# CS2040 2020/2021 Semester 2 Midterm

## MCQ

This section has 10 questions and is worth 30 marks. 3 marks per question.

Do all questions in this section.

1. What is the time complexity of the following method?

```
public int foo(int n) {
    if (n <= 0) {
        return n;
    }
    int ans = foo(n/2);
    while (n > 0) {
        ans = ans + n;
        n = n / 2;
    }
    return ans;
}
```

a. O(log n)
**b. O((log n)$^2$)**
c. O(n)
d. O(n log n)

2. The following array is the result of running 2 iterations of a certain sorting algorithm on an initially unsorted array of integers:

| 7 | 9 | 13 | 4 | 18 | 48 | 29 | 67 | 80 |
|---|---|---|---|---|---|---|---|---|

Which of the following algorithms (as taught in lecture) could be the sorting algorithm in question?

i. Bubble sort (1 iteration is defined as 1 run of the outer for loop)
ii. Insertion sort (1 iteration is defined as 1 run of the outer for loop)
iii. Quick sort (2 iterations here means the 1st 2 recursive calls to quicksort)

a. (i) and (ii)
b. (ii) and (iii)
c. (i) and (iii)
**d. (i), (ii), and (iii)**

3. The numbers 11, 21, 24, 35, 43, 46 are inserted one at a time into an initially empty hash table of size 11. The hash function is h(key) = key % 11, and quadratic probing is used as the collision resolution technique. No deletions are involved, and the exact order in which the numbers are inserted is unknown. Which of the following hash tables could be a possible result of these insertions?

(i)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | | 35 | 24 | 21 | | | 46 | | | 43 |

(ii)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | | 46 | 43 | | | 24 | 35 | | | 21 |

(iii)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 43 | 11 | 24 | 46 | | | 35 | | | | 21 |

a. (iii) only
b. (i) and (ii)
**c. (ii) and (iii)**
d. (i), (ii), and (iii)

4. Given an array containing **n** integers in descending order, we wish to sort it into ascending order by using a sorting algorithm as discussed in lecture (as opposed to simply reversing the array). Choose the best sorting algorithm (in terms of worst case time complexity) to solve this problem.

a. Insertion sort
b. Improved bubble sort
**c. Merge sort**
d. Quick sort

5. Which of the following data structures can support all queue operations in O(1) time?

i. Tailed linked list
ii. Circular linked list (only head reference, no tail reference, singly linked)
iii. Circular linked list (only head reference, no tail reference, doubly linked)

a. (i) only
**b. (i) and (iii)**
c. (ii) and (iii)
d. (i), (ii) and (iii)

6. You are given a queue of **n** integers (the value of **n** is provided to you), and you wish to reverse the contents of the queue. You are only allowed to use one additional data structure, though you can declare a few other primitive variables, where necessary. Additionally, your program should run in **less than $O(n^2)$** time in the worst case. Which data structure would be suitable for this task?

i. A stack
ii. Another queue
iii. A HashSet

Note that the options are independent ie. picking (i) and (ii) means it is possible to solve the problem with either a stack, or another queue.

**a. (i) only**
b. (ii) only
c. (i) and (iii)
d. (ii) and (iii)

7. We attempt to insert the following keys (in this order) into an initially empty hash table.

   13, 18, 21, 14, 18, 15

   The hashing function is h(key) = key % 7, but the collision resolution technique used by the hash table is unknown. However, it is known that the hash table is of size 7.

   What is the load factor of the hash table (rounded to 2 decimal places)?

a. 0.57
**b. 0.71**
c. 0.86
**d. Insufficient information to determine for certain**

8. What is the time complexity of the following function?

```
public int bar(int n) {
    int ans = 0;
    for (int i = 0; i < n; i++) {
        ans = ans + n;
        n = n - i;
    }
    return ans;
}
```

a. O(log n)
b. $O((\log n)^2)$
**c. $O(n^{0.5})$**
d. O(n)

9. You are given an array of sorted distinct integers. You are asked to find the k-th smallest element in this array. The best algorithm to do so can run in:

**a. O(1) time**
b. O(log n) time
c. O(n) time
d. O(n log n) time

10. You are given an unknown data structure X, which is initially empty. You are unsure as to whether it is a stack or a queue, but you are allowed to use the following methods on X:

    **insert(int e):** adds the integer to the top of the stack (push(e)) if X is a stack, or adds the integer to the back of the queue (enqueue(e)) if X is a queue.

