Keenan Miller
CSCI-261 HW1

1. Equivalent functions are listed in the same line delimited by commas. This list of classes are ranked in order of least to greatest growth rate from a to t.
    a. $3,\ n^{1/logn}$
    b. $log\ logn$
    c. $log_3 x$
    d. $logn,\ log(n^3)$
    e. $(logn)^3$
    f. $n^{1/3}$
    g. $n/logn$
    h. $n - logn$
    i. $nlogn$
    j. n^2
    k. $n^2 logn$
    l. $10^{100}n^2 + n^3,\ n^3$
    m. 8^{logn}
    n. 2^{3n}
    o. 3^{2n}
    p. $8^{(n-1)}$
    q. 2^{n^3}
    r. $n^{log\ logn},\ (logn)^{logn}$
    s. $n!$
    t. n^n

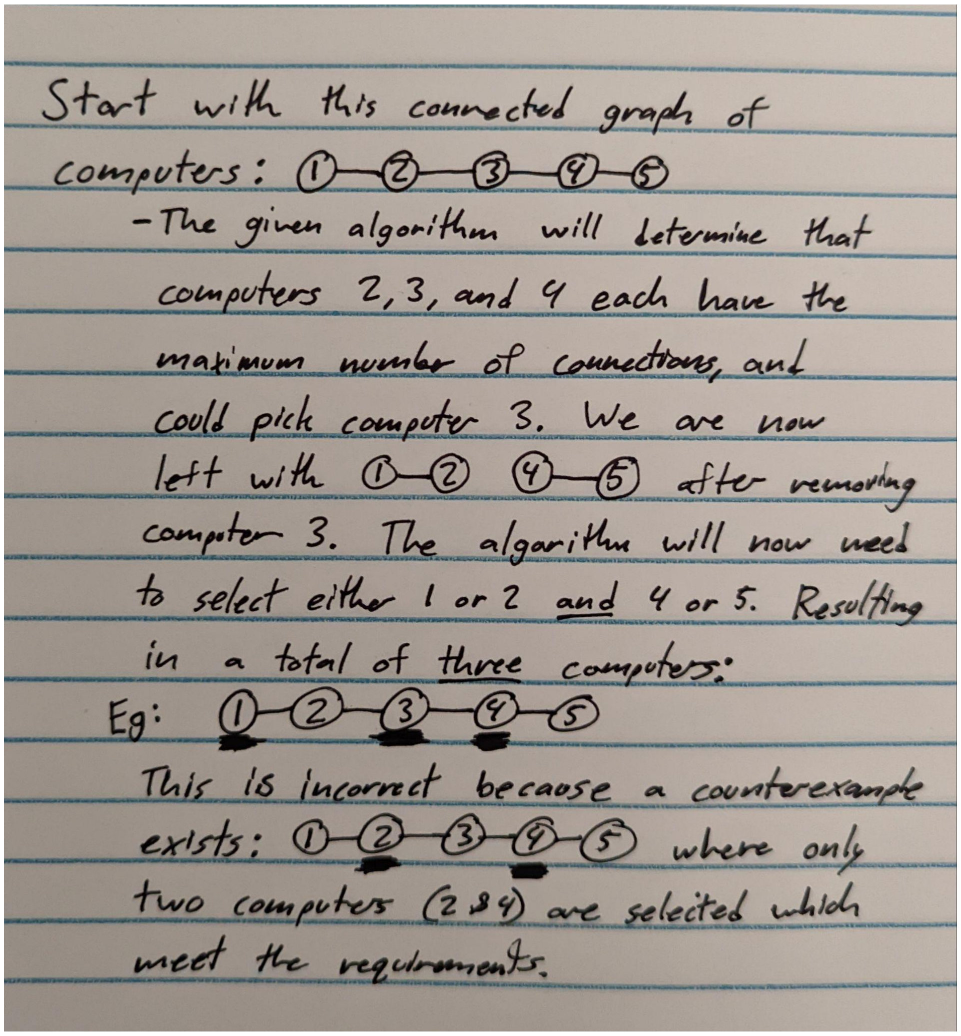
2.

