CS2040 2023/2024 Sem 2 Midterm (75 marks total)

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

1. What is the runtime complexity of classic quicksort (as taught in lecture) when all elements in the array have the same value?

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

Ans: d

The quicksort discussed in lecture always uses the first index as the pivot. The resulting partition will be imbalanced, with 0 elements less than the pivot and n-1 elements greater than or equals to the pivot.

2. An array A is being sorted using the quicksort algorithm discussed in the lecture. After running for some time, the program was terminated and A becomes [2, 1, 3, 5, 4, 6, 7]. Which of the following array is NOT a valid initial ordering of A?

a. [2, 4, 3, 5, 1, 6, 7]
b. [7, 4, 3, 5, 1, 6, 2]
c. **[6, 5, 1, 2, 7, 3, 4]**
d. [6, 3, 1, 7, 5, 4, 2]
e. None of the above

Ans: c
This question tests if the student understands the partitioning algorithm covered in lecture (This is also known as Lomuto's partitioning algorithm)
In Lomuto's partition algorithm, we maintain 4 variables (i,m,k,j) to define 3 regions: S1 (i+1..m) where the elements are < p, S2 (m+1..k-1) where all the elements are >= p, and unknown (k..j)

When an element less than the pivot is encountered, it is put in the first region (S1) by first expanding m and then swapping the current index k with m.

In the first array, we when we encounter '1', we swap it with '4':
[2, 4, 3, 5, 1, 6, 7] -> [2, 1, 3, 5, 4, 6, 7]
This matches the array captured after the program has been terminated.

We can transform the second array [7, 4, 3, 5, 1, 6, 2] to the first array by running partition() once. Since the first array is a valid initial ordering, so is the second array.

For the last array, the transformation is as follows:
[6, 3, 1, 7, 5, 4, 2] (start)
[6, 3, 1, 5, 7, 4, 2] (insert 5 to S1)
[6, 3, 1, 5, 4, 7, 2] (insert 4 to S1)
[6, 3, 1, 5, 4, 2, 7] (insert 2 to S1)
finished 1st partitioning: [2, 3, 1, 5, 4, 6, 7]
starting 2nd partitioning on [2, 3, 1 ,5 ,4]
[2, 1, 3, 5, 4, 6, 7] (insert 1 to S1, found)

For the third array, the transformation is as follows:
[6, 5, 1, 2, 7, 3, 4]
[6, 5, 1, 2, 3, 7, 4]
[6, 5, 1, 2, 3, 4, 7]
finished 1st partitioning: [4, 5, 1, 2, 3, 6, 7]
[4, 1, 5, 2, 3, 6, 7]
[4, 1, 2, 5, 3, 6, 7]
[4, 1, 2, 3, 5, 6, 7]
finished 2nd partitioning: [3, 1, 2, 4, 5, 6, 7]
[3, 1, 2, 4, 5, 6, 7]
finished 3rd partitioning: [2, 1, 3, 4, 5, 6, 7]
finished 4th partitioning: [1, 2, 3, 4, 5, 6, 7]

As seen, at any point in time, the array does not match with [2, 1, 3, 5, 4, 6, 7].
The closest array to option c that produces [2, 1, 3, 5, 4, 6, 7] is [6, 5, 1, 2, 7, 4, 3]

3. What is the time complexity of f(N) in terms of N?

```
public static void f(int N) { // pre-cond : N is positive
  int i = 0, jump = N/2;
  while (i < N) {
    i += jump;
    for (int j = i; j < i + jump; j++) {
      System.out.print("*");
    }
  }
}
```

a. O(log N)
b. O(N)
c. O(N log N)
d. $O(N^2)$
e. O(N!)

Ans: b
i ends up somewhere in between N to 2N-2 inclusive. j increases from 0, 1, 2, … and ends up at the final value of I with no repeated values of j in the middle. Therefore the print statement is executed O(N) times.

4. What is the time complexity of g(N) in terms of N?

```
public static int g(int N) {
  int x = 0, y = 0;
  for (int i = 1; i < N; i *= 2) {
    for (int j = 1; j < N; j *= 2) {
      for (int k = 0; k < i; k++)
        x++;
      for (int m = 1; m < i; m *= 2)
        y += x;
    }
  }
  return x-y;
}
```

a. O(N)
b. O(N log N)
c. $O(N (\log N)^2)$
d. $O(N (\log N)^3)$
e. $O(N^2)$
f. $O(N^2 \log N)$
g. $O(N^3)$
h. $O(N^4)$

Ans: b
The number of times the second loop (over j) runs is independent of the first loop (over i)
The number of times the third loop (over k) runs is independent of the second loop (over j) and fourth loop (over m) but dependent on the first loop (over i)
The number of times the fourth loop (over m) runs is independent of the second loop (over j) and third loop (over k) but dependent on the first loop (over i)

As the third and fourth loops contain only O(1) statements, we can just analyze how many iterations are made in those 2 loops independently, and add them together (as they occur one after the other)

Suppose the second loop performs x iterations per first loop iteration

Third loop

i =	1	2	4	8	…	N/8	N/4	N/2
k iterations	1 * x	2 * x	4 * x	8 * x	…	N/8 * x	N/4 * x	N/2 * x

Subtotalling O(Nx) time

Fourth loop

i =	1	2	4	8	…	N/8	N/4	N/2
m iterations	1 * x	log2 * x	log4 * x	log8 * x	…	log(N/8) * x	log(N/4) * x	log(N/2) * x
=	x	1 * x	2 * x	3 * x	…	(logN - 3) * x	(logN - 2) * x	(logN - 1) * x

Subtotalling O(x log N) time

Therefore the total time complexity is O(Nx) time, and since x = O(log N), the entire algorithm takes O(N log N) time

5. The following 2D array is used to simulate circular linked lists *cll1*, *cll2*, *cll3*.

   Its first row stores character data and second row stores the index of the next reference in the array (consider this to be 0-based indexing).

