PS4

201300069 邓嘉宏

Problem 1

(a)

```
BuildMaxHeap(data[1...n])
    data_size=n
    for i=floor(data_size/2) down to 1
        MaxHeapify(i)
```

输入数组$[5(1),13(2),2(3),25(4),7(5),17(6),20(7),15(8),4(9)]$, $()$中表示该元素的下标.$step1$: 对于$id(4)=25$, 因为$id(8)$和$id(9)$均小于25, 所以不交换, 原数组不变.

$step2$: 对于$id(3)$, 因为$id(7)>id(6)>id(3)$, 所以交换$id(3)$和$id(7)$, 得到数组$[5(1),13(2),20(3),25(4),7(5),17(6),2(7),15(8),4(9)]$.

$step3$: 对于$id(2)$, 因为$id(4)>id(2)>id(5)$, 所以交换$id(2)$和$id(4)$, 得到数组$[5(1),25(2),20(3),13(4),7(5),17(6),2(7),15(8),4(9)]$, 此时又因为$id(8)>id(4)>id(9)$, 所以交换$id(4)$和$id(8)$, 得到数组$[5(1),25(2),20(3),15(4),7(5),17(6),2(7),13(8),4(9)]$.

$step4$: 对于$id(1)$, 因为$id(2)>id(3)>id(1)$, 所以交换$id(2)$和$id(1)$, 得到数组$[25(1),5(2),20(3),15(4),7(5),17(6),2(7),13(8),4(9)]$, 此时又因为$id(4)>id(5)>id(2)$, 所以交换$id(4)$和$id(2)$, 得到数组$[25(1),15(2),20(3),5(4),7(5),17(6),2(7),13(8),4(9)]$, 此时又因为$id(8)>id(4)>id(9)$, 所以交换$id(8)$和$id(4)$, 得到数组$[25(1),15(2),20(3),13(4),7(5),17(6),2(7),5(8),4(9)]$, 即为最终结果.

(b)

```
HeapSort(data[1…n]):
    heap = BuildMaxHeap(data[1…n])
    for i=n down to 2
        cur_max = heap.HeapExtractMax()
        data[i] = cur_max
```

对于(a)中构建好的堆$[25(1),15(2),20(3),13(4),7(5),17(6),2(7),5(8),4(9)]$, $step1$: $cur\_max=id(1)=25, id(1)=id(9)$, 得到$[4(1),15(2),20(3),13(4),7(5),17(6),2(7),5(8),4(9)]$, 此时因为$id(3)>id(2)>id(1)$, 交换$id(1)$和$id(3)$, 得到$[20(1),15(2),4(3),13(4),7(5),17(6),2(7),5(8),4(9)]$, 此时因为$id(6)>id(3)>id(7)$, 交换$id(3)$和$id(6)$, $id(9)=cur_max$得到$[20(1),15(2),17(3),13(4),7(5),4(6),2(7),5(8),25(9)]$.

$step2$: $cur_max=id(1)=20, id(1)=id(8)$, 得到$[5(1),15(2),17(3),13(4),7(5),4(6),2(7),5(8),25(9)]$, 此时因为$id(3)>id(2)>id(1)$, 交换$id(1)$和$id(3)$, 得到$[17(1),15(2),5(3),13(4),7(5),4(6),2(7),5(8),25(9)]$, 此时因为$id(3)>id(6)>id(7)$, 所以不交换, $id(8)=cur\_max$, 得到$[17(1),15(2),5(3),13(4),7(5),4(6),2(7),20(8),25(9)]$.

$step3$: $cur\_max=id(1)=17, id(1)=id(7)=2$, 得到$[2(1),15(2),5(3),13(4),7(5),4(6),2(7),20(8),25(9)]$, 此时因为$id(2)>id(3)>$

$id(1)$, 交换$id(2)$和$id(1)$, 得到$[15(1), 2(2), 5(3), 13(4), 7(5), 4(6), 2(7), 20(8), 25(9)]$, 此时因为$id(4) > id(5) > id(2)$, 交换$id(4)$和$id(2)$, $id(7) = cur\_max = 17$, 得到$[15(1), 13(2), 5(3), 2(4), 7(5), 4(6), 17(7), 20(8), 25(9)]$.

