CS2040 2022/2023 Sem 2 Midterm

MCQ: 63 Marks, 21 Questions (3 marks each)

1. Give the time complexity of the code below, assuming the initial call to the function is rec(n, n), where n is a global variable accessible by the function.

```
void rec(int a, int b) {
    if (a + b <= n) {
        return;
    }
    System.out.println("*");
    rec(a-1, b);
    rec(a, b-1);
}
```

a. O(n)
b. O(nlogn)
c. $O(2^n)$
d. $O(4^n)$

Ans:

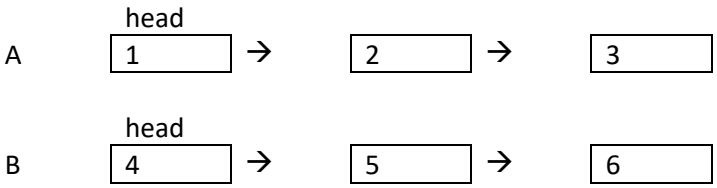
Notice that the initial value of (a + b) is 2n, and this decreases by 1 at each level of the recursive tree (regardless of whether it is a that is decremented, or b), before terminating when (a + b) == n.

This is effectively just a disguised version of the following function, supposing the initial function call is rec2(n):

```
void rec2(int a) {
    if (a <= 0) {
        return;
    }
    System.out.println("*");
    rec2(a-1);
    rec2(a-1);
}
```

The time complexity of this function is $O(2^n)$.

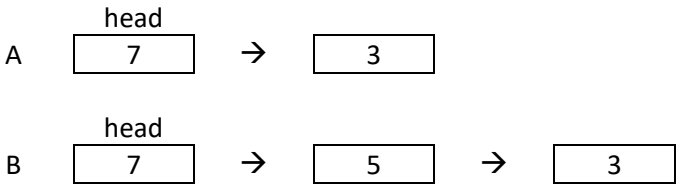
2. Given the following two basic linked lists A and B:



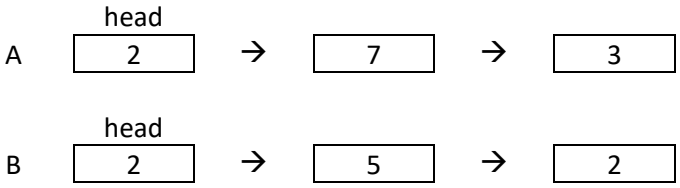
Show the contents of both linked lists after the following operations. If there are any nodes that are unreachable by A or B, do not include it in the answer. If the same nodes appear in both A and B, they should be presented as separate nodes in the answer.

```
A.head = A.head.next;
A.head.next.value = 7;
B.head.next.next = A.head.next;
B.head.value = A.head.value;
```

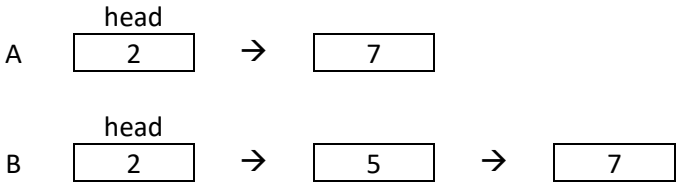
a:



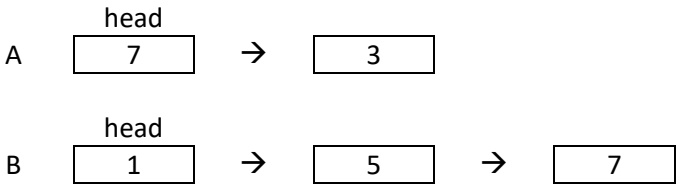
b:



c: (answer)



d:



3. You are given the following array:

4	7	3	8	6	2	5

Bubble sort (in ascending order) is run on the array, and a snapshot of the array (ie. a copy of the array) is made after every swap. Which of the following options show a possible snapshot of the array?

a:

4	3	7	2	5	6	8

b:

3	4	5	2	6	7	8

c: (answer)

3	2	4	5	6	7	8

d:

4	3	2	5	6	7	8

Ans:

The following arrays that could have occurred are the most similar ones to the arrays presented as options:

a:

4	3	7	**6**	**2**	**5**	8

After 4 swaps

b:

3	4	**2**	**5**	6	7	8

After 10 swaps

c:

3	2	4	5	6	7	8

After 11 swaps

d:

2	3	**4**	5	6	7	8

After 12 swaps

4. The following hash table of size 7 was created using linear probing with hash function h(key) = key % 7:

Index	0	1	2	3	4	5	6
Key	(empty)	22	(deleted)	29	(empty)	19	(empty)

"(deleted)" refers to a key that was removed from the hash table. What is the load factor of the hash table (rounded to 2 decimal places)?

a. 0.29
b. 0.43
c. 0.57
d. 0.75

Ans:

The hash table contains 3 keys in a size 7 hash table, so the load factor is 3/7 = 0.43 when rounded.

