CISC2006: Algorithm Design and Analysis
Homework 2 **Due Date:** 23:59, 03 March 2023

Rules: Discussion of the problems is permitted, but writing the assignment together is not (i.e. you are not allowed to see the actual pages of another student). You can get at most 100 points if attempting all problems. Please make your answers precise and concise.

1. (15 pts) Given two sorted arrays A and B, design a linear ($O(|A|+|B|)$) time algorithm for computing the set C containing elements that are in A or B, but not in both. That is, $C = (A \cup B) \setminus (A \cap B)$. You can assume that elements in A have different values and elements in B also have different values. Please state the steps of your algorithm clearly and analyze its running time.

want linear time algorithm.

input: sorted array A, B.

idea: have 2 pointers, ∵ it's sorted, so can start compare from first element, if same element, then move to next element. if different, store in array C.

```
program(A, B):
    size_A = size(A)
    size_B = size(B)
    A[size_A + 1] = 99999
    B[size_B + 1] = 99999
    i = 1, j = 1, k = 1
    set C to be empty.

    while i ≤ size_A or j ≤ size_B:
        if A[i] < B[j]:
            C_k = A[i]
            k = k + 1
            i = i + 1
        elif A[i] > B[j]:
            C_k = B[j]
            k = k + 1
            j = j + 1
```

```
elif A[i] = B[j] :
        i = i+1
        j = j+1

output : C array.
```

Analyse running time as O(|A|+|B|). :

So the major code contribute to running time is the "while" loop. Because for the each "if-else" statement, it just running in constant time. Therefore, in "while" loop, maximum running time is the size of A + size of B, which is $|A| + |B|$, $O(|A| + |B|)$ #.

$i<j$, but $a_i > a_j$

inversion : a_i , a_j

$l > r$.

2. (20 pts) Given a sequence of numbers A, design an algorithm that counts the number of inversions, where an inversion is a pair (a_i, a_j) such that $i < j$ and $a_i > a_j$. Please state the steps of your algorithm clearly and analyze its running time.

   (a) (10 pts) Given two sorted arrays L and R, design a linear $(O(|L| + |R|))$ time algorithm that counts the number of pairs (l, r) such that $l \in L$, $r \in R$ and $l > r$.

$L = \{l_1, l_2, \ldots, l_m\}$

$R = \{r_1, r_2, \ldots, r_n\}$

```
Count_Pair (L, R)
    i=1, j=1, count = 0
    while i ≤ m :
        if L[i] > R[j] :
            count = count + (m - i)
            j = j+1
        elif L[i] ≤ R[j]
            i = i+1
    return count #.
```

the main runtime is on while loop, and it run at most $i+j \leq n+m$ times

$O(n+m) = O(|L| + |R|)$.

   (b) (10 pts) Suppose we have a linear time algorithm for question (a), design a $O(n \log n)$ time algorithm that computes the number of inversion in A.

here we can use divide and conquer

$A = \{a_1, a_2, \ldots, a_n\}$ not sorted.

idea: divide array A into left L and right R, and recursively do it, until base case where only 1 element in array.

When merging, use algorithm in (a), to count inversion and fix the inversion. So it recursively on left & right, and can get the total number of inversion of A and a sorted array A^*.

running time : it's basically merge sort, but add algorithm (a)

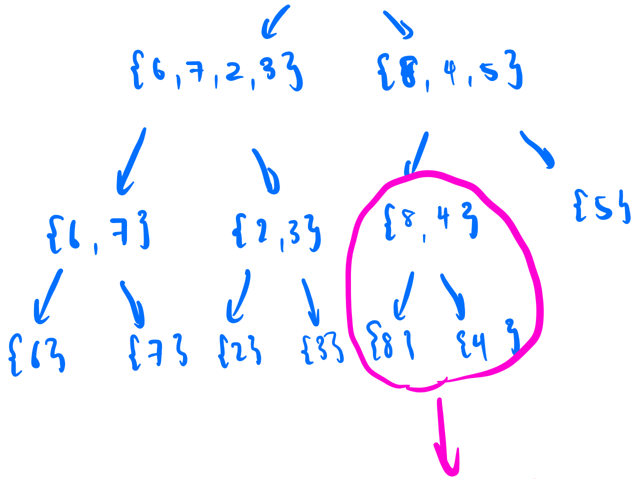
$\therefore T(n) = O(n \log n) + O(n) = O(n \log n)$ #

↑

complexity of merge sort

example.