$step4 : cur\_max = id(1) = 15, id(1) = id(6) = 4$, 得到$[4(1), 13(2), 5(3), 2(4), 7(5), 4(6), 17(7), 20(8), 25(9)]$, 此时因为$id(2) > id(3) > id(1)$, 交换$id(2)$和$id(1)$, 得到$[13(1), 4(2), 5(3), 2(4), 7(5), 4(6), 17(7), 20(8), 25(9)]$, 此时因为$id(5) > id(2) > id(4)$, 交换$id(5)$和$id(2)$, $id(6) = cur_max$得到$[13(1), 7(2), 5(3), 2(4), 4(5), 15(6), 17(7), 20(8), 25(9)]$.

$step5 : cur\_max = id(1) = 13, id(1) = id(5) = 4$, 得到$[4(1), 7(2), 5(3), 2(4), 4(5), 15(6), 17(7), 20(8), 25(9)]$, 此时因为$id(2) > id(3) > id(1)$, 交换$id(2)$和$id(1)$, 得到$[7(1), 4(2), 5(3), 2(4), 4(5), 15(6), 17(7), 20(8), 25(9)]$, $id(5) = cur_max$得到$[7(1), 4(2), 5(3), 2(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$.

$step6 : cur\_max = id(1) = 7, id(1) = id(4) = 2$, 得到$[2(1), 4(2), 5(3), 2(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$, 此时因为$id(3) > id(2) > id(1)$, 交换$id(3)$和$id(1)$, 得到$[5(1), 4(2), 2(3), 2(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$, $id(4) = cur_max$得到$[5(1), 4(2), 2(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$.

$step7 : cur\_max = id(1) = 5, id(1) = id(3) = 2$, 得到$[2(1), 4(2), 2(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$, 此时因为$id(2) > id(1) = id(3)$, 交换$id(2)$和$id(1)$, 得到$[4(1), 2(2), 2(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$, $id(3) = cur_max$得到$[4(1), 2(2), 5(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$.

$step8 : cur\_max = id(1) = 4, id(1) = id(2) = 2$, 得到$[2(1), 2(2), 5(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$, 此时因为$id(2) = id(1)$, 不交换, $id(2) = cur_max$得到$[2(1), 4(2), 5(3), 7(4), 13(5), 15(6), 17(7), 20(8), 25(9)]$.此即结果.

Problem 2

取出所有k个列表的第一个元素组成小顶堆,然后取出小顶堆的根节点,若该根节点不是所在列表的最后一个元素,则将其所在列表的下一个元素放入小顶堆根节点并修复小顶堆,若该根节点是所在列表的最后一个元素,则将小顶堆的最后一个元素放入根节点并修复小顶堆,如此不断取出小顶堆的根节点,则完成排序.

伪代码:

```
Sort(k1[n1],k2[n2]...kk[nk])
    result=[n]
    min_heap=BuildMinHeap(k1[1],k2[1],...kk[1])
    for i=1 to n
        cur_min=Extract_Add()
        result[i]=cur_min
    return result

Extract_Add()
    heap_size=k
    cur_min=min_heap(1)
    if cur_min is not the last element in its list
        min_heap(1)=the next element of cur_min
        min_heap.heapify(1)
    else
        min_heap(1)=min_heap(heap_size)
        min_heap.heapify(1)
        heap_size--
    return cur_min
```

runtime: 由BuildMaxHeap()可知,BuildMinHeap的时间是O(n),因此,Sort(k1[n1],k2[n2]...kk[nk])d的运行时间由for循环决定. 在每次循环进行一次Extract_Add(),Extract_Add()的时间由heapify()决定 为O(lgk). 因此,算法的总运行时间为:$n \times \lg k = n \lg k$.

