The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework06.html`

1. Describe a method for maintaining the median of an initially empty set, S, subject to an operation, insert(x), which inserts the value x, and an operation, median(), which returns the median in S. Each of these methods should run in at most $O(\log n)$ time, where n is the number of values in S.

   Solution. There are two solutions. This one uses a min-heap and a max-heap.

```
class median_data_structure:
    def __init__(self):
     # constructor
     self.lo = max_heap() # empty
     self.hi = min_heap() # empty

    # we maintain the invariant:
    # self.lo.size() - self.hi.size() is always equal to 0 or 1

    def median(self):
      # median-structure is empty
      if self.lo.size() == 0:
        return None
      # median_data_structure has an odd number of elements
      if self.lo.size() > self.hi.size()
         return lo.max()
      # median-structure has an even number of elements
      return (self.lo.max() + self.hi.min())/2

    def insert(self, x):
      # idea: insert x into appropriate heap.
      # If the invariant is broken as a result, rebalance by
      # transferring an excess element into the other heap.
      if self.median() is None or x <= self.median():
        self.lo.insert(x)
        if self.lo.size() - self.hi.size() == 2:
          y = self.lo.remove_max()
          self.hi.insert(y)
      else:
        self.hi.insert(x)
        if self.lo.size() - self.hi.size() == -1:
          y = self.hi.remove_min()
          self.lo.insert(y)
```

   If we only have min-heaps available, we can define a max-heap in terms of a min-heap as follows:

```
class max_heap:
    def __init(self)__:
        # constructor
        self.h = min_heap() # empty

    def insert(self, x):
      self.h.insert(-x)

    def max(self):
      return - self.h.min()

    def remove_max(self):
      return - self.h.remove_min()
```

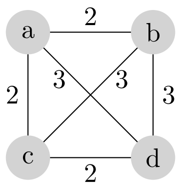
The second solution uses a balanced binary search tree in which each node contains an additional piece of data called size, which stores the size of the subtree at that node. (Technically this new data structure is called an order-statistic tree, where order is what we call size).

The procedure for finding the median is similar to quickselect for finding the k-th sorted item. Given k, start at the root, visit each of its children to find the size of their subtrees. If k is equal to the size of the left subtree, return that node. If k is strictly less than the size of the left subtree, go to the left subtree and repeat. Otherwise, decrease k by the size of the left subtree, go to the right subtree and repeat. To find the median, if n is odd, do this procedure using $\lfloor n/2 \rfloor$. If n is even, do this procedure twice with $k = n/2$ and $k = n/2 + 1$ and take their average.

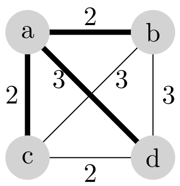
To insert x, do a typical insertion of the binary search tree. Keep a pointer to to the inserted node before all the necessary rotations are done. Then use the pointer to go to the node. Update the size of that node by summing the size of its two children, plus 1 (for the node itself). Then, go to the node's parent and update its size. Repeat this update all the way up to the root of the tree. •

2. Give an example of a weighted, connected, undirected graph, G, such that the minimum spanning tree for G is different from every shortest-path tree rooted at a vertex of G.

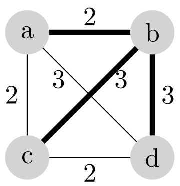
   Solution. Here is an example.



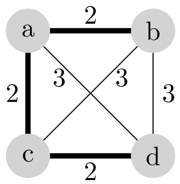
original graph



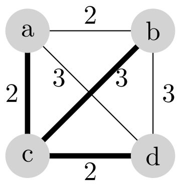
shortest path tree rooted at a



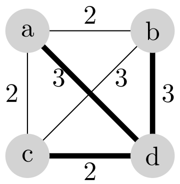
shortest path tree rooted at b



minimum spanning tree (the only one in this graph)



shortest path tree rooted at c



shortest path tree rooted at d

●

3. (a) Write an algorithm in code or pseudocode for solving the longest increasing subsequence problem introduced in class, but instead of returning the length of the subsequence, return an actual subsequence of that length.

