Homework 9

1. Exercise 11.1-3 (page 393).

Find a trivial lower-bound class for each of the following problems and indicate, if you can, whether this bound is tight.

a. finding the largest element in an array

b. checking completeness of a graph represented by its adjacency matrix

c. generating all the subsets of an n-element set

d. determining whether n given real numbers are all distinct

A:

a. When finding the largest element in an array, you have to traverse through the entirety of the array, making sure that all n elements are indeed processed. If you don't process all of the elements, and one of those end up being the true max, then you are missing information, making the solution incorrect. Then, just one item needs to be returned, whether it is the value of the largest element, or the element itself (as a position). This makes the trivial lower bound in the class of $\Theta(n)$. This seems to be tight, since usual algorithms would just pass through the array once, comparing the elements, making it a linear algorithm
b. The completeness problem has you check if there is an edge between all vertices. If the graph has n vertices, then there are a total of $n*(n - 1) / 2$ vertices for a graph, making the trivial lower bound for this problem in the class of $\Theta(n^2)$. A typical brute force algorithm would just check all the elements until there are no unchecked elements left, or if a missing edge is found between a pair, so this bound is tight
c. For a set with n elements, there will be a total of 2^n subsets for the output, which makes the trivial lower bound for this problem in the class of $\Theta(2^n)$. Then this algorithm is bounded exponentially. This bound is tight
d. The size of the input is n elements. The trivial lower bound is then in the class of $\Theta(n)$. This bound is not tight, however, since the real tight bound is about n log n

2. Exercise 11.3-5 (page 410).

Design a polynomial-time algorithm for the graph 2-coloring problem: determine whether vertices of a given graph can be colored in no more than two colors so that no two adjacent vertices are colored the same color.

A: As a form of pseudo structured code in C++

This is a BFS implementation

```
Bool twoColorability(graph G, myArray, )
{
    //This array is a coloring mapping, using true as color 1 and false as color 2
    //→ Bool myArray[G.numberofVertices];
    //Linked list which shows adjacency between each vertex
    //→ LinkedList<> adj;
    For (auto: every v in G.V)
    {
        If (myArray[v] == null)        // This vertex has not been visited
        {
            Queue q;       // Create an empty queue
            myArray[v] = true;    // Set it as the first color
            q.enqueue(v);         // Add the vertex to the queue
        }
        While (q.isEmpty() == false)         // While the queue is not empty
        {
            Vertex u = q.dequeue();
            For (auto: every vertex w in adjArr[u])
            {
                If (myArray[w] == null)        // w has not been visited
                {
                    If (myArray[u] = true)          // This vertex is color 1
                    {
                        myArray[w] = false;  // Make the neighbor color 2
                    } else  // This case is the opposite

                    {

                        myArray[w] = true;   // Make the neighbor color 1

                    }

                    q.enqueue(w);         // Add this vertex to the queue

                } else if (myArray[w] == myArray[u])        // Two connected vertices have the same color

                {

                    Return false; // Fails 2 colorability

                }

            }

        }

    }

    //Passes all of the checks and processing

    Return true;

}
```

3. Show that the following problem is NP-complete, i.e., show that it is in NP as well as NP-hard:
    - input: a graph *G*;
    - output: *true* iff the vertices of G can be colored using 5 colors (numbered 1, 2, 3, 4, 5) such that if { i , j } is an edge, then the colors of i and j must be different.

A:

Proof of NP: Supply a solution checking algorithm that runs in polynomial time

```
bool check-5-colorability-sol(graph G, map<vertex, RGBYP> m)
```

```
{
      for(auto e: G.E)
      {
      vertex v1 = e.first, v2 = e.second;
      if (m[v1] == m[v2])
      return false;
      }
      return true;
}
```
Runs in polynomial time Θ(|E|) in the size of the input, which is in the class of Θ(n)

Proof of NP hardness: Take a known NP hard problem and reduce it to our problem

Use the 5 SAT problem and reduce it to the 5 colorability problem

```
bool 4-color-solver(graph G)
{
      graph H = f(G);
      return 5-color-solver(H);
}

graph f(graph G)
{
      graph H = G;
      H.add_vertex("nv");
      for (auto v: G.V)
      H.add_edge(v, "nv");
      return H;
}
```
As we saw in class, the 4 colorability problem is NP hard, because the 3 colorability problem, which is already known to be in NP, was reduced to the 4 colorability problem. Similarly, since we know that the 4 colorability problem is NP hard, we use the same algorithm for the 5 colorability problem, in which the 4 colorability problem reduces to the 5 colorability problem, proving that it is also NP hard

We proved that this problem is both in NP and is NP hard, and thus the 5 colorability problem is NP complete

4. (Practice problems; do not turn in)
    - Exercise 11.1.7 (page 394).
    - Exercise 11.3.6 (page 410).