Andrew DiBella
Hw5
Sorting

1) Quicksort/ Bucket Sort
   Driver for Time testing:

```
class SortDriver{
public static void main(String [] args){
  int n = 2000000;
  int m = 10000;

  int[] random = new int[n];

  for (int i = 0; i < random.length; i++){
    random[i] = (int)(Math.random() * m);
  }
  // quickSort
  float qstart = System.nanoTime();
  int[] quickSorted = quickSort(random,0,random.length-1);
  float quickSortTime = System.nanoTime() - qstart;



  // bucketSort
  float bstart = System.nanoTime();
  List<Integer> bucketSorted = bucketSort(random, m);
  float bucketSortTime = System.nanoTime() - bstart;



  System.out.println("Bucket Sort: ");
  for (int x : bucketSorted){
    System.out.println(x);
  }
  System.out.println("Quick Sort: ");
  for (int x : quickSorted){
    System.out.println(x);
  }

  System.out.println("bucketSortTime: " + bucketSortTime);
  System.out.println("QuickSortTime: " + quickSortTime);
```

Quicksort:

```
/*
 * Method QuickSort
 * recursive implementation of quicksort
 * utilizes partion method to allow recursive indexes
 * and swaps similar to merge sort
 *
 */
static int[] quickSort(int[] random, int start, int end){

  if (start < end){
    int index = partition(random, start, end);

    quickSort(random, start, index -1);
    quickSort(random, index+1, end);
  }
  return random;
}
static int partition(int[] ray, int start, int end){
  int pivot = ray[end];
  int index = start - 1;

  for (int i = start; i < end;i++){
    if (ray[i] <= pivot){
      index++;

      int swap = ray[index];
      ray[index] = ray[i];
      ray[i] = swap;
    }
  }
  int tmp = ray[index + 1];
  ray[index + 1] = ray[end];
  ray[end] = tmp;

  return index + 1;

}
```

Bucketsort:

```
static List<Integer> bucketSort(int[] random, int max){
   int numBuckets = (int)Math.sqrt(random.length);
   List<List<Integer>> buckets = new ArrayList<List<Integer>>();
   for (int i=0; i<numBuckets;i++){
     buckets.add(new ArrayList<>());
   }

   for (int x: random){
     buckets.get(findBucket(x, max, numBuckets)).add(x);
   }
   Comparator<Integer> comparator = Comparator.naturalOrder();
   for (List<Integer> bucket: buckets){
     bucket.sort(comparator);
   }
   List<Integer> sorted = new ArrayList<>();
   for (List<Integer> bucket: buckets){
     sorted.addAll(bucket);
   }
   return sorted;
 }

 static int findBucket(int i, int max, int numBuckets){
   return (int)((double) i / max * (numBuckets - 1));
 }
```

Results:

N	M	Quick Sort nanoTime	Bucket Sort nanoTime
1000000	**100**	**7.046**	**1.006**
100	**1000000**	**0.00**	**3.355**
1000000	**1000**	**1.006**	**1.006**
100000	**100000000**	**3.3554432E7**	**3.3554432E7**

An N/M with a 1000:1 ratio will sort at the same speed for both algorithms. Quicksort works very well with a small n and a large m as input, whereas, bucketsort works well with a large N and a small M as input.

2) Linear time algorithm using Permutation of sorted array and quickSort

```
/*
 *Permutation method returns the indexes in order based on the
 * orginal random array with the
 *  we can create a newly sorted array using the permutation
 * with only a single pass O(N) but of course the original
 * method for finding the permutation takes O(n^2)
 */

static int[] permutation(int[] random){
  int[] original = random;
  int[] permutation = new int[random.length];
  int[] sorted = quickSort(random, 0, random.length -1);


  for (int i = 0; i < random.length; i++){
    for (int j = 0; j < random.length; j++){
      if (original[i] == sorted[j]){
        permutation[i] = Math.abs(j - i);
      }
    }
  }
  return permutation;

}
```

