Solution for Problem Set 5

201300035 方盛俊

Problem 1

(a)

Overview:

QuickSelect(A, i) will divide an array to two parts via the i-th smallest element in time $O(n)$. We can use QuickSelect(A, i) to k-sort an arbitrary array in $O(n \log k)$ time. If $n/2$ is even, select median and partition it by QuickSelect(A, n/2). If $n/2$ is odd, remove a small part by QuickSelect(A, n/k) and then it will be same with case that is even. Each times we use $O(n)$ time and it will be $\log k$ times, so the final time complexity is $O(n \log k)$.

Algorithm:

Algorithm 1 K-Sort

```
function K-SORT(A[1...n], k)
    if n == k then
        return
    end if
    if n/2 is even then
        QuickSelect(A[1...n], n/2)
        K-Sort(A[1...n/2], k)
        K-Sort(A[n/2+1...n], k)
    end if
    if n/2 is odd then
        QuickSelect(A[1...n], n/k)
        QuickSelect(A[n/k+1...n], (n-n/k+1)/2)
        K-Sort(A[n/k+1...(n+n/k)/2], k)
        K-Sort(A[(n+n/k)/2+1...n], k)
    end if
end function
```

Correctness:

After the first call, the array will be divided into two or three part, and $A[1...n/2] \prec A[n/2+1...n]$, or $A[1...n/k] \prec A[n/k+1...(n+n/k)/2] \prec A[(n+n/k)/2+1...n]$.

After several times of this process, the length of smallest part will be k and $A[1...n/k] \prec A[n/k+1...2n/k] \prec ... \prec A[n-n/k+1...n]$. It is k-sorted.

Time Complexity:

When $n = k$, $T(n) = \Theta(1)$.

When n is even, $T(n) = 2T(\frac{n}{2}) + O(n)$.

When n is odd, $T(n) = 2T(\frac{n}{2} - \frac{2n}{k}) + O(n) \leqslant T(\frac{n}{k}) + 2T(\frac{n}{2} - \frac{2n}{k}) + O(n)$.

So by the recursive tree method we can know that the final time complexity is $T(n) = O(n \log k)$.

(b)

The are k blocks and total size is n, so there are $\frac{n!}{((n/k)!)^k}$ different possible permutations for the k blocks.

Each comparision as a question can divide it into two parts, so it is like a binary tree. The thing we need to do is find the height of the tree.

$$\log(\frac{n!}{((n/k)!)^k}) = \log(n) + \log(n-1) + \cdots + \log(2) - k(\log(\frac{n}{k}) + \log(\frac{n}{k} - 1) + \cdots + \log 2) = \Omega(n \log n - k \cdot \frac{n}{k} \log \frac{n}{k}) = \Omega(n \log k)$$

So any comparision-based k-sorted algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

Problem 2

(a)

We can think that we are putting coins into two boxes one by one.

$$p = \frac{1}{2} \cdot \frac{1}{2} + (1 - \frac{1}{2})(1 - \frac{1}{2}) = \frac{1}{2}$$

(b)

Overview:

At the beginning, we randomly choose $n/2$ of n coins to put on one pan, and the remaining $n/2$ coins on the other pan. The are two cases. if it is unbalanced, we find the fake coins by `FindTheLighter` and `FindTheHeavier` function. If it is balanced, we recall the function recursively in the two parts. Finally we will get the fake coins.

Algorithm:

Algorithm 2 FindFakeCoins

```
function FindTheLighter(A[1...n])
    if n == 1 then
        return A[1]
    end if
    if n == 2 then
        lighter = Balance(A[1], A[2])
        return lighter
    end if
    lighter group = Balance(one part of A randomly, other part of A)
    return FindTheLighter(lighter group)
end function
function FindTheHeavier(A[1...n])
    if n == 1 then
        return A[1]
    end if
    if n == 2 then
        heavier = Balance(A[1], A[2])
        return heavier
    end if
    heavier group = Balance(one part of A randomly, other part of A)
    return FindTheHeavier(heavier group)
end function
function FindFakeCoins(A[1...n])
    let B = one part of A randomly, C = other part of A
    is it balanced, lighter group, heavier group = Balance(B, C)
    if it is balanced then
        isFound, lighter, heavier = FindFakeCoins(B)
        isFound, lighter, heavier = FindFakeCoins(C)
        return is it found, lighter, heavier
    else
        return true, FindTheLighter(lighter group), FindTheHeavier(heavier group)
    end if
end function
```

The times of using Balance in FindTheLighter(A[1...n]) or FindTheHeavier(A[1...n]) is $\log n$.