Data	A	B	C	D	E	F	G	H	I	J	K	L
Next	8	0	3	6	11	10	9	4	5	2	1	7
Index	0	1	2	3	4	5	6	7	8	9	10	11

   The tail references of *cll1*, *cll2* and *cll3* are in index 7, 8 and 9 respectively.

   With a function **getDataAtM(int tailIndex)** that takes in an integer **tailIndex** representing the tail and returns the data at the **M**-th position (where **M** = floor(size/2)), where position is 0-indexed (i.e. tail is at (size-1)-th position)

   What is the output of **getDataAtM(7)**, **getDataAtM(8)**, **getDataAtM(9)**?

a. E, K , C
b. E, K, D
c. E, K, G
d. L, B, C
e. L, B, D
f. **L, B, G**

Ans: f

cll1 - E > L > H (middle L)
cll2 - F > K > B > A > I (middle B)
cll3 - C > D > G > J (middle G)

6. In lecture 5 of the module, we introduced the DoublyLinkedList where each DListNode contains two references that points to the previous and next node, and contains an integer element.
   Below is an augmented implementation of binary search on a DoublyLinkedList of integers.

```
public static int binarySearch(DoublyLinkedList list, int len,
int x) {
  int mid, low = 0;
  int high = len - 1;

  while (low <= high) {
    mid = (low + high) / 2; // integer division

    int integerAtMidIndex = list.getItemAtIndex(mid);

    if (x == integerAtMidIndex) {
      return mid; // Element found
    } else if (x > integerAtMidIndex) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1; // Element not found
}
```

   The method getItemAtIndex(int index) returns the integer at the specified position in the DoublyLinkedList.
   You can assume that the getItemAtIndex method traverse from the front if index < size(list) / 2, from the back otherwise.

   What is the worst-case time complexity of the function when it runs on a sorted DoublyLinkedList of size N? (choose the best option).

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

Ans: c

Worst-case is bounded by traversing between quarter and half the list log N times.

7. A double hashing scheme is used to create a hash table of size 13. The primary and secondary hash function are h1(k) = k mod 13 and h2(k) = (k mod 19) + (k mod 3), respectively. Assuming that the index returned by the first two probes are already filled, what would be the index of the third probe for k = 30.

a. 4
b. 3
c. 2
d. **0**
e. 7
f. None of the above

Ans: d

Index = (h1(30) + 2* h2(30)) mod 13
= (4+2* (11+0)) mod 13 = 0

8. You are given the following three hash functions to implement a hash table of size 10:

   Given that the input to the hash table contains integers from 0 to 100, which of the hash functions below distribute the keys uniformly over buckets 0 to 9?

   i. h1(k) = (13*k) mod 10
   ii. h2(k) = (7*k^2) mod 10
   iii. h3(k) = k^3 mod 10

a. i only
b. ii only
c. iii only
d. i and ii
e. ii and iii
f. i and iii
g. None of the above

Ans: f

The sequence of keys generated by the hash functions (we only need to check first 10):

h1(k) = (13*k) mod 10

0, 3, 6, 9, 2, 5, 8, 1, 4, 7

Covers all slots from 0 to 9

$h2(k) = (7*k^2) \bmod 10$

0,7, 8, 3, 2, 5, 2, 3, 8, 7

Don't cover the slots 1,4,6,9

$h3(k) = k^3 \bmod 10$

0, 1, 8, 7, 4, 5, 6, 3, 2, 9

Covers all slots from 0 to 9

9. Given a Stack S which contains the following unique integers from top (leftmost) to bottom (rightmost)

   1,2,3,4,5

   If S can only re-order its content (without using recursion) by making use of another initially empty queue Q in the following way:

   1. Can only pop an integer from S and enqueue it into Q
   2. Can only dequeue an integer from Q and then push it into S
   3. 1 and 2 may perform repeatedly to transfer items to and fro between S and Q.

   Which of the following options is not a valid re-ordering of S (Choose all options that apply). If all are valid, choose option e.

a. 5, 4, 3, 2, 1
b. 4, 5, 3, 2, 1
c. 5, 3, 2, 4, 1
d. 1, 2, 5, 4, 3
e. all the above are valid

Ans: e

It is possible using only an extra queue to get all possible permutations of the elements in the stack. Thus any re-ordering is possible.

Proof of why this is possible:

Proof by induction:

1. Let the items from top to bottom of the stack form a sequence S1.
2. Now any contiguous subsequence starting from the 1st item in S1 (top of the stack) can be reversed by popping the involved items and offering them into the queue Q1 and then polling them from Q1 and pushing them back into S. Let call this operation reverse(1,x) where 1 is the 1st item in S1 from the top and x is the xth item in S1 from the top.

For S containing 1 item: There is only 1 permutation so trivially true.

For S containing 2 items: There are only 2 permutations, the original sequence and the reverse, to obtain the reverse, call reverse(1,2).

