Question 1 [50 marks]

a) Consider the following (naive) algorithm that computes the maximum and the minimum elements in a list.

- Set *min* and *max* to the first element in the list.
- Run through the rest of the list comparing each element to *max* and *min*, replacing the values if appropriate.

How many comparisons does this algorithm use?

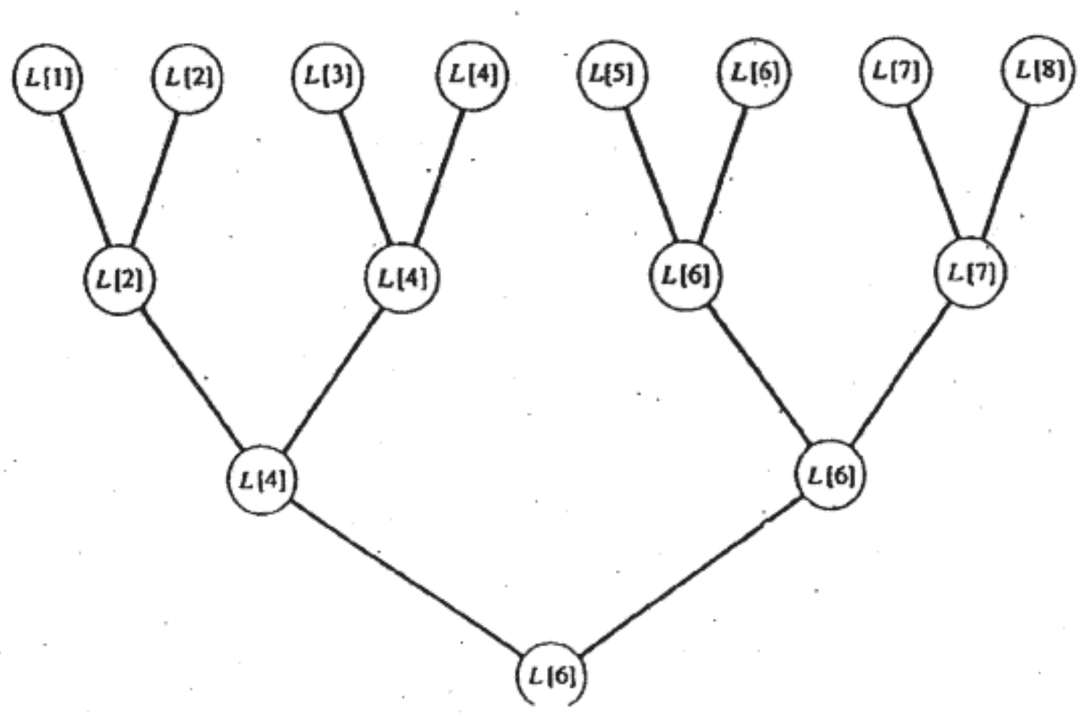
b) Consider the following algorithm for the same problem:

- pair up the elements
- use one comparison to distinguish the maximum and minimum of each pair.
- choose the maximum of all the maxima.
- choose the minimum of all the minima.

How many comparisons are used by this algorithm?

Question 2 [50 marks]

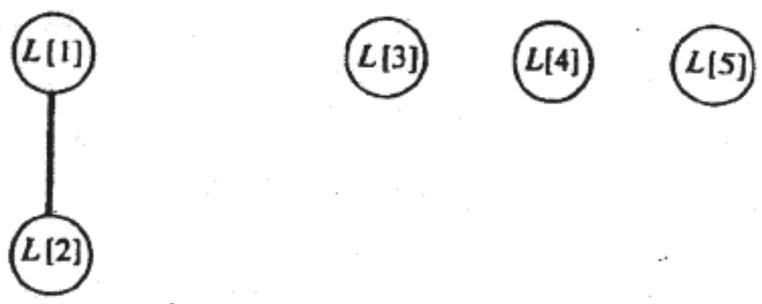
a) Consider the problem of finding the second largest in a list of (distinct) items. Consider an instance of a standard tournament defined by the diagram below. Which three elements are the *only* candidates for second best?
Please just mark them on the diagram, unambiguously.



2/7

b) Remember that there is an adversarial strategy that will ensure that the maximum item will be compared directly to at least $\lceil \lg n \rceil$ other items.
Show the answers *the adversary* would give in response to the following questions, if it were following the text book strategy.
Illustrate the answers by way of POSET pictures.
Hint: the first answer is given.

Algorithm: "Compare L[1] and L[2]?"
Adversary: "L[1] > L[2]"
POSET:



Algorithm: "Compare L[1] and L[3]?"
Adversary:
POSET:

Algorithm: "Compare L[4] and L[5]?"
Adversary:
POSET:

Algorithm: "Compare L[1] and L[5]?"
Adversary:
POSET:

c) Point out how the result confirms the claim for this strategy. (It should!)

