NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
Midterm (30%)
AY2019/20 Semester 2

CS2040 – Data Structures and Algorithms

7 March 2020 Time allowed: 90 minutes

INSTRUCTIONS TO CANDIDATES

1. Do **NOT** open the question paper until you are told to do so.
2. This question paper contains TWO (2) sections with sub-questions. Each section has a different length and different number of sub-questions. It comprises Ten (10) printed pages, including this page.
3. Answer all questions in this paper itself. You can use either pen or pencil. Write **legibly**!
4. This is an **Open Book Quiz**. You can check the lecture notes, tutorial files, problem set files, CP3 book, or any other books that you think will be useful. But remember that the more time that you spend flipping through your files implies that you have less time to actually answer the questions.
5. When this Quiz starts, **please immediately write your Matriculation Number** and **Tutorial Group (if you don't know your tutorial group, write your TA name and time slot)**.
6. The total marks for this paper is **100**.

TUTORIAL GROUP

STUDENT NUMBER: | A | | | | | | | | |

For examiners' use only		
Question	***Max***	***Marks***
Q1-4	12	
Q5	35	
Q6	35	
Q7	18	
Total	**100**	

Section A – Analysis (12 Marks)

Prove (the statement is correct) or disprove (the statement is wrong) the following statements below. If you want to prove it, provide the proof or at least a convincing argument. If you want to disprove it, provide at least one counter example. 3 marks per each statement below (1 mark for circling true or false, 2 marks for explanation):

1. The tightest time complexity of the following code fragment is $O(n)$. **[true/false]**

```
for (int i=1;i <= n; i=i*8) {
    System.out.println("outer loop");
    for (int j=1; j <= i; j=j*2) {
        System.out.println("Inner loop");
    }
}
```

False.
The outer loop variable i increase by a multiple of 8 each iteration and stop when it reaches n. Thus the values of i is as follows 1,8,16,24…n →$8^0, 8^1, 8^2 \dots 8^{lg_8 n}$
The inner loop variable j increase by a multiple of 2 each iteration and stop when it reaches i. Since total number of times the entire fragment is executed is the number of times the inner loop is executed, this can be represented by the sum

$$1 + lg_2 8^0 + lg_2 8^1 + lg_2 8^2 + \cdots + lg_2 8^{lg_8 n}$$

$$= 1 + lg_2 2^{3*0} + lg_2 2^{3*1} + lg_2 2^{3*2} + \cdots + lg_2 2^{3*lg_8 n} = 1 + 3 * 1 + 3 * 2 + \cdots + 3 * lg_8 n$$

$$= 1 + 3 * (1 + 2 + 3 + 4 + \cdots + lg_8 n) = O\left(1 + 3 * \frac{lg_8 n * (1 + lg_8 n)}{2}\right)$$

$$= O\left(1 + 3 * \frac{lg_8 n + lg_8^2 n}{2}\right) = O(lg_8^2 n) = O(lg_2^2 n)$$

Answers that did not sufficiently argue for a time complexity < O(n) (eg. arguing that since the outside loop is O(log n), the time complexity must therefore be < O(n); see Tutorial 1 Q2d for a counterexample) get 1 mark for explanation.

2. Using a basic linked list we can only insert and remove from the front in O(1) time. However using a tailed linked list we can insert and remove from both the front and back in O(1) time. **[true/false]**

False.
We cannot remove from the back in O(1) time, because we need a reference to the 2nd last node to remove the last node, and we can only point to that node by moving from the head in O(n) time.

3. If separate chaining is used as the collision resolution technique for a hashtable, and load factor cut off is set at 10. This means that whenever the chain in any row exceeds length 10 we will rehash everything into a bigger hashtable. **[true/false]**

   False.
   Load factor 10 means that if the number of entries/size of hashtable is 10 then we will resize the hashtable. We can have a hashtable of size e.g 17 and have all 10 insertions to be to a particular row so the chain in that row will be 10 but the load factor is only 10/17 < 1.

   Full marks are given for correctly identifying the relation of load factor as the average length of a chain, rather than a maximum length. Directly quoting the meaning of "load factor", without mapping it to what it means in terms of separate chaining would get reduced marks.

4. For insertion sort, we need to perform a linear scan from the back of the sorted portion to find the correct position to insert each value in the unsorted portion of the array. This causes insertion sort to be O(N^2). We can improve this to O($NlogN$) by performing a binary search to find the insertion point instead.
   [true/false]
   False.
   Even if we can find the insertion point in O(logN) time using binary search where N is the current size of the sorted region, we still need to shift all values to its right to introduce the gap for the insertion, thus negating the speedup brought about by the binary search.

