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)*

Searching in any linked list is in O(n) since there is no random access. This question tests if students have any confusion between arrays, sorted arrays and linked lists.

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

All the three sorting algorithms are comparison sorts. This question tests if students understand how the sorting algorithms work. (An important consequence of this is the lower bound of Ω(n log n) for any comparison sort.)

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

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
b) 10
c) 11
d) 12

Similar to Q3, this question tests if students can simulate hash table insertion but instead with linear probing.

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)

Tailed linked list can implement the queue with capacity simply by maintaining a size variable, and not allow any insertion to the tail once at max capacity

Array can implement the queue with capacity by initialising an array of size C, and maintaining a "front" and "end" variable to make it a circular array.

(Doubly) Linked lists cannot do both enqueue and dequeue without the tail reference.

This questions tests if students are familiar with the structure and operations of the array and linked list data structures, and whether they can work with restrictions / assumptions that are different from the norm.

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

- Insertion sort
- Optimised bubble sort
- Selection sort

a) (i) and (ii)
b) (i) and (iii)
c) (i), (ii), and (iii)
d) None of the algorithms will run in *O(kN)* time.

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).

This question tests if students understand the underlying mechanisms of how the sorting algorithms work. For students who have difficulty with this problem, you are advised to revisit the invariants of each algorithm, i.e. how they "grow" the sorted area during the algorithm.

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.

To reverse a queue, we need to find some way to "extract" the last element. With only an additional queue, we need to cycle through the original queue (dequeue then enqueue again) from the first element to the last element, before we can insert into the additional queue. Repeating this for all elements, the number of operations is $2*[N + (N - 1) + \ldots + 2 + 1] = O(N^2)$.

The stack solution is straightforward since we can get a "natural" reversal of the order.

For the hash table solution, we can insert (index, element) as the key-value pairs while removing each element from the queue, then reinsert elements in the queue based on the reversed index order.

This questions tests if students are familiar with the operations and properties of the various data structures, and the time complexity of the operations.

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

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

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

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

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

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

This questions tests if students understand how a basic bloom filter works, and whether they are able to simulate the bloom filter insertion and search query.

*Due to the 0's and 1's being flipped, this questions have been voided, and students given 3 marks.

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 $h = \log_{3/2} n$.

Essentially, we need to solve the recurrence relation $T(n) = 3 * T(2n/3) + c$. For simplicity we will set $c = 1$.

$T(n) = 3 * T(2n/3) + 1$

$= 3 * [3 * T((2n/3)^2 n) + 1] + 1$

$= 3^2 * T((2n/3)^2 n) + 3 + 1$

$= \ldots$

$= 3^{\log_{3/2} n} + \ldots + 3^2 + 3^1 + 3^0$

$= (1 - 3^{(\log_{3/2} n)+1}) / (1 - 3)$ using GP formula

$= (3^{(\log_{3/2} n)+1} - 1) / 2$

$= O(3^{\log_{3/2} n})$

$= O(n^{\log_{3/2} 3})$ by property of logarithm

This question is more challenging, and tests if students can analyse algorithms with non-standard time complexity.

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.

The algorithm is in fact a sorting algorithm called stooge sort.

We can prove that this algorithm is correct by using an inductive proof. Here is a rough sketch:

1. Prove that it works for arrays of size 1 and 2.
2. The inductive hypothesis is that the algorithm works (i.e. sorts) for arrays of length < n. (We need to prove that it also works for arrays of length n)
3. Split the array into 3 parts: [0, t-1], [t, 2t-1], [2t, n-1] where t = floor(n / 3).
4. Apply the algorithm to first two parts: By the inductive hypothesis, the elements in [0, 2t-1] are now sorted. Observe that the largest of the first 2t elements will all be in range [t, 2t – 1].
5. Apply the algorithm to the second and third parts: By the inductive hypothesis, the elements in [t, n-1] are now sorted. Observe that the largest of all the elements in the array are in range [2t, n-1], in sorted order.
6. Apply the algorithm to first two parts: By the inductive hypothesis, the elements in [0, 2t-1] are now sorted.
7. From 5 and 6, all elements in [0, n – 1] are in sorted order since all elements in [0, 2t – 1] are smaller than elements in [2t, n – 1].
8. By strong induction, the algorithm correctly sorts the array.

