CS2040S 2020/2021 Sem 1 Midterm

MCQ: 30 Marks, 10 Questions (3 marks each)

1. What is the time complexity of the following pseudo-code:

```
1. initialize boolean array A of size N+1 (N>2) to true

2. for (int i=2; i <= N; i++)
     for (int j=2; j*i <= N; j++)
       A[j] = false;
```

a. O(N)
b. O(1)
c. O(N^2)
d. O(NlogN)

For first iteration of other loop the inner loop is executed N/2 times, for the 2nd iteration it is executed N/3 times, and so on and so forth, thus we have the following series of summations for the total iterations executed by the code.

N/2+N/3+N/4+...+N/N = N*(1/2+1/3+1/4+...+1/N)
=O(N*(sum from 1 to N of harmonic series))
=O(NlogN)

2. Given two **nonempty** data structures X and Y, both containing elements in ascending order (the first element to be removed from the DS is the smallest element in the DS), we want to merge the contents of Y into X, such that X contains the elements of both X and Y, in ascending order. **Elements are distinct across both X and Y (so a value will appear at most once in X and Y combined). You should do this in O(m+n) time, where m and n are the number of elements in X and Y respectively.** You are not allowed to use any other DSes, apart from the provided X and Y, and are not allowed to use recursion. This task is possible if:

i. X is a queue and Y is a queue

ii. X is a stack and Y is a stack

iii. X is a queue and Y is a stack

iv. X is a stack and Y is a queue

ii) use standard merge sort, but instead of adding to a new DS, add elements to the back of X(either from X or from Y). Eventually, Y will be empty. Once done, poll(), store the value (temp), and offer() back to X. Peek() X, and see if the result is < temp. If it is, then X is now in ascending order. Otherwise, keep polling, storing the value, and offering

If X[0] < Y[0] it is the smallest item to the back of X, next compare X[1] with Y[0] -> standard merge sort

the time complexity is O(m+n) cause we will iterate through both X and Y once

a. i and ii
b. i, iii and iv
c. ii only
d. ii and iv

3. Given an unsorted array of N distinct integers with values from the range [1..N], we can find the k^{th} smallest element in:

a. O(1)
b. O(logN)
c. O(N)
d. O(NlogN)

just return k

4. We want to implement a new ADT, which supports the 4 operations from the Queue ADT (**offer()**, **poll()**, **peek()**, **isEmpty()**), as well as a new operation, **cut(x)**, which removes the first x elements of the queue and keeps the other elements. The **cut(x)** method should run in O(1) time, while the original Queue ADT methods should still have a worst-case time complexity of O(1). You may assume that if you use an array, the number of valid elements in the ADT will always be less than the array size, at any point in time. The most appropriate data structure to do so is:

a. Circular Linked List (tailed linked list with tail pointing to head)
b. Doubly Linked List with tail reference
c. Array
d. Circular Array

just set the list to add at pos x and as the num of additions get larger, the new values will override the old

5. Given a sorted array (containing non-distinct integers), we want to find out how many times a certain integer appears in the array. We can do so in:

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

In the case of finding out how many times a certain integer appears, we need to find the indices of the first occurrence (leftmost) and the last occurrence (rightmost) of this integer. Suppose this integer is k. We start by performing binary search as usual, until the center of the partition to be searched contains k (or if k is not found, then k is not present in the array, and we return 0

6. Given the following hash table using separate chaining:

0		->	14	->	7		
1	null						
2	null						
3		->	3				
4		->	11	->	4	->	18
5		->	5				
6	null						

The load factor of this table (rounded to 2 decimal places) is:

a. 0.57
b. 1.00
c. 1.43
d. 1.75

7 keys / 7 spaces

7. A bloom filter is created with the following hash functions:

h_1 = (key % 37) % 11

h_2 = (key % 31) % 11

h_3 = (key % 23) % 11

Below is the bloom filter after inserting keys 142 and 97.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	1	0	0	1	1	0	1	0	1	0

We then attempt to check if the following keys are present in the bloom filter. Which of these keys would give a negative result (i.e return false)?

