CS2040S 2020/2021 Sem 1 Midterm

MCQ: 30 Marks, 10 Questions

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)

Answer: **d.**

This algorithm sets all prime indices from 1 to N as true and all other indices as false.

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

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

Answer: **b.**

Option i: use standard merge sort, but instead of adding to a new DS, add elements to the back of X. 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.

Option ii: not possible, unless using recursion

Option iii: see option i

Option iv: see option i, but instead of adding to the back of X, add to the back of Y. After that, repeatedly poll() and push() to X. Now, X contains elements in descending order (largest element at the top). Pop() and offer() from X to Y. Now, the front of Y contains the largest element. Poll() and push() to X. Now the smallest element is at the top of X.

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)

Answer: **a.**

Just return k. (so many got this wrong!)

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

Answer: **d.**

Options a and b: no way to shift the front of the queue to the new position in O(1) time via cut(x)

Option c: may need to resize, as "removed" elements may take up space at the front of the array

Option d: can shift the front index by x slots. Elements at the front are effectively marked as deleted, and can be replaced with new elements as they are inserted, removing the possibility of resizing.

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)

Answer: **a.**

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).

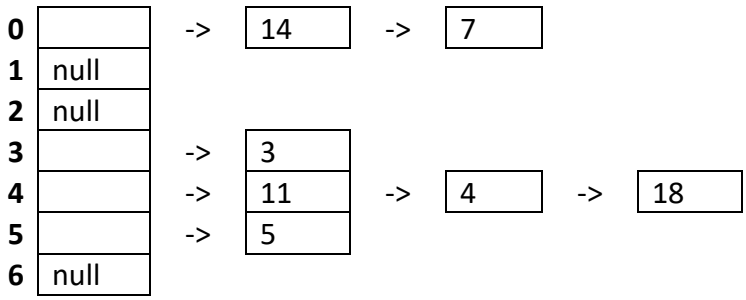
We now know that the first and last occurrence of k must be within this partition. In the worst case, this partition covers the entire array (ie. the center element of the array is k). We then have to perform binary search on both sides of the partition (including the center itself on both sides), though the code will be slightly different on both sides.

On the left side, if center == k, then we set center to be the rightmost element of the partition. On the right side, if center == k, then we set center to be the leftmost element of the partition. Owing to this definition, it may be possible that the partition never converges to a size of 1. To resolve this, we switch to sequential search at a small partition size (eg. when size is 3).

On the left side, the sequential search is done from left to right, and we stop upon encountering the first instance of k. On the right side, the sequential search is done from right to left, and we stop upon encountering the first instance of k. Once we have the two indices (leftmost and rightmost), we can return a result of (rightmost – leftmost + 1).

Total time taken is O(log n)

6. Given the following hash table using separate chaining:



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

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

Answer: **b.**

It is simply (number of keys in hash table)/(size of hash table)

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 <u>negative</u> 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

answer: **a.**

287: slots are 6, 8, 0. All slots have value 0

142: obviously exists in the bloom filter

384: slots are 3, 1, 5. Slots have values 0, 1, 1

545: slots are 5, 7, 5. All slots have value 1

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

answer: **a.**

The if condition is never met, so no new elements are inserted. The function terminates after polling out all 4 elements in the queue.

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

Answer: **c.**

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 $O(n^2)$ time

Option b: Always runs in $O(n^2)$ time

Option c: 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 $O(n^2)$ 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

answer: **d.**

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

Answer

True or False accepted depending on argument

False:

If we can bound the smallest_key_value and largest_key_value then simply create a bit array that is as large as largest_key_value-smallest_key_value since we have infinite memory. Then only use 1 hash function (a perfect hash)

h(key) = key+|smallest_key_value|

thus all keys are mapped to a unique bit in the bit array and there is no collision and thus a 0% false positive rate.

True:

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.

marking scheme:

true or false both get 2 marks.

