Problem Set 3 Answer Sheet

151220131 谢旻晖

Problem 3.4

算法 B 的过程如下: 利用 A 找到当前的中位数，利用中位数作为 pivot, 进行 partition 操作将数组划分为左数组、median、右数组。

如果 $k == \frac{n}{2}$ ，则已经找到

如果 $k < \frac{n}{2}$, 则在左子数组中递归找 k 阶的数

如果 $k > \frac{n}{2}$, 则在右子数组中递归找 $k - \frac{n}{2}$ 阶的数

Problem 3.5

1.

直接对 n 个数进行排序，直接可得前 k 大的数.

2.

维护一个最大堆：首先花 $O(n)$ 的时间对原集合建立最大堆，接着用类似堆排序的方法依次取出堆顶，再将最后一个数填入堆顶并作修复，每次花费 $O(\log n)$ 的时间修复堆，一共 k 次为 $O(k \log n)$.

3.

利用线性时间选择算法花 $O(n)$ 的时间选出第 k 大的数，同时得到前 k-1 大的数集，再利用 $O(k \log k)$ 的时间对前 k-1 大的数进行排序。

Algorithm 1 FINDTHECLOSESTKELE$S[0...n-1], k$

```
SORT(S,n)
mid = n/2
p = mid − 1
q = mid + 1
ANSWER_SET=∅
while |ANSWER_SET| < k do
    if |S[p] − S[mid]| < |S[q] − S[mid]| then
        //S[p] 离中位数更近，选 S[p]
        ANSWER_SET=ANSWER_SET∪S[p]
        p = p − 1
    else
        //S[q] 离中位数更近，选 S[q]
        ANSWER_SET=ANSWER_SET∪S[q]
        q = q + 1
    end if
    return ANSWER_SET
end while
```

Problem 3.6

1.

算法思路: 先对集合 S 进行排序，然后从中值 M 开始往两端走，比较两边的数谁离 M 更接近一点，直到取满 k 个数。

算法见: **Algorithm 1**

2.

算法分为以下几步:

STEP 1: 利用线性时间选择算法求出数组 S 的中位数的值，为 mid

STEP 2: 计算数组每个数与中位数差的绝对值，转而存于另一个数组 B 中 O(n)

STEP 3: 求出数组 B 中第 k+1 小的数 x（注意到其中有一个是中位数自己与中位数的差为 0）

STEP 4: 数组 S 中与 mid 差的绝对值小于等于 x 但不等于 0 的数都是最接近中位数的数

STEP 5: 对选出来的最接近中位数的 k 个数排序

整体的时间复杂度为 $O(n+n+n+n+k\log k) = O(n+k\log k)$

Problem 3.7

1.

Algorithm 2 UNION_MEDIAN_2($A[1...n], B[1...n], k$)

```
1: //basecase
2: if k == 1 then
3:     return min(A[1],B[1])
4: end if
5: if A[(k+1)/2] > B[(k+1)/2] then
6:     A = A[1...(k+1)/2]
7:     B = B[(k+1)/2 + 1, ...n]
8:     UNION_MEDIAN_2(A, B, (k+1)/2)
9: else if A[k/2] < B[k/2] then
10:    B = B[1...k/2]
11:    A = A[k/2 + 1, ...n]
12:    UNION_MEDIAN_2(A, B, k/2)
13: else
14:    return A[k/2]
15: end if
```

算法思路：取出 $A[\frac{k}{2}]$ 和 $B[\frac{k}{2}]$, 比较他们的大小，如果 $A[\frac{k}{2}]$ 大，则更新数组 A 为 $A = A[1, 2...\frac{k}{2}]$, 更新数组 B 为 $B = B[\frac{k}{2}+1, ..., n]$, 反之如果 $B[\frac{k}{2}]$ 比较大亦然，再递归的对缩小规模的 A、B 数组进行选择阶为 $\frac{k}{2}$ 的元素。

算法的复杂度为:

$$T(n) = T(\frac{n}{2}) + O(1)$$

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

算法伪代码见:**Algorithm 2**

2.

算法思路: 我们取各个数组的中位数，将他们的下标之和与 k 比较：若大于 k，则舍弃中位数最大的那个数组的后一半进行递归；若小于 k，则舍弃中位数最小的那个数组的前一半进行递归；若等于 k，则是 k 阶元素是当前所有数组的最大中位数，已经找到。

3.

同上。只是数组个数不同而已。

Problem 3.8

算法思路：维护一个最大堆和一个最小堆，最大堆中存放小于等于中位数的数（包括中位数本身），最小堆中存放大于中位数的数。这样有

| 最大堆 |=| 最小堆 | 或 | 最小堆 |+1。

插入元素：当新来一个元素的时候，判断他与中位数 (最大堆顶) 大小关系并插入对应的堆，此时如果两个堆的大小关系不满足上式，我们就将元素多的那个堆的堆顶取出并放入另外一个堆中，并对两个堆作相应修复。

时间复杂度为 $T(n) = O(2\log n) = O(\log n)$

发现中值: 最大堆堆顶即为当前中位数。

删除中值：删除中位数 (最大堆堆顶)，将最小堆堆顶取出并加入最大堆中，并对两个堆进行修复。

Problem 3.9

1.

