORT function, and call it INVERSIONS-COUNT. Here we need to count inversions of two subarrays, and also there can be inversions between elements of different subarrays. And in the end we're adding these three elements to each other and return that value․

In order to count the inversions between elements of different subarrays we will have another function, which is called INVERSIONS. It is the modified version of MERGE function.

The difference between INVERSION and MERGE functions is that in INVERSION we create a new variable called “inversions” and initialize it with 0. Each time we find an inversion we are going to increase the value of the variable by one․ And the inversions we find in for loop.