CS2040S 2022/2023 Semester 1 Midterm

MCQ

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

Do all questions in this section.

1. We have 5 algorithms whose running time are described as the following functions:

$100n^2,\ (\sqrt{2})^{\log n},\ n^{\frac{1}{\log n}},\ (logn)!,\ 3^n$

What would be the correct way of ranking the above algorithms in terms of time complexity from fastest to slowest?

a) $100n^2 \prec (\sqrt{2})^{\log n} \prec 3^n \prec n^{\frac{1}{\log n}} \prec (logn)!$

b) $n^{\frac{1}{\log n}} \prec (\sqrt{2})^{\log n} \prec 100n^2 \prec 3^n \prec (logn)!$

c) $100n^2 \prec n^{\frac{1}{\log n}} \prec (\sqrt{2})^{\log n} \prec (logn)! \prec 3^n$

d) $n^{\frac{1}{\log n}} \prec (\sqrt{2})^{\log n} \prec 100n^2 \prec (logn)! \prec 3^n$

Ans: d)

For the first algorithm, $100n^2$ = O(n^2).
For the second algorithm,
$\log((\sqrt{2})^{\log n}) = \frac{1}{2}\log n \Rightarrow (\sqrt{2})^{\log n} = 2^{\log((\sqrt{2})^{\log n})} = (2^{logn})^{\frac{1}{2}}$ = O($\sqrt{n}$)
For the third algorithm,
$\log(n^{\frac{1}{\log n}}) = 1 \Rightarrow n^{\frac{1}{\log n}} = 2$ = O(1)
Now, to compare the three functions O(n^2), O($(logn)!$) and O(3^n),
we can use m = log n to convert the functions to O(4^m), O(m!) and O(3^{2^m}), respectively.
We have O(4^m)$\prec$ O(m!) $\prec$ O(3^{2^m}), and therefore, $100n^2 \prec (logn)! \prec 3^n$
So finally we have $n^{\frac{1}{\log n}} \prec (\sqrt{2})^{\log n} \prec 100n^2 \prec (logn)! \prec 3^n$

2. What is the time complexity for the following program?

```
void foo(int n){
  if (n < 1)
    return;
  for (int i = 0; i < n * n; i++)
    System.out.println("*");

  foo(n/2);
  foo(n/2);
}
```

a) $O(n^2)$
b) $O(n^2 \log n)$
c) $O(n^3)$
d) $O(n^3 \log n)$

Ans: a)
. The recurrence relation given by this program is
$T(n) = 2T(n/2)+n^2 = 4T(n/4)+ 2n^2/4 + n^2 = \cdots = nT(1) + (1 + 1/2 + 1/4 + \cdots + 1/n)n^2 = O(n^2)$

3. Consider a doubly circular linked list that looks like the following:

- We have a head pointer pointing to the first node, but there is no tail pointer
- Each node contains the item as well as a prev and next pointer that points to the previous and next node, respectively,
- The prev pointer of the first node points to the last node, and the next pointer of the last node points back to the first node.

If we want to implement the **insert(newnode, i)** method that inserts a node **newnode** at a certain index **i**, what are the minimum number of pointers that we need to modify in all cases? You may assume the index **i** is always a valid position.

a) 2
b) 3
c) 4
d) 5

Ans: b)

This happens when the initial list is empty. When inserting an item, we need to set the head pointer to the new node, while the next and the prev pointer of the node are set to point to itself.

4. Given the following array

2	4	9	15	3	5	7	13	1	20

Which of the following would be the best way to sort the array in ascending order?

a) Merge Sort
b) Quick Sort
c) Divide the array into 2 halves of length 5 each, use improved Bubble Sort on each half then merge them together using the merge method of Merge Sort.
d) Divide the array into 2 halves of length 5 each, use Insertion Sort on each half then merge them together using the merge method of Merge Sort.

ans: d)

We can observe that each sub-array of length 5 is almost sorted with some small elements having a large index. This is the case where improved bubble sort would fail miserably and insertion sort will still be fast.

This is basically TimSort that is used in Java's sorting function.

5. We have a hash table of size 11 using linear probing as collision resolution technique. Suppose we already have 4 distinct keys inserted. Which of the following would be the worst possible layout of the 4 keys in terms of the average number of probes needed for the 5th insertion?