    **remove()**: removes and returns the top of the stack (pop()) if X is a stack, or removes and returns the front of the queue (dequeue()) if X is a queue.

    Calling any of the above methods counts as 1 operation. How many operations do you need to determine for sure if X is a stack or a queue?

a. 2
**b. 3**
c. 4
d. 5

## Analysis

This section has 3 questions and is worth 12 marks. 4 marks per question.

**Please select True or False and then type in your reasons for your answer.**

Correct answer (true/false) is worth 2 marks.

Correct explanation is worth 2 marks. Partially correct explanation worth 1 marks.

Do all questions in this section.

11. The time complexity of F(n) as given below is $O(n^3)$.

```
public int F(int n) {
  if (n == 0)
    return 0;
  else {
    x = F(n-1);
    int m = 0;
    for (int i = 0; i < x+1; i++)
      m += 1;
    return m;
  }
}
```

False
F(N) will call F(N-1) which will call F(N-2) ….. F(0)
Tracing backwards:

F(0) will return 0
F(1) <- For loop will loop 0+1 = 1 time and return 1
F(2) <- For loop will loop 1+1 = 2 time and return 2
…
F(N) <- for loop will loop N times and return N
Thus total iterations = N+(N-1)+(N-2)+…+1 = O(N^2).

12. In the separate chaining collision resolution technique, if we bound the load factor to a constant C (i.e once the load factor exceeds C, we re-hash to a larger hash table to keep the load factor ≤ C), we can ensure that insertion, deletion and searching in the hash table can be done in **worst case** O(1) time.

    False.

    Counter example: a Hash table of size N where N items are inserted and they all hash to one particular index in the hash table. The load factor would be N/N = 1 but the worst case for insertion, deletion and searching would be O(N) since the linked list at that index is of size N.

13. It is possible for a stack implemented using circular linked list with only a tail reference and no head reference to achieve worst case **O(1)** for pop() and push(item).

True.

You can simply use tail.next as the head reference and perform removeFront for pop() and addFront for push.
More specifically, they can be implemented as follows:

```
push(item)
 let n be node created from item
 if tail == null
   tail = n
   tail.next = tail
 else
   n.next = tail.next
   tail.next = n

pop()
 if tail == null
   return null
 else
   temp = tail.next
   if (tail.next == tail)
     tail = null
   else
     tail.next = tail.next.next
   return temp
```

## Structured Questions

This section has 5 questions and is worth 58 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.

Partial marks will be awarded for correct answers not meeting the time complexity required.

**Q14 and Q15 are related. Consider them part a and part b of the same question.**

14. Implement a queue ADT as efficiently as possible using a stack ADT (you can use multiple stacks) and some extra variables if necessary.
The only operations you can make use of in for the stack ADT is **empty()**, **push(item)**, **pop()**. You may assume all the operations run in **O(1)** worst case time. The operations you must implement for the queue ADT are **offer(item)**, **poll()**. Give the time complexity for your implementation of each of the queue operations.

**[11 marks]**

```
Let S be the stack

offer(item) {
   S.push(item) <- O(1) time
}
Time complexity of offer is O(1)

poll() {
   if (S.empty())
     return null
   Let S1 be a stack
   While (!S.empty()) <- O(N)
      S1.push(S.pop())
   X = S1.pop() <- O(1)
   While (!S1.empty()) <- O(N)
      S.push(S1.pop())
   return X
}
Time complexity of poll() is O(N) where N is number of items in the queue
```

// another way is to make offer O(1) and poll O(n) by maintaining the correct order in the stack

A more advanced version where amortized poll() is O(1)

Let S1 and S2 be 2 stacks

```
offer(item) {
   S1.push(item) <- O(1) time
}
```
Time complexity of offer is O(1)

```
poll() {
   if (S2 and S1 is empty)
      return null
   if  (S2.empty())
      while (!S1.empty()
         S2.push(S1.pop())
      X = S2.pop() <- O(1)
   else
      X = S2.pop() // everything in S2 is already in correct ordering
   return X
}
```

For each item, it is only every push and popped 2 twice (once to S1 then pop and pushed to S2 and the finally popped from S2) so amortized cost to dequeue/poll N items is only O(1) time.