   Answers that attempted to argue that the time complexity was < O(n^2) eg. by saying the array was already sorted were not accepted, as we are looking for time complexity in the general case.

Section B – Applications (88 Marks)

Write in pseudo-code. Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes. Some **partial marks** will be awarded for correct answers not meeting the time complexity required.

5. **Genome analysis [35 marks]**

Given a gene represented as a **basic linked list** (as defined in the lecture notes) of characters from the set {'A', 'C', 'G', 'T'}, you are to implement 2 common operations in genome analysis as follows:

a.) *Gene Translocation* – given 2 genes K and L of length $|K|$ and $|L|$ respectively, where $|K|,|L| > 1$, swap the portion of K from index 0 to i with the portion of L from index j to $|L| - 1$, where $0 \leq i < |K| - 1$ and $1 \leq j < |L|$.

E.g K = ACCGTC , L = AGGTCCCT, i=2 and j=4, translocation will result in

K = CCCTGTC, L = AGGTACC

Implement gene translocation in time $O(|L| + |K|)$. **[18 marks]**

```
prev = K.getNodeAtIndex(i-1) →O(|K|)
cur = K.getNodeAtIndex(i) →O(|K|)
prev' = L.getNodeAtIndex(j-1) → O(|L|)
cur' = L.getNodeAtIndex(j) → O(|L|)
tail = L.getNodeAtIndex(|L|-1) → O(|L|)
prev'.next = K.head
K.head = cur'
prev.next = null
tail.next = cur
```

Total time take is O(2*|K|)+O(3*|L|) = O(|K|+|L|)

Minor mistakes (eg. off by one, abbreviated as "OB1" as grading remarks) get **-1 mark**.
Major mistakes (eg. wrong time complexity for a section of code, incorrect iteration code) get **-2 marks**.
For grading remarks, the terms K_a, K_b, L_a, and L_b may have been used in your script, when attempting to illustrate the final results of your answer. K_a refers to the section of gene K from K_0 to K_i, while K_b refers to the section of gene K from K_{i+1} to $K_{|K|-1}$. Similarly, L_a refers to the section of gene L from L_0 to L_{j-1}, while L_b refers to the section of gene L from L_j to $L_{|L|-1}$. If there was an 'r' before it, it means that the contents were reversed eg. rK_a refers to the section from K_i to K_0, in that order.

b.) *Longest Common Suffix Length* – given 2 genes K and L of length $|K|$ and $|L|$ respectively, where $|K|,|L| > 1$, return the length of the longest common suffix of the 2 genes (0 if there is no common suffix). The suffix of a gene refers to the part of a gene from

some index i ($0 <= i < length\ of\ gene$) to the end of the gene. The longest common suffix is the longest suffix of K and L that match each other.

E.g K = ACCGTC, L = AGGTCGTC, the longest common suffix is as underlined and of length 4.

Implement longest common suffix length in time $O(|L| + |K|)$. You cannot use any additional data structures (no array, no linked list, no stack, no queue etc …) to help you. You can modify the K and L if necessary. **[17 marks]**

```
Let cur = K.head, cur' = L.head

If (|K| < |L|)
    cur' = L.getNodeAtIndex(|L|-|K|)
else if (|K| > |L|)
   cur = K.getNodeAtIndex(|K|-|L|)

let c be the length of the longest common suffix which is initialized to 0

while (cur != null)
  if (cur.character == cur'.character)
      c += 1
  else if (cur.character != cur'.character && cur.next != null)
      c = 0 ← reset c as current longest common substring is not a suffix
  cur = cur.next
  cur' = cur'.next
return c
```

Since you will go through L and K once total time taken is O(|K|+|L|)

An alternative answer is to reverse the K and L and find the longest common prefix instead.

```
prev = K.head, cur = prev.next, next = cur.next, cur.next = prev

while next != null
  prev = cur, cur = next, next = next.next
  cur.next = prev
K.head = cur

Do the same for L as above

cur = K.head, cur' = L.head, c = 0
while cur != null && cur.character == cur'.character
  c += 1
  cur = cur.next
  cur' = cur'.next
return c
```

Total time to reverse K is O(|K|), total time to reverse L is O(|L|). Total time to go through both reversed list to find longest common prefix is O(Min(|K|,|L|)). In total it is O(|K|)+O(|L|)+O(Min(|K|,|L|)) = O(|K|+|L|)

Answers that used additional data structures were capped at **6 marks**, including attempting to modify the provided basic linked lists into tailed doubly linked lists.