3/7

Question 3 [50 marks]

Consider the following naive sorting algorithm:

- find the largest element, call it *max* in the unsorted section of the list
- swap *max* with the last element in the unsorted section {*max* is now part of the sorted section}
- repeat until entire list is sorted.

a) Write out this algorithm in pseudo-code (say like Pascal), assuming that the list is represented by a one dimensional array L containing n elements.

b) How many comparisons does the algorithm use in the worst case? On the average?

c) How many swaps (count an entire exchange as one operation) does it use in the worst case? In the best case?

Question 4 [50 marks]

Consider the following algorithm, a version of Bubblesort.

```
for i := n downto 2 do
  for j := 1 to i-1 do
    compare items j and j+1 and swap if out of order;
```

a) Prove (one or two sentences could be enough) that the inner loop moves the largest item in the list 1 thru' i to position i.

b) Set up a loop invariant assertion suitable for describing the effect of the outer loop

c) Use a) to prove that the assertion is indeed invariant

d) Use b) and c) to show that the algorithm does sort the list. (Since the loops are **for** loops, you may assume that they terminate).

4/7

Question 5 [50 marks]

Quicksort is usually expressed recursively, because the stacking of intermediate results (lists to be partitioned) is then done automatically by the system, thus relieving the programmer of a serious chore.
Suppose the programmer chose to program the stack explicitly and made an error so that the stack turned out to be a FIRST IN FIRST OUT queue.

a) What effect would this have (if any) on the worst case and average case time complexity? (Only consider asymptotic results, do not worry about small changes). Justify your contentions.

b) What effect would this have on the worst case and average case space complexity (which is the space used by the queue). Justify your contentions.

Question 6 [50 marks]

a) Consider merging two, equal length, sorted lists each of length n. Argue that there exists an instance of the problem *requiring* (no matter what algorithm is used) at least $2n - 1$ comparisons.
Hint: Consider the the case where the resulting sorted list is composed of elements taken alternately from the two input lists. Show that every pair of adjacent elements in the output *must* have been compared directly to each other.

b) Suppose you want to merge two lists, and one list only has two elements. How would you minimise comparisons, assuming each list is represented by a one dimensional array?

c) Of course, not only comparisons are required, but also data rearrangement, so how much movement is required, in the worst case, in your answer to b)?

Question 7 [50 marks]

a) Arrange the nine characters of the word "COMPLEXITY" in a nine element Heap. (Draw a diagram). Assume the usual alphabetic order is defined on the characters.
Hint: Any Heap will do, it is not necessary to apply any particular algorithm in building it.

b) Remember that one can define a procedure that *Inserts* a new element into an existing Heap by placing the new element in a new leaf adjacent to the last existing leaf, and letting the new element filter up to its correct level.

Suppose we choose to build a Heap from nothing to n elements by repeatedly *Inserting*.
Prove that this procedure takes time bounded below by $\Omega(n \lg n)$ in the worst case.
Hint: You must point to some specific instance of the problem (for any n) *and* show that instance uses this time.

c) Why is this unimpressive?

5/7

Question 8 [50 marks]

Consider the problem of sorting 1000 records whose keys are 5 character strings drawn from the alphabet "A" through "Z".

a) About how many key comparisons would you expect Quicksort to make, assuming the constant of proportionality (in the order notation) is about 1.4?

b) About how many machine instructions would you expect Quicksort to use (in making these comparisons), assuming each character has do be compared one at a time and each comparison requires at least one instruction?

c) Now consider solving this task by using Radix sort. How much comparison work is done in terms of the length of the key (5) and the number of records?

d) Which method would likely be superior and does this remain the case for larger and larger sets of records?

Question 9 [50 marks]

a) Run Depth First Search on the graph below. Start at node 1, assume that "neighbours" are ordered in numerical order in the adjacency lists. Note that it is a directed graph.
Write the order in which the nodes are *first* visited in the spaces provided. i.e. line them up left to right in the order visited.

b) Now draw in the directed edges on the same diagram. What do you notice that is interesting?