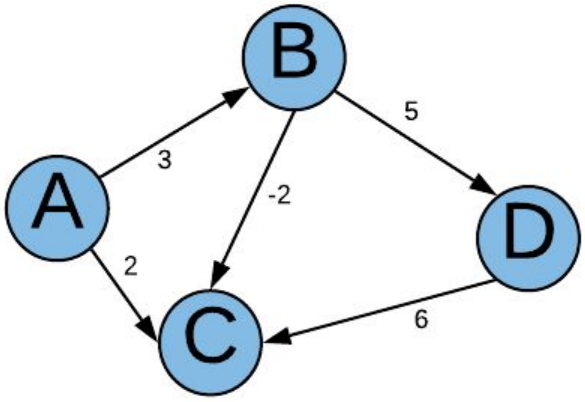
3) 9.1
   One possible topological sort:
   s G D A B H E I F C t

4) 9.5
   a) Shortest path from A to each node:
      i) A -> B
      ii) A -> C
      iii) A-> C-> D
      iv) A -> B -> G -> E
      v) A -> B -> G-> E-> F

vi) A -> B -> G
vii) A -> C -> D -> A

b) Shortest unweighted path from B to each node:
i) B-> C -> D -> A
ii) B -> E ->D -> A -> B
iii) B-> C
iv) B-> E -> D
v) B -> E
vi) B-> E -> F
vii) B-> G

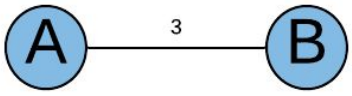
5) 9.7 Dijikstra's shortest path Algorithm does not work with negative weights considering the algorithm looks for the shortest weight to the destination node from the current node. The algorithm does not look ahead and check the rest of the path. Therefore if there is a negative weight.

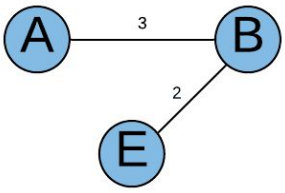


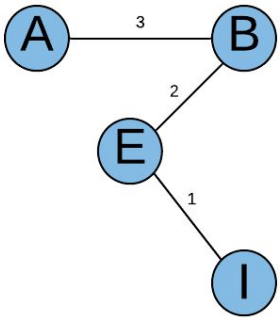
For example in this graph the Dijikstra's shortest path algorithm would produce the shortest path from A to C as A-> C(weight 2) but the actual weighted shortest path would be A->B->C (weight 1).

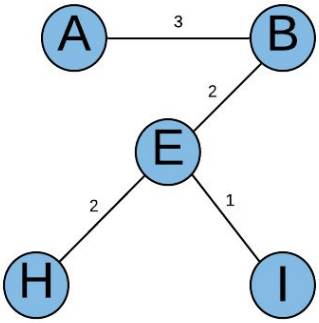
6) 9.15

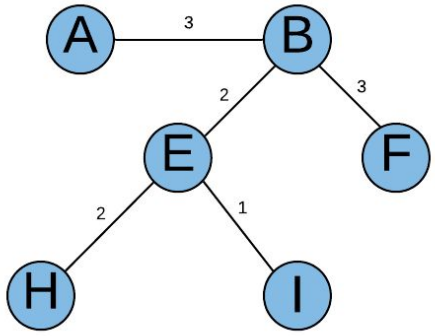
a) Minimum spanning tree using Prim's algorithm:

