PS4-231830106

Problem 1

(a):

修改后的TRQuicksort和Quicksort的不同仅仅在于：在Quicksort第二次递归排序右子数组`A[q+1……r]`时，TRQuicksort通过循环更新p的值为q+1，结合`while(q<r)`的循环控制，使得在p不断增大接近r的过程中`A[q+1……r]`也被排序好，这样左子数组`A[p……q-1]`和`A[q+1……r]`都被排序好，自然TRQuicksort的正确性就得到了证明。

(b):

数组A已经排好序，且每次Partition都恰好选择了最大的数字时，每次递归只能减少一个元素，因此函数栈的深度能达到Θ(n)。

(c):

思路：在每次Partition后用递归处理较小的子数组用循环处理较大的子数组，此时每次最坏的情况也能用循环处理1/2的数，栈深度为Θ(lg n)。
伪代码：

```
func TRQuicksort(A,p,r):
        while(p<r)do
                q = Partition(A,p,r)
                if (q-p<r-q) :
                        TRQuicksort(A,p,q-1)
                        p = q+1
                else:
                        TRQuicksort(A,q+1,r)
                        r = q-1
```

时间复杂度：
每一层都遍历了数组有n次操作，而递归栈至多有$\lg n$层，因此时间复杂度为$O(n \lg n)$。

Problem 2

(a):

1. 思路：先将n个数存入一个长度为k+1的数组，用于统计每个大小的数的出现次数，再构建另一个数组统计大小为0-k的数的个数。最后只需要用0-b的个数减去0-(a-1)的个数即可。
2. 伪代码：

```
func Preprocess(k,A,a,b):
        # 构建计数数组
        count = [0]*k
        for i in range(len(A))
                count[A[i]] += 1

        # 重构数组成统计数组
        for i in range(1,k):
                count[i] = count[i-1]+count[i]

        # 返回答案
        if a ==  0:
                return count[b]
        else:
                return count[b]-count[a-1]
```

3. 时间复杂度：构建count数组时接受了n个输入，因此时间复杂度为$O(n)$；重构的count数组其长度为k，重新生成其中的每一个元素需要一个操作，因此时间复杂度为$O(k)$，因此预处理的时间复杂度为$O(n+k)$，而在最后求解时时间复杂度显然为$O(1)$。

(b):

1. 思路：先找到位数最多的数的位数，然后依次对每个位排序即可
2. 伪代码：

```
func RadixSort(A,n):
        # 获取最多位数的数的位数
        d = Findmaxdigit(A)

        # 对每个位数排序
        for i in range(d):
                DigitSort(A,i)

        # 返回排序后数组
        return A

func DigitSort(A,i):
```

```
        # 初始化计数数组
        count = [0]*10

        # 统计每个数出现次数
        for i in range(n):
                digit = GetDigit(A,i)
                count[digit] += 1

        # 计算统计出现量
        for i in range(1,10):
                count[i] = count[i] + count[i-1]

        # 放入数字
        result = [0]*n
        for j in range(n,1,-1):
                digit = GetDigit(A[j],i)
                result[count[digit]] = A[j]
                count[digit] -= 1

        return result


func GetDigit(num,i):
        return (number / 10^(i-1)) % 10

func Findmaxdigit(A):
        max = 0
        for num in A:
                i = 0
                while num != 0
                        num = num/10
                        i += 1
                max = max(i,max)
        return max
```

3. 时间复杂度：DigitSort的时间复杂度为A中所有有第i位的数的个数，因此所有DigitSort的时间复杂度的和为A的所有数的位数和为n，其中GetDight函数和FindDigit函数总时间复杂度都为O(n)复杂度，因此整个算法的时间复杂度为$O(n)$。

Problem 3

使用二分查找：每次询问这个数字是否小于(n/2)，一直缩小范围。最后最多需要问$\lceil \lg_2 1000000 \rceil = 20$次，所以最坏情况下需要提问20次即可。

1. 上限证明
   伪代码：

```
function GuessNumber(min, max):
        while min < max:
                mid = (min + max) // 2 # 计算中间值
                print("Is the number less than or equal to", mid, "?")
                response = get_response() # 获取伊芙的回答，是或否
                if response == "yes":
                        max = mid # 如果是，缩小范围到 [min, mid]
                else:
                        min = mid + 1 # 如果否，缩小范围到 [mid + 1, max]

        # 最终找到的数字 print("The number is:", min)
```

因此至多20次必然可以找到所想数字
2. 下限证明：
每次提问可以提供一个信息位，又因为是“是/否”提问，因此可以看成是2进制信息位，x次提问可以看成一个x位的2进制数，这个二进制数需要能表示出1000000个信息，因此$x \geq \lceil \lg_2 1000000 \rceil$，至少需要提问20次
3. 所以提问次数>=20又<=20，可证明最后提问次数即为20。

Problem 4

(a):

从2n中选出n个数组成一个链表剩下自动组成另一个，而链表都已经排好序因此选择的不同只与链表中元素的不同相关，总个数为$\binom{n}{2n} = \frac{(2n)!}{n!n!}$。