15. Now implement the queue ADT as efficiently as possible using a hashtable (you can only use 1 hashtable) and some extra variables if necessary. For the hashtable, you have the **insert(key,value)**, **retrieve(key)** and **delete(key)** operation which you may assume take worst case O(1) time. Again the operations you must implement for the queue ADT are **offer(item)** and **poll()**. Give the time complexity for your implementation of each of the queue operations. **[11marks]**

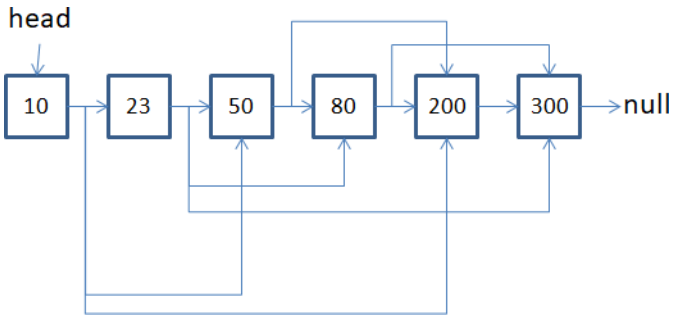
```
Let H be the hashtable
Let count = 0
Let min = 0

offer(item) {
   H.insert(count,item)
   count++
}
Time complexity of offer is average O(1)

poll() {
   if (min > count)
     return null
   let X = H.retrieve(min)
   H.delete(min)
   min++
   return X
}
Time complexity of poll() is average O(1)
```

16. You are given a linked list of **N** sorted integers (stored in the **val** attribute of the node) where each node at index **i** ($0 \le i \le N-1$) does not only have a next reference to the next node in the sorted list, but has a reference to the nodes at $i+2^0$, $i+2^1$, $i+2^2$ until $i+2^k \le N-1$. You can assume that for each node at **i** there is an array **R** containing all these references with **R**[0] containing the reference to node at $i+2^0$ , **R**[1] containing the reference to node at $i+2^1$and so on. There is only a head reference to the first node of this sorted linked list and no tail reference.

An example is shown below:




In the above diagram head.val will give you 10 the value stored in the first node. head.R[0] will access the node containing 23 and head.R[1] will access the node containing 50. For the last node, R would be null.

Now give an algorithm that return true if a given integer **x** is in the sorted list else return false. Your algorithm should run in worst case **O(logN)** time. **[13 marks]**

```
m = 0, i = 0
k = floor(log(N-1))
cur = head
While (true)
   if (k < 0) return false
   for j from k down to 0
      if (cur.R[j].val == x)
         return true
      else if (cur.R[j].val < x)
         cur = cur.R[j]
         i = i+2^j
         k = floor(log((N-1)-i))-m ← important! In the new node we jump to, we don't start from the
                                     furthest node we can jump to but rather the furthest node within the
                                     rightmost boundary of the current range
         break
      else
         m++  ← keeps track of reduction of the rightmost  index
```

The worst case occurs when x is not in the sorted list. Now if we start from the head node, the index of the left and right most nodes we have to consider if 1 and N-1 respectively (so the range is O(N) at the start)

Scenario 1: each time we go into the else if clause the leftmost index will increase by 2 times so we will have halved the range.

Scenario 2: each time we move down the index of the R (else clause), the rightmost index will decrease by half so again we will have halved the range.

We can at most increase the leftmost index O(lgN) times before leftmost index > rightmost index or we can at most decrease the rightmode index O(lgN) times before the rightmost index < leftmost index. Thus total iterations of the for loop is at most O(lgN) for all iterations of the while loop, and the time complexity if O(lgN)

17. The Arithmetica continent is home to a strange kind of creature called numbered animals. As the name suggests each animals is born with a pattern that resembles a positive integer and each animal will have a unique number (no repeats) that can be arbitrarily big. There are in total **M** (**M** > 100) number of such creatures.

    The animals are organized into **20** domains with each domain being ruled by a queen. The queen has the largest number among all animals that belong to that domain. As an example:

    If there are only 3 domains, one having a queen with number 50, one having a queen with number 200 and the last one having a queen with number 300; the one with number 50 can only rule over at most 49 animals (number 1 to 49) and the one with number 200 can only rule over at most 149 animals (number 51 to 199), and the one with number 300 can only rule over at most 99 animals (number 201 to 299).

