算法设计与分析作业三

作者：吴润泽　　　**学号**：181860109

Email：181860109@smail.nju.edu.cn

2020 年 3 月 21 日

目录

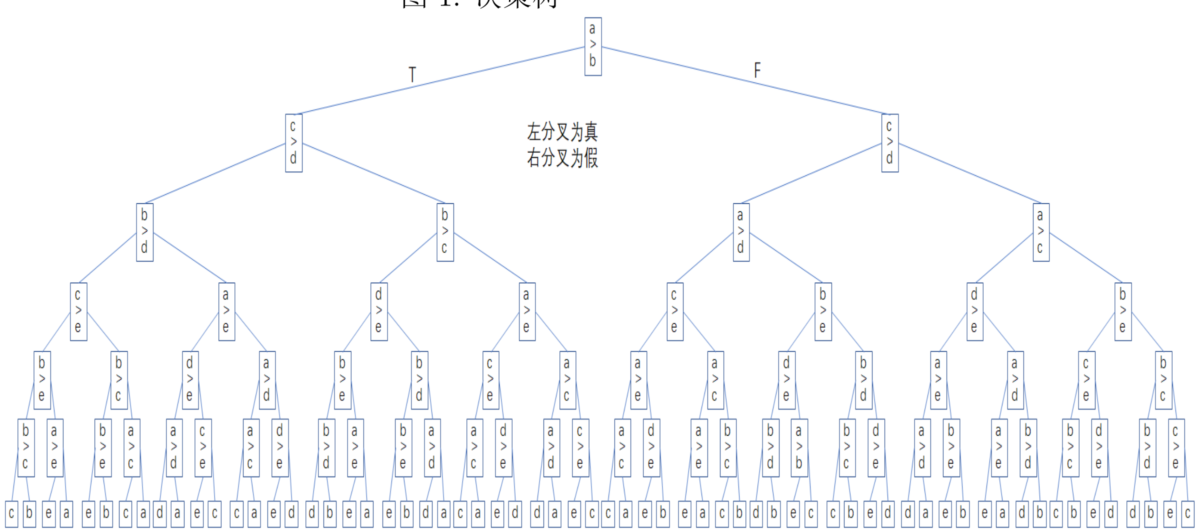
SELECTION

problem 8.2

算法 假设有 5 个数 a，b，c，d，e

1. 比较 a b，将较小者放入 a，较大者放入 b
2. 比较 c d，将较小者放入 c，较大者放入 d
3. 比较 a c，将较小者放入 a，较大者放入 c，若 a 和 c 发生交换则同时交换 b 和 d，即保证 a b 和 c d 原有大小关系不变。此时有 $a < c < d, a < b$，则 a 不可能是中位数
4. 比较 b e，将较小者放入 b，较大者放入 e
5. 比较 b c，将较小者放入 b，较大者放入 c，若 b 和 c 发生交换则同时交换 d 和 e，即保证 b e 和 c d 原有大小关系不变。此时有 $b < c < d, b < e$，则 b 不可能是中位数
6. 比较 c e，将较小者放入 c,较大者放入 e,此时有 $c < d < e$,且 $c > b, c > a$，即 c 就是中位数。

图 1: 决策树



problem 8.4

算法 设数组 array 元素个数为 n，不妨设 n 为奇数，阶为 k 的元素为 m；

1. 当 k 等于 $\frac{n+1}{2}$ 时，直接调用算法 A，即可找到 m；
2. 当 k 小于 $\frac{n+1}{2}$ 时，遍历数组 array，记录其元素最小值为 min，对于原数组 array，有 $n-k$ 个元素大于 m，$k-1$ 个元素小于 m。
3. 开辟新数组 temp，将前 $n-2k+1$ 个元素值赋为 min-1，并将 array 的 n 个元素放入其后。则数组 temp 中有 $n-k$ 个元素大于 m，$n-k$ 个元素小于 m。因此 m 为 temp 中位数，调用算法 A，即可找到 m。
4. 当 k 大于 $\frac{n+1}{2}$ 时，同理记录其元素最大值为 max，开辟数组 temp，在数组 temp 中前 $2k-n-1$ 个赋值为 max+1。同样使得 m 变为 temp 中位数，调用算法 A，即可找到 m。