a. **Pseudo code**:

```
var computers; // list of computers - eg: A, B, C
var cables;  // list of pairs of computers connected by cables - eg:(A,B),(B,C)
var result = 0; // this will be the result

while computers is not empty {
      create map of computers to integers initialized at 0;
      for cable in cables {
            increment map counter for one computer connected by cable
            increment map counter for other computer connected by cable
      }
      // now each entry is a map of the computer to the number
      // of cables it is connected to - eg: (A->1), (B->2), (C->1)

      // Note: This doesn't necessarily need to be a map, I just used that term
      // for readability and understandability. For our purposes it is a list
      // with normal O(n) lookup and tuples of computers to integers. Each
      // lookup occurs within only the outer while-loop so no additional time
      // complexity is introduced (as this algorithm is already O(n^2)).

      let maxComputer = first entry-key
      let max = first entry-value
      for entry in map {
            if entry_value > max {
                  maxComputer = entry_key
                  max = entry_value
            }
      }

      remove maxComputer from computers
      for cable in cables {
            if cable has maxComputer {
                  remove cable from cables
            }
      }
      increment result
}
return result
```

b. This algorithm loops over each item in the list of computers. Within this outer loop it performs two loops over the similarly-sized list of cables and one loop over the list of computers. Within each of these three nested loops, only constant-time operations are performed. This algorithm should run in O($3n^2$) time since 3 loops are sequentially running within another loop. Simplified, this can be put as O(n^2).

c.



Start with this connected graph of computers: ① — ② — ③ — ④ — ⑤

- The given algorithm will determine that computers 2, 3, and 4 each have the maximum number of connections, and could pick computer 3. We are now left with ① — ② ④ — ⑤ after removing computer 3. The algorithm will now need to select either 1 or 2 and 4 or 5. Resulting in a total of three computers:

Eg: ① — ② — ③ — ④ — ⑤

This is incorrect because a counterexample exists: ① — ② — ③ — ④ — ⑤ where only two computers (2 & 4) are selected which meet the requirements.

3. Planter problem:
   a. The algorithm I wrote to solve this problem begins by adding all planters, occupied by plants or vacant, to a single array (array A). The array is then sorted using an implementation of mergesort. Array A is then looped over from the greatest size planter to the smallest size planter. Each planter is first checked for vacancy. If the largest planter in A is vacant, the planter is simply moved to the end of another array (array B). If the largest planter in A is occupied, we attempt to plant it in the largest vacant planter in

array B (which we have a counter for to track the index of) if that planter happens to be larger than the current planter (we 'replant' by moving our 'max empty planter' index to the next planter in array B). Once we replant the plant from the largest pot in array A, the now-vacant pot is moved to the end of array B. Repeat this process, if we at any point cannot replant the largest plant from array A into a pot from array B, return "NO". Once array A has been fully looped over, return "YES".

b. **Pseudo code**:
```
Planter-array allPlanters = (read in data);
mergeSort(allPlanters) // from smallest to largest

int maxEmptyPlanterIndex = 0;
Planter-array emptyPlanters = [];
for(planter in allPlanters from largest to smallest) {
    if(planter is vacant) {
        add planter to the end of emptyPlanters;
    }
    else if(emptyPlanters[maxEmptyPlanterIndex] exists
             and is smaller than planter){
        maxEmptyPlanterIndex++;
        mark planter as vacant;
        add planter to the end of emptyPlanters;
    } else {
        return "NO";
    }
}
return "YES;
```

c. This algorithm always produces the desired result. Since the array of all planters is sorted and looped over from largest to smallest, we give any smaller plants the opportunity to be planted in any of the pots that are larger than it. To further ensure this, each plant, when replanted, is always the largest plant that has not yet been replanted and will always be replanted in the largest available planter. Since larger plants have fewer viable planters in which to be replanted, beginning with the largest plant will ensure that, when possible, all plants are successfully planted.
d. This algorithm runs at the desired complexity: ϴ(nlogn).
e. This algorithm begins with a standard mergesort, which we discussed in class to run with a complexity of ϴ(nlogn). After the mergesort is completed, we loop over each planter in the list of planters and perform only constant-time operations within that loop. This loop, which is not nested, runs with ϴ(n) complexity. Since this algorithm begins with a ϴ(nlogn) component followed by a ϴ(n) component, the overall complexity of this algorithm is ϴ(nlogn + n) which simplifies to ϴ(nlogn) because the n term is negligible compared to the nlogn term.

