ECE 606, Fall 2019, Assignment 5

Zhijie Wang, Student ID number: 20856733

zhijie.wang@uwaterloo.ca

October 8, 2019

1. (a) *Proof.* Here we use induction to prove this.

   Base case: if $r-p+1=2$, then, q should be r in the return of PARTITION(A,p,r). And $A[p]<A[r]$, which is correct.

   Step: Our induction is given that $r-p+1=n$, the return of QSNEW(A,p,r) where $A[\underbrace{0,\ldots,n-1}_{n}], p=0, r=n-1$ is sorted as an ascending array. Now if $r-p+1=n+1$, then, the input array will be $A[\underbrace{0,\ldots,n-1}_{n},n]$. Here we do a case analysis: (1) $A[n]>A[r]$, (2) $A[n]<A[p]$, and (3) $A[p]<A[n]<A[r]$.

   (1) If $A[n]>A[r]$, then, A is already sorted. PARTITION will not change the position of any elements of the input array because $A[r]$, where $r=n$ is bigger then any other elements of the array. Hence, the return of PARTITION is $q\leftarrow r$. Therefore, $q-p>r-q$, the algorithm will invoke QSNEW$(A,r+1,r)$, which will terminate because $r+1>r$.

   (2) If $A[n]<A[p]$, then, the only thing will happen in PARTITION is that $A[p]$ will exchange with $A[r]$, and the return will be $q\leftarrow p$. Since $p>q-1$, therefore QSNEW$(A,p,q-1)$ will terminate, and QSNEW$(A,q+1,r)$ will still only exchange the $A[p]$ and $A[r]$ in this function. Finally, the function will terminate and then A will be sorted.

   (3) If $A[p]<A[n]<A[r]$, then, assume $i\in A[0,\ldots,m]<A[n]<j\in A[m+1,\ldots,n-1]$. Hence, the return of PARTITION will be $q\leftarrow m+1$. And from then on, each time QSNEW will exchange the first item with the last one starting from $p=m+1$ and return $q\leftarrow p$. Finally the function will terminate and then A will be sorted.

   Therefore, this version of Quick Sort is correct.

   (b) $\Theta(n)$.

   For the worst-case, every time the return of PARTITION will be $\lfloor(p+r)/2\rfloor$, therefore, QSNEW will be invoked n times, and we need a call stack which has a size of n. Therefore, the worst-case space-efficiency is $\Theta(n)$.

2. Let's denote the equivalence tester as CHECKEQUIVALENT(x,y), where x,y are two account numbers, and it will return $True$ if equivalent and $False$ else. Assume the number of n cards are stored in an array $A[0,1,\ldots,n-1]$ with size n.

   HALFEQUIVALENT(A)

   ```
   1: if HalfCheck(A, 0, n − 1) ≠ None then
   2:     return True
   3: else
   4:     return False
   ```

   HALFCHECK(A,p,r)

   ```
   1: while r − p + 1 > 3 do
   2:     q ← ⌊(p + r)/2⌋
   3:     m1 ←HalfCheck(A, p, q)
   4:     m2 ←HalfCheck(A, q + 1, r)
   5:     return CheckValid(A, p, r, m1, m2)
   6: return CheckValid(A, p, r, p, r)
   ```

```
CheckValid(A, p, r, m_1, m_2)
 1: if CheckCount(A, p, r, m_1) then
 2:     return m_1
 3: else if CheckCount(A, p, r, m_2) then
 4:     return m_2
 5: else
 6:     return None

CheckCount(A, p, r, m)
 1: if m = None then
 2:     return False
 3: count ← 0
 4: for i from p to r do
 5:     if CheckEquivalent(A[m], A[i]) then
 6:         count ← count + 1
 7: if count > ⌊(r − p + 1)/2⌋ then
 8:     return True
 9: else
10:     return False
```

Here we give a brief analysis about the algorithm above. The basic idea is that if there is a number appears more than half of size's times in an array, then, this number will still appears more than half of size's times in one of arrays which is half of the original array. Therefore, HalfCheck will be invoked $\log n$ times until there are several arrays which has a size of 2 or 3. Then, we invoke CheckValid to check if there is a number appears more than half of size's times in each array. And combine the result of each two arrays, then, check again. Finally, HalfCheck will return a value, if it is not *None*, then, there is such a number appears.

