数据结构与算法第四次作业

麻超　201300066

南京大学人工智能学院

2021 年 10 月 20 日

Problem 1

a

可以通过改进快速排序算法以得到一个 k-sorted 数组. 具体方法为快速排序递归时当 right-left<k 时就取消递归.

由于快速排序的特性, 每一次排序完成后, 在主元左边的数总是比主元小, 在主元右边的数总是比主元大. 这样设置主元时，最左边的 n/k 个元素一定是最小的 k 个, 在这之后这些元素不参加排序, 那么接下来 n/k 个元素一定是次小的几个. 这样递推下来, 一定是符合 k-sort 的要求的.

在最优情况下, 即每次都能平分数组时, 可以得到时间复杂度为 T(n)=T(n/2)+n.

其中这个递归树在最优情况下一共需要递归 $\log_2 k$ 次, 即递归树深度为 $\log k$, 每层需要遍历所有元素, 故相乘之后时间复杂度为 $O(n\log k+n)=O(n\log k)$.

b

对于一个数组而言, 其最小的 n/k 个数字是确定的, 但顺序不定. 即每个子数组中的数据都是确定的. 然而任何基于比较的排序算法都是基于二叉树排列的, 即对所有的 $(k!)^{n/k}$ 种情况都能够做出排列, 即该二叉树必须至少有这么多叶节点, 所以决策树的高度至少为 $2^h \geq (k!)^{n/k}$

$$h \geq \frac{n}{k}\cdot lg(k!) \geq \frac{n}{k}\cdot(\frac{k\ln k-k}{\ln 2}) = \frac{n\ln k-n}{\ln 2} = \Omega(n\lg k)$$

即至少需要 $\Omega(n\lg k)$ 次比较.

Probelm 2

a

分为两组硬币后, 依然有两个假币, 一个比较重一个比较轻. 当这两个不在同一边时, 其重量不同, 当这两个在同一边时, 其重量相同. 故要求的概率就是两个假币在同一边的概率. 当 n 较大时, 可以近似认为每一个硬币在前一堆和后一堆的概率都是 1/2. 故 $P = 2 * 1/2 * 1/2 = 1/2$. 即两边重量相同的概率是 1/2.

b

过程 1: 每次将硬币分为两堆, 再根据 1 的结论, 判断两个假币在同一边或者不在同一边. 如果其在同一边, 则需要对两个大小为 n/2 的硬币堆都再次进行判断, 直到找出重的和轻的硬币分别在哪一堆. 当找到这两枚假币分别在哪一堆之后就可以用新的算法来分别找出这两枚硬币.

过程 2: 新的算法也很常见, 就是将该硬币堆分为三堆 (事实上当 n 为 2 的幂时无法被 3 整除, 此时只需要尽量保证三堆的元素个数近似相同且至少有两堆元素个数相同), 然后将数目相同的两堆进行比较 (此时假设需要找的是轻的假币), 若有一堆比较轻, 则证明假币在这一堆里面, 如果两堆重量相同, 则证明假币在第三堆中, 再次递归循环, 直到找出假币所在位置, 重的假币同理.

在过程一中, 即找出两个假币分别在哪一堆时, 有 1/2 的概率没有办法找出, 另有 1/2 的概率可以找出 (来自 (1)), 在找出之前, 每次都维持着原来的问题规模不变, 一旦找到, 问题规模就会缩小至 $n(\frac{1}{2})^k$(k 为过程 1 的执行次数). 在找出之后, 每次递归可以减少至原来 1/3 的复杂度. 由于过程 1 可以近似理解为服从参数 p 为 1/2 的几何分布, 故执行次数的期望 E(x)=2. 即期望意义下需要执行两次过程 1. 在这种情况下, 会将问题缩小至 n/2 的规模 (两个 n/4 规模大小的问题).

故时间复杂度 $T(n) = T(n/2) + O(1) = T(n/6) + O(1) = ... = O(\log_3 n)$

Probelm 3

a

首先对数组 W 进行循环求和, 计算出 w(S), 以供后续比较.