(example): A = {6,7,2,3,8,4,5}.

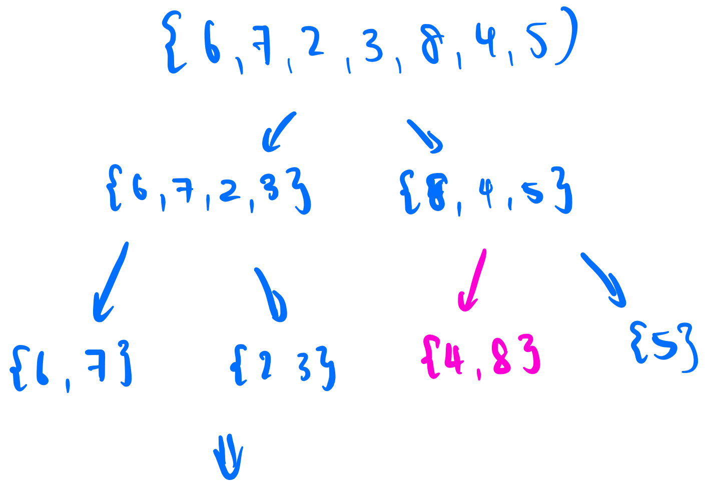


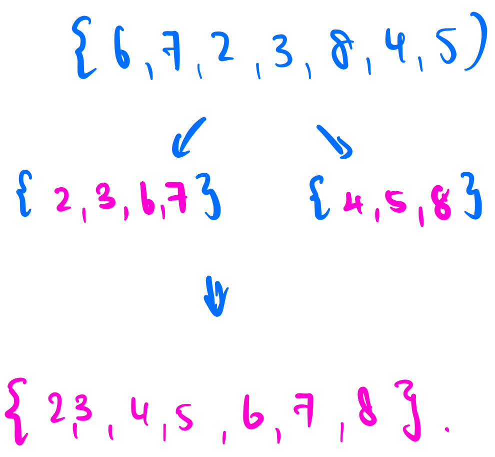
(divide into base case)

When merging, use algorithm (a),

∴ count +1,

and fix the order after merge.





3. (15 pts) Suppose we have $T(n) \leq c = O(1)$ for all $n \leq 3$, and for every $n \geq 4$, we have

$$T(n) \leq T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + T\left(\left\lfloor \frac{3n}{4} \right\rfloor\right) + c \cdot n.$$

lemma:

Use Mathematical Induction to prove that $T(n) = O(n \log n)$ for all $n \geq 4$.

want to prove $T(n) \leq c\, n \log n$.

Base Case : for $n = 1, 2, 3$:

$T(n) \leq c \leq 2cn\log n = O(n \log n)$

for $n = 4$:

$T(4) \leq T(1) + T(3) + 4n \leq 6c \leq 2c\, n\log n$.

$= O(n \log n)$

Induction :

(Induction hypothesis) for $k \leq n-1$, $T(k) = O(k \log k)$

$= 2c\, k \log k$.

∴ consider case of n,

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + cn$$

$$\leq c\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) + c\left(\frac{3n}{4}\right)\log\left(\frac{3n}{4}\right) + cn$$

$$\leq c\left(\frac{n}{4}\right)\log n - \frac{cn}{2} + c\left(\frac{3n}{4}\right)\log(3n) - \frac{3cn}{2} + cn$$

$$\leq c\left(\frac{n}{4}\right)\log n + c\left(\frac{3n}{4}\right)\log(3n) - cn.$$

$$\leq c\left(\frac{n}{4}\right)\log n + c\left(\frac{3n}{4}\right)\log n + \frac{3cn}{4}(\log 3) - cn$$

$$\leq c\left(\frac{n}{4} + \frac{3n}{4}\right)\log n + \frac{3cn}{4}(\log 3) - cn$$

$$\leq cn\log n + cn\left(\frac{3\log 3}{4} - 1\right).$$

$\leq 2cn \log n$

$T(n) = O(n \log n)$ # proved.