For S containing 3 items: To illustrate let S1 = i_1,i_2,i_3

There are the following permutations:

i_1,i_2,i_3
i_1,i_3,i_2
i_2,i_1,i_3
i_2,i_3,i_1
i_3,i_1,i_2
i_3,i_2,i_1

Keeping i_3 as the last item, I can get all permutations of i_1,i_2 since the they form a stack of 2 items. This is all permutations where i_3 is the last item.

All I need now is to have each of the other items i_2, i_1 to be in the last position and generate all permutations where they are the last item. This will be all possible permutations of 3 items.

To move i_1 into the last position, reverse S1 to be i_3,i_2,i_1 using reverse(1,3) now I can get all permutations of i_3,i_2.

To move i_2 into the last position, make it the 1st item by performing reverse (1,2) so that S1 is now i_2,i_1,i_3 now reverse the entire S1 to be i_3,i_1,i_2. Now I can get all permutations of i_3,i_1.

Now assume we can get all permutations where S contains n-1 items, the induction follows for S containing n items:

S1 = $i_1,i_2 \ldots i_n$.
Since all permutations can be generated for S containing n-1 items, we can get all permutation of $i_1,\ldots,i_{n-1}$, i.e all items from the 1st to the (n-1)th, keeping i_n where it is.

Now make i_{n-1} to i_1 take turns to be at the last position and generate all possible permutations of the other subsequences from position 1 to n-1.

To make i_x be at the last position
1. reverse i_1 to i_x using reverse(1,x) so that i_x is now the 1st item
2. reverse the entire sequence using reverse(1,n) so now i_x is the last item
3. all possible permutation of the other items from the 1st to the (x-1)th can now be generated.

10. A doubled ended stack is a stack where you can both push and pop from the top or the bottom of the stack.
    Let's call these four operations push_top(), pop_top(), push_bottom() and pop_bottom() respectively.
    Which of the following data structures can be used to implement a doubled ended stack such that all the above 4 operations runs in worst or amortized case O(1) time.

    i. 1 or more array(s)
    ii. 1 or more singly linked list(s)
    iii. 1 or more doubly linked list(s)
    iv. 1 or more hash table(s)

    Note: if the option has 2 or more of the above data structures, they are to be used independently and not in combination to implement the double ended stack.

a. i only
b. iii only
c. i and iii
d. iii and iv
e. i, ii and iii
f. all of i, ii, iii and iv

Ans: c

A double ended stack in this case is simply the same as a double ended queue.

To add or remove from the front and back in O(1) worst case or amortized time only array or doubly linked list is possible.

Note doubly linked list is possible in this case because unlike the teque lab assignment, there is no get operation required.

hashtable is not possible to get worst or amortized case O(1) insert and remove unless we know the maximum number of items from the start and so can make use of a DAT.

2 singly linked list may be used to implement the double ended stack, where the head of one linked list represent the top of the double ended stack and the head of the other represents the bottom of the double ended stack. However you can have a case of all n operations being push_top or push_bottom. In this case for every 2 push you need to re-balance the 2 singly linked list by removing the node at the end of one of the singly linked list and adding it to the end of the other linked list, this will still result in O(n^2) for all n push operations and thus amortized cost of O(n) per operation.

Analysis: 15 marks, 3 problems (5 marks each)

Question 11 and 12 refers to the problem given below:

11. **[5 marks]** In the array-based implementation of a list/stack/queue, whenever the array is full, the strategy is to create a new array that is double the size of the current array. This results in an amortized cost of O(1) time for addBack()/push()/offer() of list/stack/queue respectively.

    Claim: If we change the array enlarging strategy as follows, the amortized cost of addBack()/push()/offer() is still O(1) time.

    New enlarging strategy:

    Start with an array of size $1^2 = 1$, when it is full create an array of size $2^2 = 4$, once it is full create an array of size $3^2 = 9$ etc.

    That is, at the x^{th} time you need to create an array, create an array of size x^2 (x starts from 1).

    Select the option that is most correct about the claim and gives the most correct explanation.

a. True because this is simply the doubling strategy itself
b. True because if you make n addBack()/push()/offer() calls you will need to create O(logn) arrays of size 1,4,9... etc and fill each of them up which result in the same O(1) time amortized cost.
c. True because if you make n addBack()/push()/offer() calls you will need to create O(sqrt(n)) arrays of size 1,4,9... etc and fill each of them up, which result in the same O(1) time amortized cost.
d. False because if you make n addBack()/push()/offer() calls you will need to create O(sqrt(n)) arrays of size 1,4,9... etc and fill each of them up, which result in O(sqrt(n)) time amortized cost
e. False because if you make n addBack()/push()/offer() calls you will need to create O(n) arrays of size 1,4,9... etc and fill each of them up, which result in O(n) time amortized cost.
f. False because if you make n addBack()/push()/offer() calls you will need to create O(logn) arrays of size 1,4,9... etc and fill each of them up, which result in O(logn) time amortized cost.

Ans: d

Now the array sizes grow quadratically that is the 1st array is of size 1^2, the second array is of size 2^2, and so ... the last array need to be of size at least n to store n items so for the last array which is the x^{th} array you have

$x^2 = n$

x = sqrt(n)

Thus you need to create O(sqrt(n)) arrays.

