Homework1

Zuyao Chen 201728008629002

1 Question1

a). **algorithm description:**

Let "query(X, k)" denotes the k^{th} smallest number of A or B and we call two databases $A = \{a_1, a_2, ..., a_i, ..., a_n\}, B = \{b_1, b_2, ..., b_j, ..., b_n\}$ arranged in ascending order (In fact,it does no matter to care the sequence order owing to "query").

query(A, i) can be written as a_i,as well as query(B, j). Making sure $i + j = n$,

- if $a_i < b_j$, the median lies in $\{a_{i+1}, ..., a_n\} \bigcup \{b_1, b_2, ..., b_j\}$;
- if $a_i = b_j$, the median is a_i or b_j;
- if $a_i > b_j$, the median lies in $\{a_1, a_2, ..., a_i\} \bigcup \{b_{j+1}, ..., b_n\}$

We initialize $i = n/2, j = n-i$,that equals to comparing the median of each database. Then the search area can be narrowed down to half length of the last until we just need to find 1^{th} smallest number between two separate arrays.

pseudo-code:

Algorithm 1 finding the median of two separate databases via query

Input: Two separate databases A,B,length n. (initializing $i, j = 0, k = n$)

Output: the median(the n^{th} smallest) of $A \bigcup B$

function FIND_KTH(A, i, B, j, k)
    if $k = 1$ **then**
        return $\min\{\text{query}(A, i+1), \text{query}(B, j+1)\}$
    end if
    if $i = 0$ (initial) **then**
        $i \leftarrow k/2, j \leftarrow k - i$
    end if
    if $\text{query}(A, i) < \text{query}(B, j)$ **then**
        $k \leftarrow k - k/2$ (each discards $k/2$ numbers)
        $i \leftarrow i + k/2, j \leftarrow j - k/2$
        if $k = 1$ **then**
            $j \leftarrow j - 1$
        end if
        return FIND_KTH(A, i, B, j, k)
    else if $\text{query}(A, i) > \text{query}(B, j)$ **then**
        $k \leftarrow k - k/2, i \leftarrow i - k/2, j \leftarrow j + k/2$
        if $k = 1$ **then**
            $i \leftarrow i - 1$
        end if
        return FIND_KTH(A, i, B, j, k)
    else
        return $\text{query}(A, i)$
    end if
end function

b). **subproblem reduction graph**

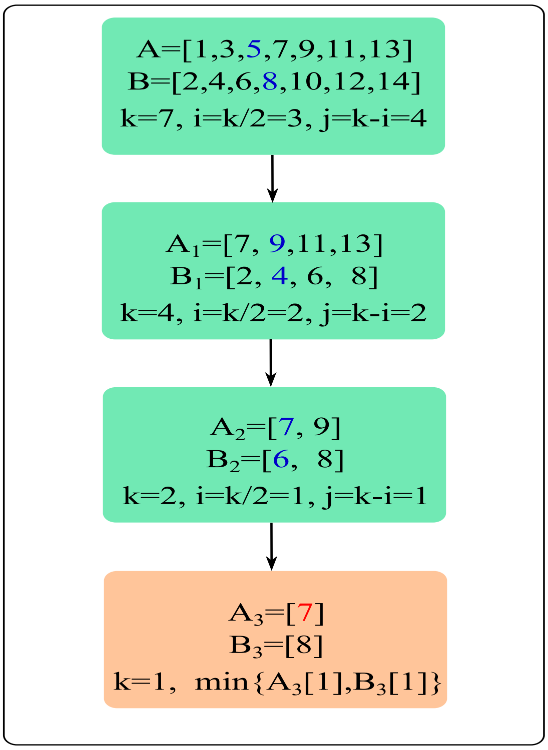

Figure 1: problem instance

c). **proof of the correctness**
Obviously, $k = 1$ means that we just want the 1^{th} smallest number of $A \bigcup B$. In order to find the median of $A \bigcup B$,we initialize $i = n/2, j = n - i, k = n$. Let "$A[i]$" denotes "query$(A, i)$", then compare $A[i]$ with $B[j]$:

i. if $A[i] < B[j]$, we can surely say that $\{A[1], A[2], ..., A[i]\}$ must lies in the left of the median and $\{B[j + 1], ..., B[n]\}$ must lies in the right of the median.For instance, if $B[j + 1]$ is the median,then there has $i + j = n$ numbers smaller than $B[j + 1]$ (each element of $\{A[1], ..., A[i]\} \cup \{B[1], B[2], ..., B[j]\}$ is smaller than $B[j + 1]$).

ii. if $A[i] = B[j]$, the median is $A[i]$ (or $B[j]$).

iii. if $A[i] > B[j]$,it's the opposite of i.

