PS5

201300069 邓嘉宏

Problem 1

(a)

运用类似快速排序中Partition的方法, 先用O(n)时间挑出A数组中第 $(n/k)*(k/2)$ 小的元素, 此时A[1...(n/k)*(k/2)-1]<=A[(n/k)*(k/2)]<=A[(n/k)*(k/2)+1...A.size], 然后再递归划分A[1...(n/k)*(k/2)-1]和A[(n/k)*(k/2)+1...A.size], 直到每一块的规模均为n/k.

伪代码:

```
k_sort(A[1...n],k)
    if (A.size==n/k){
        return;
    }
    m=QuickSelect(A,(n/k)*floor(k/2))//返回值m不重要, QuickSelect后A的左半部分小于右半部分
    k_sort(A[1...(n/k)*floor(k/2)-1],floor(k/2))
    k_sort(A[(n/k)*floor(k/2)+1...A.size],floor(k/2)+1)
```

正确性: 每次QuickSelect选出"中间元素"并且将数组划分成左半部分小于右半部分, 当递归划分到数组规模均为n/k时, 则完成k-sorts.

时间复杂度: 不难写出递归式: $T(n)=2T(n/2)+O(n)$, 基础情况为T(n/k), 所以递归树高lgk, 总时间 $T(n)=n\lg k$.

(b)

对于k-sorts的决策树, 将每 n/k 个元素看做同一类元素, 然后进行排列, 因此决策树的叶子数≥ $\binom{n}{n/k,n/k,...n/k}=\frac{n!}{((n/k)!)^k}$, 树高≥ $\lg\frac{n!}{((n/k)!)^k}=n\lg k$. 因此在最坏情况下, 比较次数为 $\Omega(n\lg k)$.

Problem 2

(a)

即求两个假币在同一盘中的概率: $\dfrac{\binom{n-2}{n/2-2}+\binom{n-2}{n/2}}{\binom{n}{n/2}}=\dfrac{n-2}{2n-2}$.

(b)

首先, 随机划分这堆币直到两盘重量不同, (即重量较轻的假币在位置较高的有n/2枚币的盘中,重量较大的假币在位置较低的有n/2枚币的盘中), 然后分别筛选两盘币: 对较重的那盘, 每次随机划分成两份, 取较重那份继续随机划分, 当最后两枚币中有一枚较重假币时, 取一枚真币分别和它们称量,找出较重假币; 寻找较轻假币同理.

伪代码:
假设总硬币为数组m[1...n], 两盘分别为数组A,B

```
randomly divide m into A and B
while(A and B is balanced){
    randomly divide m into A and B
}
for A:
    randomly divide A into two parts until one of two coins is fake

balance each of the two coins with a true coin to identify the fake one;

for B:
    randomly divide B into two parts until one of two coins is fake

balance each of the two coins with a true coin to identify the fake one;
```

正确性: 将所有硬币划分成不平衡的两份保证了两枚假币分别在两盘中, 对每一盘使用"二分法"可找出其中的一枚假币.

预期称量次数: 根据(a)可知, 将所有硬币划分成重量不等两份的概率为 $\dfrac{n}{2n-2}$, 所以划分成重量不等两盘的预期称量次数为 $\dfrac{2n-2}{n}$, 对每一盘使用二分法的预期划分次数为:$\lg\dfrac{n}{2}-1+1.5$, 其中 1.5 为最后两枚硬币的称量次数.
所以预期总称量次数为: $\dfrac{2n-2}{n}+2*\lg\dfrac{n}{2}+1.$

Problem 3

(a)

(按值排序S[1...n]时同步排序W[1...n],保证S[i],W[i]对应同一元素)
先按值排序数组S[1...n], 然后计算出w(S[1...n]), 再从S[1]开始累加S[i].weight, 直到w(S[1...m])>w(S[1...n])/2, 那么S[m-1]就是 magical-mean.

伪代码:

```
RndQuickSort(S[1...n])//随机化快速排序时同步排序W[1...n]
for i=1 to n:
    total_weight+=W[i]
m=1
some_weight=0;
while(1){
    some_weight+=W[m];
    if(some_weight>total_weight/2){
        m--;
        break;
    }
    m++;
}
return S[m],W[m]
```

正确性: 按值排序后, 前一元素的值小于后一元素, 从第一个元素开始逐一累加weight, 当weight大于total_weight的一半时, 停止累加, 此时我们就找到了magical_mean.

