# **COMP3506/7505: 2021 exam answers**

## [**UQAttic**](http://uqattic.net)

## **Get more out of your study time. Join UQAttic's revision chat.**

#### [**Other exam papers**](https://docs.google.com/folder/d/0B6_D4T6LJ-uwYzY1YWMzNjYtMzUyZC00OTEyLWJlMjktOGExYWUwOTc4NDE3/edit)

### Please **contribute** to these documents.

### If you're looking for an effective way to familiarise yourself with the course material, you can't go past collaborating with fellow students. We have laboured to put these up, and so at the very least point out where you think we are wrong!

### You'll get more out of the course, you'll do better in the exam, and other students will benefit from your input as well.

### To get editing permissions, simply go to the [chatroom](http://uqattic.net) and provide us with your Google Account address.

### **Style.**

### Type answers in blue beneath each question.

### If you're unsure of your answer, highlight your answer text then hit Ctrl+Alt+M to create a comment beside the text. Once you're satisfied with the answer, click the "Resolve" button on the comment.

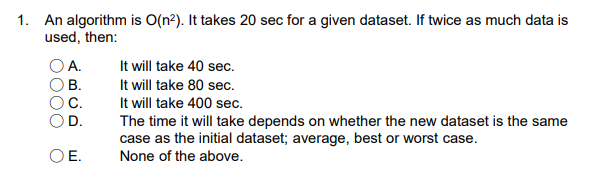
### If you want some extra explanation from someone else on their answer, highlight the other person's answer and repeat the procedure above.]

### **Communicate.**

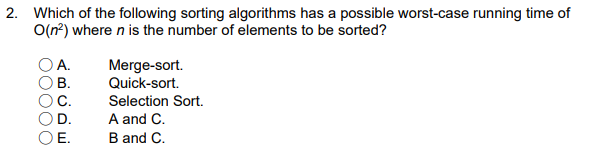
### Head over to [uqattic.net](http://uqattic.net/) and click "Chat Now!". You'll find a chatroom full of students just like you. Talk about a revision document (like this one) or swap prep tips. If you have your own IRC client, point it to irc.uqattic.net, port 6667, channel #attic.