This question is more challenging, and tests if students can determine what an algorithm is doing based on its description.

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

    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.

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.

Grading Scheme:

2 marks for correct answer
1 mark for correctly identifying insertion sort
1 mark for correct explanation/example of why insertion sort does O(N) comparisons

Notes:
No marks are awarded for the last point if student is explaining the time complexity instead of the number of comparisons. Eg. run in O(N), O(N) time, takes O(N) time
No marks are awarded for explanation if student is comparing worst-case inputs instead of best-case inputs.
Marks were not deducted for students who say n comparisons instead of n - 1 comparisons.

Benefit of the doubt is given to students who only said O(N) (without implying or mentioning time) instead of O(N) comparisons and no marks are deducted. Eg. O(N) complexity
Students should take note to be careful and make sure to not omit any details, especially so when writing explanations and/or algorithms.

Common mistakes:
Omitting explanation / example of why insertion sort has O(N) comparisons on the best-case input.
Comparing the complexity of algorithms instead of asymptotic number of comparisons or using time complexity as the conclusion while talking about comparisons.

12. John has constructed his own hashtable 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 hashtable uses another hashtable. Each sub-hashtable is of size **M'**, and uses linear probing as collision resolution.

    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 hastable is <u>worst case</u> **O(1)**.

False.

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-hashtable (at index 0 of the main hashtable) and each insertion will take O(n) time where n is the current size of the sub-hashtable (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.

Grading Scheme:
2 marks for correct answer
1 mark for valid explanation / example of why collisions still occur (in both the main and sub-table), eg. multiples of M*M', same value inserted, values inserted all < M', pigeonhole principle
1 mark for explaining why it is not O(1) worst-case, eg. linear probing / size of the linear probing run / number of probes / primary clustering, same as open addressing, using the example to explain

Notes:
No marks are awarded for the last point if resizing is used as the reason why the worst-case is not O(1) time.
Explanations that explain the worst-case to be O(M') are also accepted.

Common mistakes:
Only mentioning, but no explanation for why collisions occur for the sub-hashtable.
Assuming worst-case that a sequence of keys will map to the same slot in big hashtable and small hashtable, but no explanation/example on what this sequence is.
Talking about O(n) insertion on almost full table. But this does not explain why there is collisions on the insertion or in previous insertions that resulted in the intended insertion slot being occupied.
Talking about resizing / inserting on full table, which is not answering the question.

13. Given the following algorithm *DoSomething*

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

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)

Grading Scheme:
2 marks for correct answer
1 mark for identifying bottleneck in *if* clause (accept if try to formulate time complexity of both clauses)
1 mark for correctly explaining why O(N^2) is wrong (i.e. calculate time complexity)

Notes:
We will also accept answers that state that there is an infinite loop (due to the mistake in the else clause loop variable *j* not updating), which will get the full 2 marks for the explanation.
Some students treat both the if and else clause as the same in terms of the calculation and do an upper bound of the number of operations. These solutions are accepted if there is sufficient working / explanation to show that it is being upper bounded, using the individual terms or the sum. Otherwise, the 1 mark is not awarded for the explanation of the bottleneck in the *if* clause.
Some students used equality (=) instead of inequality (<=) in their working. No marks were deducted.
Some students just wrote the terms out without any further explanation of how they are obtained. We gave the benefit of the doubt this time.
Students are reminded to be very careful with mathematical expressions when writing your answers, and not to assume unproven results.
A number of students calculated the sum separately for the two clauses and made mistakes in the calculation for the else clause, but would still get the correct conclusion of O(N log N). The 1 mark for the last point is not awarded in this case.

Common Mistakes:
Assuming each step in the recursion has the same amount of work done, then doing O(N) * N.
Not calculating the actual time complexity of the function.
Wrong terms formulated for each recursive step.
Wrong steps in calculating the time complexity, or no steps at all.
Assuming that half O(N) + half O(log n) = O(N log N).
Incomplete steps before concluding that it is O(N log N), or just stating that the sum is not O(N^2).
Misuse of harmonic series identity.
Calculating the sum for the else clause wrongly.

