CSC263 Fall 2017 Assignment 3

Name: Ruijie Sun, Guanchun Zhao
SN: 1003326046, 1002601847

Q1. SOLUTION

(a) Using the aggregate method,
Consider the operations order: Enqueue operations followed by 50 Dequeue operations.

i. 50 ENQUEUE means that there are 50 times to call push operations (push x to S1),
12 times to call push operations (push x to S2),
and 12 times to call pop operations (pop from S1.)

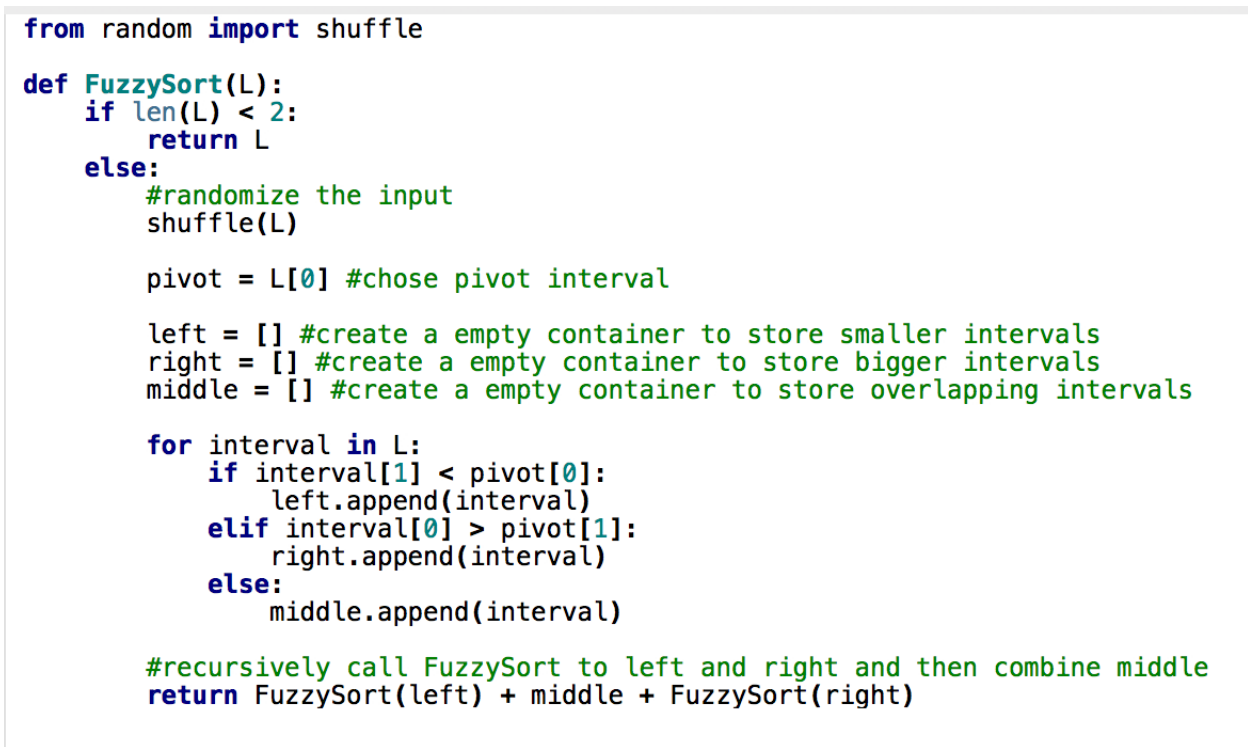
ii. 50 DEQUEUE means that there are 50 times to call pop operations (pop from S2),
(50 - 12) times to call push operations (push x to S2),
and (50 - 12) times to call pop operations (pop from S1).