i. 287

ii. 142

iii. 384

iv. 545

a. i and iii
b. ii only
c. i only
d. i,iii and iv

8. Given the following method in pseudo-code:

```
// q  is a Queue that contains integers
void foo(Queue q) {
  i = 0;
  while(!q.isEmpty()) {
    if (q.poll() == q.poll()) {
      q.offer(i++);
    }
  }
}
```

Assuming the queue contains [5, 6, 6, 7] initially, which of the following options is correct?

a. q is empty after the while loop ends   careless!
b. q = [0] after while loop ends
c. method will cause a runtime error if implemented
d. method goes into Infinite loop

9. Given an array split into half (you may assume the array size is always a multiple of 2), where the first half contains elements in sorted order, and the second half contains elements in any order (may be unsorted), choose the most efficient algorithm in terms of worst case time complexity to sort this array.

a. Insertion sort
b. Selection sort
c. Merge sort
d. Quick sort

Option a: Doesn't need to do much for the first half (other than performing 1 comparison per element), but will need to perform shifting for the remaining half of the elements. In the worst case, they may need to be shifted all the way to the left, resulting in O(n2) time

Option b: Selection sort always runs in O(n^2) time
Option c: Merge sort always runs in O(n log n) time
Option d: Half the array is already sorted. May run in O(n log n) time, but will more likely run in O(n2) time

10. The following hash table was constructed with the hash function h(key) = key % 11. No deletions were involved in the construction of this hash table:

0	1	2	3	4	5	6	7	8	9	10
null	null	4	null	15	48	null	29	62	null	21

The collision resolution technique could be:

i. Modified linear probing with step size d = 3
ii. Quadratic probing
iii. Double hashing with second hash function g(key) = 7 - (key % 7)

a. i only
b. ii only
c. ii and iii
d. i, ii and iii

Original slots for numbers (without probing)
Keys 4, 15, 48: slot 4
Keys 29, 62: slot 7
Key 21: slot 10

Modified linear probing: Consider the slots which have more than 1 key directly mapped to it (without probing): slots 4 and slot 7. Suppose the number of slots to move is d.

When adding the second key onwards for the same slot, the slot at ((original slot + d) % 11) should not be null, as this means the slot is available, and therefore should be filled by the second key. Therefore, we can conclude the following numbers cannot be d:

From slot 4: 2, 5, 7, 8, 10
From slot 7: 2, 4, 5, 7, 10
This leaves the only possible values for d as 1, 3, 6, 9.

d can be 3, given the following key insertions: [legend: key (probe sequence)]
15 (4), 29 (7), 21 (10), 4 (4 -> 7 -> 10 -> 2), 48 (4 -> 7 -> 10 -> 2 -> 5), 62 (7 -> 10 -> 2 -> 5 -> 8) d cannot be 1, 6, or 9

Quadratic probing: keys are inserted in the following order: [legend: key (probe sequence)] 15 (4), 48 (4 -> 5), 29 (7), 62 (7 -> 8), 21 (10), 4 (4 -> 5 -> 8 -> 2)

Double hashing: keys are inserted in the following order: [legend: key(result of second hash function: probe sequence)]
15 (6: 4), 48 (1: 4 -> 5), 29 (6: 7), 62 (1: 7 -> 8), 21(7: 10), 4 (3: 4 -> 7 -> 10 -> 2)

Analysis: 18 marks, 3 questions

Choose the correct answer (2 marks) and give your reasons for it (4 marks)

11. A bloom filter cannot be designed such that the false positive rate is 0%, even if we have infinite memory.

    a. True
    b. False

True. even though inf memory, values allocated based on hash function and multiple values can produce same hash key and occupying bit array.

Even if we have infinite memory, if we cannot bound the smallest_key_value and the largest_key_value then we cannot create a perfect hash function thus we cannot guarantee 0% false positive rate.