Answers that were not attempting to find the longest common suffix (eg. checking for longest common substring, checking for longest prefix of K as a subsequence of L etc.) were also capped at **6 marks**.

Answers that used Strings were capped at **9 marks**, because while String is not explicitly a data structure, it is in effect the same as an array of characters.

O(Max(|L|,|K|)^2) solution -> **9 marks**

Minor mistakes **-2 marks**
Major mistakes like get suffix finding wrong **-5 marks**

6. **Mountain Peaks [35 marks]**

a) John is creating a game where a player has to climb a series of mountain peaks of different heights which can be from 0 to M meters above sea level, where $1{,}000 \leq M \leq 1{,}000{,}000$. In order to generate the peaks, John has created an array A of 50 randomly generated positive integers between 0 and M. John then starts going through A from the 1st to the last integer and accumulate the numbers by **multiplying** them. Consider each multiplication as a step, at step i the current accumulated total c_i will be the height of the i^{th} peak. If $c_i \geq M$, c_i will be set to $c_i \% M$. The height of $c_1$the first peak is set to $A[0]$.
Once he reaches the last integer he will start from the 1st integer again and repeat the process ad infinitum.
For e.g if $A = \{3,1,2\}$ and $M = 10$, the first 5 heights will be as follows:

$c_1 = 3$, $c_2 = 3 * 1 = 3$, $c_3 = 3 * 1 * 2 = 6$, $c_4 = 3 * 1 * 2 * 3 = 18\%10 = 8$,
$c_5 = 3 * 1 * 2 * 3 * 1 = 18\%10 = 8$

In fact, the first 10 peaks generated will be 3,3,6,8,8,6,8,8,6,8 ….
John soon realize that the sequence of heights would start repeating after a while. Let the repeated subsequence be R (6,8,8 in the example above). Your job is to come up with an algorithm to return one integer that is in R (any integer in R will do, so 6 or 8 in the example above) in $\leq O(M)$ time given A the array of 50 random integers. **[18 marks]**

i.) State the data structure(s) you will use to help you (N.A if no DS required)

Use a hashset H which stores a key which is a pair of value (c,i) where c is the current height generated and i is the current index in A which generates c.

ii.) Give the algorithm

```
H.insert((A[0],0))
Let c = A[0], i = 0
while (true)
  i=(i+1)%50
  c=c*A[i]%M
  if !H.contains((c,i))
```

```
    H.insert((c,i))
  else // c is part of the repeating subsequence since we arrive
       // at c at the same index i in A so everything repeats after that
    break
return c
```

There are at most O(M*50) values hashed before a number in the repeated subsequence repeats. Thus total time taken is O(50*M) = O(M)

Totally wrong/incomplete solution -> **1 mark**
Correct O(M^2) solution -> **11 marks**

Correct O(M^3) solution -> **5 marks**

Answers which return the 1st peak repeated -> **6 marks.** From the e.g given in the question this is already not correct, as peaks outside of R can also be repeated.

The following answers or variations get **8 to 10 marks:**

Answers which return highest frequency peaks among the 1st |A| or M generated peaks (more peaks generated higher marks as more likely to have R repeated more times).
However this is still not correct solution since |R| can be huge and very few or even none of the peaks repeat within R itself, or there can be other peaks outside of |R| which also have the highest frequency.
E.g A={3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2},
and M= 1000. 3 is most frequently generated among others but not in R.
Answers which return the 1st peak with height > M (from the multiplication before mod) or the first repeated peak with height > M
Again not correct solution
E.g
A={970,448,113,820,936,980,219,469,470,470,769,327,644,602,211,480,5,926,420,983,979,906,605,535,425,136,408,504,953,57,385,323,471,802,756,849,942,201,760,967,495,501,238,406,208,505,897,962,34,481,} and M=1003

123 is one of the heights generated from multiplication of the 1st 11 integers thus > M and is repeated 2x but it is not found in R.
Answers which return the 50th peak or the 1st peak after the 50th > M
50th peak may not be in R yet, neither is it necessarily true that the 1st peak after 50th > M to be in R.
E.g

A={7,11,13,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,}, and M=1000

2 at peak 53 is actually 2*7*11*13 = 2002 = 2 which is 1st peak after 50th peak > M but 2 is not in R

Those who do not specify how many peaks generated in their solution will be treated as if they have generated M^3 peaks.