The times of using Balance in FindFakeCoins(A[1...n]) where there are no fake coins is $\sum_{k=1}^{\log n} 2^{k-1} = n - 1$.

Let $E(n)$ be the expected number of times my algorithm uses the Balance.

So, $E(n) = \frac{1}{2} \cdot 2 \cdot \log \frac{n}{2} + \frac{1}{2} \cdot ((\frac{n}{2} - 1) + E(\frac{n}{2})) = \log n + \frac{n}{4} - \frac{3}{2} + \frac{1}{2}E(\frac{n}{2})$ and $E(2) = 1$

Let $n = 2^k$, and then

$$E(n) = E(2^k) = k + 2^{k-2} - \frac{3}{2} + \frac{1}{2}E(2^{k-1}) = \sum_{k=1}^{\log n} \frac{2^k}{2^{\log n}} \cdot (k + 2^{k-2} - \frac{3}{2}) = \frac{n}{3} + \frac{14}{3n} + 2\log n - 5$$

Problem 3

(a)

Modify the merge sort algorithm. Let S be A and W be B.

Algorithm 3 MergeSort

```
function MERGE(leftA, rightA, leftB, rightB)
    m = length of leftA
    m' = length of rightA
    solA[1...(m+m')], solB[1...(m+m')]
    i = 1, j = 1, k = 1
    while i <= m + m' do
        if k > m' or j <= m and leftA[j] * leftB[j] <= rightA[k] * rightB[k] then
            solA[i] = leftA[j]
            solB[i] = leftB[j]
            j = j + 1
        else
            solA[i] = rightA[k]
            solB[i] = rightB[k]
            k = k + 1
        end if
        i = i + 1
    end while
    return solA[1...(m+m')], solB[1...(m+m')]
end function
function MERGESORT(A, B)
    if n == 1 then
        solA[1...n] = A[1...n]
        solB[1...n] = B[1...n]
    else
        leftSolA[1...(n/2)], leftSolB[1...(n/2)] = MergeSort(A[1...(n/2)], B[1...(n/2)])
        rightSolA[1...(n/2)], rightSolB[1...(n/2)] = MergeSort(A[(n/2+1)...n], B
        [(n/2+1)...n])
```

solA[1...n] = Merge(leftSolA[1...(n/2)], rightSolA[1...(n/2)], leftSolB[1...(n/2)], rightSolB[1...(n/2)])
end if
return solA[1...n], solB[1...n]
end function
function MagicalMean(A, B)
A', B' = MergeSort(A, B)
return $A'[n/2]$
end function

Correctness:

Base: $A[n]$ or $B[n]$ is only one element and it is sorted.

I.H: After merge sort, $A[1...(n/2)]$ and $A[(n/2+1)...n]$, $B[1...(n/2)]$ and $B[(n/2+1)...n]$ are sorted.

I.S:

After the i-th loop, solA[1...i] $\prec$ leftA[j+1...m] and solA[1...i] $\prec$ rightA[k+1...m], other is same. So after the loop, solA[1...(m+m')] is sorted and solB[1...(m+m')] is sorted.

Finally, we return $A'[n/2]$, which is the magical-mean.