(each X represents that a key is inserted in that slot)

a)

Slot	0	1	2	3	4	5	6	7	8	9	10
Key	X	X		X	X						

b)

Slot	0	1	2	3	4	5	6	7	8	9	10
Key	X	X								X	X

c)

Slot	0	1	2	3	4	5	6	7	8	9	10
Key	X		X		X		X				

d)

Slot	0	1	2	3	4	5	6	7	8	9	10
Key					X	X	X		X		

Ans: b)

Linear probing would perform the worst when we have a large cluster. Among the four choices, choice B has the largest cluster, with a size of 4

6. Suppose you have a hash table of size 8, and you want to design a hash function for integer keys. Among the following functions, which one would be the best?

a) h(key) = 8
b) h(key) = (12 * key + 4) % 8
c) h(key) = (7 * key + 9) % 8
d) h(key) = (key % 5) % 8

Ans: c)

A good hash function should distribute the keys as evenly across the hash table as possible. For h(key) = 8, it maps all keys to a single slot, which will lead to a lot of collisions. For h(key) = (12 * key + 4) % 8, it will only map to slots number 0 and 4. For h(key) = (key % 5) % 8, it will only map to those slots 0, 1, 2, 3, 4.

The remaining hash function h(key) = (7 * key + 9) % 8 would be the best. Since 7 and 8 are coprime, it is equally possible for a key to map to all 8 slots in the hash table.

7. Suppose you can only use the stack data structure. What is the minimum number of stacks needed to implement a queue with correct **enqueue** and **dequeue** operation?

a) 1
b) 2
c) 3
d) Impossible with a fixed number of stacks

Ans: b)

We can use 2 stack S1 and S2 in the following way:
• When we enqueue, we simply push the item into S1,
• when we dequeue, there are two cases:
– If S2 != null, we can pop from S2,
– If S2 = null, pop everything from S1 and push them into S2.
The enqueue operation will cost O(1), while the dequeue operation may cost in the worst case O(n). With amortized analysis, the dequeue would still cost O(1) (although here we are only interested in the correctness of the queue operations and not the efficiency)

8. We have an array of floats already sorted in descending order. We now want to sort them in ascending order instead. Unfortunately, your computer is really ancient, and we don't have much space left in memory. Now suppose we are to use the default implementation of arrays in Java, what would be the worst case time complexity of the best algorithm you can think of?

a) O(n)
b) O(nlogn)
c) $O(n^2)$
d) Impossible. Time to get a new computer

Ans: a)

No need to use sort algorithms, since we only need to reverse the array. We need an in-place algorithm, so we cannot create additional arrays for reversing. We can do so by swapping the first item with the last item, the second item with the second last item, and so on and so forth. In the end, each element will be swapped once, and the time complexity would be O(n)

9. If you are sorting an array of n distinct integers that are in a certain range, you can actually use a general hash table to perform sorting!

The idea is as follows:

- For each integer value i in the array, insert it into the hash table.
- Then for each integer j from 0 to the largest integer value in the array, print j if it is found in the hash table.

Now suppose we are using separate chaining as the collision resolution technique, and the largest integer value in the input array is at most O(n), which of the following is correct?

a) The algorithm is in-place
b) If the load factor of the hash table is O(n), the average case running time is O(n)
c) The worst case running time is $O(n^2)$
d) The worst case running time is O(n)

Ans: c)

The algorithm is not in-place as we need to use an additional O(n) space for the hash table.
If the load factor is O(n), then it means that the linked list in each slot of the hash table has size O(n). So each search of an integer will cost on average O(n) time, making the total time $O(n^2)$.
The worst-case running time can happen when every integer is inserted into the same slot, so it takes O(n) time to search for each integer, making the total time $O(n^2)$

10. Suppose you want to implement a special queue that has a **split(position)** operation that allows us to split the queue into 2 separate queues, where **position** is the integer index of the front of the 2nd queue if you are using an array based implementation, or is a pointer/reference to the head of the second queue if you are using a linked list implementation.

Which of the following data structures can you use to make this operation cost O(1) time in the worst case, while still allowing O(1) time **enqueue** and **dequeue** operations for both queues?

i. Array.
ii. Tailed linked list.
iii. Circular linked list (with a tailed reference).
iv. Doubly linked list.

a) All of them
b) ii, iii and iv
c) ii and iii
d) iv only

Ans: d)

We cannot use an array because although we can set the item at "position" to be the head of new array and make "split" O(1), further enqueue operation on the first queue will cost O(n) time for resizing the array.

We cannot use a tailed or circular linked list since we need to make the next pointer of previous node of the given "node" to be null, and the previous node cannot be accessed in O(1) time for both linked lists.