5. What is the time complexity of the following method when `ite(m)` is called for some integer m > 1?

```
void ite(int n) {
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
    int j = 0;
    while (j < n) {
        j = j + binSearch(arr, n, j);
    }
}
```

`binSearch()` is the method defined in Lecture 3 of the module, and is reprinted below for convenience:

```
public static int binSearch(int[] a, int len, int x) {
    int mid, low = 0;
    int high = len -1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x == a[mid]) return mid;
        else if (x > a[mid]) low = mid + 1;
        else high = mid -1;
    }
    return -1;
}
```

a. O(m)
b. O(m log m)
c. $O(m^2)$
d. Infinite (ie. program does not terminate)

Ans:

Creating and populating the array arr with values from 0 to (n-1) takes O(n) time. When attempting to binary search for the initial value of j (0), this is found at index 0 of the array, therefore 0 is added to the value of j. Since the value of j remains at 0, no progress is made, and the program loops infinitely.

6. You are provided the following information for a group of people:
   Age (an integer)
   Occupation (represented as a String)

   You would like to order them by their occupation first (in ascending lexicographical order), and if they are the same (i.e. have the same job title), to order them by ascending age (i.e. younger people appear first). Among the options below, which of the following would be the best way to do this?
   Your answer should consider correctness first, and if multiple options would give the correct sorted result, you should pick the best option in terms of speed (i.e. when both sorts are run, what is the overall big O notation). It is guaranteed that there are no ties for the best option.
   For simplicity, all sorts are assumed to sort items in ascending order.

a. (Regular) bubble sort by occupation, then radix sort the result by age
b. Selection sort by age, then insertion sort the result by occupation
c. Quick sort by occupation, then improved bubble sort the result by age
d. Radix sort by age, then quick sort the result by occupation

Ans:

Options (a) and (c) give the wrong result (sorted by age, then by occupation).

Option (d) gives the wrong result (sorted by occupation, but not necessarily sorted by age).

Option (b) gives the correct result in $O(n^2)$ time.

7. You want to create a linked list that is capable of supporting all these operations in worst case O(1) time:

    1. Adding new node to front of list
    2. Removing first node (at front of list)
    3. Retrieving first node (at front of list)
    4. Adding new node to back of list
    5. Removing last node (at back of list)
    6. Retrieving last node (at back of list)

    You start off with a basic linked list (as taught in lecture). Minimally, how many of the additional modifications below would need to be made to support all the above operations in worst case O(1) time?

    i. Adding a tail reference to the list
    ii. Having .next of the tail node point to the head
    iii. Having each node be doubly linked. If combined with (2), this also causes the .prev of the head node to point to the tail
    iv. Storing the size of the list

a. 1
b. 2
c. 3
d. 4

Ans:

A basic linked list is capable of doing (1), (2), and (3) in O(1) time already. By introducing modification (i), it is now capable of doing (4) and (6) in O(1) time. However, to remove the last node in O(1) time, the nodes need to be doubly linked (option (iii)) to access the second last node in O(1) time to update its .next reference.

8. You are given a stack A, consisting of 2N integers as elements. Suppose for the purposes of discussion that the top of the stack has index 0, and the bottom of the stack has index (2N-1). You would like to move the elements into 2 other initially empty stacks B and C such that:

   B contains element 0 at the top, and element (N-1) at the bottom.
   C contains element N at the top, and element (2N-1) at the bottom.

   You are only allowed to use the following method (assume the method is syntactically correct) to modify any of the stacks:

```
void move(Stack x, Stack y) {
      for (int i = 0; i < N; i++) {
            y.push(x.pop());
      }
}
```

   Determine the minimum number of calls to `move()` that is needed to perform the procedure above.

a. 4
b. 6
c. 8
d. The procedure is impossible under the constraints given

Ans:

The full list of move calls necessary is as follows:

move(A, C)
move(C, B)
move(A, B)
move(B, C)

9. The following is an array after one pass of partitioning/pivot swapping from quick sort:

3	2	4	5	7	6	9	8

Determine a possible original arrangement of the array (ie. how the array could have looked like before quick sort was run).

a:

3	5	4	7	2	9	8	6

b:

4	2	9	5	8	3	7	6

c: (answer)

5	2	6	8	7	4	9	3

d:

4	6	5	3	9	8	2	7

Ans:

The following is the result of one round of quick sort on each of the options. Note: there is a faster way of eliminating incorrect options rather than running quick sort on all the arrays. This is left as an exercise to the reader.

a:

2	**3**	4	**7**	**5**	**9**	**8**	**6**

b:

3	2	4	5	**8**	**9**	**7**	**6**

c:

3	2	4	5	7	6	9	8

d:

2	**3**	4	**6**	**9**	**8**	**5**	**7**

10. You are given the following (rather densely populated) hash table:

Index	0	1	2	3	4	5	6
Key	17	23	16	10	18	26	32

The hash table was initially empty, before keys were inserted into it one at a time, using modified linear probing as the collision resolution technique (the value of *d* used is unknown), and hash function h(key) = key % 7. No deletions were involved.