时间复杂度: RndQuickSort()的时间复杂度为$O(n\lg n)$, 累加过程的时间复杂度为$O(n)$, 所以总时间复杂度为:$O(n\lg n)+O(n)=O(n\lg n)$.

(b)

利用QuickSelect()中随机划分的方法, 随机划分S[1...n], 记左、右两半部分的weight之和为w1、w2, 如果w1或w2大于w(S[1...n])/2, 则递归随机划分大于w(S[1...n])/2那部分;直到划分点的左右两部分的weight和均小于等于w(S[1...n])/2.

伪代码:

```
Solve(S[1...n],l_weight,r_weight)
//l_weight和r_weight分别表示划分区域左右两边的weight
//第一次调用Solve()时,l_weight=r_weight=0

    m = MedianOfMedians(S[1...n])
    q = PartitionWithPivot(S[1...n],m)

    left_weight = sum(S[1...q-1].weight)+l_weight
    right_weight = sum(S[q+1...n].weight)+r_weight
    if(left_weight<=total_weight/2&&right_weight<=total_weight/2){
        return S[q];
    }
    if(left_weight>right_weight){
        r_weight=right_weight;
        Solve(S[1...q],l_weight,r_weight)
    }
    else{
        l_weight=left_weight;
        Solve(S[q...n],l_weight,r_weight)
    }
```

正确性: 每次随机划分后, S[q]左边的值均小于S[q],右边的值均大于S[q],每次随机划分后分别计算左右两边的weight, 当它们均小于 total_weight/2时, S[q]即为magical_mean.

时间复杂度: 根据QuickSelect()的时间复杂度分析, 我们容易得出该算法的时间复杂度:$T(n) \leq T(0.7n) + T(0.2n) + O(n)$, 所以:$T(n) = O(n)$.

Problem 4

(a)

选取该数组的最后一个元素当做主元, 其余元素依次与主元比较, 与主元相等的元素放在数组A, 大于(或小于)主元的元素放在数组B,最后,若B数组元素大于主元,则合并AB,否则,合并BA.

伪代码:

```
Sort_0_1(S[1...n])
    pivot=S[n];
    Array A,B;
    flag_is_larger=false;
    for i=1 to n-1{
        if (S[i]==pivot){
            A.add(S[i])
        }
        else{
            if(S[i]>pivot){
                flag_is_larger=true;
                B.add(S[i])
            }
        }
    }

    A.add(pivot);

    if(flag_is_larger){
        return concatenate(AB)//A数组在前合并
    }
    else{
        return concatenate(BA)//B数组在前合并
    }
```

正确性: 因为数组元素只有0或1,选出一个主元, 经过n-1次比较, 将所有0,1分别放在数组A或B,最后将元素均为0的数组放在前进行合并, 就得到排序好的数组.

时间复杂度: 显然为$O(n)$.

(b)

利用类似基数排序的方法, 将所有字符串按第一个字母(最左边一位)进行排序, 然后将第一位相同的字符串划分到同一组, 对每组进行递归排序.

伪代码:

```
SortString(S[1...m],i)//按照字符串从左往右第i位排序
    sort(S[1...m]) by the i_th character;//根据第i位按照字母顺序排序, 第一次调用时 i=1.
    for each part with the same i_th character denoted S[j1...jm]:
        SortString(S[j1...jm],i+1)
```

正确性: 第一次按照左边第一位排序, 从小到大将所有字符串分成几个"模块",再递归调用函数排序每个"模块", 显然正确.

时间复杂度: 该算法遍历每个字符 1 次, 共有 n 个字符, 时间复杂度为$O(n)$.

Problem 5

(a)

对于任何一棵树, 我们从根结点开始尝试删除该结点, 若删除后得到的每棵子树的结点数不超过n/2, 则成功删除,该结点即为要找的中间结点; 否则, 若有某棵子树的结点数大于n/2, 则删除失败,且其他子树的结点总数一定小于n/2, 接下来尝试删除结点数大于n/2的子树的根结点, 那么以该结点的父结点为根结点的子树的结点总数一定不超过n/2, 再判断删除该结点后各子树的结点数情况, 若有某棵子树的结点数超过n/2,尝试删除该子树的根结点, 如此循环, 因为以尝试删除的结点的父结点为根结点的子树的结点总数不超过n/2且逐渐接近n/2, 最坏情况下等于n/2, 此时一定能成功删除某个中间结点, 即证任何一棵树都有一个中间结点.

(b)

按照(a)中描述的方法, 先递归找出以每个结点为根结点的子树的结点数, 然后从根结点开始尝试删除结点, 直到成功, 该结点即为中间结点.