算法 1 $KthFind$ 算法

Function $KthFind\ (array, n, k)$
    if $k == \dfrac{n+1}{2}$ **return** $A(array, n)$
    $min\_num := min(array, n), max\_num := max(array, n)$
    Let $temB[1...2n]$ *be new array.*
    for $i := 1$ *to* n **do**
        $temB[i] := A[i]$
    if $k < \dfrac{n+1}{2}$ **then**
        for $i := 1$ *to* $n-2k+1$ **do**
            $temB[i+n] := min\_num - 1$
        return $A(temB, 2n-2k+1)$
    if $k > \dfrac{n+1}{2}$ **then**
        for $i := 1$ *to* $2k-n-1$ **do**
            $temB[i+n] := max\_num - 1$
        return $A(temB, 2k-1)$

时间复杂度 寻找最值和开辟新数组的时间复杂度为 $O(n)$，添加的元素个数最多为 $n-1$ 个，数组的规模仍为 $O(n)$，而找中位数算法 A，时间复杂度为 $O(n)$，因此总的时间复杂度仍为线性。

problem 8.5

(1)

使用归并排序进行降序排列，排序结果前 k 个元素即为所求。
归并排序时间复杂度为 $O(n\log n)$，输出复杂度为 $O(k)$，总时间复杂度为 $O(n\log n)$.

1.**Function** $KthOrder1$ $(array, n, k)$

2. 对原数组使用归并排序，并按照升序排列

3. **for** $i := 1$ *to* k **do**

4. $res.add(array[i])$

5. **return** res

(2)

根据原数组建立最大堆，最大堆堆顶存储当前堆的最大值，则弹出堆顶元素 k 次，k 个元素即为所求。
根据已有序列建堆的时间复杂度为 $O(n)$，弹出堆顶元素 k 次，每次修复堆时间复杂度为 $O(\log n)$，则总的时间复杂度为 $O(n+k\log n)$.

1.**Function** $KthOrder2$ $(array, n, k)$

2. 根据原数组进行建堆，得到最大堆为 heap

3. **for** $i := 1$ *to* k **do**

4. $res.add(heap.top())$

5. $heap.pop()$\\弹出堆顶元素，并修复

6. **return** res

(3)

利用 problem 8.4 中 Kth-find 算法，找到原数组中的第 k+1 大元素 m。遍历原数组，找到其中大于 m 的元素加入结果数组 res。对 res 使用归并排序，升序排列，得到 res 即为所求。

找第 k+1 大，遍历原数组为线性时间 $O(n)$，对 res 归并排序 $O(k\log k)$，总时间复杂度为 $O(n+k\log k)$.

1.**Function** $KthOrder3$ $(array, n, k)$

2. $m := KthFind(array, n, k+1)$\\\\找到数组第k+1大元素

3. **for** $i := 1$ *to* n **do**

4. **if** $array[i] > m$ **do** $res.add(array[i])$

5. 对 res 数组使用归并排序，并按照升序排列

6. **return** res

problem 8.6

(1)

使用归并排序进行排列时间复杂度为 $O(n\log n)$，遍历中位数 M 左右两侧元素，与 M 差值最小的加入 res，并移动对应侧指针，直至 res 个数为 k 即可。总时间复杂度为 $O(n\log n+k)$.

1.**Function** $KthNear1$ $(array, n, k)$

2. 对原数组使用归并排序

3. $l := \dfrac{n+1}{2} - 1,\ r := \dfrac{n+1}{2} + 1$

4. **for** $i := 1$ *to* k **do**

5. **if** $array[l] + array[r] > 2M$ **do** $res.add(array[r++])$

6. **else** $res.add(array[l--])$

7. **return** res

(2)