To create and fill up each of the arrays take total time

$O(1^2+2^2+3^2+ \ldots + sqrt(n)^2)$

this is a sum of squares with sqrt(n) terms which sum up to sqrt(n)(sqrt(n)+1)(2*sqrt(n)+1)/6 = O(n*sqrt(n))

thus per insertion the amortized cost = O(n*sqrt(n))/n = O(sqrt(n)).

Question 12 to 13 refers to the following problem

12. **[2 marks]** Consider the following alternative implementation of insertion sort:

```
public static int binarySearch(int[] arr, int start, int end,
int target) {
  int low = start;
  int high = end;
  while (low < high) {
    int mid = (low + high) / 2; // integer division
    if (arr[mid] < target) {
      low = mid + 1;
    } else {
      high = mid;
    }
  }
  return low;
}

public static void insertionSort(int[] arr) {
  for (int i = 1; i < arr.length; ++i) {
     int correctIdx = binarySearch(arr, 0, i, arr[i]);
     int toInsert = arr[i];
     for (int j = i; j > correctIdx; --j)
       arr[j] = arr[j-1];
     arr[correctIdx] = toInsert;
   }
 }
```

Instead of checking the elements from the back, we exploit the fact that the array from 0 .. i-1 is already sorted. We can use binary search to find the correct index of the element that we want to insert in O(log N) time.

Claim: This version of insertion sort is stable, and its best case and worst case time complexity is O(N) and O(N log N), respectively.

a. True
b. False

13. **[3 marks]** Give your rationale for your answer to the previous question.

Ans:

This version is not stable, because the binary search will return the first element which is not less than the target. In other words, if we have a sorted list of [1,2,3,4,5] and we want to insert 2 to the list, the new 2 will be inserted before the existing 2.

Just like normal insertion sort, the best case complexity is when the array is already sorted. In this case, no swap will happen. However, instead of checking the last element of the sorted array which takes O(1), the binary search algorithm will need to check the entire sorted array in log(i) time, where i is the current iteration.

The total of work done is log1 + log2 + log3 + ... + logN, which is O(N log N) (recall tutorial 1)

The worst case for this algorithm is when the input is sorted in reverse order. This way, although the cost of finding the correct index is logn, the cost of shifting the elements is still O(N), leaving us with $O(N^2)$.

Grading Scheme:

There are three (wrong) statements made in Q12:
- The sorting algorithm is stable
- The best time complexity is O(N)
- The worst time complexity is O(NlogN)

To disprove the whole claim, it's sufficient to disprove any one of the three statements above.

3 marks:
- Correctly explained why any of the three statements in Q12 is wrong.

2 marks:
- Explained why the claim is wrong, with minor mistakes.
  - e.g. assumes the best case of binary search is O(1)
- Correctly explained some statements but made mistake(s) in the other parts.
  - e.g. correctly identified the best case but gave incorrect worst case analysis.

1 mark:
- Correctly pointed out the correct statement (i.e. unstable, best case O(NlogN), worst case O(N^2)) but not enough explanation.
- Made some valid points regarding the statements.

Common mistakes

- Assumes the binary search terminates as soon as it found the element equals to the target. In the given snippet, it always takes O(logN).
- Only explained that binary search takes O(logN). This statement alone is not enough to prove/disprove anything.
- Assumes that the sort is stable/unstable just from the existence of swapping. Note that it's important to see how the swaps are done to check whether the sort algo is stable or not.
- Explained that the values can be re-ordered or there is no guarantee that the order is maintained to disprove the stability claim. In this case, it's guaranteed that the ordering of elements with equal value will be reversed.
- Forgot to consider the shifting required after finding the correct index.
- Stated the worst case as O(N^2 * logN). There are two common cases:
    - forgot that O(N + logN) executed N times is O(N^2 + NlogN), which is just just O(N^2)
    - incorrectly derived the time complexity as 1*log1 + 2*log2 + ... + N-1*log(N-1)

Common answers and their respective marks

- Correctly mentioned the time complexity of binary search, but does not use that information to answer the question regarding the claim -> 0 mark
- Saying that the algorithm has runtime of O(NlogN) -> 0 mark without proper explanation, mostly since it's not clear whether the student is talking about the best case or worst case.
    - Note that claiming O(NlogN) in the context of best case analysis is correct, but O(NlogN) in the context of worst case analysis is wrong.

Stability

- Saying sort is unstable, without explanation -> 1 mark
- "repeated values **might** be reordered", "position of elements are affected during insertion", "**cannot ensure** the elements with same value are not reordered" -> 1 mark
- "repeated values **will** be reordered" -> 3 mark

Best case analysis

- Saying best case is not O(N), without explanation -> 0 mark
- Saying best case is not O(N) because binary search itself ALWAYS takes logN but did not explicitly mention the overall complexity of O(NlogN) -> 3 mark
- Saying that searching the index takes N * O(logN) but mentioned that the best case is O(1) (thus O(n) total) -> 1 mark
- Saying the best case is not O(N) because binary search takes O(logN) and insertion takes O(N) -> 1 mark (insertion should be O(1) in the best case)

Worst case analysis

- Saying worst case is not O(NlogN), without explanation -> 0 mark
- Saying worst case is not O(NlogN) because it takes O(N) to insert the element, but did not explicitly mention O(N^2) -> 3 mark
- Saying worst case is O(N^2 * logN) -> 1 mark with proper explanation. otherwise, 0 mark
- Correctly pointed out that each iteration takes O(N) in the worst case:
    - If the overall time complexity is claimed as O(N) / O(NlogN), then it's considered wrong (0 mark). Many student tried to 'prove' that the worst case is indeed O(NlogN), but said that shifting is required.
    - If the overall time complexity is claimed as O(N^2 * logN), then 1 mark is awarded. It's technically correct, but not a tight bound.