(b):

可以构建一个这样的决策树，它的每个叶子节点是(a)中的不同分类情况。每次决策可以看成是合并过程的一步：是将A链表的元素放入合并后链表中还是B链表的元素放入合并链表中。因此决策树可以看成是一个二叉树，而叶子节点的数量$\frac{(2n)!}{n!n!}$，因此树的高度最低为$\lg_2 \frac{(2n)!}{n!n!}$，而树的每一层可以看作是进行了一次比较。因此最少比较次数即为$\lg_2 \frac{(2n)!}{n!n!}$，通过斯特林近似可以认为是$2n - o(n)$次比较。

(c):

将分别在A链表的x和在B链表的y放入合并链表后，则两者的大小关系必然通过中间数发现或通过直接比较发现，又因为x和y在链表中相邻所以并不存在在x和y之间的数，因此x和y必然进行了一次比较

(d):

由c中结论可得，合并后的链表有2n个数，既有2n-1个相邻关系因此至少进行了2n-1次比较。

Problem 5

(a):

分组为7和3时，时间复杂度的表达方式分别为：
$T(n) = T(\frac{5}{7}n) + T(\frac{7}{1}n) + O(n) \quad T(n) = 2T(\frac{1}{3}n) + O(n)$。
最后时间复杂度都为O(n),因此不会影响。

(b):

伪代码：

```
def Find(A):
    frequency = {}  # 用于存储每个值的频率

    for value in A:
        if value in frequency:
            frequency[value] += 1
        else:
            frequency[value] = 1

        # 检查当前值的频率是否超过 n/4
        if frequency[value] > len(A) / 4:
            return True  # 如果找到了，直接返回 True

    return False  # 遍历完都没有超过 n/4，返回 False
```

时间复杂度：
只遍历数组一次因此时间复杂度为$O(n)$。

Problem 6

(a):

1. 思路：先将S和W组合后按S中的值大小排序，然后计算出w(S)/2。接着从小到大依次验证S中的x是否为magical-mean即可
2. 伪代码

```
def magical_mean(S, W):
    n = len(S)
    # 将（值，重量）对进行排序
    paired = sorted(zip(S, W), key=lambda x: x[0])
    S_sorted, W_sorted = zip(*paired)

    w_S = sum(W_sorted)
    weight_below = 0
    weight_above = w_S

    for i in range(n):
        weight_below += W_sorted[i]
        weight_above -= W_sorted[i]

        if weight_below <= total_weight / 2 and weight_above <=
total_weight / 2:
            return S_sorted[i]

    return None  # 没有找到神奇平均值
```

3. 时间复杂度：排序时间复杂度为$O(n\lg n)$，验证遍历了一遍数组时间复杂度为$O(n)$，因此总时间复杂度为$O(n\lg n)$。

(b):

1. 思路：将(a)中寻找magical-mean的方式改为验证中位数即可而不是通过排序
2. 伪代码：

```
def magical_mean(S,W):
        n = len(S)
        total_weight = sum(W)
        mid = Quickselect (S, W ,n//2)

        weight_below = sum(W[i] for i in range n if S[i] < mid)
        weight_above = sum(W[i] for i in range n if S[i] > mid)

        if weight_below <= total_weight / 2 and weight_above <=
total_weight / 2:
```

```
        return median

        return None

def Quickselect(S,W,k):
        if len(S)==1:
                return S[0]

        pivot = S[random.randint(0, len(S) - 1)]
        low = [s for s in S if s < pivot]
        high = [s for s in S if s > pivot]
        low_weight = [W[i] for i in range(len(S)) if S[i]<pivot]
        high_weight = [W[i] for i in range(len(S)) if S[i]>pivot]

        if len(low) > k:
                return Quickselect(low,low_weight,k)
        if len(low) == k:
                return pivot
        else:
                return Quickselect(high,high_weight,k-len(low)-1)
```

3. 时间复杂度：Quickselect时间复杂度为O(n)，因此总时间复杂度为$O(n)$

Problem 7

(a):

设OneInThree返回1的概率为P_1，则有

$$P_1 = \frac{1}{2} \times 0 + \frac{1}{2} \times (1 - P_1)$$

解得$P_1 = \frac{1}{3}$

(b):

设OneInThree期望调用FairCoin次数为E，则有

$$E = \frac{1}{2} \times 1 + \frac{1}{2} \times (E + 1)$$

解得$E = 2$

(c):

```
def OneInTwo():
        x = BiasedCoin()
        y = BiasedCoin()
        if x != y:
                return x
        else:
                return OneInTwo()
```

说明：当x!=y时函数返回x，此时x为1和x为0概率相等即有1/2概率返回1，而若x=y则继续递归调用直至x!=y而产生返回值。

(d):

设每次OneInTwo中x!=y的概率为P_0，BiasedCoin返回1的概率为p则有

$$P_0 = 2p(1-p)$$

不妨设设OneInTwo期望调用BiasedCoin次数为E，则有

$$E = P_0 \times 2 + (1 - P_0) \times (E + 2)$$

综上两式解得$E = \frac{1}{p(1-p)}$