利用 problem8.4 中的查找中位数算法，得到中位数为 M，时间复杂度为 $O(n)$。 将原数组分为大于 M 和小于 M 的两数组 L,S，时间复杂度为 $O(n)$。 与 problem 8.5(3) 同样思想，找到 L 的前 k 小 (查找第 k+1 小，即第 n-k-1 大)LK 和 S 的前 k 大元素 SK，时间复杂度为 $O(k)$。对 LK 进行归并排序按照升序排列，对 SK 进行归并排序按照降序排列，时间复杂度为 $O(k \log k)$。之后遍历 LK 和 SK 找到与 M 最接近的 K 个元素即可。

Function $KthNear2\ (array, n, k)$

  $M := A(array, n)$\\\\找到中位数 M

  划分原数组为大于 M 和小于 M 两部分:L[1..n1],S[1..n2]

  $large := KthFind(L, n1, n1 - k - 1)$\\\\找到L数组第k+1小元素

  $small := KthFind(S, n2, k + 1)$\\\\找到L数组第k+1大元素

  找到 L 数组小于 large 的 k 个元素, 并升序排列, 得到 LK

  找到 S 数组大于 small 的 k 个元素, 并降序排列, 得到 SK

  $l := 0,\ r := 0$

  for $i := 1\ to\ k$ **do**

    if $LK[l] + SK[r] > 2M$ **do** $res.add(SK[r++])$

  else $res.add(LK[l--])$

  return res

problem 8.8

算法设计 使用两个堆，大根堆 q1 维护较小值，小根堆维护较大值，令大根堆 q2 元素个数为 m，小根堆元素个数为 n：

使得小根堆的堆顶是较大数中最小的，大根堆的堆顶是较小数中最大的；

将大于大根堆堆顶的数放小根堆，小于等于大根堆堆顶的数放大根堆；
对于大根堆的堆顶元素，有 n 个元素比该元素大，$m-1$ 个元素比该元素小；
对于小根堆的堆顶元素，有 m 个元素比该元素小，$n-1$ 个元素比该元素大；
在维护 $|m-n| \leq 1$ 之后，当 $m=n$ 时，两堆顶元素均为中位数，当 $m \neq n$ 时，元素个数较多的堆顶元素即为当前中位数；
易知插入和删除的时间复杂度均为 $O(\log n)$，查找中值为常数。

查找中值

if $(q1.size()+q2.size())\%2 == 1$ **then**
  if $q1.size() > q2.size()$ **do** $mid := q1.top()$
  else $mid := q2.top()$
else $mid := (q1.top + q2.top())/2$

插入操作

if $input > q1.top()$ **do** $q2.push(input)$
else $myq1.push(input)$\\\\大根堆放较小数，小根堆放较大数
while $|q1.size() - q2.size()| > 1$ **do**
  if $q1.size() > q2.size()$ **do** $q2.push(q1.pop())$
  else $q1.push(q2.pop())$

删除操作

if $(q1.size()+q2.size())\%2 == 1$ **then**
  if $q1.size() > q2.size()$ **do** $q1.pop()$
  else $q2.pop()$
else $q2.pop()$\\\\当为偶数时任意弹出一个

problem 8.9

(1)

对中位数 x_k，设 n 为奇数，有 $\frac{n+1}{2}-1$ 个元素小于 x_k，$\frac{n+1}{2}-1$ 个元素大于 x_k，则 $\sum_{x_i>x_k}\frac{1}{n}=\sum_{x_i<x_k}\frac{1}{n}=\frac{1}{2}-\frac{1}{2n}<\frac{1}{2}$. 对 n 为偶数，同样成立。

(2)

建立以 (w,x) 为元素的结构体数组 ori，w 为权重，x 为其对应下标。对其进行归并排序，时间复杂度为 $O(n\log n)$。遍历 ori，对权重进行累加，当权重和大于 $\frac{1}{2}$ 时，对应结构体元素的 x 即为加权中位数，时间复杂度为 $O(n)$。因此总的时间复杂度为 $O(n\log n)$。

1.**Function** $WeightMid1\ (ori,n)$
2.　对结构体数组 ori 使用归并排序
3.　$cur\_weight := 0$\\\\当前权重
4.　**for** $i := 1\ to\ n$ **do**
5.　　**if** $cur\_weight + ori[i].w > \frac{1}{2}$ **do**
6.　　　**return** $ori[i].x$
7.　　**else** \\\\更新权重和权重
8.　　　$cur\_weight := cur\_weight + ori[i].w$