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

1. Create a DAT B of size 10000001 (assuming 0-indexing so we will sacrifice index 0), initialized to 0.
2. for (i = 0 to N-1) ←O(N) time
    let L = A[i]
    let C be a boolean array of size 10000001 initialized to false // ensure each integer is counted once
                                                                    // in the current arraylist
    for (j = 0 to size of L) ← O(M) time
      if (C[L[j]] == false)
        B[L[j]]++
        C[L] = true
3. Go through B print out every index k where B[k] == N

total time is O(N)*O(M) = O(M*N)

Grading Scheme:

Correct solutions:
1. O(M*N) solution ← 11 marks
2. O(M*N*logM) solution ← 8 marks
3. $O(M^2*N)$ or $O(M*N^{-2})$ solution ← 5 marks
4. $O(M^2*N^2)$ solution ← 3 marks

Deduction for errors in correct solution:
1. Printing out unique integers (never check for occurrence) ← -5
2. As long as integers are common to 2 arraylists and not all N arraylists, it will be printed out. ← -4 marks
3. count multiple occurrences of a common integer in the same arraylist (should only count once from the same array list) ← -3 marks

4. Other minor errors ← -1 mark

wrong solutions:
1. printing out unique integers in all the arraylists regardless of their occurrence: 3 marks
2. other wrong solutions: 1 or 2 marks (depending on how wrong)

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 hashtable... However, the setter of the question is evil and has restricted you to use up to 2 stacks 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.

```
Let A and B be the 2 input strings, and S1 and S2 be the 2 stacks
for i = 0 to N-1
   S1.push(A[i])
count = 0
while (true)
   while (S1 is not empty) // repeated pop from S1 to S2 until top of S1 matches B[count]
      if (B[count] == S1.peek()) // if match found, move on to next character in B, discarding top of S1
         S1.pop()
         count++
      else
         S2.push(S1.pop())
      S1 = S2 //change reference or you can just pop back everything into S1 from S2.
              // Doesn't change time complexity
      if (S1 is empty and count != N) // character at B[count] cannot be matched to any character in A
         return false
      else if (S1 is empty and count == N) // each character in B is matched to 1 character in A
         return true
      // Otherwise, you continue the outer while loop
```

The worst case input will be when the 2 strings are anagrams of each other, but the character that matches is always at the bottom of the stack thus in the 1st iteration of the outer while loop in step 5, count is incremented only by 1 and the inner while loop will run N times, in the 2nd iteration again count is incremented by 1 and the inner while loop will run N-1 times and so on ... so we have N+(N-1)+(N-2) ... = O(N^2)

Grading Scheme:
7 marks for correctness

- 7 marks for correct algorithm
- 5-6 marks for minor flaws in algorithm
- 3-4 marks for major flaws in algorithm
- 0-2 marks for incorrect algorithm

5 marks for efficient solution

- 4 marks for O(N^2) algorithm
- 3 marks for any other algorithm better than or in $O(N^3)$
- 2 marks for any other algorithm worse than $O(N^3)$

(Marks are awarded for efficiency only if student has scored at least 3 marks for correctness)

Notes:
For solutions that violate the 2 stack restriction (more than 2 stack, including implicit recursive stack), 2 marks are deducted from the overall score.
For solutions that modify the input / violate stack's character only restriction / no other ADT or data structures restriction, a penalty of 50% (round down) of the original marks awarded will be applied.

Common Mistakes:
Just essentially checking that each character is the same at each index.
Loading both strings into s1 and s2 respectively. Then check top of s1 to every character in s2, and if found "reset" with all characters in B and repeat. This does not work because the algorithm is either incorrect, or you need to store all characters seen before, violating the restriction. Eg. A = cabc, B = dabc
Algorithms that assume / do not consider if the two strings have unequal number of repeated characters.
Failed attempts to sort the characters of each string using the stacks, or not explaining how to get characters in sorted order using the stacks.
Missing the transfer / reset of second stack or the swapping of S1 and S2 references.
Almost correct insertion sorts but forgot to change loop conditions for the second string.
Correct insertion sorts that violate the ADT restrictions.