### 2022 Exam is at the end of this document +1+1+1

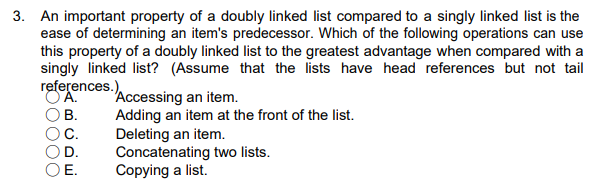
# Multiple Choice



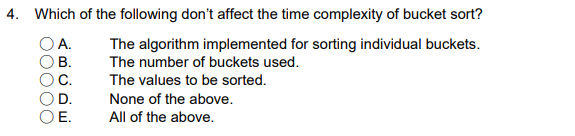
| D +3 |
| --- |



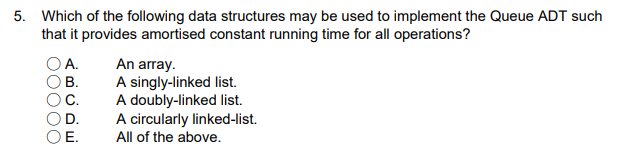
| E +3 |
| --- |



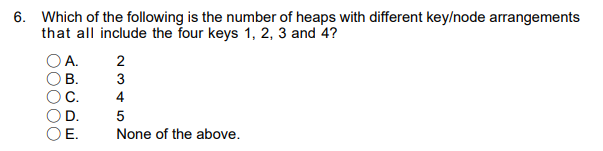
| C +2 |
| --- |



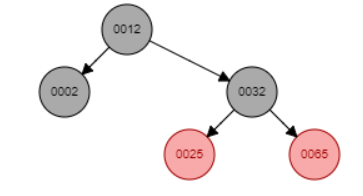
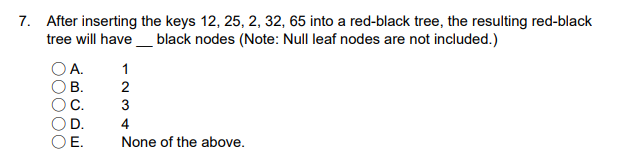
| D+2  C: the time complexity of sorted value will be the same.  Answer A implies multiple items may be stored within the same bucket -> the values to be sorted matter, as if every item is stored in one bucket, the time complexity is determined by the bucket sorting algorithm. Therefore, **D**. (can google this as well lots of sites corroborate this, e.g <https://iq.opengenus.org/time-and-space-complexity-of-bucket-sort/> ) |
| --- |



| E +1  Would this not be E (all of the above)? A queue needs to be able to access/remove its first element, insert a new element as the new first element, and check for the size/if the queue is empty. The size/isEmpty operations can always be implemented in constant time regardless of the data structure by storing the size as an integer variable within the class which is incremented/decremented whenever enqueue/dequeue is called. isEmpty then just needs to check if the size variable == 0. An array implementation can give amortised O(1) for enqueue and dequeue if using a circular array/ring buffer implementation: <https://www.geeksforgeeks.org/introduction-and-array-implementation-of-circular-queue/>. Doubly linked lists should have constant time insertion/removal if the element is always added/removed at the same known positions of the list (the start or end of the list).  Adding onto this, I don’t think it’s possible to dequeue from a circularly linked-list in O(1) time. It doesn’t have a head/tail pointer. To perform dequeue you could remove the head node from the list assuming your cursor is pointing to it, but how would you update the end of the list to point to the new head without iterating through the whole list? Seems O(n) to me… That said, it seems possible to do A, B, C, given you have head/tail nodes in singly/doubly linked lists.  Seems possible here :  <https://www.geeksforgeeks.org/circular-linked-list-implementation-of-circular-queue/>  I thought this was C and my reason was that signgly-linked lists cannot enqueue at O(1) as the tail is not stored by the List. At least with how we have been taught about singly-linked lists.  Confirmed E -> Week 5 Quiz (Quiz 4) Question 8 has this exact question, and the answer given by course staff is All of the above. |
| --- |

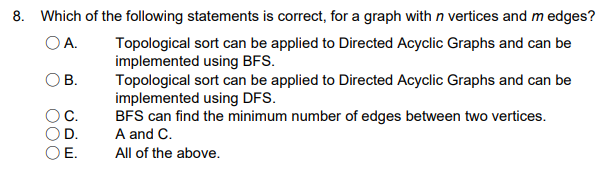


| B +4  ||also this  What about max heaps? That would make the total 6? Not a very clear question…  Given multi choice doesnt include that, think you just assume based off that |
| --- |