12. Other than an input of keys all in descending order, there is no other input that will cause optimized bubble sort to run in $O(N^2)$ worst case time.

    a. True
    b. False

only in this instance will the bubble sort require N num of passes each doing O(n) comparisons to bring the nth largest value to the back of the array

if false, give counterexample
Another input is one where all key are ascending order except the smallest key is in the position of the largest key e.g 2,3,4,5,6,1

13. Given the following code fragment, the worst case time complexity of calling f1(N) for some N that is a power of 3 is O(N).

```
void f1(int n) {
  if (n >= 1) {
    for (int i=0; i < 10; i++) {        O(10)
      f2(n);
    }
    f1(n/3);
    f1(n/3);
    f1(n/3);
  }
}


void f2(int m) {
  for (int j = 0; j < m; j++) {
    System.out.println("Hello!");   O(N)
  }
}
```

a. True
b. False

the f1 runs for log3N levels each doing N amount of work
hence the worst case time complexity is O(N logN)

yay correct ans! full marks

Structured Questions: 52 marks, 7 questions

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040S must be described, there must be no black boxes.

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

Questions 14 to 16 refer to the following problem description

New operations for Linked List and Stack

We want to include new operations to the Link List and Stack ADT. Assume that the Link List and Stack contains integer values.

The first new operation is a Stack operation and is as described:

Rotate(Stack s, int dir, int m) -

The Stack s will be rotated in the direction given by dir a total of m times. dir == 1 means rotate left, dir == 2 means rotate right.

Example of stack rotation:

Given s = 1,2,3,4,5   where 1 is the top of the stack and 5 the bottom of the stack

Rotate(s,1,1) will rotate s left 1 time and change s to be s = 2,3,4,5,1

Rotate(s,1,3) will rotate s left 3 time and change s to be s = 4,5,1,2,3

Rotate(s,2,1) will rotate s right 1 time and change s to be s = 5,1,2,3,4

Rotate(s,2,3) will rotate s right 3 time and change s to be s = 3,4,5,1,2

14. Give the algorithm for **Rotate** using only the Stack operations empty(), push(), pop(), peek() and size() in **O(n)** worst case time, where n is the size of the stack. You can use at most 2 additional stack to help you and no other DSes (no arrays, LLs, queues etc ...).

[8 marks]

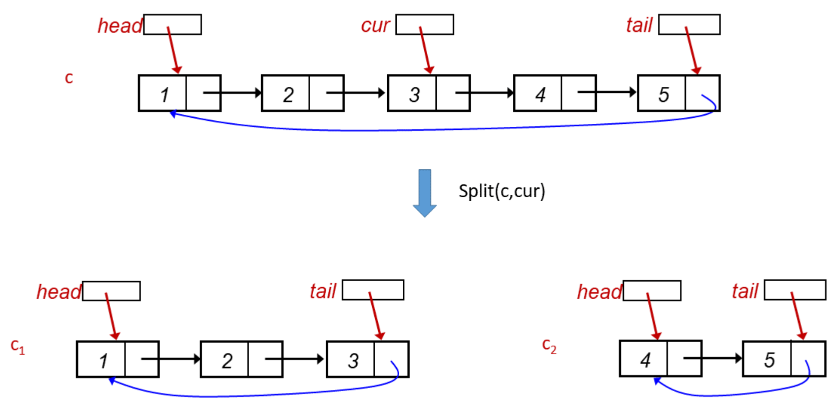
```
let s1 and s2 be two additional stacks

if dir == 1
  for(i = 0 to m)
    s1.push(s.pop) //transfer m elements from s to s1
  }

  pop each element from s1 to s2 //so that first popped element is at top
  transfer remaining elements in s to s1
  s2.push(s1.pop()) until s1 is empty //s2 will be in correct order
  s = s2;

if dir == 2
  for(i = 0 to s.size() - m)
    s1.push(s.pop) //transfer m elements from s to s1
  }

  pop remaining elements into s2
  push elements of s1 into s
  push elements of s2 into s //now s will be in correct order
```

The second new operation is a circular linked list (**A tailed singly linked list where tail.next points to head**) operation and is as described:

TailedLinkedList[] Split (TailedLinkedList c, ListNode cur) -

Given cur which points to a node of in a circular linked list c, split c into 2 circular Linked Lists.
The first one c_1: Containing head of c to cur and the head of c_1 being the head of c.
The second one c_2: Containing nodes from cur.next to tail of c,with cur.next being head of c_2.
If cur points to tail of c then return null, otherwise store c_1 and c_2 in an array of size 2 and return that array.