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

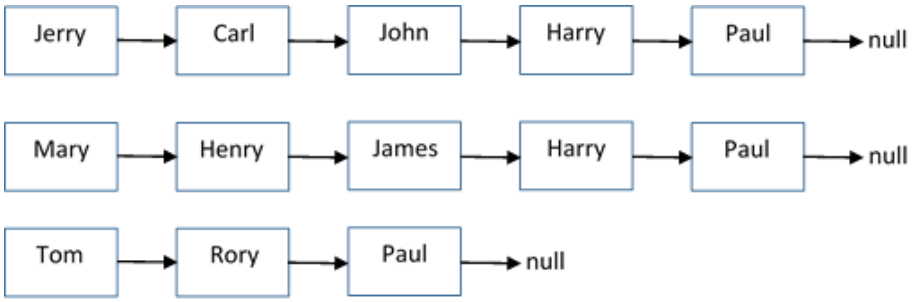
A group of **M** persons are found to be related to each other through the patriarchal line of their family (tracing back on the male ancestors). It is for certain that all their furthest traceable ancestor is the same person but there can be branchings down the line from that ancestor.

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 $\mathbf{N_{max}}$.

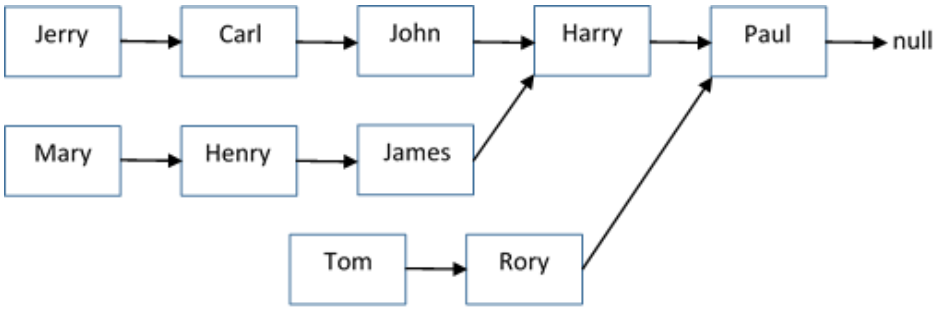
Trying to figure out how they are related is now quite difficult since there are **M** linked list. One of them in the group then suggested that they merge the linked list to form a family tree, then it would be easier to check how they are related.

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 necessary 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)

Time taken is O(N) since step 2 and 3 both takes O(N) time.

To merge 2 linked list into a family tree we have to start from the last node (the furthest common ancestor) of both linked list instead of from the head and move towards the first node. As long as the current node of both linked list have the same name they are a common ancestor and so we should only keep one of the common nodes and discard the other in the family tree. In order to start from the last node (the tail) make use of 2 stacks that will store references to each node in a linked list.

1. Let A and B be the 2 linked list, and let S1 and S2 be two stacks.
2. Go through A from head to tail and push reference to each node into S1. <-- O(N) time
   Do the same for B and S2 ←O(N) time
3. **while** (S1 or S2 is not empty) ← O(N) time
   if (S1.peek().name == S2.peek().name) // common ancestor
   retain = S1.pop() // let the one from A be the node to be retained in the family tree
   if (S1 is not empty) // point both descendent node in A and B to retained node
   S1.peek().next = retain
   if (S2 is not empty)
   S2.peek().next = retain

Time taken is O(N) since step 2 and 3 both takes O(N) time.

Grading Scheme:

Correct Solution:
1. $O(N_{max})$ time ← 10 marks
2. $O(N_{max}^2)$ time ← 6 marks

Deduction for errors in correct solution:
1. Forget about case where one linked list is sublist of the other list (i.e first node matches) ← -2 marks
2. If node1 and node2 matches, point node1.next to node2.next (this is not correct, you want to point the next of the previous node of node1 to node2). Applies to other variations of this error ← -3 marks
3. Other minor mistakes in reference pointing (broken family tree at the end) ← -1 marks each
4. Problems with traversal of the 2 linked list to check for matching nodes (e.g move through one list but not the other, or move through both at the same time is no match at the current node). ← -4 marks

Wrong Solutions
1. Other wrong solutions ← 1 or 2 marks depending on how wrong

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

A generalization of the algorithm to merge 2 linked list into a family tree from Q16. In this case, there are M stacks instead of two, and there is a need to efficiently track common nodes (common ancestors) at the top of the M stacks. This can be done using a hash table, and for common nodes, we can simply pick the 1st common node (i.e the leftmost common node) to be the one retained in the family tree and discard the rest.

1. Create an array B of M stacks.

2. for each list in A go through the list from head to tail and push reference to each node into the corresponding stack (meaning that nodes in list at A[i] will go into stack in B[i]).