Question 14 to 15 refers to the following problem

14. **[2 marks]** Given the following function (which compiles, runs, terminates and does NOT crash):

```
public static int pairUp(int[] arr) { // arr has length N
  ArrayList<Integer> list = new ArrayList<Integer>();

  // first set of nested loop
  for (int i = 0; i < arr.length; i++)
    for (int j = 0; j < arr.length; j++)
      list.add(arr[Math.min(i, j)]);// adds to back of the list
                                    // assume add is worst case
                                    // O(1) time
  int total = 0;
  // second set of nested loop
  for (int i = 0; i < list.size(); i++)
    for (int j = 0; j < list.size(); j++)
      total += list.get(i) * list.get(j);

  return total;
}
```

Claim: The time complexity of pairUp is $O(N^2)$, since the second set of nested loops is done after the first set of nested loops complete

a. True
b. False

15. **[3 marks]** Give the rationale for your answer to the previous question.

Ans: False.

The first set of nested loops adds N^2 elements into the arraylist, then the second set of nested loops does O(1) operations on every possible pair in the arraylist (including (e, e) for some element e), hence it takes $O(N^4)$ time

Grading Scheme:

3 marks to state that it runs in $O(N^4)$ with a valid explanation why it is so.
0 marks otherwise

Common Mistakes
A lot of students are stating that it is true that it runs in $O(N^2)$, but neglected to realize that the data that they are working with grew in size in relation to N.

Some students are just telling us that it does not run in $O(N^2)$, but did not analyse its runtime complexity

Structured Questions: 4 questions, 30 Marks

This section is worth 30 marks. Answer all questions.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes. For the written questions partial marks will be awarded for correct answers not meeting the time complexity required.

16. **[5 marks]**

    A palindrome is a string of characters that reads the same forward and backward. E.g. "php" and "anna" are palindromes, while "pie" and "java" are not.

    There are multiple ways to store a string of N characters in a List ADT.
    For each of the N characters, we will attempt to insert it to the List in sequence from first to last and each one will be inserted to the end of the list.

Which of the data structure(s) given in the options can be used to implement the List ADT (only 1 list) such that after inserting the string of characters, it also allows us to come up with a non-recursive algorithm that runs in O(N) worst case time complexity, and uses worst case O(1) extra space to check whether the string is a palindrome?

Note: The algorithm to check if the string is a palindrome is **not allowed** to make modifications to the data structure.

Choose all options that apply.

a. **Array**
b. Basic Linked List
c. Tailed Linked List
d. Circular Linked List
e. **Doubly Linked List**

Ans: a and e

a and e are similar in the sense that we can set curr1 and curr2 references at the start and end of list (for the array curr1 and curr2 is simply index 0 and index N-1 respectively), and then compare the characters at curr1 & curr2, and move the references inwards towards the middle.

Since you cannot modify the data structure, basic linked list, tailed linked list and circular linked list are not possible to achieve O(n) time as you have to start from the head and go to last node, then the 2nd last node, then the 3rd last node etc… and this will take O(n^2) time.

17. **[5 marks]** Mr Zak wants to design a data structure called fastList to store the most recently used emojis in his chat app. The objective is to read from and write into the fastList in O(1) average or worst case. The user first searches for emojis in fastList. If the emoji is already in fastList, it would be accessed from the fastList and also placed to the front of fastList via a **get** operation. If an emoji is not in fastList, it needs to be accessed from slowList. Once an emoji is accessed from slowList, it is added to the front of fastList via a **put** operation. If the fastList does not have the space for this new emoji, the least recently used emoji would be evicted from the fastList so that the new emoji can be added (this is all part of the **put** operation). The size of the fastList is fixed.

    The read and write mechanisms of fastList are illustrated with an example below.

    Assume that the size of the fastList is 4 and each emoji is assigned a unique integer. User accesses the emojis in the following order: 1,2,3,1,5,2,6

The fastList interface has two methods as mentioned: get(int i) and put(int i), where i is the emoji number.

fastList: (initially, it is empty)

Access emoji-1 from fastList using get(1). emoji-1 is not present in the fastList. It is read from slowList and it will be added to the front of fastList via put(1).

1			

Access emoji-2 from fastList using get(2). emoji-2 is not present in the fastList. It is read from slowList and it will be added to the front of fastList via put(2).

2	1		

Access emoji-3 from fastList using get(3). emoji-3 is not present in the list. It is read from slowList and it will be added to the front of fastList via put(3)

3	2	1	

Access emoji-1 from fastList using get(1). It would be relocated to the front of fastList via put(1).

1	3	2	

Access emoji-5 from fastList using get(5). emoji-5 is not in fastList. It is read from slowList and it will be added to the front of fastList via put(5).

5	1	3	2

Access emoji-2 from fastList using get(2). It will be relocated to the front of fastList via put(2).

2	5	1	3

Access emoji-6 from fastList using get(6). emoji-6 is not in the fastList. It is read from slowList and it will be added to the front of fastList. Item 3 is the least recently used and it would be evicted from the fastList.