Time Complexity:

$$\because T(n) = 2T(\frac{n}{2}) + O(n)$$

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

(b)

Algorithm 4 MergeSort

function MedianOfMedians(A, B)
$\langle GA_1, GB_1, ..., GA_{n/5}, GB_{n/5} \rangle$ = CreateGroups(A, B)
for i = 1 **to** n/5 **do**
MergeSort(GA_i, GB_i)
end for
MA, MB = GetMediansFromSortedGroups($GA_1, GB_1, ..., GA_{n/5}, GB_{n/5}$)
return QuickSelect(MA, MB, (n/5)/2)
end function
function QuickSelect(A, B, i)
if A.size == 1 **then**
return $A[1], B[1]$
else
ma, mb = MedianOfMedians(A, B)
q = PartitionWithPivot(A, B, ma, mb)
if i == q **then**

return A[q]
else if i $<$ q **then**
return QuickSelect(A[1...(q-1)], B[1...(q-1)], i)
else
return QuickSelect(A[(q+1)...A.size], B[(q+1)...A.size], i - q)
end if
end if
end function
function MagicalMedian(A, B)
ma, mb $=$ QuickSelect(A, B, n/2)
return ma
end function

Correctness:

In basic case, if A.size == 1, return the A[1] and B[1].

After partition, A[1...(q-1)] $\prec$ A[(q+1)...A.size] for weighted set. At the next times, if i < q, we can get the i-th value in A[1...(q-1)] by QuickSelect(A[1...(q-1)], i); if i > q, we can get the $(i-q)$-th value in A[(q+1)...A.size] by QuickSelect(A[(q+1)...A.size], i-q).

Finally, we can get the i-th value for the weighted set by MagicalMedian(A, B).

Time Complexity:

Let $T(n)$ be the time complexity of QuickSelect, so $T(0.2n)$ is time complexity of MedianOfMedians. Then the problem will be scaled down to $T(0.7n)$ at most. The time complexity of partition is $O(n)$, so

$$T(n) \leqslant T(0.7n) + T(0.2n) + O(n)$$

Using recursive tree method, we can know

$$T(n) = O(n)$$

Problem 4

(a)

Algorithm 5 Sort

---

function Sort(A)
isSame $=$ true
zero $=$ NULL
one $=$ NULL
sum $=$ 0
for i $=$ 2 **to** n **do**
if isSame $==$ true **then**

```
            result = compare(A[1], A[i])
            if result == "x < y" then
                isSame = false
                zero = A[1]
                one = A[i]
            else if result == "x > y" then
                isSame = false
                zero = A[i]
                one = A[1]
                sum = i - 1
            end if
        else
            result = compare(zero, A[i])
            if result == "x < y" then
                sum = sum + 1
            end if
        end if
    end for
    if isSame == true then
        return A[1...n]
    end if
    for i = 1 to n do
        B[i] = (i <= (n - sum)) ? 0 : 1
    end for
    return B[1...n]
end function
```

(b)

Algorithm 6 Sort

```
function BucketSort(A, i)
    k = 52
    ⟨L_1, L_2, ..., L_k⟩ = CreateBuckets(k)
    for i = 1 to A.length do
        AssignToBucket(A[i])
    end for
    for i = 1 to k do
        for j = 1 to L_i.length do
            if L_i[j] don't have (i + 1) digit then
                L_i^1.add(L_i[j])
            else
                L_i^2.add(L_i[j])
            end if
        end for
    end for
    return ⟨L_1^1, L_1^2, ..., L_k^1, L_k^2⟩
end function
function RadixSort(A, i)
    L_1^1, L_1^2, ..., L_k^2 = BucketSort(A, i)
    for i = 1 to k do
```

$L_k^1 = \text{RadixSort}(L_k^1, \text{i} + 1)$
$L_k^2 = \text{RadixSort}(L_k^2, \text{i} + 1)$
end for
return $\text{CombineBuckets}(L_1^1, L_1^2, ..., L_k^2)$
end function
function $\text{Sort}(A, \text{i})$
return $\text{RadixSort}(A, 1)$
end function

Time Complexity:

Time of BucketSort is $T_1(n) = O(n)$

And we can make sure that each BucketSort consume a layer of characters of input array. Finally, we sort the input array based on each characters of strings, and we know that the total number of characters over all the strings is n.

So, the time of Sort is $T(n) = O(n)$.

Problem 5

(a)

We assume that there are no central vertex.