Problem 3

(a)

使用归纳法.

Basis: 当 n=2 时,Cruel显然能将A[1,2]成功排序.

I.H.: 假设Cruel能将A[1...k/2]成功排序.

I.S.: 当 n=k 时, 根据假设,Cruel(A[1...k/2])和Cruel(A[k/2+1...k])分别完成左半边和右半边排序. Unusual将排序好的左右半边合并成排序好的整个数组: Unusual首先交换数组的第二和第三个四分之一部分,交换后,原本排序好的左右半边的较小部分在数组的左半部分,较大部分在数组的右半部分; Unusual(A[1...k/2])和Unusual(A[k/2+1...k])将左右部分分别完成排序,此时,整个数组的最左边的四分之一部分为排序好的最小元素,最右边的四分之一部分为排序好的最大元素; 最后一步 Unusual(A[k/4+1...3k/4])完成整个数组中间部分的排序. 此时整个数组完成排序.

根据归纳法,得证.

(b)

举个反例. 如输入数组[4,3,2,1], 若去掉for循环后,最后得到的数组为[3,1,4,2],并未成功排序.

(c)

举个反例. 如输入数组[4,3,2,1], 交换Unusual的最后两行后, 最后得到的数组为[1,3,2,4], 并未成功排序.

(d)

runtime of Unusual: 由伪代码可得递归式: T(n)=3T(n/2)+O(n). 由递归树可知一共有 $\lg n + 1$层, 第i层的和为 $3^{i-1}n$, 所以 $T(n) = n + 3n + ... + 3^{lgn}n = O(n^{1+\log_3 2})$

runtime of Cruel: 由伪代码可得递归式: $T(n) = 2T(n/2) + O(n^{1+\log_3 2})$. 由递归树可知一共有 $\lg n + 1$层, 第i层的和为 $2^{i-1}n^{1+\log_3 2}$, 所以$T(n) = n^{1+\log_3 2} + 2n^{1+\log_3 2} + ... + 2^{lgn}n^{1+\log_3 2} = O(n^{2+\log_3 2})$.

Problem 4

(a)

使用归纳法.

Basis: 当 n=1 时,显然成立.

I.H. 假设当 $n \leq k-1$时, 能成功排序.

I.S. 当 n=k 时, 根据假设,TRQuickSort(A,p,q-1)能成功排序,所以while循环每次迭代前, A[1...q]已完成排序,又因为q不断接近r,并且最终等于r,所以A[1...k]最终能成功排序.

根据归纳法,得证.

(b)

若每次Partition时取的主元是待排序部分的最大元素, 那么右半部分子数组的规模为0, 左半部分一共将进行 n-1 次递归调用, 所以递归深度为$\Theta(n)$

(c)

```
M_TRQuickSort(A, p, r)
    while p < r
        q = Partition(A, p, r)
        if q < floor((p + r) / 2)
            M_TRQuickSort(A, p, q - 1)
            p = q + 1
        else
            M_TRQuickSort(A, q + 1, r)
            r = q - 1
```

M_TRQuickSort每次对小于原数组规模一半的子数组进行递归调用. 在最坏情况下,每次Partition将原数组分成规模几乎相等的两部分,此时递归调用最多需要$\lg n$次,所以,递归深度为$\Theta(\lg n)$.

Problem 5

(a)

递归调用函数,每次让$\sqrt{n}/2$个排序好的最大数到数组的最右边.

```
Sort(A[1...n])
    k=0
    while k<=n-sqrt(n)
        SqrtSort(k)
        k+=sqrt(n)/2
    Sort(A[1...(n-sqrt(n)/2)])
```

正确性:

运用归纳法.

Basis: 当m=1,显然成立.

I.H.: 假设当 $m = n - \sqrt{n}/2$ 时,成功排序.