6	2	5	1

As shown above, the most recently used emoji would be at the front of the list and least recently used emoji would be at the back of the list. Select the most appropriate method for

the Zak to implement the fastList such that its **get** and **put** operations can each be done in average or worst case O(1) time.

a. **Doubly linked list to store the emojis (their number) and a hash table containing <key,value> entries where the value is a reference to a node of the double linked list.**
b. Singly linked list to store the emojis (their number) and a hash table containing <key,value> entries where the value is a reference to a node of the singly linked list.
c. Tailed linked list to store the emojis (their number) and a hash table containing <key,value> entries where the value is a reference to a node of the tailed linked list.
d. An array to store the emojis (their number) and hash table containing <key,value> entries where the value is an index of the array.
e. A circular linked list to store emojis (their number) and a hash table containing <key,value> entries where the value is a reference to a node of the circular linked list.
f. A circular array to store emojis (their number) and a hash table containing <key,value> entries where the value is an index of the circular array.

Ans: a

Requirements:

Eviction of the least recently used element (i.e., last element in the fastList) need to be done in O(1) on average. Arrays and doubly linked lists can do the eviction of least recently used element in O(1) on average.

In doubly linked list you can access the 2nd last node via the previous reference of the last node (pointed to by the tail) thus allowing removal of the last node (the least recently used emoji) in O(1) time.

Single linked lists, tailed single linked list, and circular single linked lists cannot be used for eviction of least recently used item in O(1) on average.

After accessing an element from the fastList, we need to move it to the front. Moving an element at arbitrary index to the front and moving the elements at the front to the right needs to be done in O(1) on average. Arrays cannot do this as moving the elements at the front to the right cannot be done in O(n) on average

(a) is correct: we have references in the hashtable for each node in double linked list, because for the <key,value> entry in the hashtable, the key will be emoji number and the value the reference to the corresponding node in the doubly linked list. Therefore, we can swap an element from anywhere in the doubly linked list to the head in constant time. Eviction of the least recently used element can also be done in constant time as the double linked list has tail pointer.

18. **[10 marks]** John's niece has finished playing a game involving n balls of different numbers which are unbounded floating point values.
Being mummy's good girl, she is placing the balls back into their container which is a long rectangular box with n (n >= 1000) holes placed along the entire length of the box from one end to the other end. Each hole is marked with the corresponding number of the ball that should go into it. The holes are arranged from smallest ball number to largest ball number from the leftmost hole (call this hole 1) to the rightmost hole (call this hole n), that is, the holes are arranged in ascending order of ball number. Eg if there are 3 balls are numbered 1.13, 3.112, 2.77, then the holes from left to right will be the holes for balls 1.13, 2.77, 3.112 respectively.

    Just as she finishes placing all the balls correctly, her baby brother runs past and kicks the box so that the balls jump out of their hole and are now possibly shifted to another hole!

    After inspecting the container, John discovers that the balls in every m^{th} hole (m = n/10) is still in the correct position (i.e hole 1*m, hole 2*m, ...).
    Call these *unmoved holes*.

    For block of holes fulfilling either one of the following conditions:
    1. holes between 2 consecutive *unmoved holes*
    2. holes between the 1st hole and the 1st *unmoved hole* (if they are not the same)
    3. holes between the last *unmoved hole* and the last hole (if they are not the same)

    There can possibly be at most 2 holes among holes in each such blocks with their balls swapped.

    Help John to devise the most efficient algorithm to place all balls correctly back to their respective holes.
    You can assume that number of the balls are unbounded floating point values in the range 1.0 to 100.0 respectively.
    The container is represented by an array A of size n, with each slot representing a hole (slot 0 is hole 1, slot 1 is hole 2 etc) and the number of the ball in the hole is stored in that slot. The value of m is also given to you.

Ans:

Since the unmoved holes are n/10 holes apart, there are then at most about 10 blocks of holes fulfilling the conditions as given in the problem definition.

Each block being about n/10 holes in size.

Since only 2 holes have their ball swapped in each block as defined in the problem definition, we can re-sort all the balls back into their correct holes in each block using Insertion sort that will take worst case O(2*n/10) = O(n) time.

To do so simply go through the array A as follows to perform the sorting.

1. Let start = 0, end = n/10
2. while (start <= n-1)
   InsertionSort(A,start,end) // insertion sort only from index start to index end
   start = end+1
   end = minimum(end+n/10,n-1)

since the while loop will loop at most about 10 times regardless of n, thus it is bounded and so worst case time complexity is O(10*time complexity of InsertionSort) = O(10*n) = O(n).

Even simpler solution:
There are at most about 10 block of holes where at most 2 balls in each block has their holes swapped. That means in total there are only about 20 balls that are out of place, meaning only 20 values that are out of their sorted order in the array. So simply use insertSort to sort the entire array in worst case O(20*n) time = O(n) time.

Grading Scheme:

1. Identified that the problem can be solved using insertion sort or wrote a code/description to swap misplaced elements in O(n) –> 10 marks
2. Gave the algorithm to swap elements in O(n), but errors in identifying the index of misplaced balls –> 8 marks
3. Identified the solution, but complexity O(n*log n); eg: merge sort -> 7 marks
4. Identified the solution with complexity O(n^2); eg: quick sort, bubble sort, selection sort - 5 marks
5. One pass bubble sort or solution that leads to incomplete swap –> 3 marks
6. Radix sort -> 2 marks
7. Wrong and incomplete description (linked lists, sort method without specifying the sort technique) -> 2 marks

Common Mistakes:

1. Ball numbers and indices of holes are assumed to be same, i.e., A[i] = i is the target. Hole numbers are integers while ball numbers are unbounded floating-point values. Eg:

Hole #	1	2	3	4	5	6	7	8
Ball #	1.2	1.8	3.6	4.1	4.5	4.7	5.0	7.1

2. Some students assume that one iteration of swapping adjacent elements of the array sorts the array: it's a one pass (iteration) bubble sort.