(3)

参考 BFRPTR 算法　假设划分的权值和为 tar, 初始 $tar=\frac{1}{2}$

1. 将所有元素分成 $\lceil\frac{n}{5}\rceil$ 组，每组 5 个元素 (最后一组可能不足 5 个元素)
2. 寻找 $\lceil\frac{n}{5}\rceil$ 组中每一组的中位数，可对每一组进行排序
3. 对于找出的 $\lceil\frac{n}{5}\rceil$ 个中位数递归进行 1,2 直到剩下一个数即为中位数，记为 m
4. 基于 m 对元素进行划分，假设有 x 个元素小于 m，n-x-1 个元素大于 m
5. 计算 x 个元素的权值和 T，若权值和 T=tar，则 m 为加权平均数
6. 若权值和 T>tar, 则在 x 个元素中递归寻找中位数，tar 值不变

7. 若权值和 T<tar, 则在 n-x-1 个元素中递归寻找中位数，tar=tar-T
8. 时间复杂度 $T(n) = T(n/5) + T(7n/10) + O(n) = \Theta(n)$，满足要求。

算法实现

Function $Partion$ (ori, l, r)\\根据 ori[l] 进行划分
    $i := l, j := r, pivot := ori[l]$
    while $i < j$:
        while $ori[j].w <= mid.w$ **and** $i < j$: $ori[i] := ori[--j]$
        while $ori[i].w >= mid.w$ **and** $i < j$: $ori[j] := ori[--i]$
    $ori[i] := pivot$
    return i

Function $FindMid$ (ori, l, r)\\寻找中位数的中位数
    $n := 0$\\计算中位数个数
    for $(i := l; r - 5; i+ = 5)$
        $sort(ori, i, i + 4)$\\对 5 个数进行排序
        $n := i - l,\ swap(ori[l + n/5], ori[i + 2])$\\将该组中位数放在数组头
    \\相同办法处理数组剩余元素
    if $n/5 == l$ **return** $ori[l]$
    return $FindMid(ori, l, l + n/5)$

Function $BFPTR$ (ori, l, r, tar)
    $FindMin(ori, l, r)$\\寻找权值中位数 m 和其下标 x
    $pos := Partion(ori, l, r)$\\根据 m 来划分 ori 为小于和大于 m 两部分
    $cur := 0$
    for $i := l$ *to* pos **do** \\计算 l-pos 的权重
        $cur := cur + ori[i].w$
    if $cur == tar$ **return** $ori[i].x$
    elif $cur < tar$ **return** $BFPTR(ori, pos + 1, r, tar - cur)$
    else return $BFPTR(ori, 0, pos - 1, tar)$

SEARCH

problem 9.4

引理 9.1 证明：

(1) 数学归纳法

当 h=0 时，内部节点有 0 个，成立

假设当 h=n-1 时，内部黑色节点个数 $x \geq 2^{n-1} - 1$ 个

当 h=n 时，为满足每条路径黑色深度相等，即最底层外部结点全部替换为黑色内部节点，增加 2^{n-1} 个黑色内部节点，所以 $x + 2^{n-1} \geq 2^n - 1$，成立。

(2)

内部节点最多时，任何一条路径上均为黑红节点交替，黑色节点位于首尾共 h+1 个，因此每条路径的红色节点为 h 个，因此树的高度 (抛去外部节点) 为 2h。且满足完美二叉树性质，内部节点个数为 $2^{2h} - 1 = 4^h - 1$，成立。

(3) 反证法

假设存在黑色节点的普通高度超过其黑色高度的 2 倍，设其黑色高度为 k，则总高度超过 2k，亦即红色结点个数超过 k，则必然存在红色结点在路径上连续，与定义相矛盾，故任何黑色节点的普通高度至少是其黑色高度 2 倍。

引理 9.2 证明：

(1)

由红黑树定义，ARB_h 可看作由红色根节点和两棵 RB_{h-1} 树 (原为子结点的黑色节点变为根节点) 组成。由引理 9.1(1)RB_{h-1} 内部黑色节点至少为 $2^{h-1} - 1$，因此 ARB_h 内部黑色节点至少为 $2(2^{h-1} - 1) = 2^h - 2$，成立。

