NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
Midterm (20%)
AY2019/20 Semester 1

CS2040 – Data Structures and Algorithms

5 October 2019 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 Thirteen (13) 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	30	
Q6	30	
Q7	28	
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 **recursion1(*N*)** of following program is $O(NlgN)$.

[true/false]

```
public static void recursion1(int i) {
  if (i <= 1)
    return;
  else
    for (int j=0; j < i; j++)
      recursion2(j/2);
}

public static void recursion2(int i) {
  if (i > 1)
    return;
  else
    recursion1(i);
}
```

Answer: false

For any n > 1, 1st call to recursion1 will go into else clause where the for loop will iterate n times. Each iteration calls recursion2 and for any i > 1 recursion2 immediately returns (base case), otherwise it calls recursion1 again however this call to recursion1 will now immediately return since i <= 1. Thus recursion2 will at most take O(1) time regardless of value passed in. So in total the time taken is simply O(N) due to the for loop.

2. Given the same input, insertion sort will always have time complexity equal to or worse than mergesort. **[true/false]**

Answer: false

If input of size N given is already sorted, insert sort will only take O(N) time since or each iteration of the outer for loop, the inner for loop for insertion sort will only execute once and determine that value at current index i is already in its correct position.
However for mergesort regardless of whether input is sorted or not it will still take O(NlogN) time.

3. If we implement a queue ADT using a circular linked list, only the time complexity for either enqueue or dequeue operation is $O(1)$ time but not both. **[true/false]**

   Answer: false

   If we maintain a tail reference, then enqueue which is equivalent to inserting to the tail of the circular linked list can be done in O(1) time as follows

   newNode.next = tail.next
   tail = newNode
   tail = tail.next

   dequeue which is equivalent to remove from the head of the circular linked list can also be done in O(1) time as follows

   tail.next = tail.next.next;

4. Given 2 arrays of size N each that contains unsorted integers that have values between 0 and 2^N, we can check whether they contain the same elements in $O(N)$ time by first performing radix sort on both in $O(N)$ time then simply go from index 0 to N-1 to check if the value at the current index in both array is the same or not. **[true/false]**

   Answer: false

   Radix sort is $O(dN)$ or more precisely $O(d(N+b))$ where b is the number of buckets/queues and d is the number of digits in the integers to be sorted. In this case the integers can be up to 2^N, so the number of digits is $O(N)$ and not bounded by a constant. Thus it will take $O(N^2)$ instead of $O(N)$.
   Even in the event where we convert the integers into a base N number and use N buckets instead and not count the time to do the conversion, there will still be $O(N/lgN)$ number of digits per base N number and so Radix sort is still $O(N^2/lgN)$ and not $O(N)$.

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. **Text manipulation [30 marks]**

a) After taking CS2040 for half a semester, you have decided to put what you've learned to good use and simulate a rudimentary text editor. To make it simple you only capture input from the keys '0' to '9', 'a' to 'z', 'A' to 'Z', the space key ' ', the enter key '\n' and the backspace key which is captured as '\b'. These inputs are then stored as a string. Given such a length N input string $\boldsymbol{A}$, give an algorithm and the associated data structure to display the final text in $O(N)$ time. **[18 marks]**

E.g: Input string $\boldsymbol{A}$: "This is CSS\b204000\b\b midterm\n"

Output text: "This is CS2040 midterm\n"

Clarification during midterm: home key brings the cursor all the way to the start of the text and end key brings it all the way to the end.