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. For the given algorithm below, the time complexity of calling mergeSort2(twoArr, N, 0, N-1) where twoArr is a 2D integer array of size N*N (N >= 1) is worst case $O(N^2 \log N)$.

```
void mergeSort2(int [][] twoArr, int N, int low, int high) {
  if (low < high) {
    int mid = (low+high)/2;
    mergeSort2(twoArr,N,low,mid);
    mergeSort2(twoArr,N,mid+1,high);
    merge2(twoArr,N,low,mid,high);
  }
  else if (low == high)
    insertionSort(twoArr[low]); // this is insertionSort as given in the
                                // lecture notes
}

void merge2(int [][] twoArr, int N, int low, int mid, int high) {
  int leftSize = (mid-low+1)*N;
  int rightSize = (high-mid)*N;
  int[] temp = new int[(high-low+1)*N];
  int[] tempLeft = new int[leftSize];
  int[] tempRight = new int[rightSize];
  int left = 0, right = 0, it = 0;

  int index = 0;
  for (int i = low; i <= mid; i++)
    for (int j = 0; j < N; j++)
      tempLeft[index++] = twoArr[i][j];

  index = 0;
  for (int i = mid+1; i <= high; i++)
    for (int j = 0; j < N; j++)
      tempRight[index++] = twoArr[i][j];

  while (left < leftSize && right < rightSize) {
    if (tempLeft[left] <= tempRight[right])
      temp[it++] = tempLeft[left++];
    else
      temp[it++] = tempRight[right++];
  }

  while (left < leftSize) temp[it++] = tempLeft[left++];
  while (right < rightSize) temp[it++] = tempRight[right++];

  index = 0;
  for (int row = low; row <= high; row++)
    for (int col = 0; col < N; col++)
      twoArr[row][col] = temp[index++];
}
```

Ans: False.

This is a version of sorting a 2D array by using merge sort in the divide phase to split the rows recursively into half and once it reaches the base case of 1 row, insertion sort is used to sort the row. Thus the recursive tree will have N base cases of 1 row each and insertion on each of them will take worst case O(N^2) so the bases cases will take a total of O(N^3) time to solve even without considering the conquer phase that uses the merge method of merge sort to merge sorted rows.

Grading Scheme:

Marks for reasoning

2m: Mentioning that the base case needs insertion sort on all of the N rows with N elements, and the total running time is O(N^3).

1m: Give the correct recurrence relation T(N) = 2T(N/2) + N^2, but forgetting the base case T(1) = O(N^2), resulting in running time O(N^2).

1m: Mentioning a time complexity greater than O(N^3), from the fact that insertion sort runs in O(N^2) time but counting the number of insertion sort runs wrongly.

0m: Analyze the algorithm as if it is non-recursive; only analyze the time for merge2; Giving a wrong time complexity based on incorrect analysis of merge2 and/or merge sort, or with little or no reasoning.

12. The effect of swapping the head and tail of the linked list (with more than 2 nodes) and then deleting the tail cannot be done in O(1) time for any linked list variant (among those given in the lecture notes) except the doubly linked list.

    Ans: True.

    To delete the tail, the node before the tail has to be made the new tail and that node cannot be accessed in O(1) time for any of the linked list variant except for the doubly linked list (prevnode = tail.prev). Even if the equivalent effect of deleting the head and then making the tail the new head (which can be done in O(1) time for circular linked list), the node before the tail must still be made the new tail and we again encounter the same problem.

    Grading Scheme:

    2m: Mentioning the need to access the second last element in the linked list, which is impossible to do in O(1) without a `prev` pointer.

    0m: Claim that swapping head and tail pointer must be done with O(N) time.

    Comment: A lot of students chose T/F wrongly (doesn't match their reasoning).

13. Given any N positive integer keys upfront, there is no hash function and hash table so that the hash table operations of insertion, deletion and retrieval on the given integer keys have worst case time complexity **better than** O(N).

    Memory is not a concern here.

    Ans: False.

    Since the keys are given up front we just use a DAT of size equal to the largest integer key as the hash table (so the hash function is the identity function) and since there is no collision at all, then retrieve, insert and delete can be done in worst case O(1) time.

    Grading Scheme:

    2m: Mentioning DAT as a hash function.

    2m: Using a very large hash table (with size > largest integer key), and one-to-one hash function.

    1m: Claim that we can find a perfect hash function/hash table without explicitly saying how to construct them.

    0m: Claim that some special collision resolution technique is enough to make it O(1); simply saying that hash tables have average case O(1); other answers.