In the worst-case, HalfCheck will be invoked $\log n$ times, and each one takes $O(n)$ times to search for m_1 and m_2. Therefore, it needs $O(n \log n)$ times of comparisons.

And I think there is another algorithm which only need O(n) comparisons. Here is the pseudo-code.

```
HalfEquivalent(A)
 1: count ← 1
 2: cur ← A[0]
 3: for i from 1 to n − 1 do
 4:     if count = 0 then
 5:         cur ← A[i]
 6:         count ← 1
 7:     else if CheckEquivalent(cur, A[i]) then
 8:         count ← count + 1
 9:     else
10:         count ← count − 1
11: count ← 0
12: for i from 0 to n − 1 do
13:     if CheckEquivalent(cur, A[i]) then
14:         count ← count + 1
15:     if count > ⌊n/2⌋ then
16:         return True
17:     else
18:         return False
```

Now we give a brief analysis on the algorithm above. Assume an account number appears more than $n/2$ times in n cards, let's denote the times of appearance as m. Then, m should bigger than the sum of any other number's appearances. Every time we select a new *cur* in Line(4) to Line(6), which means that if there is no number appears more than half of size's times before, and we only need to search if there is a number appears more than half of size' times after this selection, then say it is *cur*. (But this number might not appears more than half of size's times in the whole array.)

Hence, when the first for-loop ends, if such account number exits, than *cur* should be this number (Note

that here we do not need to copy the value to *cur*, but only the address, which is similar as a pointer operation in C++). But actually this number might not exist, so we invoke another for-loop to ensure that this number's appearance times is bigger than $n/2$. In the worst-case we invoke the tester $(2n-1)$ times, therefore, our algorithm do comparison only $O(n)$ times.

3. (a) The basic idea of proposed algorithm is to search the minimum numbers of each value. Here we denote the number of coins of each value is $[m_0, m_1, \ldots, m_{n-1}]$. And let's start from $C_0 \in C = [C_0, C_1, C_2, \ldots C_{n-1}]$. And we use the binary search between $[0, \text{MinNumCoins}(C, a)]$ for m_0, only if $m_0 + \text{MinNumCoins}(C[1:], a - m_0 * c_0) = \text{MinNumCoins}(C, a)$, then, the m_0 is exactly one of the correct answers. Therefore, for m_1, we only need to search where $\{C = C[1:\ ], a = a - m_0 * C_0\}$. And in the worst-case, assume the input size is $< n = len(C), a = 2^s >$, therefore the running-time should be $\Theta(n \log a) = \Theta(ns)$, which is polynomial-time complexity.

   Here is the pseudo code about this algorithm. (Length returns the length of list.)

   MinCoinsList(C, a)

```
1: if Length(C) = 1 then
2:    return [MinNumCoins(C, a)]
3: C_1 ← C.copy()
4: for i from 1 to (Length(C)−1) do
5:    result[i] ← BinSearch(C_1, C[i], 0, MinNumCoins(C_1, a), a)
6:    a ← a − result[i] ∗ C[i]
7:    C_1 ← C[j :  ]
8: result[Length(C)] ← MinNumCoins(C, a) − Sum(result)
9: return result
```

   BinSearch(C, v, p, r, a)

```
1: C_1 ← C.copy()
2: C_1 ← C_1[1 :  ]
3: while p <= r do
4:    med ← ⌊(p + r)/2⌋
5:    if med + MinNumCoins(C_1, (a − med ∗ v)) = MinNumCoins(C, a) then
6:       return med
7:    else if med + MinNumCoins(C, (a − med ∗ v)) = MinNumCoins(C, a) then
8:       p ← med + 1
9:    else
10:      r ← med − 1
```

   (b) a5p3b.py