Answer:

1.) Use a stack S to store each character read in from A

2.) For each character c from index 0 to N-1 in A

```
If (c != '\b')
  S.push(c)
else
  if (!S.empty())
    S.pop()
```

3.) Create a character array B of size S.size() and let i = S.size()-1

4.) while !S.empty()

```
B[i--] = S.pop()
```

5.) Go through B and print out each character.

b) After you tried out your simulator, you discovered that somehow the backspace key press has a problem and instead of a backspace it will randomly be replaced by a home key or end key. Going with the error, you now want to change your algorithm and associated data structure so that it will output the final text with the inclusion of these random home and end keys and the exclusion of the backspace key. You can assume home key is '\H' and end key is '\E'. Your algorithm should still run in $O(N)$ time. **[12 marks]**

E.g: Input string $\boldsymbol{A}$: "This is CS2040 \Hmid\Eterm\n"

Output text: "midThis is CS2040 term\n"

Answer:

1.) Create a tailed linked list L that stores strings and let i = 0, let j = 0, and let home = false

2.) while (true)

If (j == Length of A-1)

AddString(A,i,j,L,home) → linear in substring length from i to j

break

if (A[j] == '\H')

AddString(A,i,j-1,L,home) → linear in substring length from i to j

home = true

i = j+1

else if (A[j] == '\E')

AddString(A,i,j-1,L,home) → linear in substring length from i to j

home = false

i = j+1

j++

3.) Go through L and print out all strings in L → O(N)

AddString(array A, int i, int j, tailed Linked List L, boolean home)

1.) if A[j] == "\H" or "\E" // ignore A[j]

Let S' = substring of A from index i to j-1 → linear in length of S'

else

Let S' = substring of A from index i to j → linear in length of S'

2.) if (!home)

add S' to tail of L → O(1)

else

add S' to head of L → O(1)

Total time complexity is O(N).

6. **Poison manufacturing facility [30 marks]**

A facility manufactures a type of poisonous liquid. It will then bottle those liquid into bottles that can contain different integer volume M of liquid where $1\ litre < M < 10\ litre$. Batches of N bottles will be sent on a conveyor belt one after the other to be filled with the poisonous liquid. The bottle at the front will be processed first while the bottle at the back will be processed last. The bottle at the front will then be filled with up to 3 litres of liquid or until the bottle is full whichever is less, and it takes 10 seconds per litre. If the bottle is not full it will be sent to the back of the conveyer belt to be filled again (assume no time is taken here). After processing a bottle (whether it is full or not), the next bottle takes 10 seconds to come in and be positioned correctly. You may assume that at the start the 1st bottle to be processed is already positioned correctly.

a.) Given an array A of N ($N > 1$) integer values indicating the volume of N bottles to be placed on the conveyer belt from index 0 to index $N - 1$ in that order, and H the number of seconds, give the algorithm and associated data structure that will compute the number of bottles that are completely full after H seconds. **[18 marks]**

E.g1: Given A = [1,3,7,1,4] and H = 190, the number of full bottles is 3 after 190 seconds.

E.g2: Given A = [1,3,1,4,7] and H = 190, the number of full bottles is 4 after 190 seconds.

Answer:

Simulate the process using a queue!

1.) Let Q be a queue, fbcount = 0, H' = H

2.) For i from 0 to N-1

    Q.enqueue(A[i])

3.) while (H' > 0 && !Q.empty())

    If (H' < H)

      H' = H'-10

    let cur = Q.poll()

    H' = H'-min(30, cur*10)

    cur = cur-3

    if (cur > 0 and H' > 0)

     Q.enqueue(cur)

    else if (cur < 0 && H' >= 0)

      fbcount = fbcount+1

4.) Print out fbcount

[*This is a difficult sub-question]
Suddenly the facility suffers a rupture along the entire length of the pipes carrying the poisonous liquid. The facility is built around a safety moat filled with chemicals that would neutralize the poisonous liquid should it escape and flow into the moat, thus containing the liquid from spreading to the outside world. However, the facility itself is not so lucky.

This facility is a single floor rectangular structure of width **100 meters** with strangely no roof and running along the 2 sides of the facility length-wise are walls of $\boldsymbol{M} + \mathbf{1}$ meters high each where $\boldsymbol{M}$ is an integer. Along the length of the 2 side walls, $\boldsymbol{N}$ walls of integer heights ranging from **3 meters to $\boldsymbol{M}$ meters** are placed at intervals of **10 meters** perpendicular to the side walls to create rooms and since the pipes run through all the rooms, they all start filling with the poisonous liquid. Once the liquid in a room reaches over the height of either of its room walls, it will overflow into the adjoining room and so on, until it flows into the moat. You can assume the moat can contain and neutralize an infinite amount of the liquid. Your task is to now calculate what is the maximum volume of liquid contained within the facility. For simplicity sake assume walls are 0 meters thick and ignore all the equipment in the facility.