    Some student misunderstand the time complexity to be referring to the total time to insert/retrieval /delete all the N keys. This is still fine. If your explanation is that even with the best hash function/hashtable that is the identity function and a DAT it will be O(1) for each operation per key and thus still O(N) for all keys in the worst case, you will still get the marks since you are basically answering the question. Other answer like e.g assuming some bad hash function/hashtables which will cause a lot of collisions and thus result in O(N) time per operation per key will not get the marks.

    Comment: A lot of students chose T/F wrongly (doesn't match their reasoning).

Application Questions

This section has 4 questions (last 2 questions has 2 parts) and is worth 58 marks.

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.

14. **[16 marks] Garden Gnomes**

Garden gnomes are pesky creatures which operates in packs and are identified by a unique non-zero integer ID on their foreheads. These gnomes if left alone will quickly destroy one's garden.

However, as long as all the leader of a pack is caught and removed from the garden then the rest of the gnomes in the pack will also follow suit and leave the garden.

In your garden there are 2 packs of gnomes, pack A and pack B each with their own leader. To identify the leader, you only need to observe how the gnomes travel.

For both pack A and B, the entire pack will move in a linear sequence, and the ordering of the gnomes are as follows:

Pack A
1. The gnomes are arranged in a seemingly random fashion based on their gnome ID
2. There is only 1 leader
3. The leader is the only gnome who has all gnomes with IDs less than it to its left and all gnomes with IDs more than it to its right.

For example, given the following linear sequence of gnome IDs: 3,7,1,9,12,11
9 is the leader.

Pack B
1. Except for the leader the other gnomes will arrange themselves in strictly increasing gnome ID
2. The leader is always the gnome who do not follow the above rule.
3. The leader will not be the first or the last gnome in the linear sequence.

For example, given the following linear sequence of gnome IDs: 1,3,5,2,8,9,10,11
2 is the leader of the pack.

Another example: 1,2,3,5,8,11,9,10
11 is the leader of the pack

Solve the following 2 subproblems (8 marks each):

a) Given array A of size N (N > 10) containing the IDs of the linear sequence of gnomes for pack A, give the most efficient algorithm you can think of to identify and print out the leader.

b) Given array B of size M (M > 10) containing the IDs of the linear sequence of gnomes for pack B, give the most efficient algorithm you can think of to identify and print out the leader.

Ans:
a) Key observation is that the leader is the only gnome where the largest gnome ID to its left is < its ID and the smallest gnome ID to its right is > its ID. Thus now create 2 arrays L and S where L will track for each index i, the largest gnome ID to the left of the gnome ID at A[i] and S will track for each index i, the smallest gnome ID to the right of the gnome ID at A[i].

```
1. Let L and S be to arrays of size N initialized to 0
2. for i from 1 to N-1
     L[i] = max(L[i-1],A[i-1])
3. for i from N-2 to 0
     if (i != N-2)
       S[i] = min(L[i+1],A[i+1])
     Else
       S[i] = A[i+1]
4. for i from 0 to N-1
      if (i == 0 && S[i] > A[i]) return A[i]
      else if (i == N-1 && L[i] < A[i]) return A[i]
      else if (L[i] < A[i] && S[i] > A[i]) return A[i]
```

Steps 2,3 and 4 are all O(N) thus the algorithm is O(N).

b) This is the reverse of pack A, the leader is the only one not in the correct position. Everyone else is in the correct relative position. Thus any gnome at index i which has either a smaller ID than the previous gnome of a larger ID than the next gnome is a potential leader. The algorithm to identify the leader is as follows which runs in O(M) time:

```
for i from 1 to M-2
  // case where gnome at index i has smaller ID then previous gnome
  if ((B[i] < B[i-1] && i == 1) || (B[i] < B[i-1] && B[i] < B[i-2]))
    return i
  // case where gnome at index i has larger ID then next gnome
  if ((B[i] > B[i+1] && i == M-1) || (B[i] > B[i+1] && B[i] > B[i+2]))
    return i
```

Grading Scheme:

part a.)

Correct solutions:
O(N) solution. 8 marks
$O(N^2)$ solution. 6 marks
less efficient solutions. 4 marks.

