Homework 4

1. Using the quicksort algorithm discussed in class, show how quicksort would work on the following array:

   ```
   [6  10  2  12  3  14  11  5]
   ```

   In particular, explicitly list **each** call to `quicksort` and `partition` that is made, along with the arguments. Also show the contents of the entire array at the end of every call to `partition`. Make it clear which elements are swapped and when. *Sanity check: there should be 4 calls to `partition`. There will be more calls to `quicksort`, but many of them are "dead-ends" (that is, will end immediately because they will trigger the base case).*

   Do this using a recursion diagram similar to the one we used for mergesort, showing the order that calls are made. Here's a start:

   ```
   quicksort(array, 0, 7)
       pivot is 6
       swapping positions 1 (10) and 7 (5)
       swapping positions 3 (12) and 4 (3)
       swapping positions 0 (6) and 3 (3)
       array is now [3  5  2  6  12  14  11  10]
       quicksort(array, 0, 2)
           etc ….
   ```

2. We learned that on average, quicksort is an O(n log n) sort, however, it can degrade to O(n^2) in certain cases. Here, we will study what happens in one of those cases, specifically when quicksort is asked to sort an array that is already sorted.

   a. Suppose quicksort is asked to sort an array with a single number in it. For instance, suppose we call quicksort([0], 0, 0). How many total calls are there to `quicksort` (including the initial call and any recursive calls?) How many total calls are there to `partition`?

   b. Re-answer the two questions in part (a) for an array that is already sorted that has a length of 2, such as [0, 1].

   c. Re-answer the two questions in part (a) for an array that is already sorted that has a length of 3, such as [0, 1, 2]. Hopefully you see a pattern now.

   d. Generalize your answers for an already-sorted array of length n (total number of calls to quicksort, number of calls to partition). Show your work, and give an exact answer in terms of n, not big-oh.

3. Heaps

Refresher on heaps (we will assume a min-heap for this problem)

- Recall that we think of a min-heap as a binary tree, where every node must be smaller than its children. This rule is applied recursively throughout the tree. Furthermore, unlike other binary-tree-based data structures, we *actually* store the items in the heap in an array, indexed starting at 1. In this way, the children of the node at index *i* in the array are located at indices *i* and i+1. Equivalently, the parent of node at index *k* in the array is located at index *k*/2 (rounded down if *k* is odd).
- Additionally, the binary tree that represents a heap must be complete: meaning every level of the tree is full except possibly the last level, and if the last level is not full, all nodes in the last level must be as far to the left as possible. This property guarantees that the array representation of the heap will not have any gaps in it.
- We have the heap-up and heap-down algorithms (see class handout) that move items around in the heap if the item in a node is too big or too small. These algorithms form the basis of the algorithms to insert and remove an item from a heap:
- To insert an item into the heap, put it at the end of the heap (end of the array, or equivalently the "last" spot in the tree, top to bottom, left to right). Then use the heap-up algorithm to move it up the tree until it's in a valid location with respect to its parent and children.
- To delete an item from a heap (you will normally only delete the smallest item in a min-heap, which will be the root), replace the value at the root node with the last item in the heap (rightmost leaf at deepest level = last item in array), then use the heap-down algorithm to move the new root down the tree until it's in a valid location with respect to its parent and children.

Problems:

a. Add the following items in order to an empty min-heap, drawing the heap as a *binary tree* at the end of each insertion: 5, 2, 6, 4, 1. [So you should have 5 pictures in total].

b. Draw the heap at the end of part (a) as an array (remember, heaps stored in arrays are indexed starting from 1).

c. Suppose we start with the following heap:

```
      1
    /    \
   2       3
 / \     / \
4   9  7    10
```

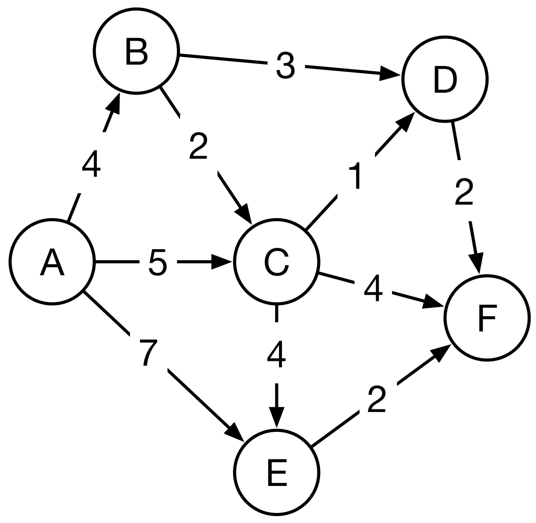
Draw the heap after deleting 1.