in each iteration, we discard $k/2$ numbers until $k = 1$.

d). **complexity of the algorithm**
The size of original problem is reduced to half at each iteration, and "query" costs $O(1)$,thus

$$T(n) = T(n/2) + cO(1) = O(\log n)$$

2 Question2

a). **algorithm description:**

- first,we randomly choose v from the array A;

- second,we split A into three categories : elements greater than v, those equal to v,and those smaller than v.Call these A_L, A_v, A_R respectively.
- then,we have

$$\text{select}(A,k) = \begin{cases} \text{select}(A_L, k) & \text{if} \quad k \leq \text{len}(A_L) \\ v & \text{if} \quad \text{len}(A_L) < k \leq \text{len}(A_L) + \text{len}(A_v) \\ \text{select}(A_R, k - \text{len}(A_L) - \text{len}(A_v)) & \text{if} \quad k > \text{len}(A_L) + \text{len}(A_v) \end{cases}$$

here "len" represents the length of an array.

pseudo-code:

Algorithm 2 find the k^{th} largest element in an unsorted array

Input: An unsorted array A and k
Output: the k^{th} largest number of A

function SELECT(A, k)
  if $k \leq 0$ or $k > \text{len}(A)$ **then**
    return error
  end if
  randomly choose v of A
  $A_L = \{\}, A_v = \{\}, A_R = \{\}$
  for $i = 1$ to len(A) **do**
    if $A[i] > v$ **then**
      $A_L = A_L \bigcup \{A[i]\}$
    else if $A[i] = v$ **then**
      $A_v = A_v \bigcup \{A[i]\}$
    else
      $A_R = A_R \bigcup \{A[i]\}$
    end if
  end for
  if $k \leq \text{len}(A_L)$ **then**
    return SELECT(A_L, k)
  else if $k \leq \text{len}(A_L) + \text{len}(A_v)$ **then**
    return v
  else
    return SELECT($A_R, k - \text{len}(A_L) - \text{len}(A_v)$)
  end if
end function

b). **subproblem reduction graph**

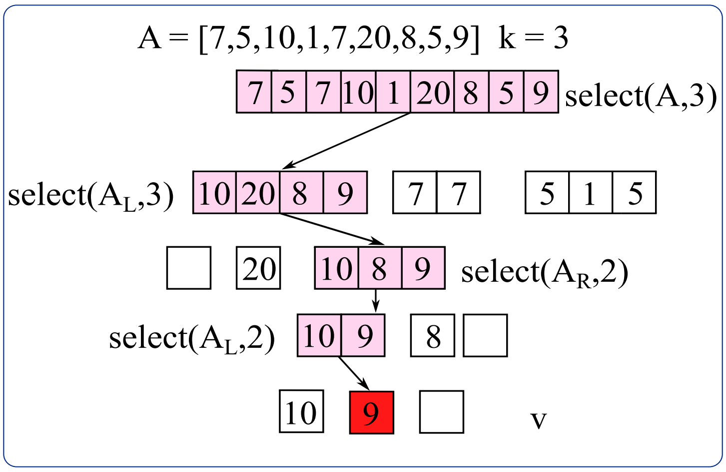

Figure 2: problem instance

c). **proof of the correctness**
In terms of the input constraints, it should throw an exception given $k \leq 0$ or $k > \text{len}(A)$. What we want is finding the k^{th} largest number of array A,there is no need to sort the array. In every recursion, the search can be narrowed down to one of three sublists until we choose the correct one of singletons.

d). **complexity of the algorithm**
Splitting A into three parts costs linear time.

- The most worst situation is that we choose the smallest number every times, then it would force our algorithm to perform

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

  or $O(n^2)$ operations.

- The best-case scenario is that we select the median at each iteration, thus it would perform

$$T(n) = T(n/2) + O(n)$$

  or $O(n)$ operations.

- good choice: select a nearly-central element , $\text{len}(A_L) \geq \epsilon\text{len}(A)$,$\text{len}(A_R) \geq \epsilon\text{len}(A)$ for a fixed $0 < \epsilon < 1$,

$$\begin{aligned} T(n) \leq & T((1-\epsilon)\text{len}(A)) + O(n) \\ \leq & cn + c(1-\epsilon)n + c(1-\epsilon)^2 n + \ldots \\ = & O(n) \end{aligned}$$

3 local minimum search

start search from the root of tree T,

- if *root* has no children, then return *root*

- if $left < root$, then $root = left$, continue search
- else if $right < root$, then $root = right$, continue search
- else return $root$

the worst condition is that searching from the root to the leaf , which costs $O(\log n)$

4 Divide and Conquer

Question:
Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.)
We use some of the terminology of problem 3. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)
Solution:

-

5 Question5

well,it's a second bracketing problem and the answer is a **Catalan number**. I don't know how to analysis it using the method of divide and conquer, so I just put up the implementation of the math formula:

$$\mathrm{tri}(n) = \mathrm{tri}(2) * \mathrm{tri}(n-1) + \mathrm{tri}(3) * \mathrm{tri}(n-2) + ... + \mathrm{tri}(n-1) * \mathrm{tri}(2)$$

where $\mathrm{tri}(2) = \mathrm{tri}(3) = 1$,$\mathrm{tri}(n)$ represents the number of triangulations of a convex polygon with n vertices. the test result is showed below:

n	tri(n)
3	1
4	2
5	5
6	14
7	42
8	132

n	tri(n)
9	429
10	1430
11	486
12	16796
13	58786
14	208012

Table 1: test result

Obviously,the complexity of the algorithm is $O(n^2)$.

6 special inversion counting

Question:
Recall the problem of finding the number of inversions. As in the course, we are given a sequence of n numbers $a_1, \cdots, a_n$, which we assume are all distinct, and we difine an inversion to be a pair $i < j$ such that $a_i > a_j$. We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair *a significant inversion* if $i < j$ and $a_i > 3a_j$. Given an $O(nlogn)$ algorithm to count the number of significant inversions between two orderings.
Solution:

a.) **Algorithm description:**
Unlike the trivial counting inversion problem, there is a point that how to count the significant inversion, it must be done with $O(n)$. pseudo-code:

Algorithm 3 counting the special inversion

Input: An unsorted sequence $a_1, a_2, \ldots, a_n$
Output: the inversion number

```
function MERGE(L = {A_1, A_2, ··· , A_m}, R = {B_1, B_2, ··· , B_n})
    inv_num = 0, i = 1, j = 1
    while i < m and j < n do
        if A_i > 3 * B_j then
            inv_num = inv_num + m + 1 − i
            j = j + 1
        else
            i = i + 1
        end if
    end while
    C = {}, i = 1, j = 1
    for k = 1 to m + n do
        if A_i <= B_j then
            append(C, A_i)
        else
            append(C, B_j)
        end if
    end for
    return (inv_num, C)
end function
function MERGE-COUNT(A = {a_1, a_2, ··· , a_n})
    if n = 1 then
        return (0, A)
    end if
    split the sequence into two halves L, R
    (r_L , L) = MERGE-COUNT(L)
    (r_R , R) = MERGE-COUNT(R)
    (r_LR, L′) = MERGE(L, R)
    return (r_L + r_R + r_LR, L′)
end function
```

b.) **correctness**

- correctness of "merge": since L , R are sorted sequence in ascending order, we scan these two lists. If $A_i > 3B_j$ then $A_i, A_{i+1}, \cdots A_m > 3B_j$, there is no need to continue scanning L for B_j, then move to B_{j+1}, as $B_{j+1} > B_j, \forall k < i, A_k \leq 3B_j$, thus $A_k \leq 3B_{j+1}$, we just need to scan L from last end point A_i , once we reach each end point of L,R, we get the inversion number of two sorted lists
- for an unsorted sequence A , we can split it into two halves L, R, each part can be reduced into two halves iteratively , then we combine the results of two halves to get the final result.

c.) **time complexity**
"merge" manipulation costs $O(n)$, original problem is divided into two subproblems, thus

$$T(n) = 2T(n/2) + O(n)$$

$T(n) = O(n \log n)$

7 local maximum in Array

Question: Given an input array $num[0, 1, \cdots, n-1]$ where $num[i] \neq num[i+1]$ for all $i = 0, 1, \cdots, n-2$, suppose $num[-1] = num[n] = -\infty$ .find one local maximum and report its position, for instance $[1, 2, 3, 2, 1]$, 3 is the maximum element and report the index number 2.give an algorithm with $O(\log n)$ complexity.
Solution:

Algorithm 4 find the local maximum in Array

Input: An array $num[0, 1, \cdots, n-1]$
Output: the position of one local maximum

```
function LOCAL-MAX(nums, start , end)
    if (end − start) <= 1 then
        if nums[start] > nums[end] then
            return start
        else
            return end
        end if
    end if
    mid = (start + end)/2
    if arr[mid] > arr[mid − 1] and arr[mid] > arr[mid + 1] then
        return mid
    else if arr[mid] < arr[mid − 1] then
        return LOCAL-MAX( nums, start, mid − 1)
    else
        return LOCAL-MAX(nums, mid + 1, end)
    end if
end function
function SOLUTION(num[0, 1, · · · , n − 1])
    return LOCAL-MAX(num, 0, n − 1)
end function
```

time complexity:

$$T(n) = T(n/2) + c$$

$T(n) = O(\log n)$