(2)

同样利用 ARB_h 由红色根节点和两棵 RB_{h-1} 树组成。由引理 9.1(2)RB_{h-1} 内部节点最多为 $4^{h-1} - 1$，因此 ARB_h 内部节点最多为 $2(4^{h-1} - 1) + 1 = \frac{4^h}{2} - 1$，成立。

(3) 反证法

与引理 9.1(3)证法相同，假设存在黑色节点的普通高度超过其黑色高度的 2 倍，则必然存在红色结点在路径上连续，与定义相矛盾，故任何黑色节点的普通高度至少是其黑色高度 2 倍。

problem 9.6

算法设计 采取二分查找的方法，设访问的元素为 $A[i]$，res 存放结果，l, r 记录查找区间；
若 $A[i] = i$，说明 $1 - i$ 没有缺少元素，则向右半边递归查找；
若 $A[i] > i$，说明 $1 - i$ 已经缺少元素，更新 $res := i$，向左半边递归查找；
当 $l > r$ 时，递归结束返回 res 即可。

算法实现

算法 1 $FindLeastNum$ 算法

Function $FindLeastNum$ $(A[1 \cdots n])$
  $l := 1, r := n$
  while $l \leq r$ **do**
    $mid := (r - l)/2 + l$
    if $A[mid] > i$ **then**
      $res := i, r := mid - 1$
    else
      $l = mid + 1$
  return res

时间复杂度 较易分析，每次缩小区间为原来的一半，因此查找时间复杂度为 $O(\log n)$。

problem 9.8

算法设计 矩阵为 A，当前行为 row，当前列为 col，待查找元素为 tar

1. 从右上角开始遍历，当 $A[row][col] > tar$，说明不在这一列中，col 减 1；
2. 当 $A[row][col] < tar$，说明不在这一行中，row 加 1；
3. 当 $A[row][col] = tar$，查找成功。否则当查找结束，返回查找失败。

算法实现

算法 2 *SearchMatrix* 算法

Function $SearchMatrix\ (A[1\cdots m][1\cdots n], tar)$

    $row := 1, col := n$

    while $row \le n$ **and** $col \ge 0$ **do**

        if $(A[row][col] > tar)\ col--$

        if $(A[row][col] < tar)\ row++$

        else return $true$

    return $false$

最坏情况 在每次比较中都可以排除一行或者一列，共有 $(m+n)$ 行列，当行或列减到 0 查找失败，因此最坏情况下需要比较 $m+n-1$ 次。

problem 9.12

(1)

算法设计 采取二分查找方法，设访问的元素为 $A[mid]$，l, r 记录区间；

1. 若 $A[mid-1] \leq A[mid] \leq A[mid+1]$，找到一个局部最小元素返回即可。
2. 若 $A[mid] > A[mid+1]$，同时 $A[n-1] \leq A[n]$，$mid-n$ 区间满足原数组性质，则向右半部分递归查找。
3. 当 $A[mid-1] < A[mid]$ 时，同理向左半部分递归查找。

算法 2 $SearchNeighborMin$ 算法

```
1.Function SearchNeighborMin (A[1···m][1···n])
2.    l := 1, r := n
3.    while l ≤ r do
4.        mid := (r − l)/2 + l
5.        if (A[mid] > A[mid − 1]) r := mid − 1
6.        elif A[mid] > A[mid + 1] l := mid + 1
7.        else return mid
```

(2) 证明

数组中一定存在最小值，当最小值不为边界时，即满足 $A[i+1] \geq A[i] \leq A[i-1]$，即一定是局部最小元素。

当最小值为边界元素时，又因为满足 $A[1] \geq A[2], A[n-1] \leq A[n]$ 则:

1. 当 $A[1]$ 为最小值时，一定有 $A[1] = A[2] \leq A[3]$，$A[2]$ 为局部最小元素
2. 当 $A[n]$ 为最小值时，一定有 $A[n-2] \geq A[n-1] = A[n]$，$A[n-1]$ 为局部最小元素。故数组中至少存在一个局部最小元素。