    The domain of the queen with the largest number (which must also be the largest numbered animal among all the **M** animals) is considered the 1st domain, the queen with the 2nd largest number the 2nd domain and so on until the 20th domain.

    Now
    1.) given array **A** of size **M** containing the number of all **M** numbered animals in no particular order
    2.) given array **B** of size 20 containing the array index (0-based) of all 20 queens in **A**, again in no particular order

    Give an algorithm to answer the following query as efficiently as possible and give its time complexity:

    DomainOfNumberedAnimal(**x**) - Given **x** a positive integer return the domain (1 to 20) that it belongs to. If it does not belong to any domain, return -1

    A small example is given below (**M** = 10, number of domains = 3):

    **A** = {5,100,142,232,11,325,8,21,10000,43}, **B** = {3,8,1}

    Here the queens will be 232, 10000, 100 respectively.

    Given the above:

    1. DomainOfNumberedAnimal(142) should return 2 (2nd domain).
    2. DomainOfNumberedAnimal(300) should return -1.

    If necessary, give an algorithm to perform a preprocessing step before any query comes in. The preprocessing step must run in worst case ≤ **O(M)** time **[13 marks]**

Preprocessing step:

1.) Go through B and for each index i get the queen's numbering from A[B[i]] and insert into another array C. ← *O(1) since B is fixed at size 20*.
2.) Sort C in ascending order. ← O(1) time since size is fixed at 20.
3.) Create hashtable H
4.) For each x in A ← O(M)

```
        For i from 0 to 19 <- constant number of iterations of inner loop, so O(1)
            If x < C[i]
                H.put(x) = 20-i
```

Time complexity for step 4 is O(M)

Total time for preprocessing step is O(M)

DomainOfNumberedAnimal(x): ← O(1) time

```
If (H.get(x) == null) return -1
else return H.get(x)
```

18. The game "**Tetritis**" is the lastest craze in town.

For each round in this game, you are given **N** (**N** $\geq$ 1) horizontal line segments which are specified by a triple (**x**,**y**,**length**) where **x**,**y** ( 0 ≤ **x**,**y** ≤ **M**-1 and **M** ≥ $\mathbf{N}^{\log \mathbf{N}}$) are integers indicating the x-coordinate and y-coordinate of the left end of the line segment, and **length** is the length of the line segment.

There will be a leftmost line segment whose left end starts from x = 0 and a rightmost line segment whose right end will end at x = m-1.These **N** line segments have no intersection with each other with regards to the x-coordinate they span, but will be contiguous in terms of the x-coordinate they span (i.e there will be some line segment occupying each x-coordinate from 0 to M-1).

You can assume the **N** line segments are stored as the above mentioned triples in an array **H** sorted by increasing x-coordinate of their left most end.

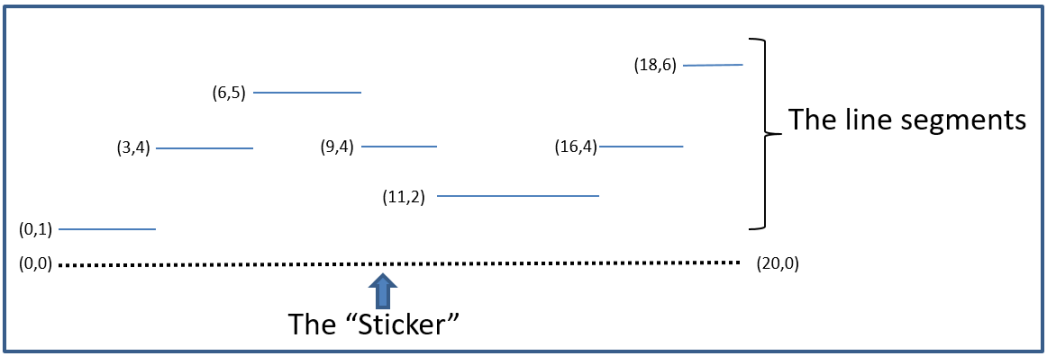
Now starting from the y = 0, a horizontal line which I will call the "sticker" spanning x = 0 to M-1 will start moving upwards.