当 $\omega_i = \frac{1}{n}$ 时，一共有 $\lceil \frac{n}{2} \rceil - 1$ 个数小于中位数，

有 $\lfloor \frac{n}{2} \rfloor$ 个数大于中位数，而对于中位数有

$$\sum_{x_i<x_k} \omega_i = (\lceil \frac{n}{2} \rceil - 1) * \frac{1}{n} < \frac{1}{2}$$

$$\sum_{x_i>x_k} \omega_i = \lfloor \frac{n}{2} \rfloor * \frac{1}{n} \leq \frac{1}{2}$$

满足加权中位数的定义。

2.

首先对这 n 个数进行从小到大的排序，然后从头开始，累加他们的权直到权之和刚刚大于等于 $\frac{1}{2}$, 其中" 压死骆驼的最后一根稻草" 就是加权中位数。

算法复杂度为：$O(n \log n + n) = O(n \log n)$

3.

Algorithm 3 WEIGHT_MEDIAN(A, p, r)

Input: A 为不相同的数，设定 weight(x) 可以得到 x 的权重

1: **if** $p == r$ **then**
2: **return** A[p]
3: **end if**
4: q=MEDIAN(A, p, r)
5: //线性选择传统中位数算法，返回 A[p...r] 的中位数的下标。注意到线性选择的同时，我们也完成了 partition 的操作。
6: $W_L = \sum_{i \in [1,q-1]} weight(A[i])$
7: **if** $W_L < \frac{1}{2} \&\& W_L + weight(A[q]) \geq \frac{1}{2}$ **then**
8: **return** A[q]
9: **else if** $W_L < \frac{1}{2}$ **then**
10: $weight(A[q]) += W_L$
11: **return** WEIGHT_MEDIAN(A,q,r)
12: **else**
13: $weight(A[q]) += 1 - W_L - weight(A[q])$
14: **return** WEIGHT_MEDIAN(A,p,q)
15: **end if**

算法思路：首先选出传统中位数，同时将整个数组 partition 为左右两个子数组，然后分治，其中注意要把被舍弃的一半的权重之和加在中位数上防止权重出现问题。算法伪代码见 **Algorithm 3**，调用时对于 $A[1...n]$, 令 $p = 1, r = n$

算法复杂度：$T(n) = T(n/2) + O(n)$,$T(n) \in O(n)$

Problem 4.1

采用折半查找的思想，每次取中间的元素 x，设他前面的元素为 a, 后面的元素为 b，分三种情况:

case 1: x>a&&x>b 最大元素就是 x

case 2: b>x>a 对数组右半边进行递归

case 3: a>x>b 对数组左半边进行递归

Problem 4.4

1.

考虑一个负载因子 α, 则闭哈希表空间消耗为 $(1+2\alpha)h_c$, 当同样的空间用于开散列，它的负载因子易算得为 $\frac{\alpha}{1+2\alpha}$。

α	空间消耗	开散列表负载因子
0.25	$1.5h_c$	0.17
0.5	$2h_c$	0.25
1.0	$3h_c$	0.33
2.0	$5h_c$	0.4

2.

考虑一个负载因子 α, 则闭哈希表空间消耗为 $(1+5\alpha)h_c$, 当同样的空间用于开散列，它的负载因子易算得为 $\frac{4\alpha}{1+5\alpha}$。

α	空间消耗	开散列表负载因子
0.25	$2.25h_c$	0.44
0.5	$3.5h_c$	0.57
1.0	$6h_c$	0.67
2.0	$11h_c$	0.73

Problem 4.6

1.

每次取出矩阵中右上角的元素 y, 比较 x 和 y 的大小:

case 1 :x==y 已经找到

case 2 :x>y 舍去矩阵的第一行，再进行递归

case 3 :x<y 舍去矩阵的最后一列，再进行递归

当矩阵只剩一行，仍要舍去该行时，抑或是只剩一列，仍要舍去该列时，则无法找到。

2.

算法 worst case 分析: 可以看到本算法每次比较消去一列或一行, 而矩阵为 $m*n$ 规模，算法通过 $m+n-2$ 次比较将规模降为 $1*1$ 后，仍需要一次确认最后一个元素是不是要找的 x, 所以最坏情况下需要比较 $m+n-1$ 次。

Problem 4.7

(a)

维护一棵红黑树，对 n 个元素依次作如下操作: 不妨当前元素为 x

step 1: 在已有的红黑树中查找关键码为 $a-x$ 的结点是否存在。若存在，则已经找到，若不存在进入 *step 2*。

step 2: 将 x 插入红黑树

当 n 个元素全部操作完，仍然没找到，则 S 中不存在两个元素的和为 a。

复杂度分析: 红黑树的插入、查找均为 $O(\log n)$，最坏情况下进行 n 次插入和查找，所以整体复杂度为 $O(n\log n)$

(b)