For true:

if argued that knowledge of the keys in advance is needed for a perfect hash function <- 4 marks

For false:

if mentioned that can map into distinct slots such that no keys overlap <- 4 marks

However, if "infinite number of hash functions" is mentioned, only 2 marks (infinite number of hash functions would need infinite time to finish)

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

Answer:

False

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

marking scheme:

Showing a counterexample, or describing a correct counterexample <- +4 marks

Saying "any other input apart from already sorted input (ie. ascending order) takes O(n^2) time" gets 0 marks for explanation

Mentioning only part of the input (eg. "first element is sorted") gets 0 marks, since it is still possible to hit O(n) time under those conditions

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++) {
      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!");
```

```
    }
}
```

a. True
b. False

Answer:

False

Time complexity = $3^0*(10xN)+3^1*(10*N/3^1)+3^2*(10*N/3^2)+...+3^{lgN}*(10*N/3^{lgN})$

= $10N*(1+1+....1)$ ← lgN ones

= $10N*lgN$

= $O(NlgN)$

marking scheme:

Showing that it is O(n log n) <- 4 marks

Incorrectly determining the time complexity, but showing something that is not O(n) gets 2 marks

Structured Questions: 7 questions

This section is worth 52 marks. Answer all 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 relate to the problem description below

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** where the Stack only operations allowed are empty(), push(), pop() and peek() 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]**

Answer:

```
Rotate(s, dir, n)

   1. Create stacks s1 and s2
   2. Let size = 0
   3. Pop from s and push into s1 until s is empty, incrementing
      size for each item popped
```

```
4. Pop each item in s1 and push into s until s1 is empty.
5. If size == 0 return s // nothing to rotate
6. Let m = m % size // n might be larger than stack size
7. if (dir == 1)
     for 1 to m
       s1.push(s.pop())
     while !s.empty()
       s2.push(s.pop())
     while !s1.empty()
       s.push(s1.pop())
     while !s2.empty()
       s.push(s2.pop())
8. if (dir == 2)
     for 1 to size-m
       s1.push(s.pop())
     while (!s.empty())
       s2.push(s.pop())
     while (!s1.empty())
       s.push(s1.pop())
     while (!s2.empty())
       s.push(s2.pop())
return s
```

Marking Scheme:

O(N) solution – 8 marks

O(NM) solution – 4 marks

Deductions for O(N) and O(NM) solutions: 1 mark deducted for each of the following

1. Never handle n > stack size (mod required)
2. Never handle empty stack properly
3. Not finding stack size if required
4. Wrong destination stack

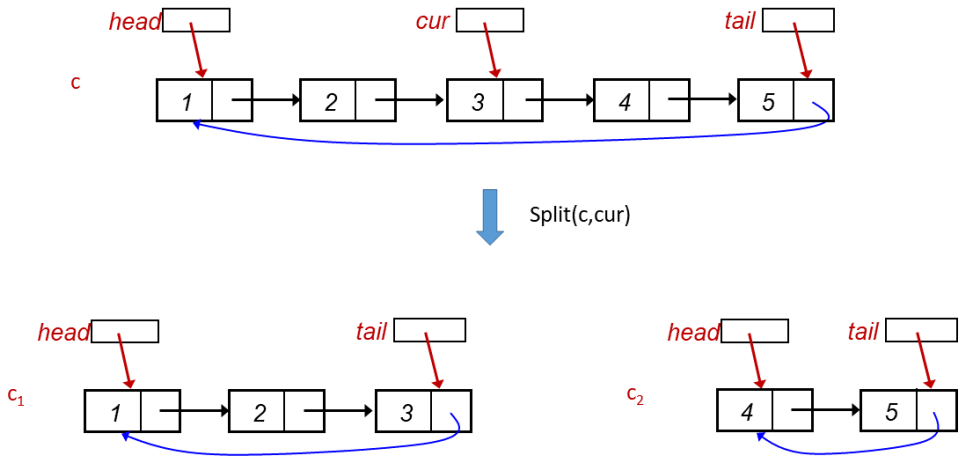
Some attempt to do rotation but completely wrong – 1 to 2 marks

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]**

Answer:

```
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
```

Marking Scheme:

O(1) solution – 6 marks

O(N) solution – 3 marks

Marks deducted for the following:

1. Head or tail or tail.next of resultant circular lists not set properly: -1 mark for each not set properly
2. Never handle the special case stated in the question (cur points to tail of c) properly: -1 mark

No attempt to detach the circular list into 2 circular list – 2 marks

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]**

Answer:

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

marking scheme:

O(1) solution – 4 marks

O(N) solution – 2 marks (1 mark if head is not updated when the 1st node is the 2nd last node and thus removed)

No marks given if use prev reference thinking this is a doubly linked list

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]**

    Answer:

    1. Radix sort A ← O(N) time
    2. Let i = 0, j = N-1 and numpair = 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 $O(N^2)$ pairs of x and y that add up to z, so a O(N) algo is not possible.

    Marking scheme:

    vague = not enough details given and/or some details given wrong

    O(N) solution -> 7 mark
    vague O(N) solution -> 5 marks

    O(N^2) solution -> 4
    vague O(N^2) solution -> 3

O(NlogN) solution -> 5
vague O(NlogN) solution -> 3 or 4 marks (depending on details given)

some attempted solution but totally Incorrect -> 1 or 2 mark

Incomplete solution -> 1 marks

Quite a few students create another boolean array B of size 100,000,000 initialized to false and then go through A and set B[A[i]] to true for each index i of A.
Then go through A again and check if B[A[i]-z] is true. If true then increment pair count.
This solution uses B as a direct addressing table and is essentially hashing, thus it is not acceptable as a solution.

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]**

Answer:
Can't do radix sort efficiently due to the large integer values….

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)

marking scheme:

O(N+M) solution -> 7 marks
O(MlogN) or O(max(MlogM,NlogN)) solution -> 5 marks
O(NM) solution -> 4 marks

Slower solutions -> 3 marks

Hashing solution that does not take care of duplicate values (i.e does not keep frequency count of unique integers in A) -> 4 marks

Doing binary search to check if every value in B is also in A after sorting A and/or B O(NlogN+MlogM) -> 3 marks

Double for loop to check if each element in B is in A -> 2 marks
Checking if B is sorted prefix subarray of A -> 2 marks

other wrong solutions -> 1 to 2 marks

incomplete solution - 1 mark

Questions 19 and 20 relate the problem description below

Lots of points

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

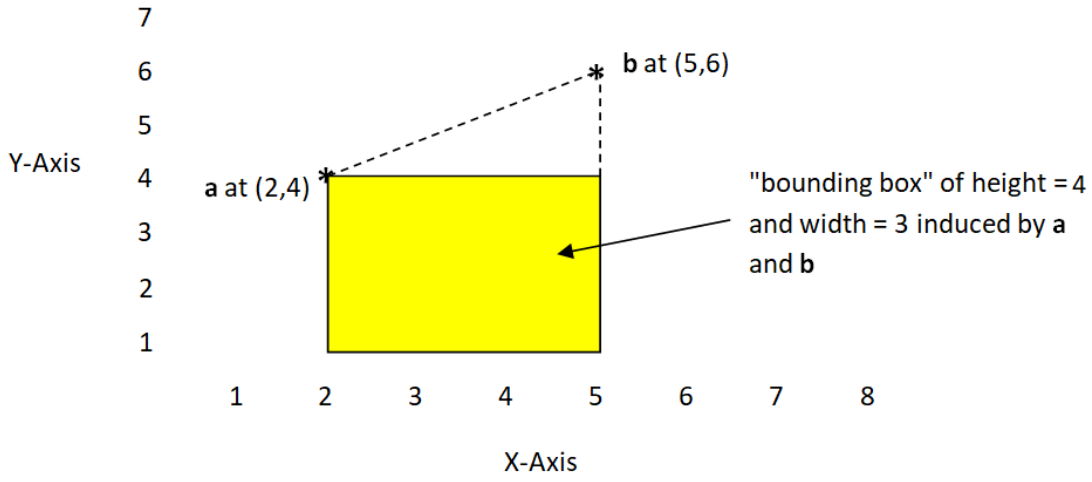
These N points in A is neither sorted by x-coordinate or y-coordinate.

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|

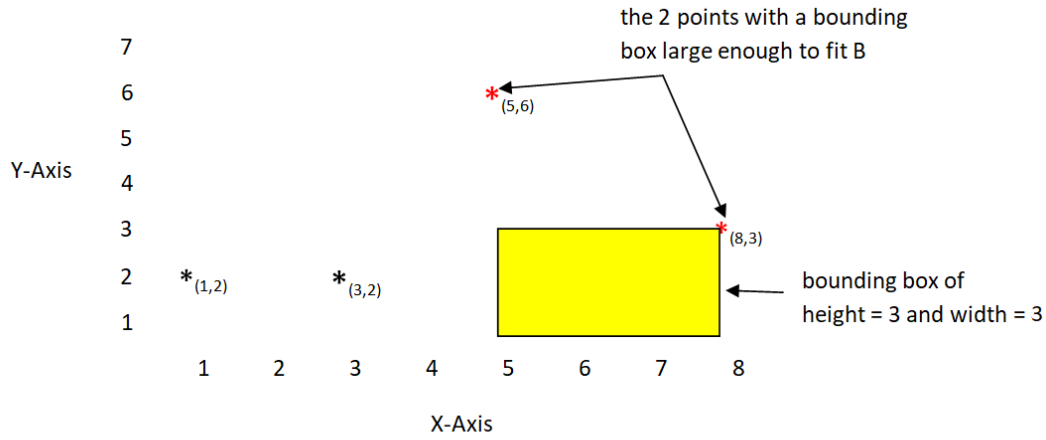
This is illustrated diagrammatically below



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

Answer:

1. Let X_{min} = 1,000,000,001 and X_{max} = -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 < X_{min})
        X_{min} = p.x
        p1 = p
    b. If (p.x > X_{max})