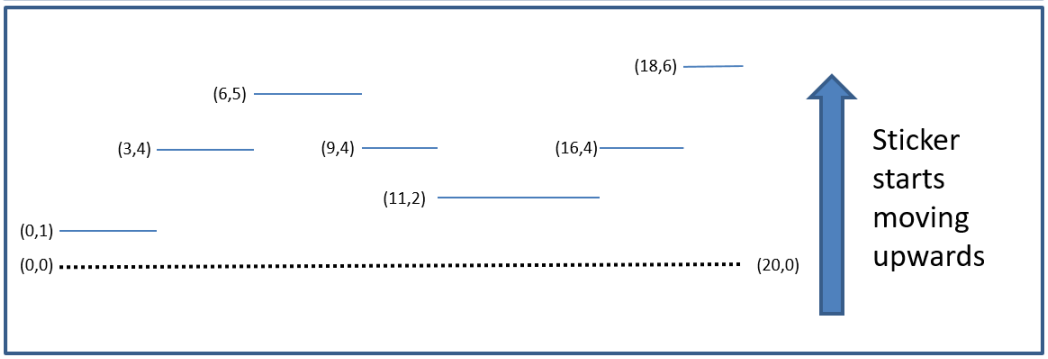
For each line segment encountered, the sticker will "break" with the portion spanning the line segments encountered being "stuck" to the line segment and the other parts (sticker segments) will continue moving upwards. So at some point you can have multiple sticker segments moving upwards. Even if the sticker segment matches the line segment exactly, it is also considered a "break". Now, regardless of how many line segments are encountered by a particular sticker segment at a particular y-coordinate, if there is ≥ 1 line segment that causes a "break" it is considered as 1 "break" not multiple breaks.

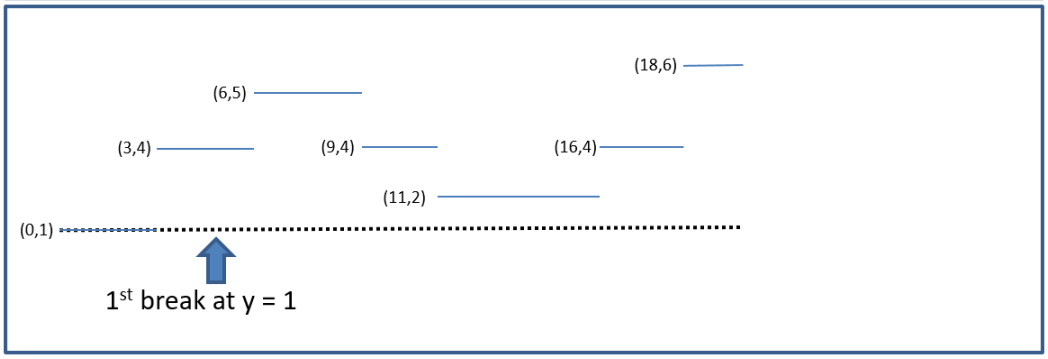
This breaking and moving upwards of the sticker/sticker segments will continue until all line segments have some sticker segment being stuck to them at which time the round will end.

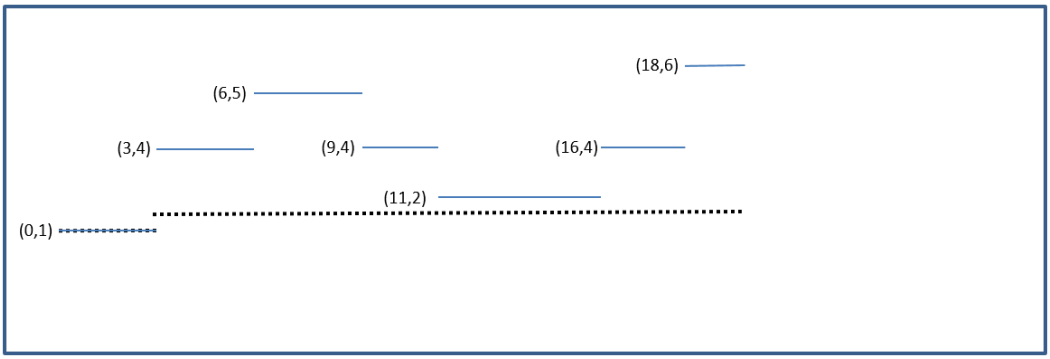
The point of the game is for the player to guess how many "breaks" will have happened when the round ends. The closer to the actual number, the more points.