Thus, the total cost T(n) =
(50 + 12 + (50 - 12)) * 2 + (12 + 50 - 12 + 50) * 3 = 500
Therefore, by the aggregate method, the amortized cost = T(n) / the number of operations
= 500/100 = 5 units of time

(b) Using the accounting method,
Give each operation 6 units of times.
No matter what the order of these operations is, the worst case (not popping from an empty stack) always inserts m elements and pops them all. Besides, ENQUEUE at least costs 2 units because of the last line. Similarly, DEQUEUE at least costs 3 units. Thus, not counting each while loop cost, the remaining time for each operation is at least 3 units, and 3*(2m) units in total. For the while loop cost, since there are m element insertions in total and there is no procedure in ENQUEUE and DEQUEUE such that an element can be moved from S2 to S1, the only function in while loop is moving elements from S1 to S2, and can be called at most m times in all while loop.That means the total while loop cost is 5m.
As aforementioned, not counting each while loop cost, the remaining time for each operation is at least 6m units in total. Therefore, there is enough remaining money to pay for the total while loop cost.
Thus, by the accounting method, the amortized cost is 6 units of time.

Q2. SOLUTION

(a). Pseudo Code:



```
from random import shuffle

def FuzzySort(L):
    if len(L) < 2:
        return L
    else:
        #randomize the input
        shuffle(L)

        pivot = L[0] #chose pivot interval

        left = [] #create a empty container to store smaller intervals
        right = [] #create a empty container to store bigger intervals
        middle = [] #create a empty container to store overlapping intervals

        for interval in L:
            if interval[1] < pivot[0]:
                left.append(interval)
            elif interval[0] > pivot[1]:
                right.append(interval)
            else:
                middle.append(interval)

        #recursively call FuzzySort to left and right and then combine middle
        return FuzzySort(left) + middle + FuzzySort(right)
```

(b).

Let X be the random variable representing the number of comparisons.

We define an indicator random variable:

$$X_{ij} = \begin{cases} 1 & \text{if interval i and interval j are compared} \\ 0 & \text{otherwise} \end{cases}$$

As we have already known in class, $E(X) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(\mathrm{X}_{ij}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \in \Theta(nlogn)$

When all interval overlap, the left container and right container in Pseudo Code will always be empty. So the total number of comparison is 2n. And other codes cost constant time. Thus the expected runtime $\in \theta(n)$.

Q3. SOLUTION

For the following parts, we define n = len(A).

(a) Pseudo Code:

```
# A is the array searched, and x is the value searching for
def f(A, x):
    checked = []

    # When not checking all the elements in A,
    while len(A) != len(checked):
        # randint(a, b) returns the random number c such that a <= b <= c.
        i = random.randint(0, len(A) - 1)
        if A[i] == x:
            return i

        if A[i] not in checked:
            checked.append(A[i])

    return -1 # x is not found in A
```

(b) i. Sample space: any permutation of any n elements. Note that there is exactly one element in A equals x.

ii. Let M be the number of indices into A that we must pick before we find x and Random-Search terminates. Note that M is $\in N$

Let p = 1/n, and q = (n-1)/n. Then,

$$\text{P(M)} = \begin{cases} p & \text{,if } M = 1 \\ qp & \text{,if M = 2} \\ q^2p & \text{,if M = 3} \\ q^3p & \text{,if M = 4} \\ \ldots \end{cases}$$

iii. We could notice that M follows Geometric distribution (Geo(p)). Thus, E(M) = 1/p = n. Thus, the expected number of indices into A is n.

(c) i. Sample space: any permutation of any n elements. Note that there is exactly k elements in A equals x.

ii. Let M be the number of indices into A that we must pick before we find x and Random-Search terminates. Note that M is $\in N$

Let p = k/n, and q = (n-k)/n. Then,

$$\text{P(M)} = \begin{cases} p & \text{,if } M = 1 \\ qp & \text{,if M = 2} \\ q^2p & \text{,if M = 3} \\ q^3p & \text{,if M = 4} \\ ... \end{cases}$$

iii. We could notice that M follows Geometric distribution (Geo(p)). Thus, E(M) = 1/p = n/k. Thus, the expected number of indices into A is n/k.