其次, 利用一个时间复杂度为 O(nlg n) 的排序算法 (可以是快速排序, 归并排序等...) 对数组 S 进行排序, 并在对数组 S 排序时调整数组 W 中元素的位置, 以确保二者元素可以一一对应. 接下来只需要对数组 S 进行遍历, 依次求出 S 对应的 weight 的值的和, 并将其与 w(S)/2 进行比较, 直至其可以满足所述条件.

时间复杂度: 第一次计算总和的时间复杂度为 O(n), 排序算法的时间复杂度为 O(nlg n), 后面计算部分和并与之比较的时间复杂度也为 O(n), 故总的时间复杂度为 O(nlg n).

b

第一步和 (a) 一样首先计算所有 weight 的和, 以便后边进行比较. 首先按照 Select 函数寻找主元的方法找到主元, 其次按照所得到的主元的值对数组 S 进行 Partition 划分, 将小于主元的划分在 S 左边, 将大于 S 的划分在 S 右边, 并同时调整数组 W 中元素的值, 使之与 S 可以一一对应. 计算主元两边数组 S 中元素对应的 weight 的和, 如果可以满足题目中对 magical-mean 的定义, 则返回该主元. 如果不满足 magical-mean 的定义, 则再次进行递归操作, 即若左边一部分 weight 的和大于 w(S)/2, 则对左边部分进行递归, 直到左边刚好满足小于 w(S)/2. 右半部分同理.

时间复杂度: 第一次计算总和时为 O(n), 后面进行递归操作时由学过的知识可知该递归不会改变时间复杂度, 故该算法时间复杂度为 O(n).

Problem 4

a

由于只有两种元素 1 和 0, 所以只需要找一个参考元素, 用其他的元素与之比较即可. 然而选取的这个元素不能与 0 或者 1 比较, 否则其至少需要 n 次比较才可以完成. 可以利用与后面元素的判断来确定该参考元素的值. 具体操作方法为选定一个元素 a[0], 令 a[1],a[2],...a[n-1] 与之比较, 若 a[1]<a[0], 将其放入 0 的一队中, 若 a[1]>a[0], 将其放入 1 的一队中, 若 a[1]=a[0], 则先等待直到能够确定 a[0] 的值. 另

外在比较过程中, 如果出现了 a[i]<a[0] 的情况, 则可以确定 a[0] 的值为 1, 否则则可以确定 a[0] 的值为 0. 在整个比较过程中, 只需要比较 a[0] 与其他所有元素值的大小, 即需要比较 n-1 次.

b

利用桶排序的思想对其进行排序. 首先建立 26 个桶, 分别对应字母 a-z. 然后根据每个字符的首字母将其放入 26 个桶内, 然后再在每一个桶内对这个桶内的字符串的第二个字母 (没有第二个字母时放在最前面) 排序, 依次类推, 最后按照桶的顺序和桶内元素的顺序将字符串放入数组内, 排序即可完成. 由于所有的字符串字符总和为 n, 故假设第一次共有 k 个字符, 排序需要 O(k) 时间, 后面的排序同理, 总的时间复杂度为 O(n).

Problem 5

a

利用数学归纳法证明结论.

奠基: 当 n=1 时, 则该根节点即为 central vertex.

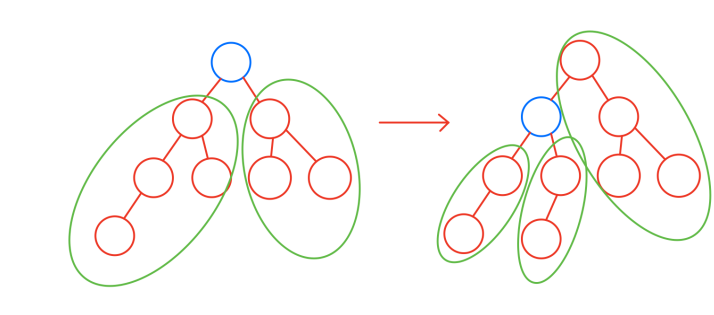
归纳假设: 设当 n=k 时该结论可以成立, 即当 n=k 时该树存在一个 central vertex 使得该树可以被分为三个子树, 且每个子树含有的顶点树均不大于 n/2.