Determine which of the following could be a possible value of *d*.

(Select all options that are a possible value of *d*)

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

Ans:

Consider the slots which each key should map to:

Slot 2: 16, 23
Slot 3: 10, 17
Slot 4: 18, 32
Slot 5: 26

With this information, we know that keys 23, 17 and 32 (in indices 1, 0, and 6 respectively) were placed there as a result of some collision. We now try to perform probing using each of the possible values of d:

d = 2:
2 -> 4 -> **6** -> 1
3 -> 5 -> 0
4 -> 6

Only key 23 requires another key to be inserted first (in this case, 32), so a possible insertion order would be 16, 10, 18, 26, 32, 23, 17.

d = 3:
2 -> 5 -> 1
3 -> **6** -> 2 -> 5 -> **1** -> 4 -> 0
4 -> **1** -> 5 -> 2 -> 6

Key 17 requires key 32 to be inserted first, which in turn requires key 23 to be inserted first (which is also needed by key 17). A possible insertion order would be 16, 10, 18, 26, 23, 32, 17.

d = 4:
2 -> **6** -> 3 -> **0** -> 4 -> 1
3 -> 0
4 -> **1** -> 5 -> 2 -> 6

Keys 23 and 32 require each other to be inserted first, to be placed in their respective indices, which is impossible under the constraints given.

d = 5:

2 -> **0** -> 5 -> 3 -> 1
3 -> **1** -> 6 -> 4 -> 2 -> 0
4 -> 2 -> **0** -> 5 -> 3 -> **1** -> 6

Keys 23 and 17 require each other to be inserted first, to be placed in their respective indices, which is impossible under the constraints given.

11. **Refer to the following binary search algorithm snippet for questions 11 to 14.**

```
boolean binarySearch(List list, int value) {
    int low = 0;
    int high = list.size()-1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (list.getItemAtIndex(mid) == value) return true;
        else if (list.getItemAtIndex(mid) < value) low = mid+1;
        else high = mid-1;
    }
    return false;
}
```

*Assume list contains N integer items.

If List is implemented using an **array**, the **best case** time complexity of binary search will be

a. O(1)
b. O(log N)
c. O(N)
d. O(N log N)
e. $O(N^2)$

12. If List is implemented using an **array**, the **worst case** time complexity of the binary search will be

a. O(1)
b. O(log N)
c. O(N)
d. O(N log N)
e. $O(N^2)$

13. If List is implemented using a **singly linked list** (basic linked list in the lecture notes), the **best-case** complexity of the binary search will be

a. O(1)
b. O(log N)
c. O(N)
d. O(N log N)
e. $O(N^2)$

14. If List is implemented using a **singly linked list** (basic linked list in the lecture notes), the **worst case** complexity of the binary search will be

a. O(1)
b. O(log N)
c. O(N)
d. O(N log N)
e. $O(N^2)$

Ans:

Best-case is when we find the answer in 1 step, worst-case is when we need log N step.
If we use singly linked list, we need to multiply everything by N as it does not have a random access.

15. **Please refer to the following quick sort algorithm snippet for questions 15 to 16.**

```
void quickSelect(int[] arr, int k, int low, int high) {
   // Same quickSelect algorithm as explained in tutorial 2,
   // Running this algorithm will ensure arr is arranged as  follows:
   // * arr[k] contains the (k-low+1)-th smallest element from arr[low…high]
   // * arr[i] < arr[k] for all low ≤ i < k
   // * arr[i] > arr[k] for all k < i ≤ high
}

void quickSort(int[] arr, int low, int high) {
    if (low >= high) return;

    int pivot = another_partition(arr, low, high)
    quickSort(arr, low, pivot-1);
    quickSort(arr, pivot+1, high);
}
```

Suppose that `another_partition(arr, low, high)` is as follows:

```
int another_partition(int[] arr, int low, int high) {
    quickSelect(arr, (low+high)/2, low, high);
    return (low+high)/2;
}
```

The **worst case** time complexity of calling `quicksort(arr, 0, N-1)`, where arr is an integer array containing N integers, will be

a. O(NlogN)
b. $O(N^2)$
c. $O(N^2 logN)$
d. $O(N^3)$
e. Will not terminate

16. Suppose that `another_partition(arr, low, high)` is as follows:

```
int another_partition(int[] arr, int low, int high) {

    quickSelect(arr, low, low, high);

    return low;

}
```

a. O(NlogN)
b. $O(N^2)$
c. $O(N^2 logN)$
d. $O(N^3)$
e. Will not terminate

Ans:

Note that quickSelect has a worst-case time complexity of $O(N^2)$.
For Q15, the time complexity is then $T(N) = O(N^2) + 2T(N/2)$. This will be $O(N^2)$.
For Q16, the time complexity is then $T(N) = O(N^2) + T(N-1)$. This will be $O(N^3)$.