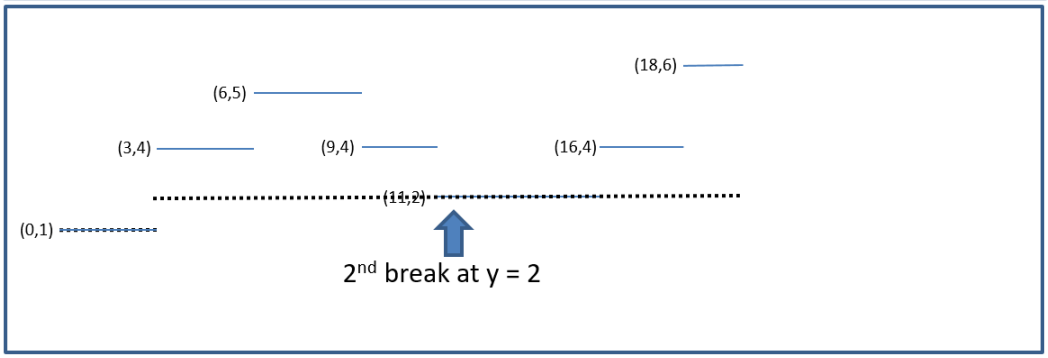
An example round where **M** = 20, **N** = 7 and
**H** = {{(0,1,3),(3,4,3),(6,5,3),(9,4,2),(11,2,5),(16,4,2),(18,6,2)} is shown below (look from top picture to bottom picture).