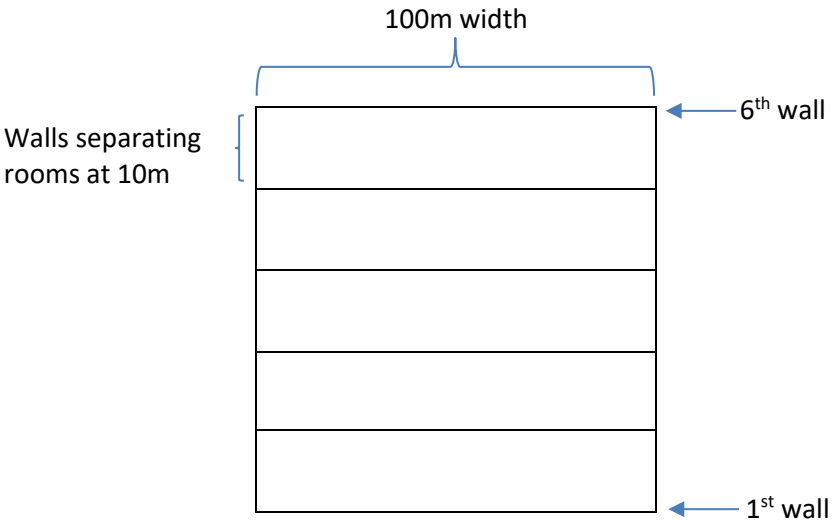
When the facility contains the maximum volume of liquid, the liquid will fill the facility such that for all pairs of walls W and W' (assuming without loss of generality that W is to the left of W') of height h and h' respectively such that

1. there are no walls taller than $\min(h, h')$ between W and W'
2. If $h \leq h'$ (W is the shorter wall), there are no walls taller than h to the left of W
3. If $h' \leq h$ (W' is the shorter wall), there are no walls taller than h' to the right of W'

then all rooms between W and W' will be submerged up to height $\min(h, h')$.

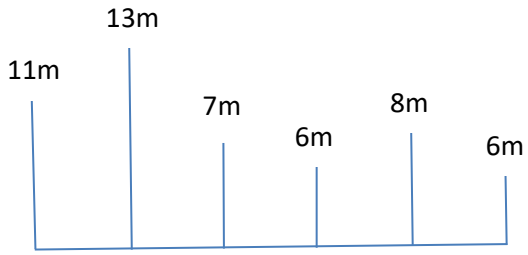
The input given to you is an array $\boldsymbol{A}$ of the height of the walls separating the rooms starting from the 1st wall to the N^{th} wall.

Example top down view of the facility:



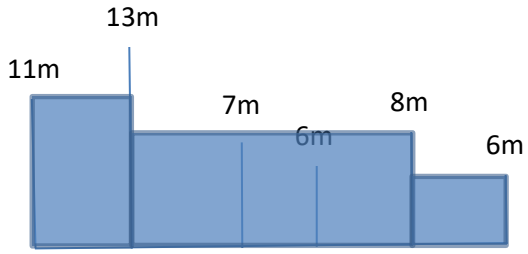
The 2 side walls are height M+1 each while the 6 walls have integer height ranging from 3 to M.

Example side view of the facility (ignoring the side wall):



The heights of the walls from the 1st to the 6th is 11,13,7,6,8,6 meters respectively.

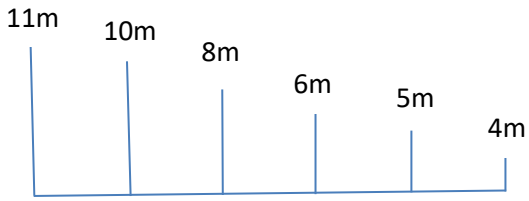
In the example above, the maximum volume of poisonous liquid contained in the facility can be shown below based on how the liquid submerges the rooms.



