CS2040S 2021/2022 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. Searching for an element in a doubly linked list (as given in lecture notes) of size **N** will require worst case time complexity

a) O(1)
b) O(log n)
[✓ c) O(n)]
d) O(n log n)

worst case is when the value is in the middle of the linked list which has a better constant 0.5n but it is still O(n)

2. Haftek has a box of apples, each having a distinct weight. Haftek wants to sort the apples, but he does not have a digital weighing scale toget the exact weights. Thus, he can only use his hands to compare two apples at a time to know which of the two apples is heavier. Which of the following algorithms can be used by Haftek?

a) Insertion sort
b) Merge sort
c) Quick sort
[✓ d) All of the above]

3. Consider a hash table using separate chaining for collision resolution, with table size of 10 (keys 0 to 9) and the hash function *h(x) = (3x + 1) % 7*. After inserting the keys 4, 11, 18, 25, 32, 39 into the hash table, the length of the linked list for key 6 is

a) 4
[✓ b) 5]
[✓ c) 6]
d) Insufficient information to determine

since they all hash to key 6 all of them will join that list and the length will be 6

All elements inserted hash to the same value 6. This question tests if students can simulate hash table insertion with separate chaining. A bonus shortcut would be if students are able to recognise that the inserted keys have successive differences of 7, which is the table size.

4. Consider a hash table using linear probing for collision resolution, with table size of 11 (keys 0 to 10) and the hash function *h(x) = x % 11*. The total number of probes for inserting the keys 3, 7, 13, 15, 2, 29 is

a) 9

the first probe is counted as 1 probe

[✓ b) 10]

c) 11

d) 12

5. Oizne Mak wants to design a queue which has a maximum capacity of **N**. This queue supports the operations *dequeue* and *peek* which should work like the standard queue ADT. The *enqueue* operation is slightly different: when there are less than **N** elements in the queue it is the same as the standard queue ADT's *enqueue*, but should do nothing when the queue has **N** elements. Oizne Mak wants all operations to be done in **O(1)** time in the worst case.

Which of the following data structures can Oizne Mak use? The options are independent, eg. picking (i) and (iii) means that it can be done with either a linked list alone, or a tailed linked list alone.

i. Linked list
ii. Doubly linked list (without tail reference)
iii. Tailed linked list
iv. Array

a) (i) only

b) (ii) and (iii)

[✓ c) (iii) and (iv)]

d) (ii), (iii) and (iv)

6. An array is **k**-*sorted* if each element is at most **k** positions away from its sorted position. Which of the following sorting algorithms (as given in the lecture notes) can sort a **k**-*sorted* array of size **N** in **O(kN)** time?

i. Insertion sort
ii. Optimised bubble sort
iii. Selection sort

For insertion sort, each element will move at most k times (consecutive swaps) within the algorithm. This is the same for Optimised bubble sort, which will run for k iterations, followed by the (k + 1) "check" iteration. (In fact, if you know it is k-sorted there is no need for the (k + 1) iteration).

[✓ a) (i) and (ii)]

[✓ b) (i) and (iii)]

c) (i), (ii), and (iii)

d) None of the algorithms will run in *O(kN)* time.

7. Which of the following statements is false?

- a) We can reverse a queue in *O(N)* time with an additional stack only.
- b) We can reverse a queue in *O(N)* time with an additional queue only.
- c) We can reverse a queue in *O(N^2)* time with an additional queue only.
- d) We can reverse a queue in average *O(N)* time with an additional hash table only.

~~8. A bloom filter of size 11 is created with the following 4 hash functions:~~ qn is voided

$h_1(x) = (2x - 1) \% 11$

$h_2(x) = (2x + 1) \% 11$

$h_3(x) = (3x - 1) \% 11$

$h_4(x) = (3x + 1) \% 11$

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

The diagram above shows the state of the bloom filter after the keys 2 and 25 are inserted (top row is the index of the bloom filter). Which of the following keys will return a negative result (i.e. return false)?

- a) 10
- b) 17
- c) 29
- d) 33

9. Given the following function/method *foo*

```
function foo(L, i, j) {

    // If the left-most element i is larger than the right-
    // most element j

    if L[i] > L[j] {
        swap(L, i, j)
    }

    // If there are at least 3 elements in the array

    if (j - i + 1) > 2 {
        t = floor((j - i + 1) / 3)
        foo(L, i  , j-t)    // Recurse on the first 2/3 of L
        foo(L, i+t, j)      // Recurse on the last 2/3 of L
        foo(L, i  , j-t)    // Recurse on the first 2/3 of L

    }
    return L
}
```

Assume *L* is a 0-indexed array of size **N**.