(d) i. Sample space: any permutation of any n elements. Note that there is no element in A equals x.

ii. Let R_i be the number of picks that we need to get the i^{th} element that has not been checked. Note that R_i is $\in N$
Let p = (n-(i-1))/n, and q = 1 - p. Then,

$$\text{P}(R_i) = \begin{cases} p & \text{,if } R_i = 1 \\ qp & \text{,if } R_i = 2 \\ q^2p & \text{,if } R_i = 3 \\ q^3p & \text{,if } R_i = 4 \\ ... \end{cases}$$

iii. We could notice that R_i follows Geometric distribution (Geo(p)). Thus, E(R) = 1/p = n/(n-(i-1)).
Since we need find all the elements in A,
the total expected number = $\sum_{i=1}^{n} E(R_i)$
= $\sum_{i=1}^{n}\{n/(n-(i-1))\}$
= n$\sum_{i=1}^{n}\{1/i\}$.
= n logn + O(n) by Harmonic series on CLRS Thus, the expected number of indices into A is n$\sum_{i=1}^{n}\{1/i\}$ = n logn + O(n).

Q4. SOLUTION

For the following parts, the mentioned i means the original i picked randomly.

(a) By cases,

(1). if x in A,

①. if x != A[head],

Let j be the index of x in A

if x > A[i], then the number of comparisons is ((j - i) mod n) + 1 . Since the max value of (j - i) mod n is n - 1, the worst case number of comparisons is n.

if x < A[i], then the number of comparisons is 1 + ((j - head) mod n) + 1. Since the max value of (j - head) mod n is n -1 , the worst case number of comparisons is 1 + n.

②. if x == A[head],

The worst case for i when i != head,

and the number of comparisons is 2.

(2). if x not in A,

The algorithm at most checks A[i] once and every element once when x is bigger than all elements in A and i = head. the number of comparisons is n + 1. Note that the value of i does not affect the worst case.

Therefore, the worst case number of the comparisons in the whole is 1 + n.

(b) i. Sample S is [0, 1, 2, ..., n-1], and random variable i is uniformly chosen from S.

ii. Let t_i be the variable, which means that the worst case number of comparisons for any i $\in$ S.

iii. Since i follows Uniform distribution, p(t_i) = 1/n.

$E(t) = \sum_{i=0}^{n-1} t_i P(t_i)$ = (1/n) $\sum_{i=0}^{n-1} t_i$.

Now we need to calculate t_i for each i.

Without loss of generality, let A be an almost-sorted array and head equals 0.

Note that we do not need to divide into two cases: x in A, x not in A, since we focus on the worst case.

t_i = max(i + 2, n - i) since we let head equal 0 and in the worst case the algorithm could directly searched either from A[i] and A[0] to A[i] or from A[i] to A[n-1] and A[0].

For 0 ≤ i < $\lfloor n/2 \rfloor$, the worst case is when $x > A[n-1]$, and t_i = n - i.

For $\lfloor n/2 \rfloor \leq$ i $\leq n$, the worst case is when A[i-1]< x < A[i], and $t_i =$ i + 2.

By cases,

(1). n is even,

$\sum_{i=0}^{n-1} t_i$

$= \sum_{i=0}^{i=n/2-1}(n-i) + \sum_{i=n/2}^{n-1}(i+2)$

$= (n/2)(n) - \sum_{i=0}^{i=n/2-1}(i) + \sum_{k=n/2+2}^{n+1}(i)$

$= (n/2)(n) - ((n/2)(n/2-1))/2 + (((n+2)(n+1)) - (n/2+2)(n/2+3))/2$

$= (3n^2+4n)/4$

Thus, $E(t_i) = (1/n) * (3n^2+4n)/4 = (3n+4)/4 \in O(n)$.