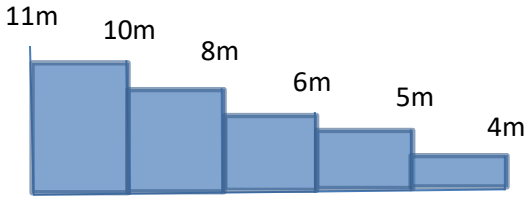
(11*10*100)+(8*30*100)+(6*10*100) = 41,000 m^3

Any excess liquid will simply flow into the moat (since there is no roof).

In another example below



the maximum volume of poisonous liquid contained in the facility can be shown as follows



10*10*100+8*10*100+6*10*100+5*10*100+4*10*100 = 33,000 m^3

b) Given input array $\boldsymbol{A}$ of the height of the N walls from 1st wall to N^{th} wall, come up with an algorithm and appropriate data structure so that the maximum volume of poisonous liquid that can be contained in the facility can be computed in $O(N)$ time. **[12 marks]**

Answer using a stack:

1.) Let S be a stack that store integers pairs <X,Y> where X represents the height of a wall, and Y represents how many rooms are submerged to a height of X to the left of the wall. S will contain walls in order of decreasing height from bottom to top of stack.

2.) Let maxVol be the maximum volume of poisonous liquid contained in the facility and is initialized to 0

3.) Go through A for index = 0 to N-1

```
if (index == 0)
  S.push(<A[0],0>)
else
  Let X' = A[index], Let Y' = 1, Let H = 0
  while (!S.empty() and S.peek().X <= X')
    H = S.peek().X
    Y'+= S.pop().Y
  If (S.empty())
    maxVol += Y'*10*100*H
    S.push(<X',0>)
  else
    S.push(<X',Y'>)
```

4.)
```
while (!S.empty())  // pop all remaining items in S and add to maxVol
  maxVol += S.peek().X*S.pop().Y*10*100
```

5.) Print maxVol

Each wall is at most pushed into the stack once and removed from the stack once. Thus there are at most O(N) push and pop operations which costs O(1) each. So total time taken is O(N).

Answer without using a stack (courtesy of Wang Zhi Jian):

First, let's assume that the heights of all walls are distinct. We will handle the case where not all heights are distinct later. We make the following claim:

Claim 1

Consider any pool of liquid surrounded by two walls on its left and right. Let the heights of the two walls be A[i] and A[j] respectively. Then, one of the following must hold:

- A[i] < A[j] (the wall on the right is taller than the wall on the left)
- A[i] > A[j] (the wall on the left is taller than the wall on the right)

Since one of the above conditions must hold for each wall, we can first find all pools that satisfy the first condition, and then find all pools satisfying the second condition. Then, we would have found all the pools that can possibly be formed.

To find all pools satisfying the first condition, we can first start from the leftmost wall, and consider that to be the left wall enclosing the first pool. Then, we iterate from left to right to find the next wall higher than the first wall to be the right wall enclosing the first pool. Then, we can let this wall be the left wall enclosing the second pool, and we repeat this process.

To find all pools satisfying the second condition, we use a similar procedure. Start from the rightmost wall and consider that to be the right wall enclosing the first pool. Then, iterate from right to left to find the next wall higher than the first wall to be the left wall enclosing the first pool. Then, let this wall be the right wall enclosing the second pool, and repeat this process.