Let the starting values of *i* and *j* be 0 and **N**-1 respectively.

What is the worst case time complexity of the algorithm *foo*?

[✓ a) $O(N^{(\log_{3/2} 3)})$]

b) $O(N^{(\log_{3} 3)})$

c) $O(N^{(\log_{2} 3)})$

d) None of the above

For this question, there is no need to know what the algorithm is doing. We can just directly analyse the time complexity of the algorithm. You can draw the recursive tree and show that the height of the tree will be h = log3/2 n .
Essentially, we need to solve the recurrence relation T(n) = 3 * T(2n/3) + c. For simplicity we will set c = 1. T(n) = 3 * T(2n/3) + 1
= 3 * [3 * T((2n/3)2n) + 1] + 1
=3^2 *T((2n/3)2n)+3+1
=...
=3^log3/2 +...+3^2 +3^1 +3^0
= (1 - 3^(log3/2 n)+1) / (1 - 3) using GP formula

10. Given the same function/method *foo* as in Q9

Assume *L* is a 0-indexed array of size **N**.

Let the starting values of *i* and *j* be 0 and **N**-1 respectively.

Which of the following statement is true?

a) The algorithm *foo* correctly sorts the first 1/3 of the array.

b) The algorithm *foo* correctly sorts the first 2/3 of the array.

[✓ c) The algorithm *foo* correctly sorts the entire array.]

d) The algorithm *foo* does not sort any part of the array.

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. Given the best case input of size **N** for each of the following sorting algorithms:

    a) Selection Sort
    b) Optimized Bubble Sort
    c) Insertion Sort
    d) Merge Sort
    e) Quick Sort

    For optimized bubble sort, the total comparisons for a best case sorted input is O(N). since it takes one- pass to determine the input is already sorted and N-1 pairs of adjacent items are compared.
    For Insertion sort for a best case sorted input does the same number of comparisons as bubble sort, i.e O(N) comparisons. This is because each iteration of the outer for loop, the inner for loop (where the comparison is made) will only run once, since the current value to be sorted will be compared to the back of the sorted region and it will be found to be already in the correct position.

    The sorting algorithm that does the least amount of comparison in terms of big-O to perform the sorting is only b) Optimized Bubble Sort.

    false. merge sort does lesser

12. John has constructed his own hash table of size **M** that uses separate chaining as a collision resolution technique and the division method (% method) as the hash function.

    However he has modified the separate chaining technique so that instead of using a linked list, each entry in the hash table uses another hash table. Each sub-hash table is of size **M'**, and uses linear probing as collision resolution.

    Co-prime numbers are pairs of numbers that do not have any common factor other than 1

    John is quite confident that for any prime number **M** and and any **M'** that is co-prime to **M** and **M'** < **M**, if he does not need to resize the main table or any of the subtables, the time complexity to insert a pair of integer (assuming bounded number of digits per integer) key value pairs into the hash table is worst case **O(1)**.

    false, he must first look through hash table of size M to check if the key is available, then he has to look through its linked list of M' to insert the key. At worst case the key is located at the M' th pos of the sub of the Mth key which is O(M + M')

    Given N input keys where the key values are a multiple of M*M', all of them will hash to index 0 of the 1st sub-hash table (at index 0 of the main hash table) and each insertion will take O(n) time where n is the current size of the sub-hash table (since you need to move down n rows before finding an empty slot to insert the new key). Thus worst case is not O(1) time

13. Given the following algorithm *DoSomething*

```
DoSomething(int n, int i, int s) {
  if (n/i <= 1)
    return
  else {
    if (s == 1) {         N
      for (int j=0; j < n/i; j++) {
        System.out.println("DoSomething1");
      }
      DoSomething(n, i+1, 2);  N, 2, 2
    }
    else
      for (int j=1; j < n/i; j*2) {
        System.out.println("DoSomething2");
      }
      DoSomething(n,i+1, 1);
  }
}
```

Calling *DoSomething*(N,1,1) for some N will run in worst case time complexity **O(N^2)**.