3. Now have a hash table with the following <key,value> pair.
- key = name of person
- value = reference to left most node with name of person at the top of the M stacks

4. **while** (true)

```
    Let H be a hash table as described in step 3
    for (i from 0 to M-1) // average O(1) per insertion, so average O(M) for the loop
        if (B[i] is not empty) // have not exhausted the list
            if (B[i].peek().name not in H) //leftmost node for name since it's 1st time encountered
                H.insert(<B[i].peek().name,B[i].peek()>)
    for (i from 0 to M-1) // average O(1) per interation, so average O(M) for the loop
        node = B[i].pop(); //cur node at top of stack i
        leftnode = H.get(B[i].pop().name).value; // leftmost node with same name as cur node
        if (B[i] is not empty) // all children will point to only 1 common parent node
            B[i].peek().next = leftnode
        else
            if (node is not leftnode) // entire list is a sublist of another list, so discard it
                A[i] = null
    allempty = true
    for (i from 0 to M-1)
        if (B[i] is not empty)
            allempty = false
            break
    if (allempty == true) // all stacks are empty, so we are done
        break;
```

Another solution without stacks and only using a hash table but require to keep a reference to the previous node. The key to note is that once 2 nodes in different linked lists have matching names every node after them must also match, thus we just need to scan through all the linked list and hash nodes with names that are seen for the first time into a hashtable and make those the only node with those names. Don't even need stacks! :

```
1. create a hash table H where key = name, value = node that contains the name
2. for i from 0 to M-1
      prev = null
      cur = A[i]
      while (cur is not null)
        if H.get(cur.name) is null
           H.insert(cur.name,cur) // make cur the only node with this name
        else
           if (prev == null) // first node matches so it is sublist of another list and should be discarded
              A[i] = null
           else
              prev.next = H.get(cur.name)
        prev = cur
        cur = cur.next
```

Grading Scheme:

Correct Solution:
1. $O(M*N_{max})$ time ← 8 marks

2. $O(M*N_{max}^2)$ time ← 5 marks

3. $O(M^2* N_{max}^2)$ time ← 3 marks

Deduction for errors in correct solution:
same as for Q16 (so errors carry forward if you use solution for q17 for q16)

Wrong Solution:

1. Trying to do repeated pairwise merging or select 1 linked list (usually A[0] or the longest) and merge all other linked list with it (meaning that all other lists only try to match nodes with that selected list). Due to branching occurring as the lists gets merged, this can result in cases where common ancestors is in another branch other than the branch of the selected linked list. ← 2 marks

For example, given the following 3 linked list

Jerry->Carl->John->Harry->Paul->Ø

Mary->Henry->Harry->Paul->Ø

Tom->Henry->Harry->Paul->Ø

If the longest list (1st list) is used as the main list, then when 1st and 2nd list has been merged, it will be

Jerry->Carl->John->Harry->Paul->Ø
^
|
Mary->Henry----------

but now in the 3rd list, the common ancestors matched will only be Harry and Paul (since it will only checked against the branch formed by the longest list) but not Henry since Henry is in the branch formed by the merged 2nd list.

2. Other wrong solutions ← 1 to 2 marks depending on how wrong

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'_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. Your algorithm must run in worst case **O(N)** time.

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)

1. Iterate through B and count number of sheep and wolf. Let NS be the number of sheep and NW be the number of wolves. ←O(N) time
2. Create an array S of size NS and an array W of size NW. ←O(N) time, O(N) space for arrays
3. Iterate through B again, adding each sheep to S, and each wolf to W.
4. Iterate through each index pair in SA, alternating the animal type between sheep and wolf (starting with sheep). For each pair (left, right), we will an animal from S or W depending on the type to each index from left to right inclusive.

   For example, if we have the first pair (0, 2), we will take 3 sheep from S and assign it to index 0,1,2 respectively in B. And if we have the second pair (3, 7), we will take 5 wolves from W and assign it to index 3,4,5,6,7 respectively in B.