The pseudocode of this algorithm is shown below

Algorithm 1: Computes total volume of all pools, assuming that heights of walls are distinct

```
Let A[1..N] be the heights of the walls from left to right.

// This function computes the volume of a single pool
function calculate_volume(width, height)
        return width * 10 * height * 100

total_volume = 0 // total volume

// Find all pools satisfying height(right) > height(left)
left_wall_index = 1 // position of left wall for current pool
for i = 2 to N
        if (A[i] > A[left_wall_index])
                total_volume += calculate_volume(i - left_wall_index,
                                                 A[left_wall_index])
                left_wall_index = i

// Find all pools satisfying height(left) > height(right)
right_wall_index = N // position of right wall for current pool
for i = N - 1 to 1
        if (A[i] > A[right_wall_index])
                total_volume += calculate_volume(right_wall_index - i,
                                                 A[right_wall_index])
                right_wall_index = i

print total_volume
```

This algorithm works if the heights of all walls are distinct. However, in the question, the heights of all walls may not necessarily be distinct. So, in the question, our conditions actually look like this:

Claim 2
Consider any "pool" of liquid surrounded by two walls on its left and right. Let the heights of the two walls be A[i] and A[j] respectively. Then, *at least* one of the following must hold:

- A[i] ≤ A[j] (the wall on the right is at least as tall as the wall on the left)
- A[i] ≥ A[j] (the wall on the left is at least as tall as the wall on the right)

Notice that now, some pools actually satisfy both conditions instead of just one condition: the pools surrounded by two walls with the same height on both sides. So, Algorithm 1 may double count some pools, and we may get an incorrect total volume.

But *exactly* which pools are actually double counted? We make another observation:

Claim 3

Let H be the maximum height over all walls. All pools where both walls surrounding the pool are of height H will be double counted.

Challenge: Prove it!

So, we can modify the algorithm to ignore all pools where both walls surrounding the pool are of height H in the second loop above, when iterating through the walls from right to left. The modified pseudocode is shown below (modifications to Algorithm 1 shown in blue):

Algorithm 2: (Solution for 6b.) Computes total volume of all pools

```
Let A[1..N] be the heights of the walls from left to right.

function calculate_volume(width, height)
        return width * 10 * height * 100

total_volume = 0 // total volume

// Find max height over all walls
max_height = 0
for i = 1 to N
        if (A[i] > max_height)
                max_height = A[i]

// Find all pools satisfying height(right) > height(left)
left_wall_index = 1 // position of left wall for current pool
for i = 2 to N
        if (A[i] >= A[left_wall_index])
                total_volume += calculate_volume(i - left_wall_index,
                                                 A[left_wall_index])
                left_wall_index = i

// Find all pools satisfying height(left) > height(right)
right_wall_index = N // position of right wall for current pool
for i = N - 1 to 1
        if (A[i] >= A[right_wall_index])
                if (A[i] > A[right_wall_index] || (A[i] == A[right_wall_index] &&
                                                   A[i] != max_height))
                   total_volume += calculate_volume(right_wall_index - i,
                                                    A[right_wall_index])
                right_wall_index = i
print total_volume
```

Time: O(N) and Space is O(1)

7. **Laser Defense [28 marks]**

a) Company W houses some of the most advanced military weapons in the world. Each floor of the company is laid out in an N by M grid (N is the # of rows and M is the # of columns), where the top left cell is at row 0, column 0, i.e (0,0) and the bottom right cell is at row $N-1$ and column $M-1$, i.e ($N-1$,$M-1$). In order to protect their assets, K ($k \leq \min(M,N)$ number of laser defenses are installed on every floor. Each laser defenses' position is given by its row and column coordinate on the grid. Now a laser defense at coordinate (X,Y) will have a laser barrier shooting out all the way along both row X and column Y. Due to this feature of the laser defense, the following non-conflict property must hold:
No laser defense can be placed on the same row or column as another laser defense otherwise their laser barrier will destroy each other.

E.g: Given the 2 following grid and laser defenses, Fig 1 is an invalid configuration while Fig 2 is a valid configuration.

(1,0)		(1,2)	
	(2,1)		
			(4,3)

Fig1: 5x4 grid, with laser defenses as indicated by their coordinates. Lasers at (1,0) & (1,2) violates the non-conflict property

(1,0)			
	(3,1)		
			(4,3)

Fig2: 6x4 grid, with laser defenses as indicated by their coordinates. No lasers violate the non-conflict property here.