True

when i is even loop 1 runs O(n)
when i is odd loop 2 runs O(log n)

recursive fcn called each time until i = n O(n)

in worst case loop 1 and 2 runs n/2 times each =
n * n/2 + log n * n/2 = O(N^2)

False.
There are N+1 recursive calls, since it hits base case when i > N and i is incremented by 1 for each recursive call.
Each recursive call before the base case keeps switching between the if and the else clause, thus for half of the recursive calls the if clause is executed (odd values of i) and for the other half, the else clause is executed (even values of i).
Let M be total number of iterations of the for loop where the if clause is executed. Let M' total number of iterations of the for loop where the else clause is executed.
Now M > M' since in the if clause the loop variable of the for loop increments by 1 while in the else clause the loop variable of the for loop increments by doubling.
Thus total iterations of for loop for all recursive calls < 2*M < O(M)
The number of iterations of the for loop then goes as follows for each successive recursive call that activates the if clause (ignoring base case):
N/1,N/3,N/5, ... N/N (assuming N is odd)
Thus, total iterations = N*(1+1/3...+1/N) < N*(1+1/2+1/3+1/4+...+1/N) < N*O(logN) = O(NlogN) since 1,1/2,1/3,1/4 .... 1/N is a harmonic series and the sum of a harmonic series with N terms is O(logN). This a tight bound since 1+1/3+1/5...1/N > 1/2+1/4+1/6...1/(N-1)
So, 1+1/2+1/3+1/4+...+1/N cannot be more than 2*(1+1/3+1/5...+1/N)

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. **[11 marks]** Given an arraylist *A* containing **N** (**N** ≥ 1) unsorted arraylists each containing up to **M** (**M** ≥ 1) possibly repeated integer values, give an algorithm which takes in *A* and print out all integers that are common to all the **N** arraylists in ascending order. The integers can take values from 1 to 10,000,000, and each list may contain repeated integer values.

    You must ensure the time complexity of your algorithm is **O(M*N)** in the worst case.

    For example, if *A* contains 3 unsorted lists with the following content

    10, 13, 30, 100, 2, 5, 11, 2
    2, 11, 13, 5, 13
    13, 3, 2, 1, 7, 3, 4, 10, 11, 100, 30

    the output from your algorithm should be

    2,11,13

15. **[11 marks]** You are given 2 input strings *A* and *B*, each of length **N**. You are supposed to check if *A* and *B* are anagrams of each other.

    2 strings are anagrams of each other if they consist of the same characters but not neccessarily in the same order. For example, "eat" and "tea" are anagrams of each other, but "idea" and "adeer" are not.

    This problem can be solved using a hash table... However, the setter of the question is evil and has restricted you to use up to 2 stacks that can only contain characters and no other ADT/data structure to solve the problem. You cannot modify the input strings *A* and *B* in any way.

    You must solve the problem (output true if *A* and *B* are anagrams and false otherwise) in at most **O(N^2)** worst case time.

Both Question 16 & 17 are related to the problem description below

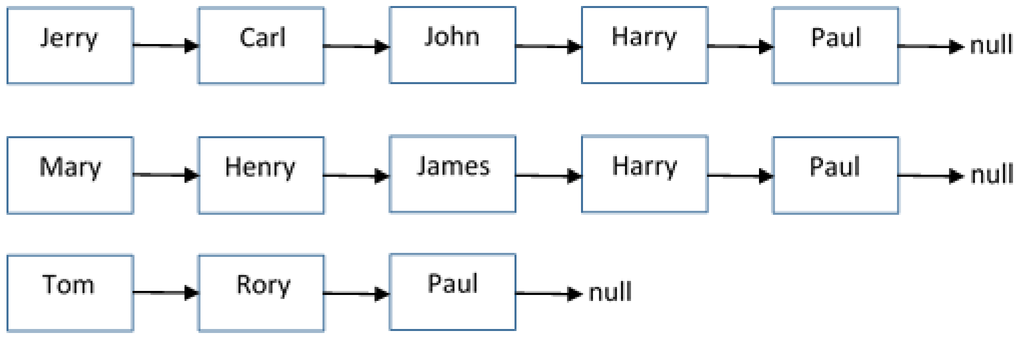
A group of **M** persons are found to be related to each other through the patriarchal line of their family (tracing back on the male ancestors). It is for certain that all their furthest traceable ancestor is the same person but there can be branchings down the line from that ancestor.

Now each of them has created a singly linked list of their patriarchal line with them at the head and their furthest traceable ancestor at the tail. Only the names are stored in each node. Each linked list is of size $\mathbf{N_1}$ to $\mathbf{N_m}$ and the largest among them is $\mathbf{N_{max}}$.