Wrong solutions:
1. sort and then return the first gnome whose position is unchanged. 3 marks
2. Linear scan of the array and just checking to left and right of a current index to determine if the value at the index is the leader. 2 marks
3. Linear scan of the array but only all values to left of the current index or to the right of the current index is checked to be < or > than the value at the current index, instead of checking both sides. 3 marks

Other wrong or vague (no details at all) solution: 0 to 1 mark

Deduction for mistakes made:
1. -1 mark for minor mistakes
2. -2 marks for big mistakes

Comment:
1. A lot of students mistake that by sorting the array, the 1st gnome whose position is unchanged must be the leader. This is not true, a gnome being in its correct sorted position does not necessarily mean it is a pivot (i.e satisfy the rule that all gnomes to its left is < it and all gnomes to its right is > it).

Counter example: 16,10,5,20,35,30,25
Here 10,20 and 30 are in the their correct sorted position but only 20 satisfy the rule that all gnomes to the left are smaller than it and all gnomes to the right are larger than it.

part b.)

Correct Solutions:
O(N) solution. 8 marks
$O(N^2)$ solution. 4 marks (e.g remove each gnome and check if resulting sequence is increasing).

wrong or vague (no details at all) solution: 0 to 1 mark.

Deduction for mistakes made:
1. -1 mark for minor mistake
2. -2 marks for big mistake (e.g only check if A[i] > A[i+1] to determine that leader is A[i])

Comments:
1. A common mistake is to assume that if A[i] > A[i+1] (i.e there is a dip) then simply return A[i+1]. This is not true because if A[i-1] < A[i+1] that means the leader was removed from its sorted position and placed in the region of IDs smaller than itself, thus A[i] is the leader, but if A[i-1] > A[i+1] that means the leader was removed from its sorted position and placed in the region of IDs bigger than itself, thus A[i+1] is the leader.

2. A similar mistake is to just check if A[i-1] < A[i] > A[i+1] or A[i-1] > A[i] < A[i+1] to determine A[i] is the leader. Again you need to check A[i-1] with A[i+1] for the 1st case and A[i-2] and A[i] for the 2nd case to be able to determine the leader.

15. **[14 marks] Saw XI**

N people (including yourself) have found themselves each trapped in one of N sequential cells (each cell is only occupied by 1 person) by a sadistic killer, where N is an odd number > 1. The cells are numbered from 0 to N-1 and each person is given a unique prisoner ID (an integer value ≥ 1).

Unfortunately you occupy cell k, where if the prisoners are ordered from lowest to highest prisoner IDs and placed into cells 0 to N-1 respectively, then k is the cell that you will occupy.

Due to this, the killer has picked you to play a deadly game. He wants you to put the people with prisoner ids < your prisoner ID into cells 0 to k-1 (in any order) and those with prisoner IDs > your prisoner id into cells k+1 to N-1 (again in any order). If you can successfully do this in accordance to the following 2 rules, then all the prisoners will be set free.

The 2 rules are as follows:

1. You can only perform pairwise swapping of prisoners to achieve the killer's goal. However, any prisoner can only be swapped **at most 1 time**.

2. You can only go through all prisoners at most a constant number of times, thus your algorithm to achieve the killer's goal can at most be **worst case time complexity O(N)**. Exceeding this will result in all the prisoners being killed.

3. Since you can only swap the prisoners from the current cells they are to any of the N cells given, that means for any algorithm you give, you can only use **at most O(1) extra space**.

Given a value k, an array S of size N where S[i] contains the prisoner id of the prisoner in cell i (you will be in cell k and S[k] will be your prisoner id), give an algorithm that will complete the killer's game successfully based on the given rules. The more rules are broken the more prisoners will be killed by the killer (meaning more marks deducted ...).

Ans:

The simplest way (which will not satisfy all the rules) is to use the partitioning algo of quicksort to solve the problem. First simply swap yourself with the prisoner at cell 0 (swap S[0] and S[k]) so you will become the pivot. Then do the partitioning as taught. However, that partitioning algo can result in prisoners being swapped multiple times (even if you modify the partitioning algo so you do not need to swap S[0] and S[k] at the start). An example is given below

E.g 10 20 5 1 13 17 2 <- where you are prisoner ID 10 and have been swapped to cell 0
using the partitioning algo as taught we have 20 and 5 which will be swapped to get 10 5 20 1 13 17 2
then we have 20 and 1 which will be swapped to get 10 5 1 20 13 17 2.
finally we will need to swap 20 and 2 to get 10 5 1 2 13 17 20 thus 20 is swapped multiple times.
Now to put 10 back into it proper position we need to swap 10 with 2 (so 2 is swapped multiple times too).