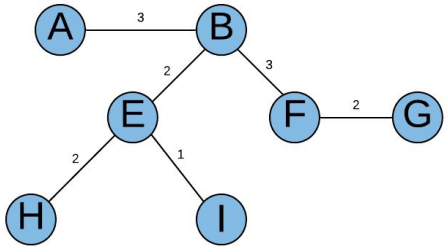


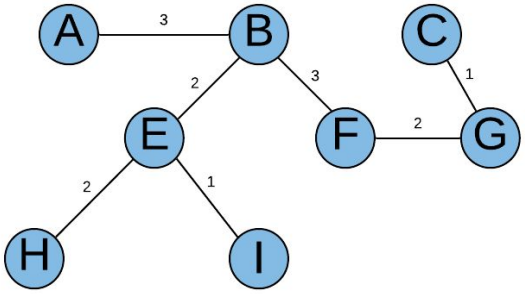


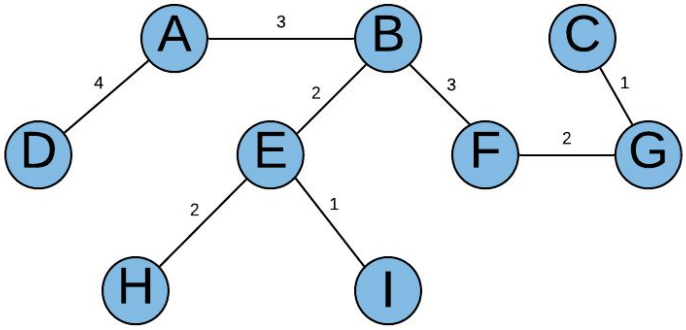


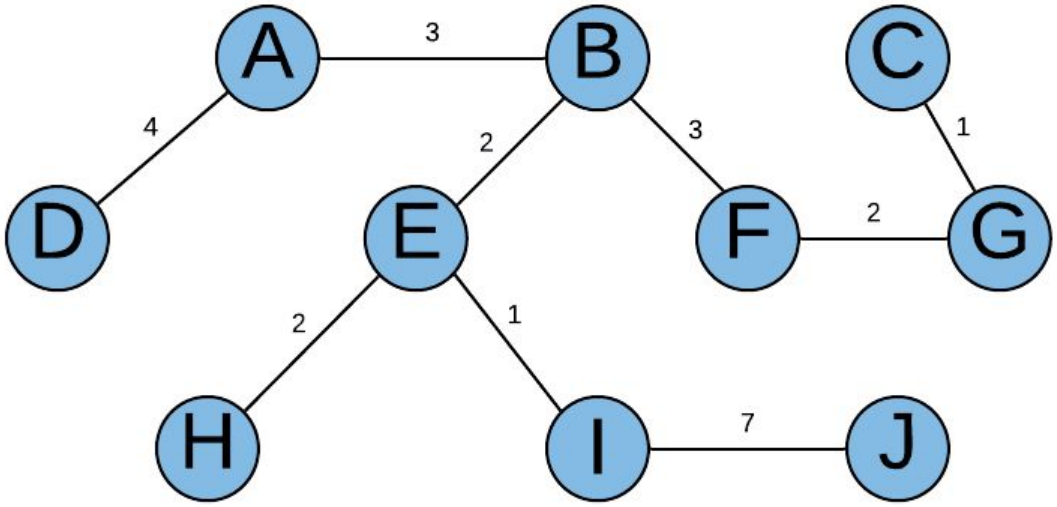



A
3
B
2
3
E
F
2
1
H
I

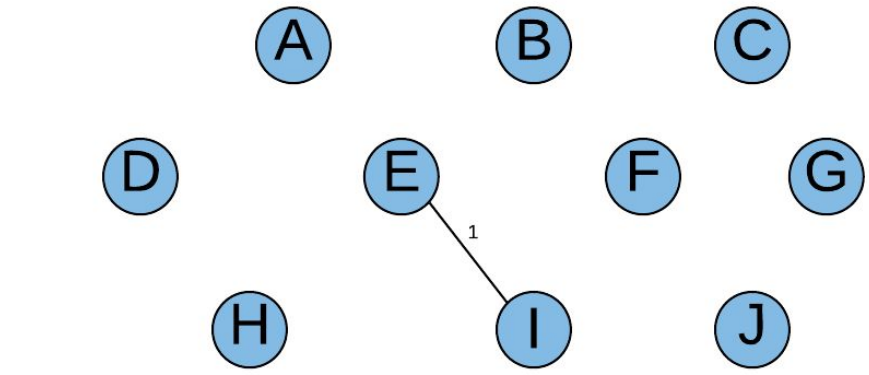

A
3
B
2
3
E
F
2
G
2
1
H