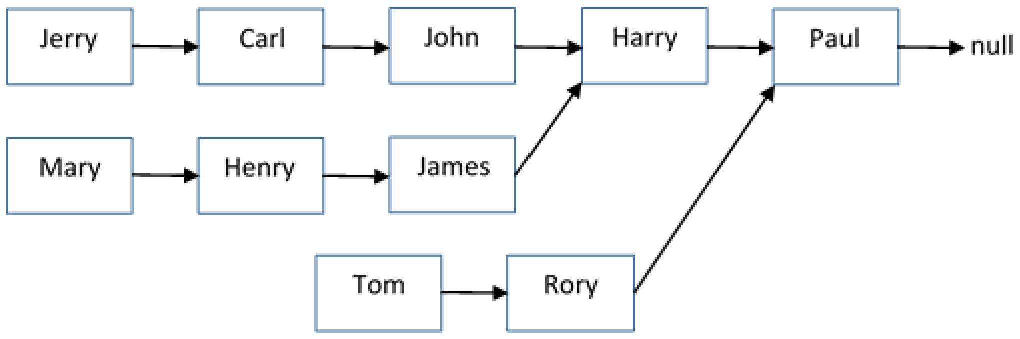
Trying to figure out how they are related is now quite difficult since there are **M** linked list. One of them in the group then suggested that they merge the linked list to form a family tree, where there should be only 1 node in the family tree for any common ancestor shared among them. Then it would be easier to check how they are related.

For simplicity sake all the names of the people involved in the family tree are unique and no more than 20 characters long, thus if some node in 2 different linked list share the same name that would be a common ancestor between the patriarchal lines.

For example, given **M**=3 and the following singly linked lists for each person:



A way to merge them into a family tree would be as follows:



Now your problem is given an array *A* of **M** references to the head node of the **M** linked list, give an algorithm to merge the linked lists into a family tree in $\mathbf{O(M*N_{max})}$ average case time or better. You can create and delete nodes as neccessary to form the family tree at the end.

16. **[10 marks]** For Q16, solve the problem for **M** = 2

    (you can solve for Q17 and use that answer here if you are confident it is correct. The suggestion is to do Q16 first then Q17)

17. **[8 marks]** For Q17, solve the problem for any value of **M** ≥ 2

Both Question 18 & 19 are related to the problem description below

In the "programming" game of “wolves and sheep”, you are given an array *B* of size **N** (**N** > 10) containing items that represent “wolves” and “sheep”.

An item will have 2 attributes

- type –> type = 1 is a sheep while type = -1 is a wolf
- id –> a unique positive integer value bigger than 0, identifying the animal

Now *B* is split up into **k** (2 <= **k** < **N**) contiguous subarrays B'_1 to $B'_{\mathbf{k}}$ and each subarray B'_i can only contain either sheep or wolves but not both. The leftmost subarray will always start as a sheep array (can only contain sheep) and then the subarrays will alternate between sheep and wolves from left to right.

Each of the **k** subarray is identified by a pair <L,R> where L is the index of the leftmost boundary and R is the index of the rightmost boundary of the subarray. These pair information are stored in an array *SA* from leftmost to right most subarray.

You are guaranteed that the number of sheep in the *B* is equal to the sum of the size of all the sheep subarrays, and similarly for the wolves.

Now at the start of the game, the wolves and sheep are all jumbled up in *B* and so they may be in the wrong subarray.

An example of a jumbled up *B* of size **N** = 13 with **k** = 4 and *SA* = {<0,2>,<3,7>,<8,9>,<10,12>} is shown below,

43	3	50	27	51	85	1	33	31	90	151	113	70
-1	1	-1	1	-1	1	1	-1	-1	-1	-1	-1	1

For each entry, top value is the id and bottom underlined value is the type.

Subarrays are color coded to show the animal it should contain (yellow is sheep, red is wolf).

You can easily see that there are sheep and wolves that are in the wrong subarray.

Now you will be given *B*, **N**, **k** and *SA*, and the point of the game is to write an algorithm to move the animals so that each subarray contains the correct type of animal. <u>Your algorithm must run in worst case **O(N)** time.</u>

A valid *B* at the end of your algorithm (among possibly many others) for the example given is as shown,

27	3	1	43	51	31	50	33	85	70	151	113	90
1	1	1	-1	-1	-1	-1	-1	1	1	-1	-1	-1

18. **[10 marks]** For Q18, solve the problem where you can use up to **O(N)** additional space for your algorithm.

    (you can solve for Q19 and use that answer here if you are confident it is correct. The suggestion is to do Q18 first then Q19)

19. **[8 marks]** For Q19, solve the problem where you can only use additional **O(1)** space for your algorithm.