Another way of performing the partitioning that will satisfy the rules (i.e only perform swapping for each prisoner at most once) is to note that the pivot (you) are already in the correct position, so from 0 to k-1 there must be as many prisoners which are out of place as there are prisoners out of place from k+1 to N-1. So you can "pair" the out of place prisoners to swap them by going from 0 to K-1 and from N-1 to K+1 simultaneously. The algo is as follows:

1. Let left = 0, right = N-1
2. while (left != k and right != k) // once either left or right reaches k the partitioning is done
    if (S[left] > S[k] and S[k] > S[right]) // both out of place so swap them

swap S[left] and S[right]
left++
right--
else if (S[left] > S[k] and S[k] < S[right]) // decrement right to find the out of place pair for S[left]
right--
else if (S[left] < S[k] and S[k] > S[right]) // increment left to find the out of place pair for S[right]
left++
else // both in correct place
left++
right--

Since you will swap and decrement right or increment left no prisoner will ever be swapped more than once and you will only go through each prisoner once, so total time complexity is O(N).

Grading Scheme:

Correct Solutions:

1. O(N) algo that satisfies all the rules. 14 marks
2. O(NlogN) algo (radix sort considered here since prisoner ID is not upper bounded) that satisfies all the rules. 11 marks
3. $O(N^2)$ algo that satisfies all the rules. 8 marks

Deduction for not following the rules

1. Swap some prisoner multiple times. -4 marks
1. For each extra array of size O(N) used. -4 marks (capped at -8 marks)

Deduction for mistakes

1. -1 mark for minor mistake
2. -2 marks for big mistakes

Wrong algorithm:

1. totally wrong algorithm. 1 mark
2. Incomplete/wrong algorithm that has some semblance of trying to move the prisoners into their correct region (e.g simply swap when S[left] < S[right] (left start from 0 and right start from N-1) instead of comparing to S[k] to determine whether to swap or increment left/decrement right). 2 to 3 marks

Comments:

1. Many students use the partitioning algo as taught without realizing that a value can be shifted/swapped multiple times during the partitioning.

2. Quite a lot of students also perform a wrong partitioning by just comparing if S[left] > S[right], where left is some index left of k and right is some index right of k, to swap S[left] and S[right] instead of comparing them to S[k], since one of them can already be in the correct region and does not need to be swapped.

16. **[14 marks] Musical Chair 2.0**

N (N > 10) persons are playing a game where they are arranged in a circle starting from person 0 to person N-1 (person N-1 is beside person 0). Each person has a status S associated with them, where S can be 1,2 or 3.

There is a given integer K ($2 \leq K < N$), and the game will now proceed for R turns ($1 \leq R < N$). For the first turn, it will start from person 0 (the current person) and go in direction L where L can be clockwise or counter-clockwise and starts off as clockwise by default.

For each turn, look at the current person's status S. If

S = 1:
Set L to be clockwise and starting from the current person skip the next K-1 persons and remove the K^{th} person according to direction L. The person next to the removed person in direction L will be the current person for the next round.

For example. we have 0,1,2,3,4,5 in a circle where 5 is next to 0. If the current person is 3, K = 4 and status of person 3 is 1 then person 0 will be removed and the current person for the next turn is person 1.

S = 2:
Same as the case of S = 1 except that L is now set to counter-clockwise..

For example, we have 0,1,2,3,4,5 in a circle where 5 is next to 0. If the current person is 3, K = 3 and status of person 3 is 2 then person 1 will be removed and the current person for the next turn is person 0.

S = 3:
Starting from the current person skip the next K-1 persons and put back the last removed person (that has not been put back) before the K^{th} person according to direction L and the person just placed back will now be the current person for the next turn. <u>If all removed persons has been placed back into the circle or there is no person removed yet, then treat this as S = 1 if L is clockwise or S = 2 if L is counter-clockwise</u>.

For example, we have 0,1,2,3,4,5 in a circle where 5 is next to 0. If the current person is 3, K = 4 and status of person 3 is 1 then person 0 will be removed and the current person for the next round is person 1. In the next round, if status of person 1 is 3, then person 0 will be placed back in between person 3 and person 4. So we now have 1,2,3,0,4,5 and person 0 will now be the current person for the next turn.

Another example, if we have 0,1,2,3,4,5 in a circle where 5 is next to 0. If the current person is 2, K = 3, L is clockwise and status of person 2 is 3, but there is no removed persons, then simply remove 4 and person 5 will be the current person for the next turn.

At the end of the R turns, print out the persons still in the circle by starting from any one person and going one round in a clockwise fashion. E.g if we have 1,2,3,0,4,5 and we start from person 0 then print out 0,4,5,1,2,3. If we start from person 1 then print out 1,2,3,0,4,5.