A rough implementation of step 4 is shown below:

```
Let s = 0, w = 0, b = 0, type = 1
for t = 0 to SA.size
  for b = SA[t].left to SA[t].right // for each subarray in B copy animal from the correct array (W
                                    // or S) into the subarray
    if (type == 1)
      B[b++] = S[s++]
    else
      B[b++] = W[w++]
  type *= -1
```

Grading Scheme:
5 marks for correctness

- 5 marks for correct algorithm
- 4 marks for minor flaws in algorithm
- 3 marks for major flaws in algorithm
- 0-2 marks for incorrect algorithm

5 marks for efficient solution

- 5 marks for O(N) algorithm
- 3 marks for any other algorithm better than or in $O(N^2)$
- 2 marks for any other algorithm worse than $O(N^2)$

(Marks are awarded for efficiency only if student has scored at least 3 marks for correctness)

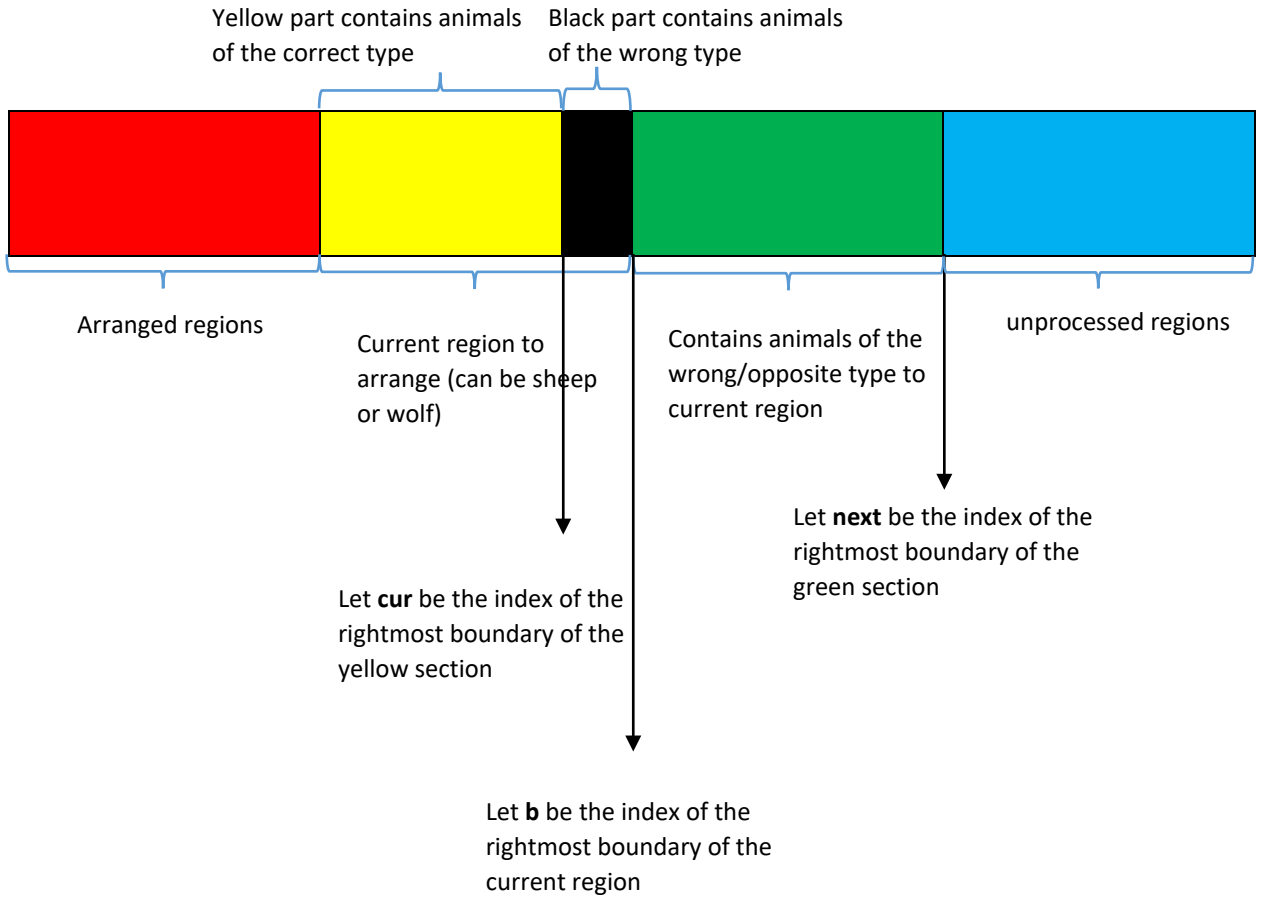
Notes:
For solutions that violate the O(N) space requirement, a penalty of 50% (round down) of the marks will be applied.