17. **Questions 17 to 19 refers to the problem described below**

You are given a hash table with hash function h(key) = key % 7 and linear probing as its collision resolution. It is known that after a sequence of insertions, the hash table is as shown below.

index	0	1	2	3	4	5	6
value	(empty)	1	23	8	18	5	12

A possible insertion sequence to get the hashtable above is

a. 1, 18, 8, 5, 12, 23
b. 23, 1, 12, 18, 8, 5
c. 18, 23, 5, 8, 1, 12
d. 1, 5, 18, 12, 23, 8
e. 18, 5, 23, 8, 1, 12

18. The number of possible insertion sequences that will generate the hashtable is

a. 180
b. 120
c. 60
d. 240
e. 30

19. Suppose we create a bigger hashtable of size x, and we rehash the elements from the original hashtable into this new hashtable with hash function h(key) = key % x. Among the following x, the one which gives the maximum length of the **longest used contiguous slot** (defined below) is

***longest used contiguous slot** is the longest unbroken sequence of slots in the hashtable that is occupied by an item. In the original hashtable is the **longest used contiguous slot** is of length 6, since all the keys occupy consecutive slots.

a. **11**
b. 15
c. 13
d. 14
e. Insufficient information to determine

Ans:

Q17. Just simulate the insertion

Q18. There are 3 groups of independent insertions: {1, 23, 8}, {18}, {5, 12}. {1, 23, 8} has 2 possible sequence, {18} has 1, and {5, 12} has 1. Thus, the number of sequences is 6!/(3!1!2!) * (2*1*1) = 120

Q19. Simulate one of the insertion sequences. The longest used contiguous slot for 12, 13, and 14 is 2, while 11 is 3. Note that this works because with linear probing, the used slots will always be the same regardless of the insertion sequence.

20. **Questions 20 to 21 refers to the problem description given below**

A string is called “almost palindrome” if we can rearrange its letters such that the resulting string is a palindrome (a palindrome is a string which reads the same forwards and backwards). For example, “acabb” and “abab” are almost palindrome because their letters can be rearranged into “abcba” and “abba”, respectively. "aacb" is not an "almost palindrome" since there is no way to rearrange the letters to form a palindrome.

Our problem is as defined below:

Given a string of length N which consists of only lower-case alphabets ('a' to 'z'), we want to know the minimum number of character to remove such that the string becomes an "almost palindrome" string. Let's call this minimum number of characters to be removed min_{rc}.

For example:

- For “aaabcc”, min_{rc} = 1, because we can remove one of the ‘a’ and the string will become an "almost palindrome", since we can rearrange the remaining characters into “acbca”.
- For “aaaabb”, min_{rc} = 0, because without removing any characters we can rearrange it into “baaaab”.

Answer the following question:

The maximum value of min_{rc} to make any possible string of length N (assume N is very large) become an "almost palindrome" string is

a. N-1
b. N/2
c. 25
d. 26
e. N

21. The ADT (you can use multiple of it, for e.g 2 lists ...) that can result in the most efficient algorithm (in terms of worst case time complexity) to solve the problem defined in the previous question is

(select all possible options that satisfy the requirement)

a. Queue
b. Stack
c. List
d. Map

Ans:

Observe that this problem boils down to counting how many characters have an odd number of occurrences.

For Q20, we can have at most 1 character with an odd number of occurrences. Thus, the maximum answer is 26-1 = 25.

For Q21, we will calculate the occurrence of each character. Notice that we can simulate a counting array using all mentioned ADT with the worst-case time complexity of O(26) = O(1). Hence, all of them are equally good in terms of the big-O of their worst-case time complexity.

Analysis: 12 marks, 3 questions (4 marks each)

Questions 22 to 24 refer to the problem below

22. John is implementing an application for a small university where he needs to insert *n* integer numbers (say, student IDs) every month. The people who use the application also need to print a sorted list of all the numbers at the end of each month. There are 2 possible ways of doing this.

    (a) John plans to implement the data structure as a regular array list.

    (b) His friend Lucy suggests a new approach: "Why not implement the array list as a sorted array list, i.e., when inserting keep the numbers sorted in the list at all times. This will make printing the numbers at the end of the month very efficient."

    Assume that for the unsorted list (idea (a)) each number is inserted at the end of the array list which is initialized to be of size *n*.