HASH

problem E1

(1)

闭散列 设其负载因子为 $\alpha, \alpha \in \{0.25, 0.5, 1.0, 2.0\}$，其初始空位为 h_C，则链表节点个数为 αh_C，因此其空间消耗为 $2\alpha h_C$。因此当 α 分别为 0.25,0.5,1.0,2.0 时，闭散列空间消耗分别为：$\begin{cases} \mathbf{0.25} : 0.5h_C \\ \mathbf{0.5} : h_C \\ \mathbf{1.0} : 2h_C \\ \mathbf{2.0} : 4h_C \end{cases}$

开散列 设其负载因子为 α_1，则其关键字个数为 $\alpha_1 h_C$，其空间消耗为 $\alpha_1 h_C$，即 $\alpha_1 h_C = 2\alpha h_C \Rightarrow \alpha_1 = 2\alpha$，则当闭散列负载因子 α 分别为 0.25,0.5,1.0,2.0 时，开散列对应的负载因子 α_1 分别为：

$$\mathbf{0.5,1,2,4}.$$

(2)

闭散列 同理其空间消耗为 $5\alpha h_C$。因此当 α 分别为 0.25,0.5,1.0,2.0 时，闭散列空间消耗分别为：$\begin{cases} \mathbf{0.25} : 1.25h_C \\ \mathbf{0.5} : 2.5h_C \\ \mathbf{1.0} : 5h_C \\ \mathbf{2.0} : 10h_C \end{cases}$

开散列 同理其空间消耗为 $4\alpha_1 h_C$，即 $4\alpha_1 h_C = 5\alpha h_C \Rightarrow \alpha_1 = \frac{5}{4}\alpha$，开散列对应的负载因子 α_1 分别为：

$$\mathbf{0.3125,0.625,1.25,2.5}.$$

problem E2

平摊分析 假设输入的规模为 n，由数组 i 的大小为 2^i 可知，最后一个不为空的数组的下标 $x=\lfloor \log n \rfloor$，即每个元素插入后最多下沉 $\lfloor \log n \rfloor$ 次。
对于插入，其 $Actual\ Cost=1$，记 $AccountingCost=\log n$
当元素下降至第 m 层时，其总的下降消耗即为从第 0 层下降至第 m 层，即元素的消耗不超过 $\lfloor \log n \rfloor$。

操作	**Amortized Cost**	**Actual Cost**	**Accounting Cost**
Insert	$1+\log n$	1	$\log n$

因此插入 n 次的总的消耗为 $n(1+\log n)=n+n\log n$，则插入 n 次的时间复杂度为 $O(n\log n)$。

problem E3

(1)

设两个栈为 A，B

Enqueue 将新元素压入 B 中;

Dequeue 如果 A 不为空，从 A 中弹出元素；如果 A 为空，B 中所有元素依次出栈并压入 A 中，即使得 A 中弹栈顺序变为原始序列输入顺序.

(2)

比较简单的 Dequeue(即 A 不为空) 的 $Actual\ Cost = 1 + 1 = 2$，即判断 empty 和 pop；

而比较复杂的 Dequeue，设 B 中元素个数为 t，则 $Actual\ Cost = 1 + 2t + 1 = 2t + 2$，判断 empty，B 中 pop t 个，向 A push t 个，最后 A pop。

对于 Enqueue，$Actual\ Cost = 1$，记 $Accounting\ Cost$ 为 3，即之后的判空，pop 和 push 的代价。

则对于复杂的 Dequeue 的 $Accounting\ Cost = -3 \times t = -3t$，即减去在 Enqueue 中的 $Accounting\ Cost$。

操作	Amortized Cost	Actual Cost	Accounting Cost
Enqueue	4	1	3
Dequeue(easy)	2	2	0
Dequeue(diff)	2	$3t + 2$	$-3t$

即两种情况下 Dequeue 的消耗相同，若规模为 n，入队出队平摊分析法得到的时间复杂度为：

$$Enqueue(n) + Dequeue(n) = O(6n).$$