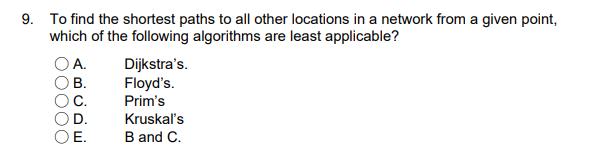


| C +3 |
| --- |

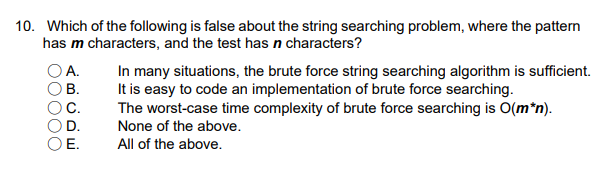
`



| E +3 for sure. see q18 on how A.) is true |
| --- |

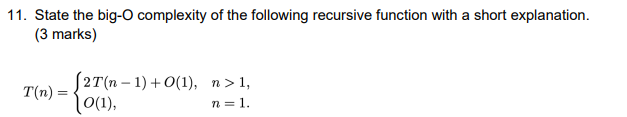


| Floyd’s is the only one not in the wk9 lecture notes, A,C and D are in O((n+m)log(n)) time. My guess is B.  It is A  Dijkstra’s algo computes the shortest path to all other locations from a point, it is the MOST applicable. Prim’s computes an MST, which does not necessarily mean the shortest paths (just the smallest total edge weights). No idea about Floyd’s - It’s C or E surely. Since we haven’t covered Floyd’s, must be C  It could be D. Since Kruskal's does not work from a "given point" but rather it looks at the graph as a whole. +1 We can choose the starting point in Prim so only Kruskal is not applicable. |
| --- |

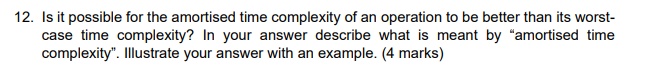


| D +3 +1 |
| --- |

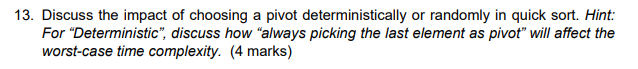
# Short Answer



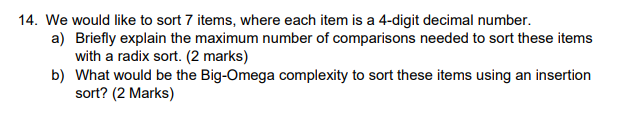
| O(2^n) ? +4  I got this summation, but I don’t know where I would go from here in an exam. Can someone do the working for this Q please? Or can we just simplify this to 2^n    I got O(2^n) from writing out the expansion as:  T(n) = O(1) + 2T(n-1)  T(n) = O(1) + 2(O(1) + 2T(n-2))  T(n) = O(1) + 2 \* (O(1) + 2 \* (O(1) + 2 \* T(n-3)))  Then removing all the O(1)s, you get  T(n) = 2 \* (2 \* (2 \* T(n-3))))  And then just guessed it was O(2^n) from there, since the base case is O(1), hence the question mark in the above answer haha. |
| --- |



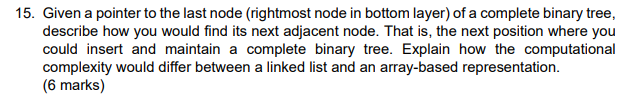
| The amortised time complexity of an operation can be better than its worst case time complexity. Amortised time complexity represents the average time complexity of an operation over the total number of calls to that operation. A simple example is inserting an element into an array, where the array resizes using a doubling strategy once it is completely full. The worst case insert time would be O(n), if the array resizing has to take place on this call to the insert method. (Resizing involves initialising a new empty array of double the size, then manually copying across all n elements into the new array, occurring in O(n) time). In cases where the array does not need to be resized, insert runs in constant time. If you average out the worst case time complexity of the operation (O(n)) over the total number of operations performed before an array resize occurs (n operations, as the array is only resized when full), then the amortised time complexity is O(1). This is less than the worst case time complexity of O(n) for insert. |
| --- |



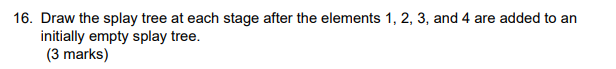
| This was discussed briefly by a tutor in a revision session (<https://uqz.zoom.us/rec/share/vPjYdpweZ48IlzsjozAmx0L-PLpA_Qczif4FI9gBQIS5PVYmUeFt_EpQSAVcdBaS.ct-dOROW7RowPELT?startTime=1668394867000> ). If quick sort is given a data set of partially or fully sorted data, and the chosen pivot is always the last element, quick sort will perform in the worst case. In the case of a fully sorted list {1, 2, …, n}, each partition of the list will result in the partition L (less than) being of size n - 1, and a partition E (equal) containing only the pivot. A tree of recursive calls in this case will be of a height of O(n). This is opposed to the expected height of O(log n), which would result from choosing a pivot randomly that sits in some threshold around the median value for all elements in the list. (Notably, the easiest way to make any divide-and-conquer algorithm become horribly inefficient is to give it a case where most/all of the recursive work ends up on one half of the recursion call tree, instead of approximately evenly divided). Whilst the ideal case with a height of around log n results in a total time complexity of O(n log n), in this particular example, the time complexity will be closer to O(n^2).  You can draw out the tree for this and calculate the summation of i which is the sum of all integers from 1 to n, which is [(n+1)\*n]/2, which is O(n^2). |
| --- |