    Do not consider deletions (the list is automatically emptied after the print operation each month).

    Lucy says her implementation is overall more efficient in terms of the worst case time complexity when considering all operations:

    (a) (John) Unsorted list with *n* insertions at the end of the array, and 1 sorted print.

    (b) (Lucy) Sorted list with *n* insertions (at the appropriate locations), and 1 sorted print.

    Lucy is right.

    [2 marks]

a. True
b. False

23. The reasons are as follows: **[1 mark]**

1. This is because the worst case time complexity of all *n* insertions (disregarding printing) for solution (a) and (b) are as follows

a. (a) n * O(1)
(b) n * O(logn)

b. (a) n * O(n)
(b) n * O(logn)

c. (a) n * O(1)
(b) (n * O(logn)) + (n * O(n))

d. (a) n * O(n)
(b) (n * O(logn)) + (n * O(n))

e. (a) n * O(logn)
(b) (n * O(logn)) + (n * O(n))

24. [1mark]
2. Further explanation is as follows

a. For (a) the n insertions are $O(n^2)$ while for (b) the insertions are only O(n log n). So (b) is more efficient.

b. For (a) we need to do sorting with O(n log n) before printing, while (b) needs no sorting. So (b) is more efficient.

c. For (a) the n insertions are O(n) while for (b) the insertions are O(n log n). So (a) is more efficient.

d. Considering all operations – insertions, sorting (for (a)), printing – the complexity for both (a) and (b) are equal.

e. For (b) the n insertions require $O(n^2)$ while for (a) all of the operations (insertions, sorting, printing) require at most O(n log n). So (a) is more efficient.

Further explanation for Q24.

Insertions: n * O(1) = O(n).
Printing: sorting O(n log n), then printing O(n). Total O(n) + O(n log n).

Insertions: we need to find the right location in the sorted array. On average this will take about O(log n) steps if the list is n items long. However, we also need to relocate the items after the insertion point. This takes about n/2 steps. Totally we need to shift 1+2+3+4+…+n/2 items for n insertions: $O(n^2)$. No sorting is needed. Printing is simple, just O(n). So the total is O(n2) + O(n). Thus, this approach has worse complexity than (a) and Lucy is wrong (False).

Question 25 to 27 refers to the following problem

25. Given the following function

```
void Func1(int arr[], int i, int j, boolean flag) {
  if (j <= 0)
    return;
  else {
    if (flag)
      // arr.length gives the size of the array
      for (int k = arr.length-i; k < i; k++)
        System.out.println(arr[k]);
    else
      for (int k = arr.length; k > i-1; k--)
        System.out.println(arr[k-1]);
    Func1(arr,i-1,j/2,true);
    Func1(arr,i-1,j/2,false);
  }
}
```

The time complexity of calling Func1(a,n,n,true) where a is an integer array of size n (n > 100) takes **O(nlogn)** time.

a. True
b. False

26. The reason is as follows: [1 mark]

1. The recurrence for the time complexity of Func1 is

a. T(n1,n2) = 2T(n1-1,n2/2) + cn1 where c is a constant,n1 and n2 start with n and base case is n1 = 0.

b. T(n) = 2T(n/2) + cn where c is a constant and base case is n = 0

c. T(n) = 2T(n-1) + cn where c is a constant and base case is n = 0

27. [1 mark]

2. Therefore the recurrence evaluates to a time complexity of

a. O(n)
b. O(nlogn)
c. O(n^2logn)
d. O(n^2)
e. O(n(logn)^2)
f. O(2^n)

Further explanation on how to get O(n^2) from T(n1,n2) = 2T(n1-1,n2/2)+cn1

T(n,n1) = 2T(n-1,n1/2) + cn
= 2[2T(n-2,n1/4) + c(n-1)] + cn
= 4T(n-2,n1/4) + 2c(n-1) + cn
= 4[2T(n-3,n1/8) + c(n-2)] + 2c(n-1) + cn
= 8T(n-3,n1/8) + 4c(n-2) + 2c(n-1) + cn
...
= nT(n-logn,0) + cn + 2c(n-1) + 4c(n-2) + ... + (n/2)*(n-logn+1)
= cn + cn + 2c(n-1) + 4c(n-2) + ... + (n/2)*c*(n-logn+1)
= cn + c(1*n + 2(n-1) + 4(n-2) + ... + (n/2)(n-logn+1)) // ignore +1 to simplify the evaluation
= cn + c([1*n + 2n + 4n + (n/2)*n] - [2+4+8+....+(n/2)*logn])
logn terms here
logn terms here so if we just take each term to be nlogn then the sum here is O(n(logn)^2) ← not tight but does not change the analysis
= O(n) + O(n^2) - O(n(logn)^2) <- dominating term is O(n^2)

= O(n^2)