利用 head、rear 两个变量指示，考虑 S[head]+S[rear] 的和与 a 的大小关系，移动相应的指示变量。算法伪代码见 **Algorithm 4**

Algorithm 4 FIND_SUM_A($S[0...n]$)

```
head = 0
rear = n − 1
while head<rear do
    if S[head] + S[rear] > a then
        rear = rear − 1
    else if S[head] + S[rear] < a then
        head = head + 1
    else
        //S[head] + S[rear] = a
        return [head, rear]
    end if
end while
return CANTFIND
```

Problem 4.8

(a)

利用数学归纳法对树高度进行归纳。如果 T 的高度为 0，则 T 必为 NIL，此时有 0 个内部结点，满足 $0 \geq 2^h - 1 = 0$。对于归纳步骤，考虑一个高度为正值且有两个子树的 T，每个子树有黑高 h 或 h-1, 取决于自身的颜色是红还是黑。由于 T 的子树的高度比 T 的高度要低，可以利用归纳假设得出每个子节点至少有 $2^{h-1}-1$ 个内部黑色节点。于是子树 T 至少包含 $(2^{h-1}-1)+(2^{h-1}-1)+1 = 2^h-1$ 个内部黑色节点。因此得证。

(b)

利用数学归纳法对树高度进行归纳。如果 T 的高度为 0，则 T 必为 NIL，此时有 0 个内部结点，满足 $0 \leq 4^h - 1 = 0$。对于归纳步骤，考虑一个高度为正值且有两个子树的 T，为了获得最多的内部节点数，两个子树应都是准红黑树以充分利用原红黑树的黑高，此时内部结点数 = 根节点 + 双子树结点 + 双子树的四个儿子子树（不妨叫孙子子树）。而这四个孙子子树的高度比 T 的高度要低，可以利用归纳假设得出每个孙子子树至多有 $4^{h-1}-1$ 个内部节点。内部节点数 $\leq 1+2++4*(4^{h-1}-1) = 4^h-1$。因此得证。

(c)

由红黑树性质，红结点的两个子节点是黑色。所以从根节点到任意黑色节点的任何一条简单路径上都至少有一半的结点为黑色。因此得证。

(d)

可以利用 **(a)** 中的结论。准红黑树的两个子树一定是红黑树，且有黑高为 $h-1$, 则 A 至少有的内部黑色结点数为两子树至少有的黑色结点数之和 $2^{h-1}-1+2^{h-1}-1=2^h-2$

(e)

可以利用 **(b)** 中的结论。准红黑树的两个子树一定是红黑树，且有黑高为 $h-1$, 则 A 至多有的内部结点数为两子树至多有的结点数之和加上根节点为 $4^{h-1}-1+4^{h-1}-1+1=\frac{4^h}{2}-1$

(f)

与 **c** 同理，RBT 和 ARBT 均满足红结点的两个子节点是黑节点的性质。

Problem 4.9

$$C_n^1*(\frac{1}{n})^k=\left(\frac{1}{n}\right)^{k-1}$$

Problem 4.10

利用并查集，首先处理所有等式约束，对于 $a=b$, 在并查集中 union(a,b); 然后处理所有不等式约束，对于 $a\neq b$, 在并查集中检查是否有 $find_set(a)\neq find_set(b)$, 如果该式不成立，则约束不可同时满足，检查完所有不等式约束后则可以 m 个约束同时满足。

Problem 4.12

首先我们容易得到 n 位比特数的加法的复杂度为 $O(1)$,n 位比特的乘法的复杂度为 $O(n)$.

算法为:

设

$$2^{\frac{n-1}{2}}\leq\sqrt{N}\leq 2^{\frac{n}{2}}$$

因此有

$$2^{\lfloor\frac{n-1}{2}\rfloor}\leq\lceil\sqrt{N}\rceil\leq 2^{\lceil\frac{n}{2}\rceil}$$

$$t_1\triangleq\lfloor\frac{n-1}{2}\rfloor+1\leq t\leq\lceil\frac{n}{2}\rceil+1\triangleq t_2$$

在 2^{t_1} 和 2^{t_2} 之间进行二分查找 $\left\lceil\sqrt{N}\right\rceil$)，对于每个 x, 利用 $O(n)$ 的时间计算出他的平方。

算法复杂度分析: 我们可列出递归式 $T(n)=T(n-1)+O(n)$, 解递归易得 $T(n)\in O(n^2)$

Problem 4.13

使用聚合分析的方法，n 个操作的代价之和为

$$\sum_{i=1}^{n}c_i=\sum_{j=0}^{\lfloor\log n\rfloor}\left\lfloor\frac{n}{2^j}\right\rfloor 2^j\leq n(\lfloor\log n\rfloor+1)$$

所以，每个插入操作的平摊代价为 $1+\lfloor\log n\rfloor$