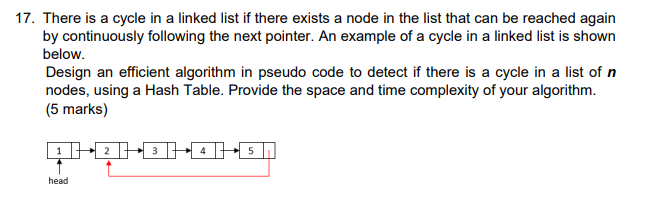
| a) 4^4  I think this might be a trick question. Radix sort isn’t a comparison sort, so it doesn’t actually make any comparisons at all (0 comparisons) +2 wow what a stupid question +2  # I got 280 comparisons. “Radix sort is in base 10. Need 10 different buckets to store the 4-digit decimal number. 10 comparisons needed to traverse through the buckets per digit. Since the decimal number is of size 4, needs to traverse the buckets 4 times. Since there is also 7 of these decimal digits, the whole operation needs to be performed 7 times. Therefore: 7 \* 10 \* 4 = 280”  It’s 0. Radix sort uses bucket sort for each digit, and bucket sort is not a comparison based sort. Minhao confirmed 0 on Ed.  b) Ω(n) Ω(7)? +2 why? Big omega isn’t the same as best case? Isn’t answer Ω(n^2). Pretty sure big omega is best case - it is, at least, the lower asymptotic bound. Hence Ω(n), as it cannot possibly be any faster, thus it is the lower bound.  Insertion is an adaptive sorting algo, so if the data is already sorted then it only iterates through the array once hence Ω(n) |
| --- |



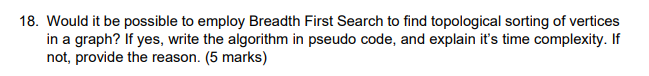
| The array implementation for this should be fairly easy. In an array, the index of the node corresponds to its element. The rank of each node is calculated such that all nodes in that tree level are assigned an increasing rank from left to right, up until that level of the binary tree is filled with the maximum number of nodes (at which point, a new level of new nodes is created with their ranks also assigned in increasing values). Because of this implementation, and because a complete binary tree can only have nodes inserted from left to right in its last level, it is very easy to determine the index of the next adjacent node, if the index of the last node is known. If the index/rank of the last node is *i*, the next adjacent node will have an index *i+1*. Accessing the array at index i+1 is a constant time operation.  For linked list implementation: assume that each element in the linked list has pointers to its parent, left child, and right child. First, check the parent of the last node, and see if the last node is a left or right node. If it’s a left node, then the parent has no right child, and we can insert the new node as the parent’s right child. If the last node is a right node, then we need to traverse up the tree/list elements and check other paths for a candidate empty position that maintains the complete binary tree property. In the worst case, the last node could be the rightmost node on the final level of the tree, where that level is completely full. In this instance, the traversal algorithm would have to go up through all ancestors in that chain to the root of the tree, then go downwards in a leftmost manner to form the first leftmost node in a new level. If a complete binary tree has a height of around log n, then it would take about 2\*log n (is O(log n)) time to do this. (Not 100% sure if this is correct, there may be a more efficient way to do this.) You could also try implementing the linked list such that prev/next pointers corresponded to level-order predecessors/successors, and then you would just be placing the next adjacent node directly after the last node in the list, although I’m not sure if you could reconstruct a unique tree from a level-order traversal or if that’s what the question wants. |
| --- |