[**This may be a difficult sub-question*]

b) After helping John with his problem in a), he now returns to you with another problem. He has included the ability for the player to rappel from a peak X to another peak Y if height of $X \geq$ height of Y and all peaks in between X and Y are shorter than X and Y. John has now stored the 1st N integers ($N > 1$) out of the sequence R from a) in an array B. He wants you to find out what is the maximum number of peaks m between a pair of X and Y among all possible pairs of X and Y in B where the player can rappel from X to Y. Your algorithm should run in time $O(N)$ or better. **[17 marks]**

E.g if B = {5,11,10,3,12,7}, the pair X=12 and Y=11 has the maximum number of peaks in-between them, i.e 2, which should be output by your program. You will get the same result if 11 and 12 were swapped i.e B' = {5,12,10,3,11,7}.

i.) State the data structure(s) you will use to help you (N.A if no DS required)

Use a stack S which stores a pair of values (h,m) for a peak v where h is B[i] for the current value of i, and m is # of peaks to the left of v which has height < h before the 1st peak with height >= h

ii.) Give the algorithm

```
int maxM = -1
S.push((B[0],0))
for (int i=1; i < N; i++)
  let v = (B[i],0)
  if (v.h < S.peek().h)
    S.push(v)
  else
    while (!S.empty() && S.peek().h < v.h)
      v.m = v.m+S.pop().m+1 // +1 to include peak popped in m
    if (S.empty())
      v.m -= 1 // don't include last peak in stack into calculation of m
    S.push(v)

while (!S.empty) // important step to get the best m
  maxM = Math.max(maxM,S.peek().m)
  S.pop()

return maxM
```

Each value in B is pushed and popped from the stack at most once, and each push and pop operation is O(1) thus total time taken is O(N).

Totally wrong/incomplete solution -> **1 mark**

O(N^2) Solution -> **9 marks**
O(N^3) Solution -> **7 marks**

Find the 2 largest peak and return the number of peaks between them -> **4 marks**

Algorithms which uses the largest peak as one of the X,Y peaks and vary the other peak to find the maximum peak counts among those X,Y peaks -> **5 to 7 marks**

Do scanning from front to back to find best X,Y and number of peaks between but forget to scan back to front or any stack based equivalent of this solution -> **10 marks**

Stack based algorithms which does not keep a peak count for each peak in the stack but instead uses a global peak count (can easily overcount) -> **6 to 8 marks**

Almost correct stack based algorithms with minor errors e.g forget to pop everything left in stack after going through B to find the maximum number of peaks -> **13 marks**

7. **Scramble [18 marks]**

You have been invited to a game show "Scramble". In this game show, you will be presented with a sequence R_o of N $(N > 1)$ non-repeating integers sorted in increasing order. Then immediately, the game show host will press a "scramble" button which as the name suggest will scramble the sequence so that the integers will no longer be in sorted order. Let the scrambled sequence be R_s.

Now you will be given an integer A which is guaranteed to be in R_s and not the smallest value, and you are supposed to pick out (in any order) including A itself the 1st X largest integers $\leq A$, where $X > 0$. If there are less than X integers $\leq A$ then pick out all integers $\leq A$. Let the set of such integers be A'.

For e.g if R_o= {1,3,7,31,51} and A = 31, X = 3 then A' = {3,7,31}.

if we change X = 5, then A' = {1,3,7,31}

The game will proceed in rounds. For each round you will pick out 2 integers B and C from R_s. Without loss of generality assume $B < C$. Once they are picked, they will be placed in their position i_B and i_C in the original sorted sequence R_o, and all integers D where $B < D < C$ will also be placed in the positions between i_B and i_C although they will also be scrambled and each integer will also be at most $\leq \lfloor 0.3 * (i_c - i_b - 1) \rfloor$ positions away from its sorted position. You can now do two things in the order listed below before the start of the next round

1. Pick out integers to be included in A'.
   An incorrectly picked integer will result in failure.
   You can also choose not to pick any integers at all for the round.
2. Remove all integers from i_B to i_C **OR** keep all integers from i_B to i_C and remove all other integers. The remaining integers are sorted (forming a new R_o) then scrambled to form a new R_s for the next round.
   Removed integers will no longer be available. So if any required integer is removed before it is included in A', the game will end in failure.

e.g of the game play:
If we start with R_s= {3,1,13,7,35,21,89,77,97,100,99} and A = 7, X = 2
for round 1 step 1, if we pick B=7, C=89 then we can have
R_s= {3,1,**7**,13,35,21,77,**89**,97,100,99} where integers between 7 and 89 is placed in the indices between 7 and 89 and they are again scrambled. Assume we don't pick any integer
to be in A' for this round.
For step 2, If we keep everything from 7 to 89 we will scramble those values and have R_s= {13,7,35,21,89,77} for round 2. If we throw everything from 7 to 89 away, we scramble the remaining integers and can have R_s= {3,1,99,97,100} for round 2.