4. Find max pairs double problem:

a. The algorithm I wrote to solve this problem begins by performing addition between each double value provided (using two nested for-loops) and appending each sum to the end of a new array, called sumSet. This array will always have size on the order of n^2 (but will be smaller). This array of sums (sumSet) is then sorted using mergesort from smallest to largest. SumSet is then looped over from largest to smallest, using a single, non-nested, for-loop containing only constant-time operations. Each iteration checks if the double value is equal to the previous value. If they are equal, one is added to a counter which tracks the number of values equal to this double. If they are not equal, the algorithm checks if the number of values equal to the previous double is greater than or equal to the maximum count so far. If it is not greater than or equal to the maximum count so far, we continue and begin counting the number of doubles equal to the current double. If it is greater than or equal to the maximum count so far, we update the maximum count to the current count and update the most-counted double so far. Once each value in sumSet has been iterated over, we return the maximum count, and the most-counted double.

b. **Pseudo code**:

```
inputs;
sumSet = [];

// O(n^2)
for every value1 in inputs from index 0 to inputs.length {
    for for every value2 in inputs from (index of value1 + 1) to inputs.length{
        add (value1 + value2) to sumSet;
    }
}

// O(n^2logn)
mergeSort(sumSet);

currentCount = 1;  // number of values equal to the
                   // currently-being-counted value

maxCount = 1;      // number of values equal to the
                   // maximum-counted-value so far

currentResult = last value of sumSet;
resultValue = last value of sumSet;

// O(n^2)
for value in sumSet from sumSet.length - 2 to 0 {
    if value is equal to previous value {
        if currentResult is equal to value {
            increment currentCount;
        } else {
            if currentCount is greater than or equal to maxCount {
                maxCount = currentCount;
                resultValue = currentResult;
                }
                // reset current counts
                currentResult = value;
                set currentCount to 1;
            }
        }
    }
    // check if the smallest value had a higher count
    if currentCount is greater than or equal to maxCount {
        maxCount = currentCount;
        resultvalue = currentResult;
    }
    return maxCount and resultValue;
```

c. This algorithm's proof of correctness is very simple. Every unique pair of input values are added together, and these sums are all added to an array. This array is sorted, and the amount of elements equal to each value (from largest to smallest) is kept track of. When a greater amount of equal values is found, that is recorded as the result. This result will be overwritten each time a greater number of equal values is found (or if an equal number of smaller values is found). Once the array has been completely searched through, the correct greatest-number of equal values will be returned (along with the value).
d. This array runs with complexity Θ(n^2logn).
e. The complexity of Θ(n^2logn) can be verified because this algorithm begins with a double-nested for-loop which has complexity Θ(n^2) because only constant-time operations are performed within. After, a newly created array of size [on the order of] n^2 is sorted using mergesort. Mergesort runs with complexity Θ(nlogn), but since the array that is being sorted has size [on the order of] n^2, this run of mergesort runs with complexity Θ(n^2log(n^2)) which is simplified to Θ(n^2logn). After the array is sorted, another single loop (with only constant-time operations) is performed on this large array which has complexity Θ(n^2) due to the n^2 size of the array. The addition of these three complex operations of Θ(n^2), Θ(n^2logn), and Θ(n^2) being performed subsequently results in the overall complexity of Θ(n^2logn). This is because the Θ(n^2) operations are comparatively small to the Θ(n^2logn) operation and can be ignored.