Questions 28 to 29 refers to the problem below

28. In Mergesort of an integer array A of size *n* where the integer values can range from 1 to n^n, the merge algorithm used in the merge step for merging the 2 sorted subarrays A[low...mid] and A[mid+1…high] into the sorted subarray A[low…high] can be replaced with **one or more** of the following sorting algorithms taught –

Insertion Sort, Improved Bubble sort, Radix Sort to sort A[low…high] and still result in a **stable sort** that runs in **worst case O(nlogn)** time. [2 marks]

a. True
b. False

29. [2 marks]

The reason for the previous question is as follows

(Select the most correct one out of all the given options)

a. Since the 2 subarrays to be merged are always sorted, **Insertion sort** which is a stable sort should run in worst case O(m) where m = high-low+1, instead of $O(m^2)$, thus it will be the same time as the merge algorithm and so Mergesort will still be O(nlogn).

b. Since the 2 subarrays to be merged are always sorted, **Insertion sort and Improved bubble sort** which are stable sorts should run in time O(m) where m = high-low+1, instead of $O(m^2)$, thus it will be the same time as the merge algorithm and so Mergesort will still be O(nlogn).

c. Since **Radix sort** is stable and runs in worst case linear time, i.e O(m) where m = high-low+1 thus it will be the same as the merge algorithm and so Mergesort will still be O(nlogn).

d. **Radix sort, Insertion sort and Improved Bubble sort** can be used for the reasons given in b and c above.

e. None of the sorting algorithms listed can be used since all of them might result in an unstable sort.

f. It cannot be guaranteed that Radix, Insertion and Improved Bubble Sort will always run in time linear to the size of the subarray A[low...high] so it is not guaranteed that the final run time of mergesort is O(nlogn).

Ans:

Option a. is not the correct reasoning. You can have the values in one subarray (s_1) being all smaller than the value in the other subarray (s_2), and they are place into A[low…high] as s_2s_1 then you will have to shifting all values in s_1 past all the values in s_2 before they are in their correct sorted position. This will take $O(m^2)$ is s_2 is of size m/2.

Option b. is not the correct reasoning. This is because insertion sort will not be O(m) as explained above. Improved bubble sort will also not be O(m) if you have the same situation as given above you will have to bubble up all the values in s_2 past all the values in s_1 before they are in their correct sorted position thus resulting in a $O(m^2)$ sort.

Option c. is not the correct reasoning because the largest number is n^n thus there are O(nlogn) digit and not a constant number of digits.

Option d is not the correct reasoning due to the reasons as given above

Option e is not the correct reasoning since insertion, bubble and radix sort are stable.

Option f is the correct reasoning.

Structured Questions: 3 questions

This section is worth 25 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.

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

Questions 30 to 32 refers to the problem given below

30. When Matthew was in middle school his mom bought him a pair of rabbits. As time progressed the rabbits had more and more young rabbits and he now has *m* rabbits, where *m* >= 2 and *m* is an even number. Recently, his mom told him that he has too many rabbits and he needs to give away half of them. However, Matthew is quite attached to all his rabbits and cannot make up his mind about which rabbits to give away.

    Finally, he decides to use an algorithm as follows:

    He will arrange all the rabbits in a circle and will label them from 1 to *m* in a clockwise manner. Then, counting from position 1 in a clockwise manner, he will give away every n^{th} rabbit until only *m*/2 rabbits are left in the circle. Every time a rabbit is marked as the n^{th} rabbit, it will be removed from the circle before the algorithm continues.

    Design an algorithm *RabbitGiveAway* that takes 2 inputs: the number *m* of rabbits in the circle, and a value *n*, where every n^{th} rabbit will be given away. Your algorithm should display the positions of those rabbits given away, in the order they are given away. The algorithm starts from position 1 and stops when there are *m*/2 rabbits remaining.

    Answer the following question to ensure you have understood the problem:

    Is the following a valid sequence for the first 6 rabbits to be given away when m = 30 and n = 9? [2 marks]

    9,18,27,6,16,26

    a. True
    b. False

31. Let us now consider some general aspects of the *RabbitGiveAway* algorithm.
Which of the following is/are **True**? [2 marks]
(Select all options you think are true)

a. The *RabbitGiveAway* algorithm only works correctly if at the beginning the value *m* is larger than *n* (i.e., *m* > *n*)

b. The *RabbitGiveAway* algorithm can work correctly until there is only 1 rabbit left

c. If we have a space-efficient algorithm of *RabbitGiveAway*, then we need O(*m*+*n*) space

d. If we have a space-efficient algorithm of *RabbitGiveAway*, then we need O(m) space

e. If the *RabbitGiveAway* algorithm starts with values *m* > *n*, then it must stop when the number of remaining rabbits reaches *n*.