4. (10 pts) Given an array $A = \{a_1, a_2, \ldots, a_n\}$ of n integers in the range $[0, n^2-1]$, design an algorithm for sorting A in $O(n)$ time. Please state the steps of your algorithm clearly and analyze its running time.

Idea: use multiple times O(n) algorithm to do the work.

(dont know how to express my idea, I will use example)

(example) $A = \{a_1, a_2, \ldots, a_{10}\}$, range = [0, 99].

First thing to do is create 50 array to store as below:

[0,1] [2,3] [4,5] [6,7] [8,9]

[10,11] [12,13]. ....

⋮

[90,91] .... [98,99]

for example, if element 1, will store in 1st array [0,1]

if element 12, will store in 7th array [12,13].

∴ after create 50 arrays, linear scan A and place it's elements into corresponding array. This linear scaning will take O(n) time.

---

since A only have 10 elements, ∴ at most 10 arrays have element. others are empty.

∴ linear scan all array, remove the array that are empty. Left array (at most 10) that have element, here take O(n) time now we have 10 arrays from left to right.

So we just output from left to right, but for each array, output the smaller value first by using simple "if else" statement.
(won't increase running time)

Again, here takes $O(n)$ time.

$\therefore$ $T(n) = O(n) + O(n) + O(n) = O(n)$ #.

5. (20 pts) You are given n numbers $a_1, a_2, \ldots, a_n$. It takes constant time to check whether two numbers a_i and a_j are of the same value. The goal is to check whether more than half of the numbers have the same value. Please state the steps of your algorithm clearly, prove that it is correct, and analyze its running time.

(1) (10 pts) Design an $O(n \log n)$ time algorithm to solve the problem.

idea : use $O(n\log n)$ to sort the number first.
initialize a counter to 1 , when scaning from left to right,
if element same with last one , counter +1 , else, reset to 1
At the same time, record counter maximum value. as max_count
When done scan all, if max_count > $n/2$, then output " Yes".

running time : $O(n\log n) + \underset{\text{scanning}}{O(n)} = O(n\log n)$. #.

(2) (10 pts) Design an $O(n)$ time algorithm to solve the problem. (Hint: Show that using linear time, the problem size can be reduced by at least half.)

idea : randomly choose k element from numbers. Since k is finite ,
and we are able to do sorting in $O(n)$ for finite element (by google)
Then, we choose the median in this k element as pivot.
choosing pivot can be achieve in $O(n)$ if in a sorted array.
Lastly, compare this pivot to all element , which take $O(n)$,
to check whether it exceed $n/2$. If yes, output "Yes".

runtime : $O(n) + O(n) + O(n) = O(n)$.

but this method doesn't guarantee $O(n)$ run time always.

#.

6. (40 pts) **Comparison of Sorting Algorithms**.

In this problem you need to implement the different sorting algorithms, and compare their running times on different inputs. Implement each of the following algorithms as a function that takes as input an array (which can be very long), and outputs the sorted version of the array (from minimum to maximum).

- InsertionSort : based on Insertion-Sort from the lecture notes;
- BubbleSort : based on Bubble-Sort from the lecture notes;
- SelectionSort : based on Selection-Sort from the lecture notes;
- HeapSort : use heap implementation of priority queue for sorting;
- MergeSort : based on Merge-Sort from the lecture notes;
- QuickSort : use median of three random elements as the pivot.

In the main function, we read an array $A = \{a_1, a_2, \ldots, a_n\}$ of different integers from a file, and use different sorting algorithm to do sorting. For each algorithm, test whether the returned array is sorted or not, and output its running time.

Several test cases of array A will be provided.