(2). n is odd,

$\sum_{i=0}^{n-1} t_i$

$= \sum_{i=0}^{(n-1)/2}(n-i) + \sum_{i=(n+1)/2}^{n-1}(i+2)$

$= ((n-1)/2)(n) - \sum_{i=0}^{(n-1)/2}(i) + \sum_{k=(n+5)/2}^{n+1}(i)$

$= ((n-1)/2)(n) - ((n-1)/2)((n+1)/2)/2 + (((n+2)(n+1)) - ((n+5)/2)((n+3)/2))/2$

$= (3n^2+n-3)/4$ Thus, $E(t_i) = (1/n) * (3n^2+n-3)/4 = (3n+1-3/n)/4 \in O(n)$.

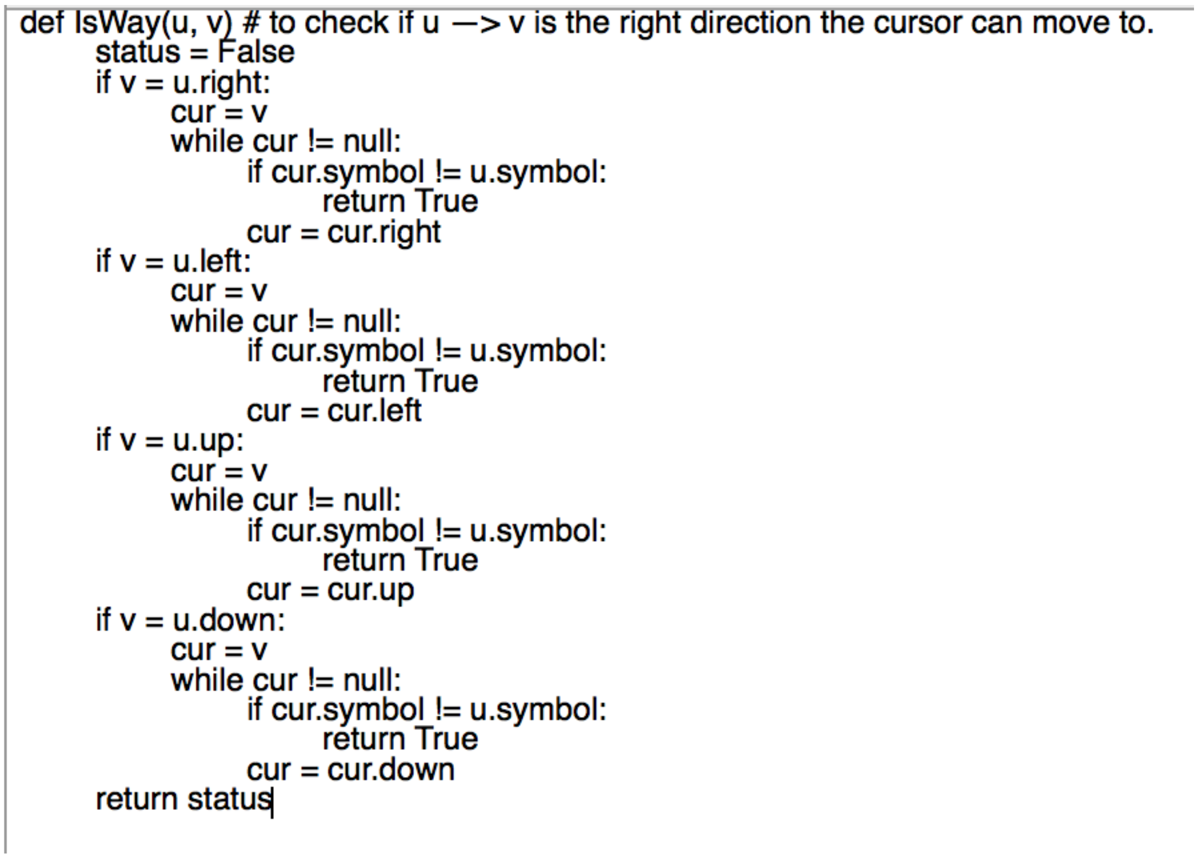
Thus, $E(t_i) \in O(n)$ and $\in \Omega(n)$.