Given an array A of size N where A[i] is the status of person i, the value K and R, using at most 2 ADTs out of any of the ADTs you have come across in CS2040S (lecture, lab, tutorials), where the 2 ADTS can be the same (for example 2 stacks), give an algorithm to solve the problem in average or worst case O(N+RK) time.

Ans:

The best ADT to solve this is a double ended queue AKA deque (to simulate the circle of people) and a stack to store the removed people.

1. Let Q be a deque that stores pairs <person, status> where person is the person number and status is the status of the person. Let S be a stack.

2. for i from 0 to N-1 // O(N) time
   enqueue <i,A[i]> to the back of Q

3. Let L = 0 // 0 == clockwise, 1 == counter-clockwise
   Let current = 0
   while (R > 0) // O(R) time
    R--
    <person,status> = peek at front of Q
    if status != 3
      L = status -1
    if (L == 0)
     dequeue from front of Q K-1 times. For each dequeue, re-enqueue to back of Q. // O(K) time
     if (status == 3)
       pop from S and enqueue to front of Q
     else
       dequeue one more time from front of Q and push into S
    else if (L == 1)
     dequeue from back of Q K-2 times. For each dequeue, re-enqueue to front of Q. // O(K) time
     if (status == 3)
       pop from S and enqueue to front of Q
     else
       dequeue one more time from back of Q and push into S
       dequeue one more time from back of Q and re-enqueue to front of Q.

4. dequeue from front of Q until it is empty and <person,status> dequeued print out person.

Total time complexity = time for step 2 + time for step 3 = O(N) + O(RK) = O(N+RK).

Grading Scheme:

- Correct Solutions:
  - **Initialization (1 mark)**: O(N) time.
  - **Simulation (13 marks)**: O(K) time for each step

  O(K) time per round in the game: using either a deque or a doubly circular linked list (14 marks)
  - O(N) time per round: using a doubly circular linked list, but use it incorrectly by starting from the head for every round. Could be better with some simple modifications (11 marks)
  - O(N) or O(K+N) time per round: using a (circular) array, which takes O(N) time for insertion and deletion (8 marks)
  - O(KN) time per round: use either array or linked list, but use lazy deletion (marking the corresponding position as NULL, for example), which will take O(KN) time in worst case to go to the correct index if there are O(N) removed players. (8 marks)
- Incorrect/Incomplete solutions: Deduct points from the marks for correct solutions of certain time complexity.
  - Incorrect indexing: The index to the player to be removed is incorrect. (minus 2-4 marks, based on how far it is to the correct solution)
  - The removed players are not stored (-3 marks) or stored in some incorrect way (e.g. queue or hash set) (-2 marks)
  - Incomplete implementation for each case (-2 marks/-2 marks/-4 marks for S==1/S==2/S==3 case, may be less depending on how complete their solutions are)
  - Other operations that significantly affect the correctness of algorithm (-2 marks)

Common mistakes:

1. A lot of students used linked lists, yet they use operator[] to access elements as if it is an array, which actually cost O(N) time.
2. Incorrect modifying of pointers in doubly linked list for removing and inserting of elements. (-1 mark)
3. Infinite while loops. (-1 mark)
4. Missing the case in S==3 where the stack is empty (-1 mark)
5. Other minor mistakes that doesn't significantly affect the correctness (-1 mark)

17.**[14 marks] Leaping Frogs**

N magical frogs (numbered from 0 to N-1) are made to race around a circular race track which is M meters in circumference (M > N).

The starting point of the race track is marked as 0 meters and there is a marking for every meter of the race track (i.e 0,1,2,3...,M-1) and after M-1 it will be 0 again. At the start each frog is placed at the i meters marking where i is the frog number. When the race starts, each frog will then have to leap a total of ≥ M meters around the track to successfully complete the race.

Each frog will make a jump every second starting from the 0th second (start of the race) and the i^{th} frog can jump forward S_i meters (an integer value ≥ 1) each time it jumps. At any time during the race if 2 or more frogs meet at the same place, those that have not leapt a total of ≥ M meters will be magically eliminated.

For example if M = 100 and frog 10 and frog 5 meet at the 20 meters mark during the race and frog 10 has leapt a total of 10 meters and frog 5 has leapt a total of 15 meters then both of them will be eliminated. However, if frog 10 has leapt a total of 110 meters and frog 5 a total of 15 meters then only frog 5 will be eliminated.