Hole #	1	2	3	4	5	6	7	8
Ball #	1.2	5.0	3.6	4.1	4.5	4.7	1.8	7.1

Balls at indices 2 and 7 are swapped.

After one iteration of bubble sort:

Hole #	1	2	3	4	5	6	7	8
Ball #	1.2	3.6	4.1	4.5	4.7	1.8	5.0	7.1

It is incomplete.

3. Some students assume that only the balls in adjacent slots are swapped, which is untrue. Therefore, after swapping adjacent pair of elements the list won't be sorted.
4. Improved bubble sort: It also takes O(n^2). Try it on the above example.
5. Some students suggested quick sort. However, quick sort is O(n^2) for almost sorted arrays.
6. Several students suggested merge sort, selection sort and bubble sorts. The time complexity of merge sort is O(n log n). The time complexity of selection sort and bubble sort are O(n^2).
7. Auxiliary data structures such as hashmaps, double linked lists, stacks, etc. are proposed without justification.

19. **[10 marks]** Ivan took **n** photographs on a trip. You are given **n**, as well as an array **r** of each photograph's rating in sequence (the ratings may be arbitrarily large integers).

    You are to write an algorithm, to <u>most efficiently</u> recommend, for each photograph **j** from **j** = 1, **j** = 2, **j** = 3, ... to **j** = (**n-1**) in that order, all the photographs **i** taken before photograph **j** that have strictly lower ratings than **j** , i.e. **i** < **j** and **r[i]** < **r[j]**, for which photograph **i has not already been recommended**.

    For each recommended pair of photographs, output (**j**, **i**) ordered by non-descending **j**, using ascending **i** to break ties.

    e.g. for input **r** = [8, 6, 3, 2, 1, 3, 1, 7, 4], the output should be:

    (5, 3)
    (5, 4)
    (7, 1)
    (7, 2)
    (7, 5)
    (7, 6)

    The photographs at indexes 3 and 4 are recommended in the photograph at index 5, but will not be recommended again with the photograph at index 7.

Ans:

This question requires identifying pairs (j, i), where j and i are **indexes of r** satisfying all of:

- i < j – comparing **indexes**, photograph i has to be taken before photograph j
- r[i] < r[j] – comparing **array elements**, photograph i has strictly lower rating than photo j
- each photograph **appears at most once** as the second element in the pair (as i within (j, i)):
  if it can appear multiple times, it should be tied to the lowest possible j – recommend, for each photograph j = 1, 2, … n-1 **in that order**, … for which photograph i has not already been recommended

The output should be ordered by (j, i) ascending.

Brute-Force O(N^3) Solution

```
ArrayList<Integer> rec = new ArrayList<>();
for (int j = 1; j < n; j++) {
      for (int i = 0; i < j; i++) { // or < n but check i<j later
            if (r[i] < r[j] && !rec.contains(i)) {
                  S.o.println("(" + j + ", " + i+ ")");
                  rec.add(i);
            }
      }
}
```

The arraylist rec stores indexes of recommended photographs. Checks between indexes and ratings of i and j are made. No sorting is required as the order of iteration already causes output to be ordered (j, i) ascending.

Using HashSet to Store Recommended Photographs, O(N^2) AVERAGE Solution

```
HashSet<Integer> rec = new HashSet<>();
for (int j = 1; j < n; j++) {
      for (int i = 0; i < j; i++) {
            if (r[i] < r[j] && !rec.contains(i)) {
                  S.o.println("(" + j + ", " + i + ")");
                  rec.add(i);
            }
      }
}
```

HashSet improves the time taken to search whether a photograph i has already been recommended from O(N) worst, to O(1) average. Do note that HashSet operations are **NOT O(1) worst case**. In the unfortunate event that there are many collisions, it is possible to get many > O(1) operations.

Lazy Deleting Recommended Photographs, O(N^2) WORST-Case Solution

```
boolean[] rec = new boolean[n];
for (int j = 1; j < n; j++) {
      for (int i = 0; i < j; i++) {
            if (r[i] < r[j] && !rec[i]) {
                  S.o.println("(" + j + ", " + i+ ")");
                  rec[i] = true;
            }
      }
}
```

We can place a mark on each photograph to note whether it is recommended or not. In order not to mutate the ratings in r, we can create a parallel array of the same length as r to note whether a photograph i has been recommended or not.

Monotonic Stack, O(N) Solution

You may realize that you can only recommend photograph i to a later photograph j when r[i] < r[j], after which the earlier photograph i is out of the picture. So if we remove all photographs which have already been recommended halfway through the algorithm (after some recommendation for some j completes), we will never see any two unrecommended photographs with the later one having a higher rating.

This means we can maintain a list of ratings that are not yet recommended, they are guaranteed to be in non-ascending order i.e. r[i] >= r[j] for i < j. Since all photographs that qualify to be recommended need to have rating < r[j], we can start from the lowest rating till either all photographs are recommended, or we find a photograph with rating >= r[j] which cannot be recommended to this photograph j.