伪代码:

```
//递归计算以每个结点为根结点的子树的结点数为每个结点的val
count_each_node(node)
    if (node.children==null) return 1 //叶结点
    if (node.left!=null)
        m1=count_each_node(node.left)
        node.val+=m1
    if (node.right!=null)
        m2=count_each_node(node.right)
        node.val+=m2
    return node.val

try_delete(node)
    val1=val2=val3=0
    if(node.parent!=null)
        val1=root.val-node.val //以父结点为根结点的子树的结点数
    if(node.left!=null)
        val2=node.left.val //以左子结点为根结点的子树的结点数
    if(node.right!=null)
        val3=node.right.val //以右子结点为根结点的子树的结点数
    return val1,val2,val3

find_central_vertex(root)
    count_each_node(root)

    cur=root
    while(1){
        val1,val2,val3=try_delete(cur)
        if(val1>root.val/2)
            cur=cur.parent
            continue
        if(val2>root.val/2)
            cur=cur.left
            continue
        if(val3>root.val/2)
            cur=cur.right
            continue
        else
            break;
    }
    return cur;
```

正确性: (a)中已证明.

时间复杂度: 递归计算以每个结点为根结点的子树的结点数时, 遍历每个结点一次, 时间O(n), 尝试删除结点时间也为O(n), 所以总时间为:$O(n)$.

Problem 6

(a)

利用类似二分查找的方法, 将k分成两份k/2, 比较A[k/2]和B[k/2], 若相等, 则A[k/2]即为合并后第k小的元素; 若A[k/2]>B[k/2], 则寻找A[1...m]和B[k/2+1...n]合并后第k/2小的元素; 若A[k/2]< B[k/2], 则寻找A[k/2+1...m]和B[1...n]合并后第k/2小的元素.

伪代码:

```
kth_smallest(A[1...m],B[1...n],k)
    if(A.size>B.size)
        return kth_smallest(B,A,k)//保证第一个数组长度不超过第二个数组
    if(k==1)
        return min(A[1],B[1])
    if(A.size==0)
        return B[k]

    k1=min(k/2,A.size)
    k2=k-k1

    if(A[k1]>B[k2])
        return kth_smallest(A[1...m],B[k2+1...n],k-k2)
    else if(A[k1]<B[k2])
        return kth_smallest(A[k1+1...m],B[1...n],k-k1)
    else if(A[k1]==B[k2])
        return A[k1]
```

正确性: 比较A[k/2]和B[k/2]时, 若A[k/2]>B[k/2], 说明B[1...k/2]小于合并后的第k小的元素,可以不考虑, 则递归查找A[1...m]和B[k/2+1...n]合并后第k/2小的元素; 若A[k/2]< B[k/2], 说明A[1...k/2]小于合并后的第k小的元素,可以不考虑, 则递归查找A[k/2+1...m]和B[1...n]合并后第k/2小的元素. 当某数组为空时或者k=1时一定能找到答案.

时间复杂度: 对k进行了二分, 所以:$T(n) = O(\lg k)$, 因为$k \le m + n$, 所以时间复杂度:$T(n) = O(\lg k) = O(\lg{(m+n)})$.

(b)

利用Morris遍历, 用cur表示当前访问到的结点, mostright表示cur左子树的最右叶结点. cur初始为root, 如果cur无左子结点,那么 cur=cur.right; 否则, mostright为cur左子树的最右结点;如果mostright无右子结点, 则mostright.right=cur, 同时cur=cur.left;如果mostright的右子结点指向cur, 则将mostright.right=null, cur=cur.right;更新完cur后循环进行以上操作, 直至cur==null.

伪代码:

```
Morris(root)
    cur=root, mostright=null
    while(cur!=null){
        mostright=cur.left
        if(mostright!=null){
            while(mostright.right!=null&&mostright.right!=cur)
                mostright=mostright.right

            if(mostright.right==null)
                mostright.right=cur
            if(mostright.right==cur){
                mostright.right=null;
                cur=cur.right
            }
        }
        else{
            cur=cur.right
        }

    }
```

正确性: 利用左子树的最右叶结点的右子结点记录当前结点(即更新cur后的前驱结点), 记录了"从哪里来", 当遍历到叶结点时, 能回到前驱结点, 顺利完成所有遍历.

时间复杂度: 左子树为空的结点遍历一次, 否则遍历两次, 时间复杂度为:$O(n)$.

空间复杂度: 额外使用了两个指针, 空间复杂度为:$O(1)$.