We choose an partition that there are tree subtrees, a_1, b_1, c_1, and $a_1 > \frac{n}{2}$. We move the central vertex to a_1 tree direction so that $a_2 < \frac{n}{2}$, because of the assumption, we can think that $b_2 > \frac{n}{2}$.

$\because b_2 = b_1 + c_1 + 1 > \frac{n}{2}$

$\therefore b_2 = b_1 + c_1 + 1 = (n - 1 - a_1 - c_1) + c_1 + 1 = n - a_1 > \frac{n}{2}$

$\therefore a_1 < \frac{n}{2}$

$\therefore a_1 < \frac{n}{2}$ contradict with $a_1 > \frac{n}{2}$

$\therefore$ the assumption is wrong

$\therefore$ every tree has a central vertex

(b)

Algorithm 7 FindCentralVertex

```
function PreorderTrav(r)
    if r != NULL then
        count = 1
        for each child u of r do
            count = count + PreorderTrav(u)
        end for
        hashTable.add(key: r, value: count)
        return count
    end if
end function
function FindCentralVertex(root)
    PreorderTrav(root)
    leftCount = hashTable(key: root.left)
    rightCount = hashTable(key: root.right)
    largerCount = leftCount > rightCount ? leftCount : rightCount
    cur = leftCount > rightCount ? root.left : root.right
    while largerCount > hashTable(key: root) / 2 do
        largerCount = leftCount > rightCount ? leftCount : rightCount
        cur = leftCount > rightCount ? cur.left : cur.right
        leftCount = hashTable(cur.left)
        rightCount = hashTable(cur.right)
    end while
    return cur
end function
```

Correctness:

Just same with **(a)**.

Time Complexity:

Time complexity of PreorderTrav is $T_1(n) = \Theta(n)$. And we know the while-loop will be executed no more than the height of tree times. So the final time complexity is $T(n) = \Theta(n) + O(n) = O(n)$.

Problem 6

(a)

Algorithm 8 kthElement

```
function kthElement(A[1...m], B[1...n], k)
    if m == 0 then
        return B[k]
    end if
    if n == 0 then
        return A[k]
    end if
```

midA = m / 2
midB = n / 2
if midA + midB < k **then**
    if A[midA] < B[midB] **then**
        return kthElement($A[midA+1...m]$, $B[1...n]$, k - midA)
    else
        return kthElement($A[1...m]$, $B[midB+1...n]$, k - midB)
    end if
else
    if A[midA] < B[midB] **then**
        return kthElement($A[1...m]$, $B[1...midB-1]$, k)
    else
        return kthElement($A[1...midA-1]$, $B[1...n]$, k)
    end if
end if
end function

Time Complexity:

Because it divide $A[1...m]$ or $B[1...n]$ into two parts with equal length, so the time complexity is $T(n) = O(\log n + \log m)$.

Because $\log(m+n) < O(\log n + \log m) = O[(\log(nm))] < O(\log[(m+n)^2]) = O(2\log(m+n))$,

We can know that $T(n) = O(\log(m+n))$

(b)

Algorithm 9 InorderTravIter

```
function InorderTravIter(root)
    if root.left is not NULL then
        last = root
        cur = root.left
    else if root.right is not NULL then
        Visit(root)
        last = root
        cur = root.right
    else
        Visit(root)
        return
    end if
    while True do
        if last.left == cur || last.right == cur then
            if cur.left is not NULL then
                last = cur
                cur = cur.left
            else if cur.right is not NULL then
                Visit(cur)
                last = cur
```

cur = cur.right
else
Visit(cur)
temp = last
last = cur
cur = temp
end if
else if cur.left == last **then**
Visit(cur)
if cur.parent == NULL **then**
Break
end if
if cur.right is not NULL **then**
last = cur
cur = cur.right
else
end if
else if cur.right == last **then**
if cur.parent == NULL **then**
Break
end if
last = cur
cur = cur.parent
end if
end while
end function

The algorithm use `last` and `cur` to replace stack, so the space complexity is $O(1)$. We use inorder travel strategy, so the time complexity is $O(n)$.