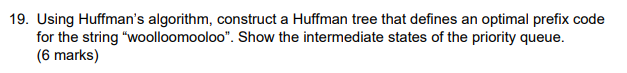
| +6 |
| --- |



| Algorithm cycle\_check(head): time: O(n), space: O(n)?  hash = int[n - 1] not 100% sure if this works, kinda rushed  count = 0  current = head  While count != n:  hashCode = current.hashCode()  if (hash[hashCode] == null):  hash[hashCode] = current  else:  Return true  current = current.next  count ++  I think the approach is right, but maybe small implementation issues - hashCode() in java will produce an equivalent hash code if two objects are equal (by equals()), but they are not necessarily distinct if the values are different. In this case, two different list nodes could by poor luck have the same hashcode and result in a collision. # This is suppose to be pseudocode anyway, so can assume uniqueness. RE: depends how much they care about specifics, it seems also like we’re adding a list node object to an array of integers/indexing with a probably huge hash code into an array of size n-1, but yea maybe they dont care about small stuff.  The correct approach is adding each node to a hashtable if it is not in there already, and returning true if it already exists, so time/space complexities correct.  **Answer 2**  Algorithm checkCycle(linkedList)  Input: linkedList the list to be checked for cycles  Output: true if a cycle exists; false otherwise  hashtable = //Initialise hashtable - we weren’t required to show how to make a hashtable,  //just use it  pointer = linkedList.head while !(hashtable.search(pointer.key, pointer.value)) do  if pointer = Null then  return false // that is - you reached the end - congratulations  hastable.insert(pointer.key, pointer.value)  pointer = pointer.next return true |
| --- |



| algo topo\_sort\_bfs (graph):  Q = new Queue<Vertex>  for (vertex v in graph.vertices()) do  if inDegree(v) == 0 then  Q.enqueue(v) {add all vertices with no  incoming edges to the queue}  R = LinkedList<Vertex> {initialize results list}  while !Q.isEmpty() do {start BFS}  vertex v <- Q.dequeue() {add element at front of queue  to result list -> topo order}  R.add(v)  for (edge e in graph.outgoingEdges(v)) do {disconnect v from graph  if e.vertex.inDegree() == 1 then  Q.enqueue(e.vertex) {add a vertex to the queue if v  was its only inbound connection}  removeEdge(e)      return R {list of vertices in topo order} |
| --- |
| Formatting is a bit messed here: wrote it in notepad, but the comments should explain the steps well enough. Starts from first vertex with no inbound connections (start of topo order) and works its way to the end - kind of like a reverse DFS implementation.  Assumes input graph is a DAG, although you can count the elements in R compared to the total input vertices to check if they are equal. If a cycle was present (no topo order possible), not all vertices will be in R. Also worth noting the input graph should probably be copied, because this code will destroy all its edges.  O(n+m)  Explanation: O(n) setup time before bfs, All vertices visited once during bfs (O(n)) + all edges visited/deleted once -> O(m). BFS total O(n+m) |



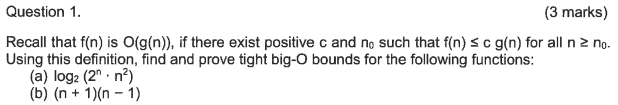
| PQ = ‘o’:8 ‘l’:3 ‘w’:1 ‘m’:1    Shouldn’t it be the other way, with 5 as left subtree and 8 as right subtree Huffman trees aren’t unique, doesn’t matter what side the nodes are on as long as they are ordered correctly  Equivalent # of bits encoding for all symbols, but the algorithm given to us in lectures maintains smaller values on the left. Can’t hurt to follow it    ^ left subtree is the smallest value in PQ, right is second smallest. |
| --- |