Example of a split where cur is not pointing to tail of c:



15. Give an algorithm to perform **Split** by only manipulating references in the original circular linked list c and having head and tail of c_1 and c_2 pointing to the correct nodes. You should not create any new nodes. You algorithm should run in **O(1)** worst case time. **[6 marks]**

```
if(cur == tail) {
  return null
} else {
  pt to cur node
  set c's tail next element to curr.next
  set curr.head as c2.head

  set curr next element to c.head
  set curr to c.tail

  Array a = [c1,c2]
  return a
}
```

```
TailedLinkedList[] Split(TailedLinkedList c, ListNode cur)
    1. Create TailedLinkedList array A of size 2
    2. if (cur == c.tail)
          return null
    3. if (cur != c.tail)
          A[0].head = c.head
A[0].tail = cur
          A[1].head = cur.next
          A[1].tail = c.tail
          c.tail.next = cur.next
          cur.next = c.head
4. return A
```

The third new operation is a basic/singly linked list operation and is as described:

DeleteSecondLast (BasicLinkedList c, ListNode cur) -

Given cur which points to the second last node of c, delete that node.

16. Give an algorithm to perform **DeleteSecondLast** in **O(1)** worst case time. You may assume that c always has ≥ 2 nodes before the second last node is deleted.

[4 marks]

go to node cur - 1
prev = cur - 1
next = cur.next
prev.setNext = next
size -= 1

1. Copy integer value from cur.next to cur
2. cur.next = null

17. Given an array A of N unsorted and unique integers between 0 and 100,000,000, find the total number of pairs x+y such that x+y = z.

You cannot use any form of hashing in your algorithm and it must run in O(N) worst case time.

[7 marks]

1. Radix sort A ———> O(N) time
2. Leti=0,j=N-1andnumpair=0
3. while (i < j) ———> O(N) time since each index of A is accessed once by i or j

case 1: (A[i]+A[j] < z) increment i by 1
case 2: (A[i]+A[j] > z) decrement j by 1
case 3: (A[i]+A[j] == z) increment numpair by 1 increment i by 1 decrement j by 1

The above algorithm is possible since all the integers are unique, if they are not then there can be O(N2) pairs of x and y that add up to z, so a O(N) algo is not possible.

18. Given 2 unsorted arrays, **A** containing N integers with values ranging from 0 to N^N and **B** containing M integers with values ranging from 0 to M^M, give the best algorithm (in terms of average time complexity) you can think of that will output true if B is a subset of A and false if it is not. Both A and B might have duplicate values.

[7 marks]

1. create new arr C with size M and all values set to False
2. for(i=0 to N)
   for(j=0 to M)
    if(C[j] == False && A[i] == B[j]) //if not duplicate and B[j] is in A
     C[j] = True
     count++ //num of values of B that are in A

//if M!=N and B subset A, count = M
//if M==N, B subset A means B=A, count = N
if(count == M II count == N)
  return true
else
  return false

1. If A.size < B.size, output false
2. Create a hashtable H where key k = each unique integer d in A and value v = frequency of d in A
3. Go through A and for each integer k encountered ———> O(N) average time
   a. If k is not in H, insert <k,0> into H
   b. If k is in H, update v associated with k by v+1
4. Now go through B and for each integer k' encountered ———> O(M) average time
   a. If k is not in H, output false
   b. If k is in H and v associated with k is 0, output false
   c. If k is in H and v associated with k is > 0, update v by v-1
5. Output true

Total time = O(N+M)

Questions 19 to 20 refer to the following problem description

<u>Lots of points</u>

Array A contains N points where each point consist of 2 integers x and y representing the x and y coordinate of the point ($1 \leq x,y \leq 1{,}000{,}000{,}000$).