        X_{max} = p.x
        p2 = p
4. If |p1.x-p2.x| < w
    return “impossible”
   else
    return p1 and p2

Time complexity = O(N)

marking scheme:

Marks awarded based on whether you were answering the question for "bounding box that is exact match with B" or "bounding box can be larger than B"

In either case,

O(n) solution <- 8 marks

slower than O(n) solution <- marks capped at 6 marks

Correctness is more valued in this question, so solutions that are incorrect (eg. points disappearing due to use of hashmap/hashset) but fast gets around 4-5 marks

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]**

Answer:

Preprocessing:

1. create a data structure called pointPlus that store 3 attributes
   a) an integer yval
   b) point p1 such that p1.y >= yval and p1.x is smallest among all points with y-coordinate >= yval
   c) point p2 such that p2.y >= yval and p2.x is largest among all points with y-coordinate >= yval

// A pointPlus represents the 2 points (p1 and p2) forming the largest possible
// bounding box among all points with y-coordinate >= yval

2. Let C be an array of pointPlus

3. Sort A by y-coordinate of the points using Radix sort. <-- O(N) time

4. Let curY = -1, let last be index of the current last entry in C. Now scan A from **right to left** and let p be the current point. <-- O(N) time

   a) if (curY != p.y)
      curY = p.y
      create a pointPlus c
      let c.yval = p.y
      let c.p1 = c.p2 = p
      if (C is not empty) // possibly "inherit" p1 and p2 from last entry in C
         if c.p1.x > C[last].p1.x
            c.p1 = C[last].p1
         if c.p2.x < C[last].p2.x
            c.p2 = C[last].p2
      insert c into the back of C
   b) else if (curY == p.y)
      if (C[last].p1.x > p.x)
         C[last].p1 = p
      if (C[last].p2.x < p.x)
         C[last].p2 = p

5. Reverse array C, so it is in ascending order of unique y-coord values <-- O(N) time

Total time for preprocessing is O(N) time.

Answering query given block B of height h and width w:

1. perform a binary search on C using h <-- O(logN)
2. if h is found at some index i, and C[i].p2.x-C[i].p1.x >= w,
   return C[i].p1 and C[i].p2 otherwise output "impossible"
   // if largest bounding box among all points with y-coordinate >= h is not big
   // enough, then there is no bounding box large enough to contain B
   // if largest bounding box among all points with y-coordinate >= h is big enough,
   // simply return the 2 points that induce that bounding box
3. if h is not found and i is the last index the binary search stopped at, perform step 2 on index i (if C[i].yval > h) or index i+1 (if C[i].yval < h and i+1 is not out of bounds). If neither i nor i+1 is feasible output "impossible", otherwise output the 2 points as in step 2.

Total time taken to answer each query is O(logN) time

marking scheme:

Solutions that are a copy-paste from Q19 (with little modification) get 1 mark, since you've already received the marks for the solution back in Q19
- most of Q19 is inapplicable to preprocessing Q20 anyway (B isn't given at the start)

Solutions meeting the query time complexity, but not the preprocessing are capped at 8 marks

Solutions not meeting the query time complexity capped at 4 marks