d. Continuing from part (c), draw the heap from the end of that problem after deleting the new root [which should be 2, since it's the next-biggest element].

e. Suppose I want to insert a new number into a min-heap that will force heap-up to take as long as possible to run. What number should I insert and how long will it take (in terms of big-oh) for heap-up to run in this situation? (You may assume that you have prior knowledge of all the values in the heap.)

f. Suppose I want to insert a new number into a min-heap that will force heap-up to take the shortest amount of time possible to run. What number should I insert and how long will it take (in terms of big-oh) for heap-up to run in this situation? (You may assume that you have prior knowledge of all the values in the heap.)

4. Suppose you have the following graph:



Use Dijkstra's algorithm to find the shortest path from A to F.

Fill in the following tables (or rewrite them on your own paper), showing the dist and prev tables, and the order that vertices are visited (this is not the final path, this is the order that the vertices come off the priority queue).

When you update an item in the tables, lightly cross out the old item so I can read it.

Order that we visit vertices: ____

Dist table		Prev table
dist[A]:	____________________	prev[A]: _________________________
dist[B]:	____________________	prev[B]: _________________________
dist[C]:	____________________	prev[C]: _________________________
dist[D]:	____________________	prev[D]: _________________________
dist[E]:	____________________	prev[E]: _________________________
dist[F]:	____________________	prev[F]: _________________________

Final shortest path distance: _____________

Final shortest path: ____

5. Many real-world (and semi-real-world) situations and problems can be turned into graphs. This is often done because then many standard graph algorithms (e.g., Dijkstra's algorithm) can be applied to these graphs to solve the problem. For instance, we did this in class with the water jug problem.

   Consider the towers of Hanoi problem (you probably saw this in COMP 142 or some other CS class, but here's a review). There are many made-up stories about the origin of this puzzle, but supposedly in Hanoi there is a temple or monastery with three tall posts. At some point in history, 64 golden disks of different sizes were placed on one of the posts, each disk rests on the disk of the next largest size. Supposedly, the priests of the temple or the monks of the monastery must transfer all the disks from the first post to the third post, but this must be done in such a way that a disk never rests on a disk below it of a smaller size, and the priests can only move one disk at a time.

   Here is a visual demo: https://www.mathsisfun.com/games/towerofhanoi.html

   This puzzle can be visualized as a graph. Let's simplify the game to use just two disks, of size 1 and 2. Label our three posts as A, B, and C. The puzzle starts with both disks on post A, and the player must move both disks to post C. The vertices of our graph will represent the current game situation: specifically, which posts hold which disks. So the starting vertex will be (1 and 2, x, x) indicating post A has disk 1 on top and disk 2 on the bottom, and posts B and C are empty. The ending vertex will be (x, x, 1 and 2), meaning post C has disk 1 on top and disk 2 on the bottom, and posts A and B are empty. [*Do not be thrown off by the vertices having more complicated representations than just letters or numbers.]* The edges of our graph will represent the ability for a player to move a disk from one post to another (obeying the rule that they can never put a bigger disk on top of a smaller disk [in this case can't put disk 2 on top of disk 1]). So for instance, the starting vertex (1 and 2, x, x) would have edges going to vertices (2, 1, x) [moving disk 1 to post B] and (2, x, 1) [moving disk 1 to post C].

   (a) Draw the complete graph representing this puzzle. Hints: this should be an *undirected* graph, since each move in the puzzle can be "undone" to go back to the previous situation (moving a disk back from where it was just moved to). This graph will also be *unweighted*, since we don't have the concept of moves costing different amounts in this puzzle. This graph will have 9 total vertices.

   (b) Find the shortest path from the starting vertex to the ending vertex. You do not need to use Dijkstra's algorithm to find this; you can do it just by examining the graph. List the vertices that form the shortest path.