Therefore, the expected worst case number of comparisons is O(n) and $\Omega(n)$.

Q5. SOLUTION

a) Pseudo Code:

```
def DrawGraph(A):
    for each grid square in the virtual keyboard:
        grid.symbol = the symbol of this grid in virtual keyboard

        grid.right = null

        grid.left = null

        grid.up = null

        grid.down = null

    for each grid square in the virtual keyboard:

        if grid's right in the virtual keyboard exists and grid.right == null:

            grid.right = grid's right in the virtual keyboard

        if grid's left in the virtual keyboard exists and grid.left == null:

            grid.left = grid's left in the virtual keyboard

        if grid's up in the virtual keyboard exists and grid.up == null:

            grid.up = grid's up in the virtual keyboard

        if grid's down in the virtual keyboard exists and grid.down == null:

            grid.down = grid's down in the virtual keyboard
```



```
def IsWay(u, v) # to check if u —> v is the right direction the cursor can move to.
    status = False
    if v = u.right:
        cur = v
        while cur != null:
            if cur.symbol != u.symbol:
                return True
            cur = cur.right
    if v = u.left:
        cur = v
        while cur != null:
            if cur.symbol != u.symbol:
                return True
            cur = cur.left
    if v = u.up:
        cur = v
        while cur != null:
            if cur.symbol != u.symbol:
                return True
            cur = cur.up
    if v = u.down:
        cur = v
        while cur != null:
            if cur.symbol != u.symbol:
                return True
            cur = cur.down
    return status
```

```
def BFS(G,s,t):
    for each vertex $u \in V - \{s\}$:
        $u.color = WHITE$
        $u.d = \infty$
        $u.\pi = null$
    $s.color = GRAY$
    $s.d = 0$
    $s.\pi = null$
    $Q = \varnothing$
    ENQUEUE(Q, s)
    u = s
    while ($Q \ != \varnothing$ and $u.symbol \ != t$):
        u = DEQUEUE(Q)
```

```
            for each $v \in \{u.right, u.left, u.up, u.down\}$:
              if $v! = null$:
                if v.color == WHITE:
                  v.color = GRAY
                  if $IsWay(u, v) == True$:
                    if v.symbol != u.symbol:
                      v.d = u.d + 1
                    else: v.d = u.d
                  $v.\pi = u$
                  ENQUEUE(Q,v)
          u.color = BLACK
      return u.d
  def MinimalStrokes(A,s)
      DrawGraph(A)
      StartPosition = upper left corner of the keyboard
      counter = BFS(A,StartPosition, s[0])
      for i in range(1,length(s)):
        counter = counter + BFS(A,s[i-1], s[i])
      return $counter + length(s) + BFS(A, s[-1], \backslash n) + 1$
```

b)

Since operation time of assigning a edge is O(1) and one node has at most 4 edges, so the total time of DrawGraph(A) = $2 \times \#ofedges \in O(E)$ which $\in O(8V) = O(r \times c)$.

For execution of IsWay(), each node is checked at most (r+c) times and each check costs constant time. So checking all nodes cost at most $V \times (r + c)$.

Since for each BFS, it costs at most $V \times (r + c)$ for IsWay() and $(V + E)$ for other parts as we have learnt in class. So each BFS cost $\in O(V + E + (r+c) \times V)$ which $\in O((r+c) \times V) = O((r+c) \times r \times c)$. So n times BFS cost $\in O(n \times (r + c) \times r \times c)$. And other parts cost constant time. Thus the total cost of our algorithm $\in O(n \times (r + c) \times r \times c)$.