Common mistakes:
Not modifying B when algorithm is complete (return the auxiliary data structure but do nothing to B).
Algorithm that only works if there is 1 wrong item per subarray.
Algorithm that only works if assume that wrong item can always be found in adjacent subarray.
O(N^2) algorithm by linearly searching for a second “wrong” item in a pair to swap. Also, wrong analysis of O(N^2) algorithm as O(N).
Other O(N^2) algorithms
Not stating type of sort or how to sort.
Missing important details in the algorithm description / explanation.
Not wrong, but there are many overly complicated algorithms. Accepted answers can be as short as 2-3 sentences if they are explained clearly.

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

Solution 1:

Since you can only use additional O(1) space, the idea is to use something similar to partitioning of quicksort (which is in-place and thus use only O(1) additional space) to efficiently arrange all the animals back into their correct subarrays/regions. The array B is divided into the following sections as the algorithm proceeds:



Once the entire current region is yellow (contains animals of the same type), it will become part of the arranged regions (red) and the algorithm will now move on to the adjacent region which will become the current region.

The algorithm is as follows:

1. Let t be index of the current region in SA. t = 0
2. b = SA[t].right
3. cur = 0 , next = 0 // at the beginning, current region is the 1st subarray in SA. There is nothing to
   // the left of that subarray thus region to the left is trivially arranged
4. let type be the animal type for the current region. type = 1 // always start with sheep
5. **while** (cur != N)
   if (B[cur].type == type) // correct animal so increment cur
   cur += 1
   if (cur > b) // current region arranged
   t +=1
   b = SA[t.right] // update to right boundary of new region
   type *= -1 // update type of animal for new region
   else // wrong animal at the boundary of the yellow section
   if (next < cur) // get new next position
   next = cur+1
   while (B[next].type != type) // now move next until an animal of the correct type, this will
   // grow the green region
   next += 1
   swap B[cur] and B[next] // swap the two animals at cur and next
   cur++ // increment cur as yellow region is now extended

The algorithm only takes O(N) since cur and next can only move through all indices until they reach the end of the array B and for every index, at most a swap is made in O(1) time.

Solution 2:
Notice that for every *k* sheep that is in a wolf "slot", there will be exactly *k* wolfs in a sheep "slot". So, all we need to do is to find all *k* pairs of these sheep+wolf in wrong slots and swap them. In the worst case, there will be only n/2 such pairs to swap, and we will only iterate through every index in B once.

```
ss = 0 // sheep slot number
ws = 1 // wolf slot number
i = SA[ss].left
j = SA[ws].left

while true
   while (B[i] != 1 and i <= n-1) // find next wolf in sheep "slot"
      i = i + 1
      if (i > SA[ss].right) // go the next sheep range
         ss = ss + 2
         if (ss >= SA.size) // no more wrong wolf / range pairs
            return
         i = SA[ss].left
   while (B[j] != -1 and j <= n-1) // find next sheep in wolf "slot"
      j = j + 1
      if (j > SA[ws].right) // go to next wolf range
         ws = ws + 2
         if (ss >= SA.size) // no more wrong sheep / range pairs (in fact this is not needed)
            return
         j = SA[ws].left
   swap(B, i, j)
```

Grading Scheme:
4 marks for correctness

- 4 marks for correct algorithm
- 3 marks for minor flaws in algorithm
- 2 marks for major flaws in algorithm
- 0-1 mark for incorrect algorithm
- Note: 0 mark for any algorithm that violates the O(1) space restriction

4 marks for efficient solution

- 4 marks for O(N) algorithm
- 2 marks for any other algorithm better than or in $O(N^2)$
- 1 mark for any other algorithm worse than $O(N^2)$
- Note: 0 mark for any algorithm that violates the O(1) space restriction

(Marks are awarded for efficiency only if student has scored at least 2 marks for correctness)

Common mistakes:
Violating O(1) space restriction
Algorithm that only works if there is 1 wrong item per subarray
Algorithm that only works if assume that wrong item can always be found in adjacent subarray
O(N^2) algorithm by linearly searching for a second “wrong” item in a pair to swap
Missing important details in the algorithm description / explanation
Doing the variable increment wrongly (for solution 2)