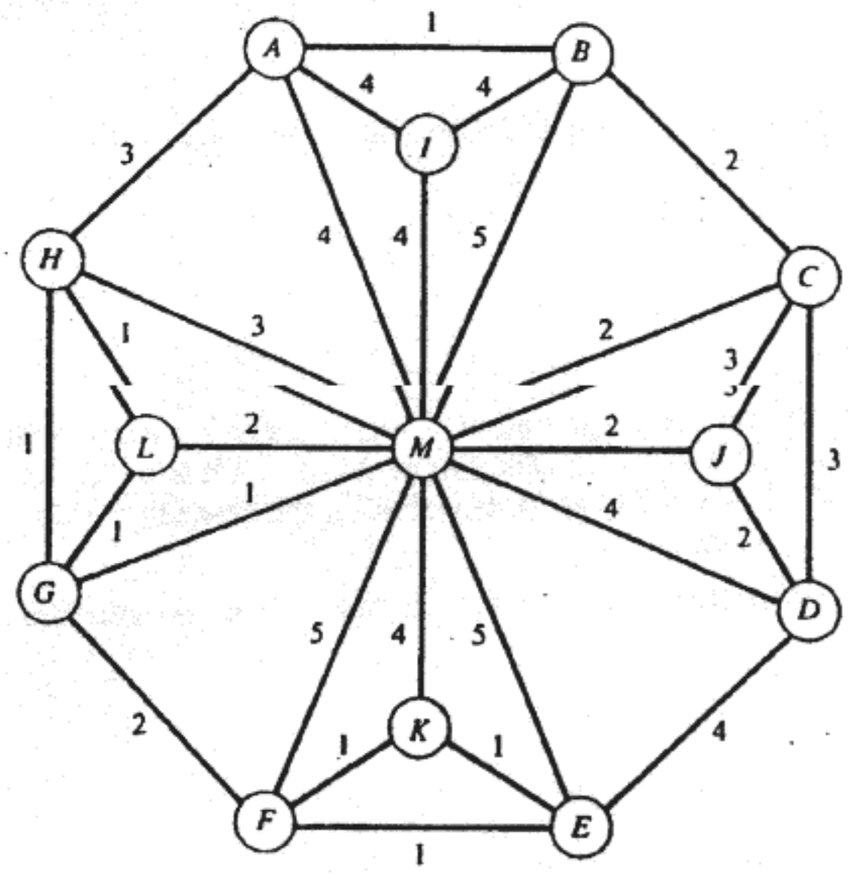
c) Suppose you run **Breadth** First Search, starting at node v in some undirected graph. Describe in principle how one could add a little to the basic *bfs* algorithm (see back of this page !!) so that each node becomes associated with an integer representing its distance (path length) from v.
Hint: Note that neighbours of v are at distance 1 from v, neighbours of neighbours at distance 2, etc.

```
procedure BreadthFirstSearch (adjacencyList: HeaderList; v: VertexType);
var
    Q: Queue;
    w: VertexType;
begin
    initialize Q to be empty;
    visit and mark v; insert v in Q.
    while Q is nonempty do
        x := RemoveFromQ(Q);
        for each unmarked vertex w adjacent to x do
            visit and mark w;
            insert w in Q
        end { for }
    end { while Q is nonempty }
end { BreadthFirstSearch }
```

6/7

Question 10 [50 marks]

a) Run Kruskal's Minimum Spanning Tree algorithm on the example below.
Mark each edge that is in the tree and, for each tree edge, append a number indicating the order in which it was chosen, (first edge "1", second edge "2", etc).
Hint: Kruskal repeatedly tries to add the next unused, cheapest edge, so long as it does not create a cycle in the forest so far formed.
Use the ordered edge list given



Sorted edges: *AB, EF, EK, FK, GH, GL, GM, HL, BC, CM, DJ, FG, JM, LM, AH, CD, CJ, HM, AI, AM, BI, DE, DM, IM, KM, BM, EM,* and *FM.*

b) How many MSTs can there be if all the edge weights are distinct? Prove your contention.
Hint: Argument by contradiction is one way to do it.

7/7

Question 11 [50 marks]

a) Let $S = \{1, 2, ... 9\}$. Assume you are to use the Union (denoted U) and Find (denoted F) algorithms as described in class. Draw the trees representing each sub-set after the *last* Union and after *each* Find operation, as specified below.
Hint: The union should make the root of the largest set the new root. The Find should include "path compression", i.e, every node visited from target node to root should be made a child of the root.
If the sizes of two trees are equal then, on Union, make the root of the second tree the new root.

U(1,2); U(3,4); U(2,4); U(6,7); U(8,9); U(7,9); U(4,9);
Result:

F(1); Result:

F(4); Result:

F(6); Result:

F(1); Result:

b) Prove that, even if "path compression" is not used, choosing the new root (on Union) in the manner defined in hint a), ensures that that tree height is bounded above by $\lceil \log n \rceil$, where there are n items in the set represented by the tree.
Hint: You need an inductive argument. Note that the union operation can do no more than add one to (which?) tree height. What can it do to the size of the tree?