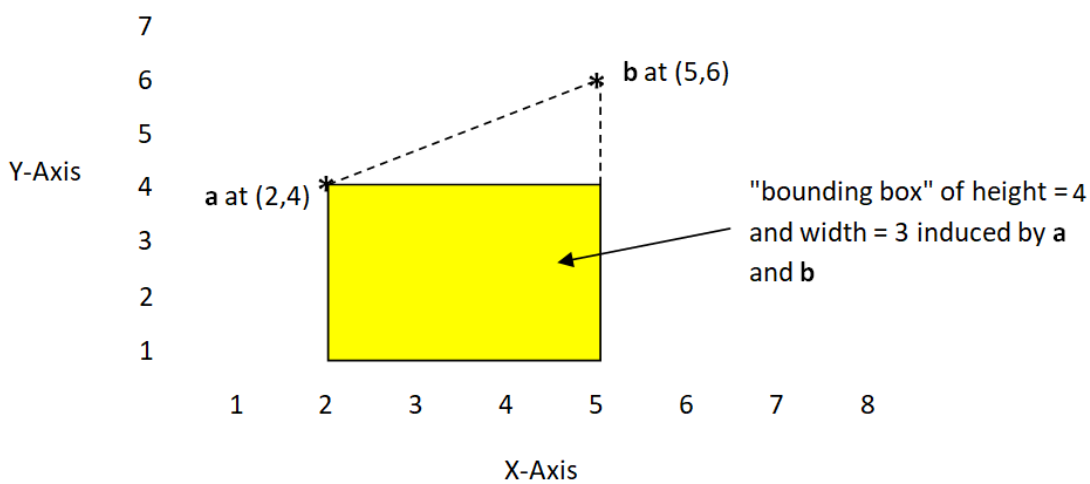
These N points in A is neither sorted by x-coordinate or y-coordinate.

Given any 2 points **a** and **b** there is a "bounding box" induced by **a** and **b**.

Height of bounding box = min(**a**.y,**b**.y)

Width of bounding box = |**a**.x-**b**.x|

This is illustrated diagrammatically below

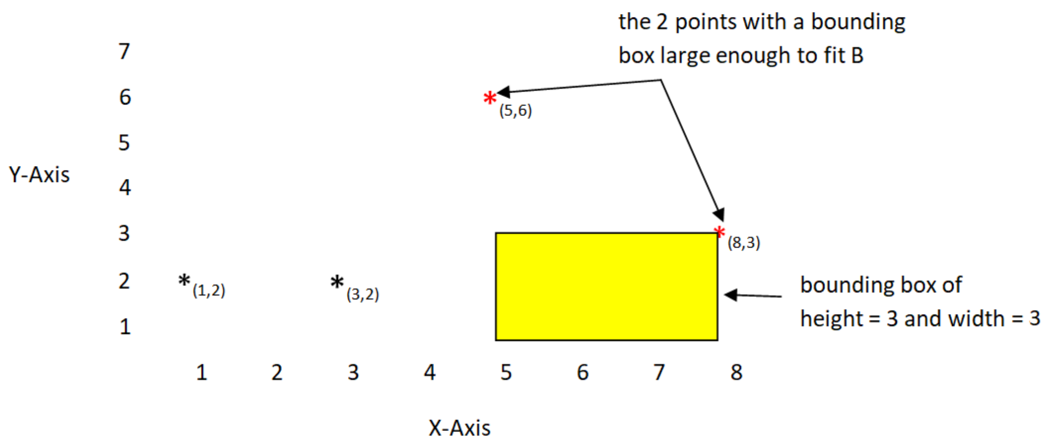


19. Now given a rectangular block B of height **h** and width **w**, return 1 pair of points in A such that B can fit within the bounding box induced by the pair of points (without rotating B). If there is no such pair, output "impossible".

    Your algorithm must run in O(N) worst case time.

[8 marks]

An example is shown below



Given a rectangular block B of height = 3 and width = 3, B can fit into the bounding box induced by the 2 points as indicated, and those are the only 2 points with a bounding box large enough to fit B

```
1. Let Xmin = 1,000,000,001 and Xmax = -1
2. Let p1 and p2 be the 2 points to be returned
3. Scan through A and for each point p where p.y >= h ———> O(N) time
   a. If (p.x < Xmin)
        Xmin = p.x
        p1 = p
   b. If (p.x > Xmax)
        Xmax = p.x
        p2 = p
4. If |p1.x-p2.x| < w
      return "impossible"
   else
      return p1 and p2

Time complexity = O(N)
```

20. Now with pre-processing that does not exceed $O(N)$ worst case time, design a $O(logN)$ worst case time algorithm that given the height **h** and width **w** of any block B, will return a pair of points in A with a bounding box large enough to fit B. If there is no such pair output "impossible".

    Describe your algorithm for preprocessing (including any DS used) and also the algorithm to answer each query.

[12 marks]