Assignment 2 marking scheme (draft)

Question 1

a) A BST node stores its children such that the maximum value in the left subtree is less than the value of the current node which is less than the smallest value in the right subtree. A max-heap node stores its children such that the largest element of both subtrees is less than the value stored at the current node. The values in the two subtrees have no particular order relative to one another.

b) Since a heap can be constructed in linear time, no linear algorithm can exist that visits heap nodes in sorted order, otherwise we would have an O(n) comparison sort which constructs and then walks the heap.

Question 2

This algorithm computes all pairs of line segments, sorts the gradients of all line segments starting at each given point. The points are not completely triangulable if and only if a duplicate gradient exists at one or more starting points,

```
# Input: two arrays of the same size containing x and y coordinates
#  respectively.
# array.sort() modifies array in place using heapsort O(n log n).
# Discussed in week x slide y
def completely_triangulable(xs, ys):
    for i in range(len(xs)):
        x0, y0 = xs[i], ys[i]
        # "gradients" could also be "angles" using appropriate trig
        # functions.
        gradients = []
        for j in range(i + 1, len(xs)):
            num_inf = 0  # Need this if using gradients, angles avoids this complication
            dx, dy = xs[j] - x0, ys[j] - y0
            if dx != 0:
                gradients.append(dy / dx)
            elif num_inf == 0:
                num_inf += 1
            else:
                return False
        gradients.sort()  # O(n) sorts occur, each taking O(n log n) time.
        # NB: students must specify what kind of sort. Heap and Merge
        # have acceptable performance here.
        for j in range(1, len(gradients)):
            if gradients[j - 1] == gradients[j]:
                return False
    return True
```

Question 3

There were two classes of good solution here: ones based on heapsort, and ones based on merge sort.

In these solutions, a size k heap is maintained of the smallest elements in each of the input lists. Each of the n elements in the final list is pushed and poped from that list at most once, each operation costing O(log k). The overall algorithm is thus in O(n log k).

Correctness is guaranteed by the fact that at all times the min heap is guaranteed to contain the smallest un-processed element in each input list, since these lists are sorted this guarantees that the minimum element in the heap must be the next one in the result array.

```
import heapq

# "lists" is an array of arrays.
def k_merge_lk(lists):
    kSortedList = []
    H = []
    head = []

    # note range is inclusive/exclusive.
    # i.e. the loop below means "for j <- 0 .. len(lists) - 1"
    for j in range(len(lists)):
        key = lists[j][0]                  # head for list j
        data = j                           # remember list j
        H.append([key, data])              # store head of list j
        head.append(0)                     # index of current head for list j

    heapq.heapify(H)                       # create min-heap in O(k) time (week 9 slide 5)

    while H:
        elem = heapq.heappop(H)            # pop min element in O(log k) (week 9 slide 2)
        kSortedList.append(elem[0])        # store it
        j = elem[1]                        # remember the list j of the popped element
        next_head = head[j] + 1            # move to the index of next head on list j

        if next_head < len(lists[j]):      # if list j still has a head
            elem[0] = lists[j][next_head] # store its value
            heapq.heappush(H, elem)        # insert elem in O(log k) (week 9 slide 3)
            head[j] = next_head            # store index of next head for list j

    return kSortedList
```

Merge-based:

In these two solutions, we halve the number of lists each call, meaning O(k) merge calls in total, and O(log k) recursion depth. Note that each item in the final array is merged at most once in each recursive level, each recursive level therefore does at most O(n) work, so the overall complexity is O(n log k).

Correctness is guaranteed by the fact that merge(l, r) returns a sorted list if both of its inputs are sorted. In each merge_k call the return value is either the result of a merge call, or an element of the input lists which are already sorted. All recursive calls are either merges of input lists, or original input lists, either of which must be sorted.

```
def merge_k_bottom_up(lists):
    if len(lists) == 1:
        return lists[0]
    half_lists = []
    if len(lists) % 2 == 1:
        half_lists.append(lists[-1])
    for i in range(len(lists)/2):
        l0, l1 = lists[2*i], lists[2*i+1]
        # "merge" takes two sorted lists/arrays and returns a new
        #  sorted list, discussed in lecture ...
        half_lists.append(merge(l0, l1))
    return merge_k_bottom_up(half_lists)


def merge_k_top_down(lists):
    if len(lists) == 1:
        return lists[0]
    l = merge_k_top_down(lists[0:len(lists)/2])
    r = merge_k_top_down(lists[len(lists)/2:len(lists)])
    # "merge" takes two sorted lists/arrays and returns a new
    #  sorted list, discussed in lecture ...
    return merge(l, r)
```