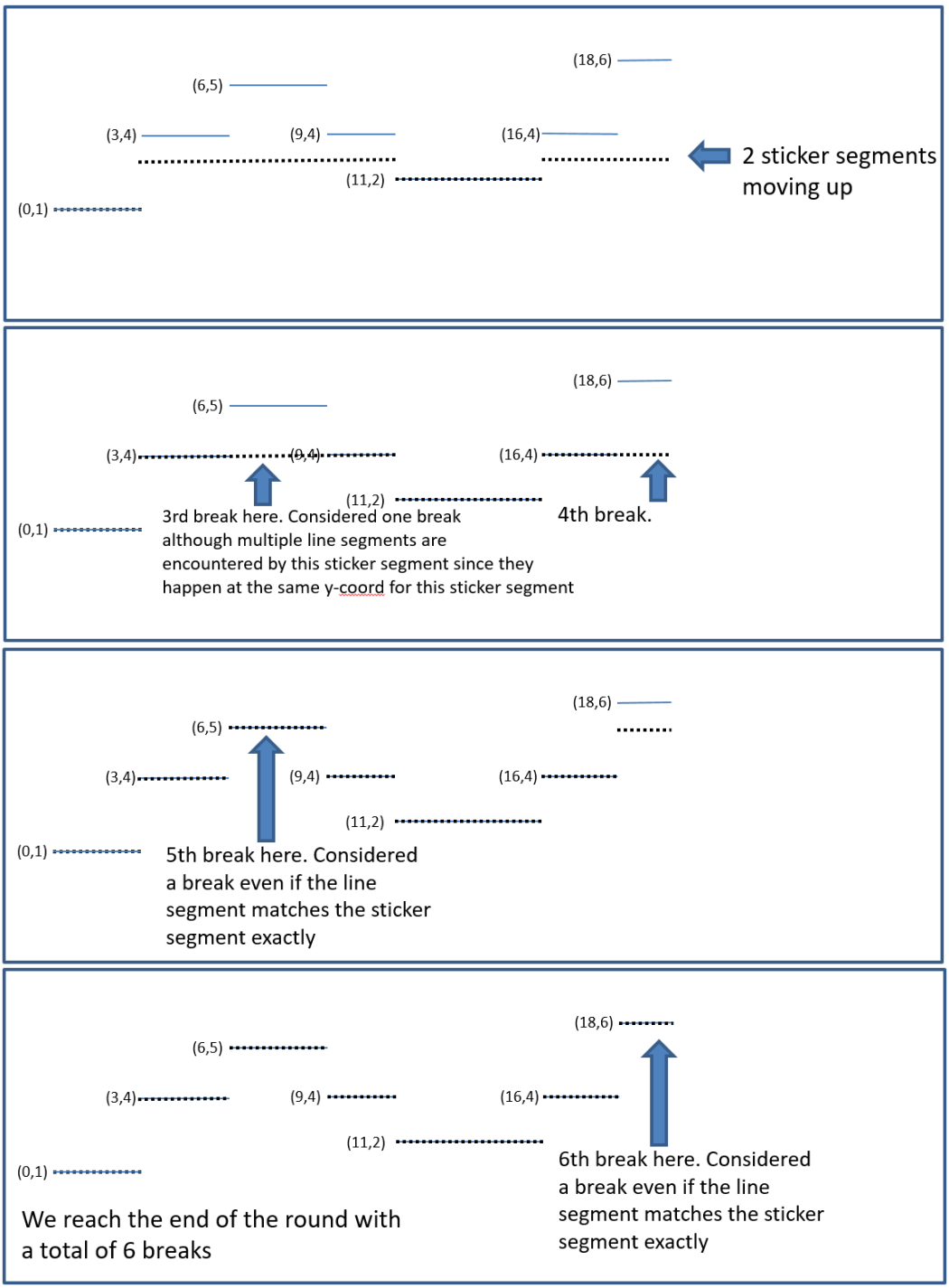

(18,6)
(6,5)
(3,4)
(9,4)
(16,4)
(11,2)
(0,1)
(0,0)
(20,0)
The line segments
The "Sticker"


(18,6)
(6,5)
(3,4)
(9,4)
(16,4)
(11,2)
(0,1)
(0,0)
(20,0)
Sticker starts moving upwards


(18,6)
(6,5)
(3,4)
(9,4)
(16,4)
(11,2)
(0,1)
1st break at y = 1


(18,6)
(6,5)
(3,4)
(9,4)
(16,4)
(11,2)
(0,1)


(18,6)
(6,5)
(3,4)
(9,4)
(16,4)
(11,2)
(0,1)
2nd break at y = 2




You are also a dedicated player of the game but instead of guessing you want to write a program to generate the actual answer (the number of breaks).

So given the value of **M,N** and the array **H** for a round of the game, give an algorithm that will output the total number of breaks when the round ends in worst case **O(N)** time. **[10 marks]**

Observation: If we scan the line segments from left to right (based on x-coord of left end)

i. If there is a series **S** of line segments going upwards (increasing y-ccord), then each one can potentially will trigger a break, now if we encounter a line segment **s** that goes downwards (decreasing y-coord) each line segment in **S** that is higher than **s** will definitely trigger a break. If there is a line segment in **S** at exactly the same height as **s**, it will only result in 1 break (since the sticker segment between that line and **s** will encounter both at the same time).

ii. If there is a series **S'** of line segments going downwards, then definitely each line will trigger a break (sticker will encounter the rightmost and lowest line segment in **S'** and break then encounter the $2^{nd}$ rightmost and $2^{nd}$ lowest line segment in **S'** and break … etc).

iii. If there is a series S'' of line segments at the same height, they will only trigger 1 break as a whole.

Solution Idea: We can use a stack to just keep track of line segments with increasing y-coord as we scan left to right. If we encounter a line with smaller y-coord as the top of stack keep popping and incrementing break count until we hit a line that has a smaller y-coord (push the current line onto stack) or has the same y-coord (ignore the current line since the line on the top of the stack will already account for the break). If no more line segments, pop entire stack and increment break count each time.

```
1.) Let S be a Stack
2.) Let count = 0
3.) S.push(H[0].y)
4.) For i = 1 to N-1
        if (H[i].y > S.peek())
           S.push(H[i].y)
        else if (H[i].y < S.peek())
           while (!S.empty() && H[i].y < S.peek())
              S.pop()
              count++
            if (S.empty() || S.peek() < H[i].y)
              S.push(H[i].y)
        else
           continue
5.) While (!S.empty())
        S.pop()
        count++
6.) Return count
```

Time complexity is at most O(N) since we perform a linear scan of H (O(N) time) and the stack S will at most push and pop each triplet in H at most once (again O(N) time).