I

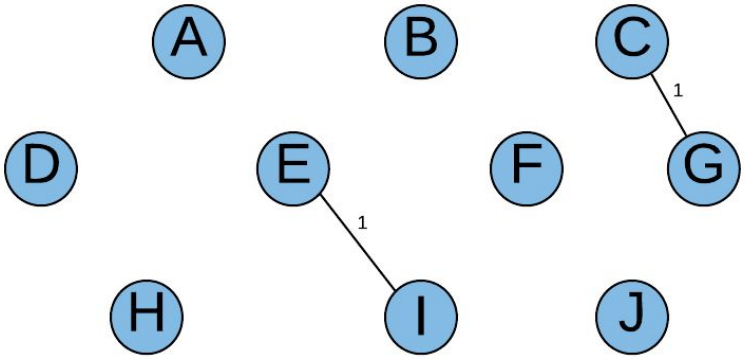

A
3
B
C
2
3
1
E
F
2
G
2
1
H
I

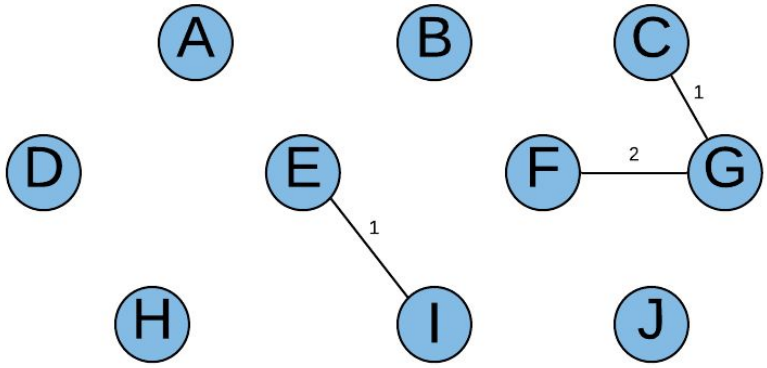

A
3
B
C
4
2
3
1
D
E
F
2
G
2
1
H
I

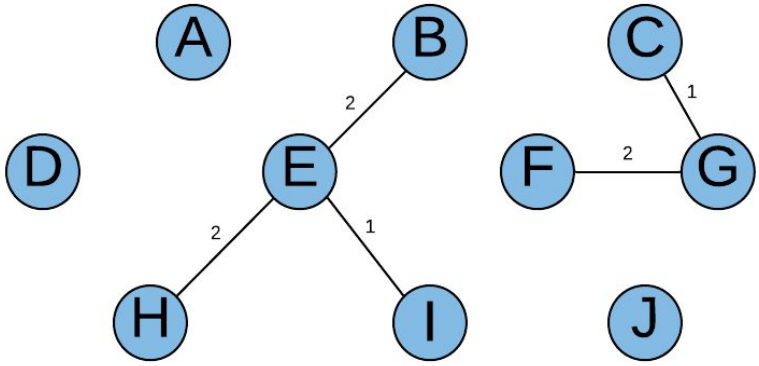


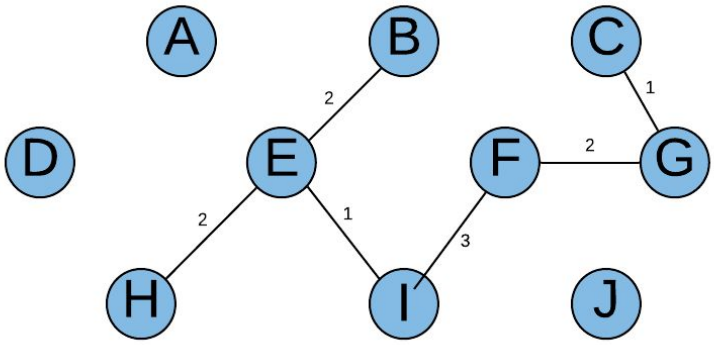
Kruskal's Algorithm:



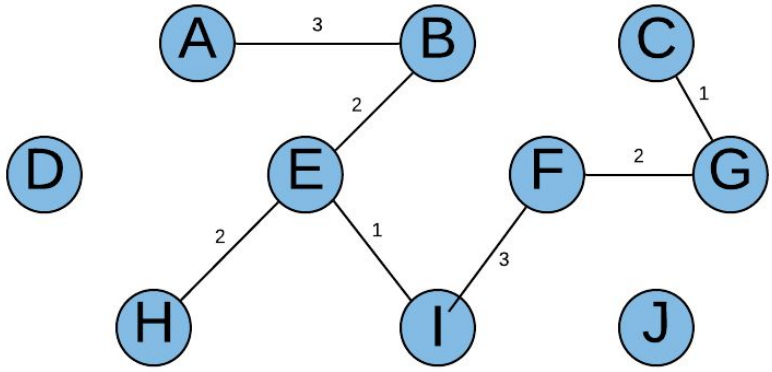


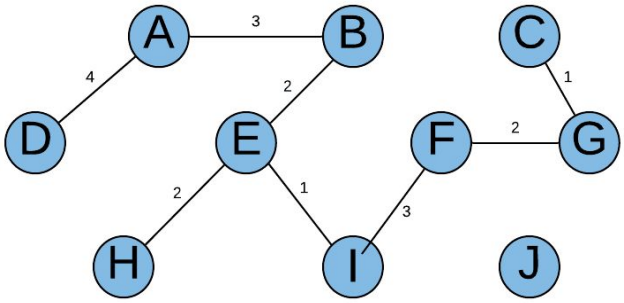


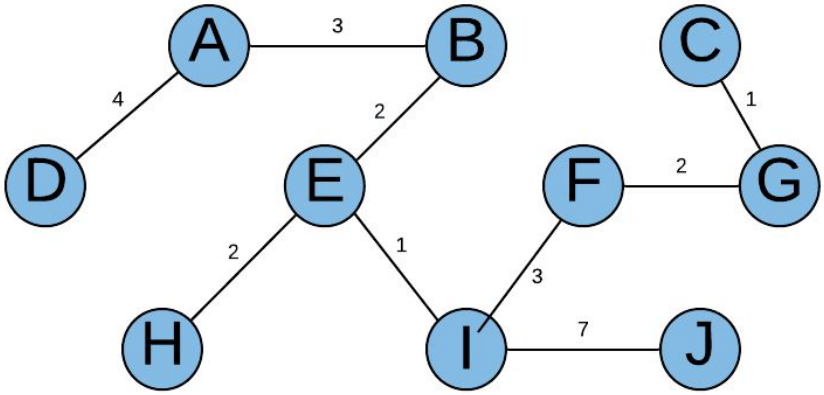




** Not unique Spanning tree compared to Prim's algorithm ie path from node F to I







b) The Spanning tree for this graph is not unique considering at node F it has two seperate possible paths to either I or B due to the fact that both have a weight of three. In the first example in Prim's algorithm the B and F are connected but in one version of Kruskal's F and I are connected.

7) Both Prim's and Kruskal's algorithms do work for negative weights. Both algorithms focus on finding the least weight and ensure that each node can reach another. There is no dependency for the weights being, both will ensure that the least weight from the current

node is depicted in the diagram until there is a connection to each node. For example if you were to change each weight to a negative number the order of the spanning trees would be the same.