32. Give an algorithm that runs in O(m*n) time, which utilizes an ADT/data structure learned in CS2040 to implement *RabbitGiveAway*. [7 marks]

Ans:

The problem can be solved using multiple different data structures. One compact solution is to use a queue. Put *m* numbers into a queue. Read *n*-1 numbers from the front of the queue and add them to the back; read one more number from the front, print and discard. Continue until the queue is reduced to size *m*/2.

Example:

```
create a queue Q
for i = 0, i < m, i++
  Q.enqueue(i)
for i = 0, i < m, i++
  for j = 0, j < n-1, j++
    Q.enqueue(Q.dequeue)
  print(Q.dequeue)
```

The outer for-loop takes O(m) and the inner for-loop takes O(n), so overall it is O(m*n). (Of course you can also call the functions Q.offer() and Q.poll()).

The problem can equally efficiently be solved with a doubly linked circular list and also with a singly-linked circular list. However, to remove a node from a singly linked list we have to be careful and stop at the node *before* the one that needs to be deleted. Otherwise, the deletion is not efficiently possible if the node to be deleted is the current node (because we have no pointer to the previous node in a singly-linked list).

Grading Scheme:
Note: minor typos (e.g.: mixing up n and m) are not penalized and the below scheme applies.

7m: A solution algorithm that correctly uses either a queue, a doubly linked or singly-linked circular list correctly and runs in O(m*n).

6m: A solution algorithm that uses a singly-linked list, but it is unclear if the deletion of nodes is efficiently supported. So the computation time takes > O(m*n).

5m: A solution algorithm that uses various data structures in a way where the complexity is clearly > O(m*n). For example, data structures where removed rabbits are only marked as deleted, but are not actually removed in the data structure. This results in the access of the next rabbit to become longer and longer, i.e., > O(n). Examples of such data structures are circular arrays and some linked list implementations. In some cases, deleted rabbits are removed, but the removal takes time (e.g., closing the gap if a rabbit is removed in a circular array). Some hashmap, hashset and hashtable solutions also have this problem, i.e., removed rabbits are only marked, not actually removed.

4m, 3m: Various errors in the code that make it not work correctly or not understandable, or missing details in the algorithm. For example, a variable is used but it is unclear what values it may take or from which value it starts, etc. Also, if the loop through the list of rabbits stops after one pass.

2m: The description or solution only states the data structure and, for example, that there is a loop. Incomplete, and missing details.

1m: The description or solution only states the data structure (e.g., “use circular linked list”). Incomplete, and missing details.

0m: No description or solution provided. Or the provided description cannot really be matched to the problem statement.

Questions 33 to 37 refers to the problem given below

33. You have an unsorted array A of size N containing integers with values from the set S = {w,x,y,z} where w < x < y < z. There is at least 1 integer of each value in A.

    An example array A is [1,10,1,13,13,10,1,15,1] where w = 1, x = 10, y = 13 and z = 15 and S = {1,10,13,15}

    You have run a sorting algorithm on A to sort it in non-decreasing order but the program crashed before the sorting finished.

    Good thing is that for each step of your sorting algorithm you have saved the partially sorted array and some information on how much of the array is sorted.

    After you rebuild A from the latest saved data, you gather from the information saved that **only integers of value w have been fully sorted**.

    To make sure you have understood the question, is the following a possibly valid A rebuilt from the saved data based on the problem description above if S = {1,10,13,15}

    [1,1,1,13,1,10,10,15,10]

    [2 marks]

a. True
b. False

34. Assume you have the following functions/methods:

    linearSearch(A,i,j,k) - perform a linear search of array A from index i to index j for the search key k and return the index of k the first time it is found. If k is not found, return -1

    binarySearch(A,i,j,k) - perform a binary search of array A from index i to index j for the search key k and return the index of k the first time it is found. If k is not found, return -1.

    In order to speed up the rest of the sorting of A, you want to quickly identify the starting index **t** of the unsorted portion so that you can start the sorting from there.

    The algorithm to identify index **t** consists of 2 steps as follows:

    Step 1: identify starting index i and ending index j of A to search for index **t**

Step 2: search for index **t** between index i and j of A.

Choose the best option for each step such that the algorithm will be correct and as time efficient as possible: [2 marks]

Select the best option for **Step 1**

a. **i = 0**
j = N-1

b. i = 0
j = N/2

c. i = N/2
j = N-1

d. i = binarySearch(A,0,N-1,x)
j = N-1

e. i = 0
j = binarySearch(A,0,N-1,y)

f. i = 0,
Compute j using the following algorithm:

```
j = 1
while (true)
  if (A[j] == w)
    j = j*2
  else
    break
  if (j >= N-1)
    j = N-1
    break
```

g. i = j/2 where j is computed as follows:

```
j = 1
while (true)
  if (A[j] == w)
    j = j*2
  else
    break
  if (j >= N-1)
    j = N-1
    break
```

Ans:

Options d & e are out since A is not fully sorted so you cannot use binary search.