**GOOD LUCK!! +1**

**Yes**

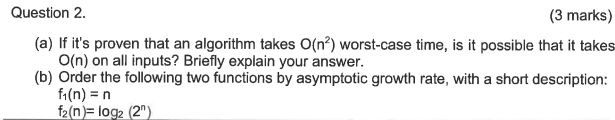
**2022 COMP3506 Exam**

**Q1**

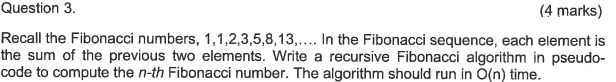
****

| **Can find the answer in week2 tut question sheet** |
| --- |

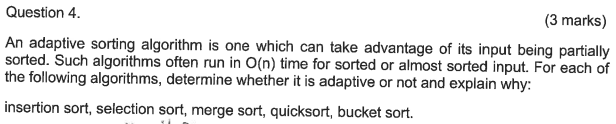
**Q2**

****

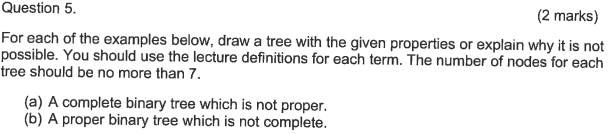
| a)  It is possible that all inputs can be done in O(n), which still follows this upper bound. Big O gives us the asymptotic upper bound, so some inputs can take less time than that  b)  F1 = n  F2 = nlog2(2) = log2(2^n)  F2 is asymptotically slower than f1 because it increases at a slower rate  b)  F1= n = O(n)  F2 = log2(2^n) = nlog2(2) = O(n)  As both functions are equal in asymptotic growth rate, F1 = F2 in asymptotic growth rate. +2 |
| --- |

****

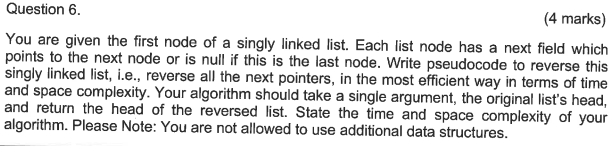
| Is this the correct way?    How would you write this in pseudo-code?  **I would say:**  **Algorithm fib(n):**  **If n is 1 then (I think there should be: if n <= 2 then)**  **Return 1**  **Return fib(n-1) + fib (n-2)**  [**Data Structure & Algorithms Fibonacci Series**](https://www.tutorialspoint.com/data_structures_algorithms/fibonacci_series.htm) **Check out this link for more info**  **\*The “return fib(n-1) + fib(n-2)” seems to be O(2n) not O(n)**  **https://stackoverflow.com/questions/360748/computational-complexity-of-fibonacci-sequence** |
| --- |

****

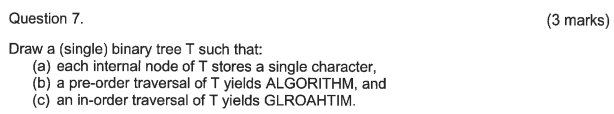
| Insertion sort = adaptive. It reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.  Selection Sort = not adaptive. This is because the initial order of the array has no impact on the number of comparisons.  Merge Sort = not adaptive. It is not adaptive to existence of ordering among the elements. Thus, it has the same computational complexity in any case. This is because, no matter if an array is sorted or not, merge sort will still go in for comparisons and then merge.  Quicksort = adaptive. This is because this algorithm only performs operations on the sorted array if it is not equal to the operations performed on the unsorted array. Can someone please explain this better? +1  I’m like 95% sure that quicksort isn’t adaptive! Lol <https://www.geeksforgeeks.org/is-quick-sort-algorithm-adaptive-or-not/>  THE ANSWER IN THE TUT SHEETS.  +1  Bucket sort = not adaptive. As time complexity doesn't vary based on input order. |
| --- |