If you can pick out all required integers in **at most $O(logN)$ rounds**, then you will win 1,000,000 dollars!

For each round there is a time limit and since you can only do computation mentally, an additional restriction is that you can only take $O(N)$ time in total to play the game and use at most $O(1)$ additional space.

Now given the afore mentioned restrictions, come up with an algorithm that will solve the problem in at most $O(logN)$ rounds and take in total $O(N)$ time and additional $O(1)$ space.

Non-viable solutions include memorizing the original sequence (it is too long), and using radix sort since it is too mentally challenging to perform (also requires too much space).

The idea is the use a modified form of QuickSort/QuickSelect to find the 1st X largest integers $\leq A$

0.) Let count = X
1.) Scan through R_s and get the smallest value M. ←*O(N) time*
2.) Pick B = M and C = A and let the number be placed in their positions (as well as all values in between which are still scrambled)
3.) Scan from B to C and count the number of items X' in that range.
If X' <= count, select everything value from B to C to be in A'. End of game
If X' > count,
i.) don't pick any value to be in A' ← *You're not sure which values should be in*
ii.) keep everything from B to C for round 2 ←*everything in A' must be found somewhere from B to C, so keep that range of values throw away everything else. Time taken is at most O(N)*

For subsequent rounds do the following
4.) Pick C = largest number in R_s ← again can *find this by scanning through* R_s. *O*($|R_s|$)
Pick B = value in middle of R_s. If this value is the largest number in R_s (i.e C) simply pick the one to the left or right ← *find the middle position by scanning through* R_s. *O*($|R_s|$)
5.) Scan from B to C to count number of items X' in that range.
If X' <= count,
i.) count = X-X'
ii.) select every value from B to C to be in A' ← *O*($|R_s|$) *time required.*
iii.) throw away everything from B to C keep the rest.
If X' > count, keep everything from B to C throw away the rest.
6.) if count == 0 then end of game, else repeat 4.)

Analysis:
In each round picking out B, C and picking the integers to include in A' only takes time linear to the size of R_s of the current round.
Now for each round after the 1st round, you are always removing at least 20% of the value since we pick the C to be the largest value and B to the value in the middle index which in the worst case is the integer that is either 0.3*$|R_s|$ to the left or right of the middle index. Regardless we will throw away at least 0.2*$|R_s|$ of the values
So the size of $|R_s|$ is represented by the following sequence for each round

$$\left(\frac{4}{5}\right)^0 * N, \left(\frac{4}{5}\right)^1 * N, \left(\frac{4}{5}\right)^2 * N, \dots, 2$$

We will stop when we have 2 values left. The number of round is then reflected in the power of 4/5.
We stop when

$$\left(\frac{4}{5}\right)^k * N = 2 \rightarrow \left(\frac{4}{5}\right)^k = \frac{2}{N} \rightarrow klg\frac{4}{5} = lg\frac{2}{N} \rightarrow k = \frac{\lg\frac{2}{N}}{lg\frac{4}{5}} \rightarrow k = \frac{lg2 - lgN}{lg4 - lg5} \rightarrow k = \frac{lgN - lg2}{0.32} \rightarrow k = O(lgN)$$

The total time taken is then sum of the same sequence and we have

$$\left(\frac{4}{5}\right)^0 N + \left(\frac{4}{5}\right)^1 N + \left(\frac{4}{5}\right)^2 N + \cdots + \left(\frac{4}{5}\right)^k N = N\left(\left(\frac{4}{5}\right)^0 + \left(\frac{4}{5}\right)^1 + \left(\frac{4}{5}\right)^2 + \cdots + \left(\frac{4}{5}\right)^k\right) = O(N)$$

A lot of answers do not say what is the B and C to be picked for the round. That is crucial to picking all X largest integers <= A in O(lgN) round.

Some completely wrong solution -> **1 mark**

Correct solution requiring O(N) rounds and fulfilling requirements -> **9 marks**

Correct solution using sorting or correct solution using more than O(1) space-> **3 marks**
(fails requirement that only O(N) time/O(1) space in total. There is no array of the input, the sequence is simply presented to you so you have to use O(N) already to remember the entire sequence.)

Incomplete/incorrect solution that have the idea of picking C=A and B=smallest element in 1st round, but then
1. C and B not specified correctly for subsequent round
2. Did not specify how to pick the integers to be included in A'
3. Unclear or wrong in which is the part of R_s to keep for the next round
Any combination of the above
-> **4 to 6 marks**

Minor errors to correct solution -> **-2 marks** per minor error.

== END OF PAPER ==