Therefore we can compare r[j] with the last-inserted element of the list which is guaranteed to have the (or one of the) lowest rating. This means the list can be viewed as a stack, since we only add, access and remove elements from one end. Since we need the indexes i and j, we should store the photograph index, not the rating (which may not be unique).

Every photograph i on top of the stack that qualifies to be recommended can be popped out of the stack and paired with the current photograph j. With j moving from 0 to n-1 inclusive, and the latest (yet unrecommended) photograph i being popped from the stack, the output is ordered: (j ascending, i descending). Therefore, the indexes popped from the stack for each j need to be first reversed.

```
// one photo is a <rating, index> pair
// stack invariant is that the:
//    photos are in chronological order (latest at top)
//    ratings of i non-ascending (one of the smallest at top)
Stack<Pair> unRec = new Stack<>();
for (int j = 0; j < n; j++) {
      int newRating = r[j];

      // reverse the 'i's with strictly lower ratings
      Stack<Integer> toRec = new Stack<>();
      while (!unRec.isEmpty() && unRec.peek().rating < newRating)
            toRec.push(unRec.pop().index);

      // output (j, i) pairs, j already ASC, now also i ASC
      while (!toRec.isEmpty())
            S.o.println("(" + j + ", " + toRec.pop()+ ")");

      // photo j is not yet recommended
      unRec.push(new Pair(r[j], j));
}
```

Each photograph gets pushed at most once into unRec, popped at most once from unRec, pushed at most once into toRec, and popped at most once from toRec. Though a photograph at the top of the stack may be accessed more than once, over the entire algorithm, there will be at most n-1 peek() calls which do not result in the top being popped (because j would advance to the next photograph in such a case).

Grading Scheme:

Correct solutions:

- 10 marks – O(N) worst-case
- 8 marks – O(N log N) worst-case / O(N log N) average / O(N) average
  Typically, unnecessary sorting for O(N log N)
  Typically, unnecessary use of hash table for O(N) average
- 6 marks – $O(N^2)$ worst-case
- 5 marks – $O(N^2)$ average
- 4 marks – $O(N^2 \log N)$ worst-case
  Typically, unnecessary repeated sorting when each pair is recommended
- 3 marks – $O(N^3)$ worst-case

1 mark deducted – Each minor mistake

2 marks (3 marks) deducted – Not ordering 1 dimension (2 dimensions) correctly

Major mistakes (affecting whether identified pairs meet the 3 conditions):

- 3 marks deducted – Using HashMap with rating as key, thereby losing elements
- Max 1 mark – allowing photo i that appears AFTER j to be recommended to j
- Max 1 mark – not comparing rating of photo i with respect to photo j
- Max 1 mark – not filtering only photographs that are not yet recommended
- Max 1 mark – messing up array indexes greatly and relying on the wrong indexes

0 marks – 2 or more major mistakes, as there is a simple check for each of the 3 required conditions

No special credit for:

- creating linked lists
- reorganizing data using hashing
- sorting / swapping / partitioning / divide-and-conquer

as unfortunately any one of these strategies does not advance closer to the goal

Providing alternatives – You are at risk of having the solution with the lowest marks awarded!

Common Mistakes

- Declaring the outermost loop control variable to be i, the inner loop's be j. Swapping variable names is technically not wrong but MANY students confused themselves doing so
- Using ratings in place of indexes – these are NOT interchangeable for this question as ratings can be very large (you don't even know if they are negative or not). If you loop over each rating, you run the risk of getting very bad time complexity, and you may go out of bounds if there are negative ratings. If you mark or search for indexes based on ratings then you may lose track of position once there are duplicates

- Confusing ratings r[i] r[j] with indexes i j; confused/unclear about “for i in r” vs “for i = 0 to < n”
- Returning instead of outputting/printing – Return generally terminates the function immediately, and the program does not provide any output to the system
- Searching for already recommended photo by comparing i against a (j, i) pair in the output
- Looping over i starting from 1 and missing out index 0; allowing i to be larger than j when iterating; looping over i from j-1 downwards but checking only i < j and forgetting about i >= 0
- Searching for a key or key-value pair in a HashMap by value; searching for the next smaller key (i.e. lower / predecessor) in a Hashtable when it is an unordered data structure
- Attempting to sort a Hashtable – Hashtable is an unordered data structure, if you mess up the index each element is at, then you can’t use the same hash function to find the element anymore
- Wasting time creating comparators and sorting output that is already ordered by (j, i) ascending
- Using radix sort, thinking that “radix sort takes O(n) time” – forgetting about the number of digits and buckets required
- (This is more python-specific) Having an output list: output = [], and then output += (j, i) – the dimensioning is incorrect
- Using a for loop over a stack instead of popping while there are still elements within – Not really a valid pure stack operation. If used over an array-based stack, will iterate over elements from bottom to top of the list/stack
- Attempting to recommend photograph to one with next higher rating – This doesn’t take position of the photographs into account: i may not have appeared yet relative to j, or i may already be recommended to some photograph k that appeared in the middle of the two (i < k < j so would have already output (k, i))
- Attempting to recommend only adjacent photographs (i.e. (i+1, i))
- Thinking that there are somehow 2 types of photographs: i photographs and j photographs – But the question says that you need to recommend some photographs i for each photograph j = 1, 2, 3, .. n-1, i.e every photograph (except the first) could be a photograph j, regardless of whether it becomes a photograph i later or not