****

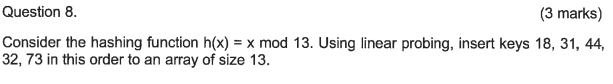
| A proper binary tree is one where all internal nodes have exactly two children  A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible  In the examples we should only use 7 nodes…the answers show the logic anyway   1. **Complete but not proper**     **b) Proper but not complete** |
| --- |

****

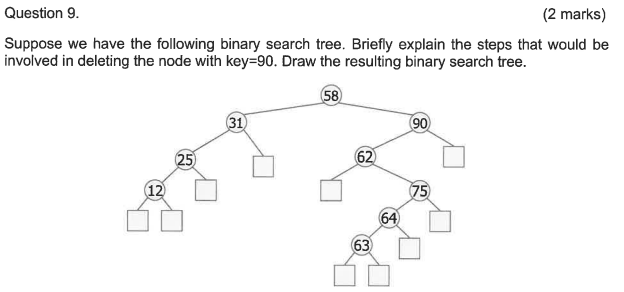
|  |
| --- |

****

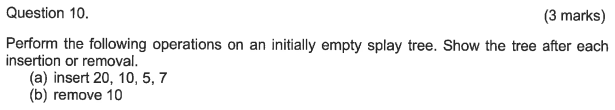
|  |
| --- |

****

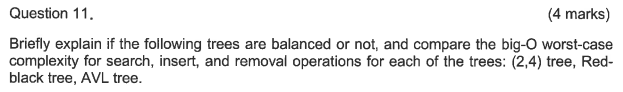
|  |
| --- |

****

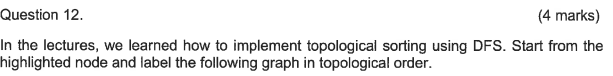
|  |
| --- |

****

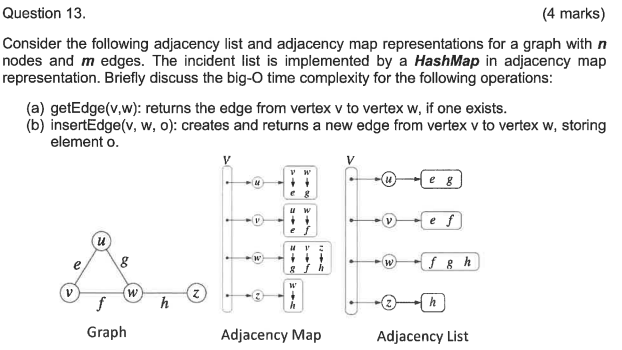
| **b)**  **7**  **/ \**  **5 20** |
| --- |

****

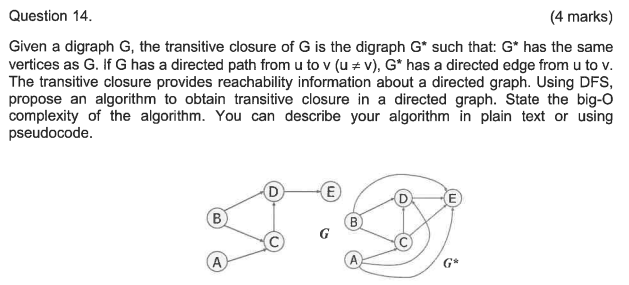
|  |
| --- |

****

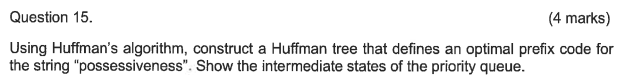
|  |
| --- |

****

|  |
| --- |

****

| **Does anyone have a solution to share here? +1** |
| --- |

****

| My answer might be different from yours depending on which element you decide to pop since there are several elements throught the intermediary with the same prioitiy.  <https://stackoverflow.com/questions/44764085/can-a-huffman-tree-enconding-be-different-from-one-person-to-anoher> |
| --- |