I.S.: 当 $m = n$时. 在while循环的每次迭代开始前,$A[k + \sqrt{n}/2 ... k + \sqrt{n}]$是$A[1...k + \sqrt{n}]$ 中排序好的最大元素,迭代停止时,$k = n - \sqrt{n}$,所以此时$A[n - \sqrt{n}/2...n]$是A[1...n]中排序好的最大元素, 根据假设,整个数组完成排序.

根据归纳法,得证.

times:

最坏情况下,while循环调用$2\sqrt{n}$次SqrtSort(k),一共需要调用SqrtSort(k):$1 + 2 + ... + 2\sqrt{n - \sqrt{n}/2} + 2\sqrt{n} = 4n(次)$.

(b)

$T(n) = T(n - \sqrt{n}/2) + 2\sqrt{n},$
根据递归树, 可知一共有$n/(\sqrt{n}/2)$层, $T(n) = 2\sqrt{n} + 2\sqrt{n - \sqrt{n}/2} + ... + 1 + 2 = 4n = O(n).$

Problem 6

(a)

设OneInThree返回0的概率为m,则有$1 - m = 1/2 \times m$,解得m=2/3,所以OneInThree返回1的概率为1/3.

(b)

调用 i 次 FairCoin的概率为 $(1/2)^i$,所以预期调用次数为 $E(X)=\sum_{i=1}^{\infty}i(1/2)^i$.

(c)

调用两次BiasedCoin生成x,y,若x,y不相等则返回x,若相等则继续调用两次BiasedCoin生成x,y,如此循环直至返回某个x.

伪代码:

```
OneInTwo
    x=BiasedCoin(),y=BiasedCoin()
    while(x==y)
        x=BiasedCoin()
        y=BiasedCoin()
    return x
```

(d)

$P(x=0,y=1)=P(x=1,y=0)=p(1-p)$

生成 i 次x,y的概率为:
$(2p^2-2p+1)^{i-1}\times(2p-2p^2)$

生成一次x,y需要调用两次 BiasedCoin,
所以预期调用BiasedCoin的次数为:$E(X)=2\sum_{i=1}^{\infty}i(2p^2-2p+1)^{i-1}(2p-2p^2)$.

Problem 7

采用类似二分法的思想,从中间数字问起,第一次问"这个数字是否大于等于500,000",若回答"yes",则继续问"这个数字是否大于等于750,000",若回答"no",则继续问"这个数字是否大于等于250,000",如此不断询问,直至问出答案;在最坏情况下,问到两次数字之间相差1则确定答案,此时问了$ceil(\lg 1000000)+1$次.

所以上界和下界均为 $ceil(\lg 1000000)+1$.

在最坏的情况下,Eve想的数字为边界数字 如:1,500000,1000000等,类似二分法的算法猜数字是最快的,需要$ceil(\lg 1000000)+1$次,所以上下界均为此.

Bonus Problem

```
HeapSort(data[1…n]):
    heap = BuildMaxHeap(data[1…n])
    for i=n down to 2
        cur_max = heap.HeapExtractMax()
        data[i] = cur_max
```

在$HeapSort$中, $BuildMaxHeap$的时间复杂度为$O(n)$, 下面分析在每次for循环中的$HeapExtractMax$.

当输入的n个元素都不同时:

最坏情况: HeapExtractMax的每次MaxHeapify都要进行$\lg n$次, 整个HeapSort需:$\lg n+\lg(n-1)+\ldots+\lg 2=\lg(n!)=O(n\log n)$, 所以HeapSort的时间复杂度为$O(n\lg n)$.

最好情况: 无论如何,叶子结点一定小于高度比叶子结点小 2 的所有结点, 每次 HeapSort都将一个叶子结点放在堆的顶部, 那么至少要进行$\lg n-2$次MaxHeapify, 整个HeapSort需:$\lg(n-2)+\lg(n-3)+\ldots+\lg 2=\Omega(n\log n)$, 所以HeapSort的时间复杂

度为$\Omega(n \lg n)$.

综上, 当输入的n个元素不相同时HeapSort的时间复杂度总是$\Theta(n \lg n)$.