归纳递推: 如果证明当 n=k+1 时, 无论 n=k 时是怎样的情况, 都可以得到该结论正确, 则由数学归纳法的基本思想可以得知该结论是正确的. 分为以下几种情况:

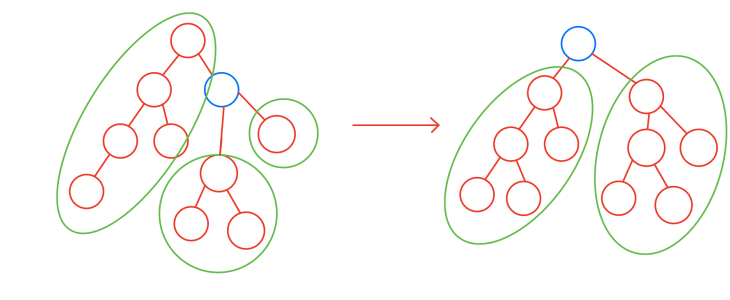
1. 如果当 n=k 时所得到的三个子树节点数都不大于 n/2, 则添加一个节点后, 无论添加在哪个位置, 它必然时这三个子树的其中一个子树的叶节点, 则这三个子树的节点数同样不会超过 (k+1)/2, 可以得知正确.

2. 如果当 n=k 时得到的三个子树其中一个子树节点数为 n/2(T1), 其他两个小于或等于 n/2(T2,T3), 但和不会超过 n/2. 当该节点添加到 T2,T3 时, 同样满足原条件, 可以证明. 以下只讨论添加叶节点到 T1 的情况. 分为以下子情况:

1). 如图所示, 当原来的 central vertex 为根节点时, 可以选择将 central vertex 向 T1 移动, 便可维持 T1 的规模不变, 将 T2(没有 T3) 的规模增加 1. 当 k 为奇数时,T2 的规模是 n/2-1, 变化后, 变为 n/2. 满足条件.



2). 如图所示, 当原来的 central vertex 不是根节点时, 可以选择将 central vertex 向根节点移动, 可维持 T1 的规模, 并且 T1 的规模保持 n/2 时, 可以保证剩下一部分 (或两部分) 的规模也不会超过 n/2. 满足条件.



3)central vertex 不可能为叶节点.

由上述数学归纳法过程可知, 对于所有的树, 都至少有一个 central vertex.

b

通过检查每一个树或子树的节点树来判断是否为 central vertex. 当前指向 root 节点, 则判断 root 节点的左子树和右子树所包含的顶点有多少个, 若二者相等 (平衡二叉树), 则该 root 节点即为 central vertex. 否则, 则指向包含顶点数多的该节点, 将该节点的左子树所含顶点数目与右子树所含顶点数目与上一轮得到的另一个子树包含的顶点数目 +1 做比较, 若三者中的最大值不超过 n/2, 则该节点即为 central vertex. 否则, 继续移动向该节点包含顶点数多的子树的顶点. 直到找到某个顶点满足 central vertex 的要求为止.

Problem 6

a

要找 A 和 B 的并集的第 k 小元素, 考虑利用二分法.

首先假设 k 为偶数, 当 A,B 的元素数量都大于 k/2 时, 比较 A[k/2-1](即 A 中第 k 个元素),B[k/2-1](B 中第 k 个元素), 比较的结果分为三种情况:

- A[k/2-1]<B[k/2-1], 此时说明 A 中最多有在 k/2-1 之前的数和 B 中 k/2-1 之前的数, 共 k-2 个数大于 A[k/2-1], 即其不可能是第 k 小的数. 此时排除 A 中 A[0,...k/2-1] 这 k/2 个元素.
- A[k/2-1]>B[k/2-1], 同理, 删除 B[0,...k/2-1] 这 k/2 个元素.
- A[k/2-1]=B[k/2-1]. 此时即找到了第 k 小的元素, 返回 A[k/2-1] 或 B[k/2-1] 即可.