You may assume there is a coordinate class with attributes x and y representing the row and column index respectively.

Now given N, M, K and an array A of K coordinates representing the position of the K lasers, give an algorithm for the method **Valid(N, M, K, A)** that returns true if the K lasers are placed in a non-conflicting configuration otherwise return false. **Valid(N, M, K, A)** must run in $O(K)$ time. **[10 marks]**

Answer:

Non-conflict property means no row value and column value can be repeated for the set of laser defense positions.

```
boolean Valid(N,M,K,A)
    1.) Let H and H' be two hash sets. H stores row values and H' stores column values
    2.) For all coordinates (X,Y) in A
          if (H.get(X) == null)
            H.put(X)
          else
             return false
          if (H'.get(Y) == null)
            H'.put(Y)
          else
            return false
    3.) return true
```

b) [*This *may be* a difficult sub-question]
Company W has changed their laser defense so that the laser defenses are not activated from the start and also the non-conflict property is no longer needed and now the number of laser defenses K is such that $4 <= K <= max(N, M)$. Sensors are placed on the floor which will detect movement, so that whenever any intruder has moved a certain distance within a floor the laser defenses will be activated. In this version, if 4 activated laser defenses are placed in a rectangular configuration and an intruder is within that rectangular region, the 4 laser defenses will erect a rectangular laser barrier to trap the intruder. An intruder may be within the rectangular region of many such laser configurations, but only the 4 laser defenses which forms the smallest rectangular region trapping the intruder will be activated (if there are many such smallest configurations, one of them will randomly be activated).

Given 4 laser defenses in some rectangular configuration, an intruder at (X_1,Y_1) is considered within their rectangular region if

X coordinate of a leftmost laser < X_1 < X coordinate of a rightmost laser
Y coordinate of a topmost laser < Y_1 < Y coordinate of a bottommost laser
E.g: In the 2 following grid, where # marks the position of the intruder, the bold and underlined laser configuration will be activated, and its rectangular region are the shaded cells.

(1,0)		**(1,2)**	(1,3)
	#		
(3,0)		**(3,2)**	(3,3)

(1,0)	**(1,1)**		**(1,3)**	(1,4)
		#		
	(4,1)		**(4,3)**	
(5,0)				(5,4)

Now given N, M, K, A as before and the coordinates P of an intruder, give the best algorithm you can think of for the method **SmallestArea**($\boldsymbol{N}, \boldsymbol{M}, \boldsymbol{K}, \boldsymbol{A}, \boldsymbol{P}$) which will return the area of the activated rectangular laser configuration that will trap the intruder. If there is no laser configuration that can trap the intruder return -1. Analysis the time complexity of your algorithm. **[18 marks]**

Answer:
O(K^2) time algorithm:

Key idea: For lasers in a rectangular configuration, if we have 2 of the laser positions that are in a diagonal, we can immediately get the positions of the 2 other lasers in the other diagonal.

SmallestArea(N,M,K,A,P)

1.) For each coordinate in A hash to a hash set H.
2.) let area = 0 where it represents area of activated laser configuration
3.) for i = 0 to K-2
    for j = i+1 to K-1
      let lp = A[i] and lp' = A[j] // get every pair of laser defense
      if (lp.x != lp'.x && lp.y != lp'.y) // lp and lp' can form a diagonal
        let minX = min(lp.x,lp'.x)
        let maxX = max(lp.x,lp'.x)
        let minY = min(lp.y,lp'.y)
        let maxY = max(lp.y,lp'.y)
        let lp1 = (lp.x, lp'.y)
        let lp1' = (lp'.x, lp.y)
        if (H.get(lp1) != null and H.get(lp1') != null) // the other diagonal exists
          if (minX < p.x < maxX and minY < p.y < maxY) // intruder within rect. region
            if (area == 0 || area > (maxX-minX-1)*(maxY-minY-1))
              area = (maxX-minX-1)*(maxY-minY-1)

4.) if (area == 0) return -1 else return area

== END OF PAPER ==