Solution.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class LongestIncreasingSubsequence {
   public static List<Integer> longestIncreasingSubsequence(List<Integer> list) {
      List<Integer> bestList = new ArrayList<>();
      for (int i = 1; i <= list.size(); i++) {
         List<Integer> LISUsingLastElement = longestIncreasingSubsequenceUsingLastElement(list.subList(0, i));
         if (bestList.size() < LISUsingLastElement.size())
            bestList = LISUsingLastElement;
      }
      return bestList;
   }

   public static List<Integer> longestIncreasingSubsequenceUsingLastElement(List<Integer> list) {
      List<Integer> bestSublist = new ArrayList<>();
      bestSublist.add(list.get(list.size() - 1));
      for (int i = 1; i <= list.size() - 1; i++) {
         List<Integer> sublist = longestIncreasingSubsequenceUsingLastElement(list.subList(0, i));
         boolean canExtendSublist = sublist.get(sublist.size()-1) < list.get(list.size()-1);
         if (canExtendSublist && bestSublist.size() < sublist.size() + 1) {
            bestSublist = sublist;
            bestSublist.add(list.get(list.size()-1));
         }
      }
      return bestSublist;
   }

   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>(Arrays.asList(1, 8, 2, 7, 3, 4, 1, 6));
      System.out.println("input:␣␣" + list);
      System.out.println("output:␣" + longestIncreasingSubsequence(list));
   }
}
```

Sample console output:

```
input: [1, 8, 2, 7, 3, 4, 1, 6]
output: [1, 2, 3, 4, 6]
```

•

(b) Write an algorithm in code or pseudocode for solving the maximumSubsetSum problem from Homework 4, but instead of returning the maximum sum, return an actual subset of the input numbers (as a list or other appropriate data structure) that achieves this maximum value.

Solution.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MaximumSubsetSum {
  private static List<Integer> maxSubsetSum(List<Integer> list, int target) {
     if (list.isEmpty())
        return new ArrayList<>();

     List<Integer> bestSubsetNotUsingFirst = maxSubsetSum(list.subList(1, list.size()), target);
     if (target < list.get(0))
        return bestSubsetNotUsingFirst;

     List<Integer> bestSubsetUsingFirst = maxSubsetSum(list.subList(1, list.size()), target - list.get(0));
     bestSubsetUsingFirst.add(list.get(0));
     int bestSumNotUsingFirst = bestSubsetNotUsingFirst.stream().mapToInt(Integer::intValue).sum();
     int bestSumUsingFirst = bestSubsetUsingFirst.stream().mapToInt(Integer::intValue).sum();
     return bestSumNotUsingFirst > bestSumUsingFirst ? bestSubsetNotUsingFirst : bestSubsetUsingFirst;
  }

  public static void main(String[] args) {
     List<Integer> list = new ArrayList<>(Arrays.asList(5, 13, 9, 8, 2));
     System.out.println(list + ",␣6:␣maxSubsetSum:␣" + maxSubsetSum(list, 6));
     System.out.println(list + ",␣8:␣maxSubsetSum:␣" + maxSubsetSum(list, 8));
     System.out.println(list + ",␣13:␣maxSubsetSum:␣" + maxSubsetSum(list, 13));
     System.out.println(list + ",␣14:␣maxSubsetSum:␣" + maxSubsetSum(list, 14));
     System.out.println(list + ",␣15:␣maxSubsetSum:␣" + maxSubsetSum(list, 15));
  }
}
```

Sample console output:

```
[5, 13, 9, 8, 2], 6: maxSubsetSum: [5]
[5, 13, 9, 8, 2], 8: maxSubsetSum: [8]
[5, 13, 9, 8, 2], 13: maxSubsetSum: [8, 5]
[5, 13, 9, 8, 2], 14: maxSubsetSum: [9, 5]
[5, 13, 9, 8, 2], 15: maxSubsetSum: [2, 8, 5]
```

●

4. How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Use the recursion-tree method to solve this problem.

```
void f(n) {
   if (n > 1) {
      print("still␣going");
      f(n/3);
      f(n/3);
   }
}
```

Solution. This function is recursive, and specifically, in the style of a divide-and-conquer algorithm. We first notice that every call to this function, regardless of input value n prints at most a single line of text. In addition, there are two recursive calls, each of which takes input $n/3$.

Therefore, at level i of the recursion tree, where i ranges from 0 to $\log_3 n$ (since the input is divided by 3 at each level and we stop recursing when $n \leq 1$), we have

- an input of value $n/3^i$ (although this information is not particularly helpful in the case where the amount of work done is constant)
- one line printed per node or call
- 2^i nodes
- and therefore the total number of lines printed by the recursive calls at this level is 2^i.

Summing over all levels, we find that the total number of lines printed is

$$\sum_{i=0}^{\log_3 n} 2^i = \frac{2^{\log_3 n+1} - 1}{2-1} = 2n^{\log_3 2} - 1$$

or $\Theta(n^{\log_3 2})$, which is $\Omega(\sqrt{n})$ and $O(n)$ since $\log_3 2 \simeq 0.6$. •