这之后, 如果没有找到, 则再次进行递归, 更新 k 的值, 即令 k=k-k/2. 继续取 A[k/2-1] 和 B[k/2-1] 进行比较.(注意此时有一个数组中删除了 k/2 个元素.) 则反复递归, 利用二分查找法找到第 k 小的值.

如果 A 和 B 中有一个数组的剩余值已不足 k/2, 则取该数组的全部元素, 不足的部分由 B 来补齐. 在这种情况下,k 后来的取值应该以排除的个数来决定, 而不是直接减去 k/2. 另外, 当 k=1 时, 可以直接输出 A 和 B 当中的最小值. 当某一个数组为空时 (已经被全部排除), 可以直接输出另一个数组第 k 小的值.

伪代码如下: 时间复杂度:m,n 为数组 A 和 B 的长度, 由于每次查找都可以缩小一半范围, 所以时间复杂度为 $O(\log(m+n))$

b

要想在 O(1) 空间内实现遍历树中的所有元素, 则需要考虑在输出某个左孩子, 右孩子都为空的节点后继续输出下一个节点, 在中序遍历中, 即代表了其某个祖先节点. 因此考虑在遍历该孩子之后, 指向下一个要遍历的元素 (其祖先节点). 得到以下方法:

在该方法中, 首先给定一个树的 root 节点, 然后寻找该 root 节点的左子树最右边的节点 (即中序遍历时其前一个元素), 该节点必定是没有右孩子的, 故令其右指针指向 root 节点, 然后指针指向 root 节点的左孩子, 继续遍历, 往此反复, 可以得到以下步骤:

1. 如果当前节点的左孩子为空, 则输出当前节点并将其右孩子作为当前节点 (指针指向下一个节点).

2. 如果当前节点的左孩子不为空, 在当前节点的左子树中找到当前节点在中序遍历下的前驱节点 (即最” 右” 的节点).

Algorithm 1 find the k_{th} smallest number in A∪B

Input: $A[], B[], k, A.length = m, B.length = n$

```
function GetKthElement(A[], B[], k)
    index1, index2 ← 0
    while True do
        if index1==m then
            return B[index2 + k − 1]
        end if
        if index2==n then
            return A[index1 + k − 1]
        end if
        if k==1 then
            return min(A[index1], B[index2])
        end if
        newIndex1 ← min(index1 + k/2 − 1, m − 1)
        newIndex2 ← min(index2 + k/2 − 1, n − 1)
        if A[newIndex1] ≤ B[newIndex2] then
            k ← k − newIndex1 − index1 + 1
            index1 = newIndex1 + 1
        else
            k ← k − newIndex2 − index2 + 1
            index2 = newIndex2 + 1
        end if
    end while
end function
```

Output: $GetKthElement(A, B, k)$

a) 如果前驱节点的右孩子为空, 将它的右孩子设置为当前节点. 令当前节点的左孩子成为当前节点.

b) 如果前驱节点的右孩子为当前节点, 将它的右孩子重新设为 NULL(恢复树的形状) 并输出当前节点. 令当前节点的右孩子成为当前节点.

3. 重复以上 1 及 2 步骤直到当前节点为空, 即该树的所有元素都被输出了.

伪代码如下: 复杂度分析:

Algorithm 2 print the tree

```
Require: TreeA, node *cur = root, *prev = NULL
  while cur!= NULL do
    if cur.left== NULL then                                    ▷ 1.
      Print(cur.value)
      cur ← cur.right
    else
      prev ← cur.left
      while prev.left ≠ NULL&&prev.right ≠ cur do
        prev ← prev.right
      end while
      if prev.right==NULL then                                 ▷ 2.a
        prev.right ← cur
        cur ← cur.left
      else                                                     ▷ 2.b
        prev.right ← NULL
        Print(cur.value)
        cur ← cur.right
      end if
    end if
  end while
```

由于在该过程中只使用了两个辅助指针, 故空间复杂度为 O(1).

每个树有 n 个结点, 故有 n-1 条边, 在整个遍历过程中每一条边最多只被走两次, 一次是为了定位到某个节点, 另一次是寻找某个节点的前驱节点, 故其时间复杂度为 O(n).