Option b is out since there is no guarantee the sorted portion is <= half the size of A.

Option c is out since again there is no guarantee the sorted portion is >= half the size of A.

Option f is not correct because you need to check if (j >= N-1) before you can check if (A[j] == w) otherwise you get index out of bounds when j is too large.

Option g is not correct because of the same problem as option f.

35. Select the best option for **Step 2** // how to search from index i to j of A to find index t

    [2 marks]

a. return linearSearch(A,i,j,x)
b. return linearSearch(A,i,j,y)
c. return linearSearch(A,i,j,z)
d. return binarySearch(A,i,j,x)
e. return binarySearch(A,i,j,y)
f. return binarySearch(A,i,j,z)
g. return minimum(linearSearch(A,i,j,x), linearSearch(A,i,j,y), linearSearch(A,i,j,z))
h. return minimum(binarySearch(A,i,j,x), binarySearch(A,i,j,y), binarySearch(A,i,j,z))

Ans:

Option d, e, f and h are out because binary search cannot be used since A is not guaranteed to be fully sorted.

Option a, b, c is out since you cannot be sure the 1st value in the unsorted region is x or y or z.

36. Your chosen algorithm will run in time [1 mark]

a. O(1)
b. O(N)

c. O(logN)

Note that you will only be given marks if a linear time algo is selected in 35. If algo that is not linear is selected and correct time complexity is given for the wrong algo, no marks are given.

37. Assume you have found the index t.

Write the most efficient algorithm you can think of to implement *SortA(A,N,t,S)*, which will take in the array A, the size N of A and the index t to sort the unsorted portion of A from index t to index N-1.
S is a sorted array of size 4 containing the four possible integer values in A.

RESTRICTION: **You can only use only swaps if needed to exchange the positions of the items in the unsorted portion in order to sort them** and you can only use an **extra O(1) space**.

State the time complexity of your algorithm.

[7 marks]

Ans:

```
// use the idea of quick sort's partitioning algorithm, but instead of a 2 way partition, make it a 3 way
// partition, since there are only 3 distinct values in the unsorted region.

1. Let mid = S[2] // the middle value in the unsorted region
2. Let i,j = t-1  // i is the rightmost boundary of the partition that contains the smallest value
                  // j is the rightmost boundary of the partition that contains the middle value
                  // initially all 3 partitions are empty and everything from t to N-1 is in the
                  // unprocessed region, so i and j is initialized to t-1
3. for (int k = t; k <= N-1; k++)
     if A[k] < mid  // perform 2 swap to get the smallest into the correct region
       i++
       swap A[k] and A[i]
       j++
       swap A[k] and A[j]
     else if A[k] == mid  // perform 1 swap to get the middle value into the correct region
       j++
       swap A[k] and A[j]
     else // just increment k when A[k] > mid, i.e the largest value
       do nothing
```

Time taken is O(N) since you only need to iterate through the unsorted region(which is at most O(N) in size) and perform a constant number of operations to put each integer in the unsorted region into the correct partition.

Grading scheme:

Some notes

- We assume quick sort to be O(N log N)
- We assume quick select to be O(N)
- We assume hash table supports all operations in O(1)
- We assume insertion sort can be implemented using swapping (easy to modify from what we've learned; many examples of insertion sort also do this)
- We assume merge sort to be NOT in-place (it is possible to make in-place merge sort, but the change from what we've learned is not trivial....)

Starting Marks

- 0m if the answer is vague
    - Note: include answers which only say "Use <some sort algorithm>" without specifying what array it should sort (as we are given two arrays, A and S). Need to specify the array A, or at least mention the following:
        - index t
        - unsorted part/region
- 7m otherwise

Major Mistakes

- Time complexity
    - -2m if the answer is O(N log N)
    - -3m if the answer is >= O(N^2)
- Space
    - -4m if the answer requires writing (not only swapping)
    - -3m if the answer requires >= O(1) space
    - -5m if both happen

Minor Mistakes

- -1m for each minor mistake such as
    - off-by-one indexing
    - does not use index t as starting index
    - if the answer is not in-place and forgot to set the values in A
    - other easy-to-fix that is not mentioned here
- -2m/-3m if the answer includes the intended idea but the details are wrong, depending on the severity.
- Capped to -3 for all mistakes here

Final Marks

- If the answer correctly sorts the unsorted part, the answer will get at least 1 mark.
- Otherwise, 0.

Common Mistakes

- Not writing what array to be worked on (this is the minimal indication that the student understands what the question is asking).
- Implementing sort on their own, without using swap (e.g some implementations of insertion sort).
- Using binary search on the unsorted region.
- Other common mistakes are already mentioned in the -1m marks in the grading scheme.