Given an array S of size N where S[i] is the distance frog i can move each time it jumps, give an algorithm that will determine the number of frogs that can successfully complete the race without being eliminated in worst case or average case $\leq O(M^2)$ time.

ans:

Idea: Simulate the race for every second (you can only go up to M seconds at most since the least distance jumped by a frog is per jump is 1 meter) and check the frogs that are in the same position to be eliminated for each second simulated.

1. Create a hashset h1 that contains frogs eliminated from or finished the race.
2. Create a hashset h2 that contains positions each frog will jump to at each step of the simulation.
3. Create a DAT h3 of size M where the index is the position and each index contains a count of the number of frogs at that index. h3 is initialized to 0. // O(M) time
4. Create an array p of size N that contains the unique positions generated per step of the simulation. p is initialized to -1. // O(N) time

```
5. Let survive = N
   for i from 1 to M // simulate race up to at most M seconds. O(M) time
    Let index = 0
    for j from 0 to N-1 // go through each frog j to get their position at step i. O(N) time
     if (j is not found in h1)  // frog j is still in the race.  Ave O(1) time
       position = (S[j]*i+j)%M
       h3[position] += 1
       if (position is not found in h2)
         h2.insert(position) // ave O(1) time
         p[index++] = position

    for j from 0 to N-1  // go through the frogs again. O(N) time
     if (j is not found in h1)
       position = (S[j]*i+j)%M
       if (h3[position] > 1) // multiple frogs at that position
         h2.insert(j) // ave O(1) time
         if (S[j]*i < M) // frog is eliminated, otherwise it has merely finished the race
           survive--
       else // check if frog has finished the race
         if (S[j]*i >= M)
           h2.insert(j) // ave O(1) time

    // clear h3 using p, then clear p and h2
    for j from 0 to index-1  // O(N)
     h3[p[j]] = 0
    clear p and clear h2 // O(N) time since p and h2 is of size at most O(N)

6. return survive
```

Time for the algorithm is dependent on step 5 which takes average O(MN) for the 1st and 2nd inner for loop and O(MN) for clearing h3,h2 and p. Thus it will be O(3*MN) = O(MN) on average.

$O(M^2)$ time is to simply clear the entire h3 in O(M) time after every step of the simulation (instead of only resetting those slots which are used in that particular step of the simulation). So in total it will be $O(M^2)$.

Grading Scheme:

- Correct Solutions:
  - O(MN) total time: using several hash tables to achieve fast searching and collision detection. (14 marks)
  - $O(N^2)$ total time: do pairwise checking on the frogs, see if they can collide before a frog reaches M meters. Very hard to do this properly, but I think it might be possible with some modulo checking. (14 marks)
  - O(M(N + log M)) total time: for each round, simulate for each frog then group frogs at same position together with radix sort. (11 marks)
  - O(MN log N) total time: similar as the case above but use other sorting function. (10 marks)
  - $O(MN^2)$ total time: Do pairwise comparison to find the frogs at the same position, where the frogs are stored in an array. (8 marks)

- Incorrect solutions: Deduct points from the marks for correct solutions of certain time complexity.
  - Infinite loop for simulation. A lot of `while(true)` loops not terminated correctly … (-2 marks)
  - Incorrect collisions at different time step: incorrectly find collisions, and delete frogs that reach the same place at different time. (-2 marks)
  - Incorrect elimination of frogs: e.g., only eliminate one of the two frogs collided, not considering their leapt distance; eliminate frogs as soon as we see collisions, ignoring collisions for future frogs at this time step (-1~2 marks, depending on their solutions)
  - Deletion of completed frogs: incorrectly remove frogs that have leapt a distance of M (-2 marks)
  - Incorrect frog positions: Simply calculate the position to be the same as initial position + leapt distance, without considering that the track is a loop of M meters (-2 marks)
  - Incorrect resurrection of frogs: somehow an eliminated frog is inserted back. (-2 marks)
  - Eliminated but still in the race: somehow an eliminated frog is still simulated and may cause collision. (-2 marks)

- Incomplete solutions/Unclear descriptions: based on the time complexity I guess your idea is achieving, deduct point depending on how complete the solution is.
  - No way of storing position/leapt distance of frogs (-2~6 marks)
  - No way of removing frogs (-2~8 marks)
  - Keep a minimum of 1~3 marks for students that write at least something that are meaningful and make sense.

Common mistakes:

1. Most people forget to consider the initial positions of the frogs. (-1 mark)
2. Most people only simulate the race for 1 second. I simply regard their implementation as one step in the simulation.
3. Other minor mistakes that don't significantly affect correctness (-1 mark)