The homework assignment is available at:
http://www.jennylam.cc/courses/146-s17/homework04.html

1. Here is a working Java solution which runs in linear time in the total length of the input lists.

```
/**
 * Give a list of integers, is there a subset of these integers whose sum is
    equal to the specified target value,
 * with the additional condition that all elements that make up the sum must be
    odd?
 * An empty set of integers is considered to have a sum of 0, and have only odd
    integers (a vacuously true statement)
 * Solve this problem without using any loops.
 *
 * @return true if such a subset exists
 */
public static boolean oddSubsetSum(List<Integer> list, int target) {
    if (list.isEmpty())
        return target == 0;
    List<Integer> restOfTheList = list.subList(1, list.size());
    boolean canMakeSubsetUsingFirstElement = oddSubsetSum(restOfTheList, target
        - list.get(0));
    boolean canMakeSubsetWithoutFirstElement = oddSubsetSum(restOfTheList,
       target);
    return list.get(0) % 2 == 0 ? canMakeSubsetWithoutFirstElement
            : canMakeSubsetUsingFirstElement ||
               canMakeSubsetWithoutFirstElement;
}

/**
 * Given a list of integers, is there a subset of exactly 8 integers whose sum
    is equal to the specified target
 * value? If the list has 7 or fewer numbers, the answer is no.
 * Solve this problem without using any loops.
 * @return true if such a subset exists
 */
public static boolean subsetOf8Sum(List<Integer> list, int target) {
    return subsetOfNSum(list, 8, target);
}

public static boolean subsetOfNSum(List<Integer> list, int n, int target) {
    if (list.size() < n)
        return false;
    if (n == 0)
        return target == 0;
    List<Integer> restOfTheList = list.subList(1, list.size());
    boolean canMakeSubsetUsingFirstElement = subsetOfNSum(restOfTheList, n - 1,
        target - list.get(0));
    boolean canMakeSubsetWithoutFirstElement = subsetOfNSum(restOfTheList, n,
       target);
    return canMakeSubsetUsingFirstElement || canMakeSubsetWithoutFirstElement;
}
```

```
/**
 * Given a non-empty list of integers, is there a non-empty subset whose sum is
     equal to the specified target?
 */
public static boolean nonEmptySubsetSum(List<Integer> list, int target) {
    return nonEmptySubsetSum(list, target, true);
}

public static boolean nonEmptySubsetSum(List<Integer> list, int target, boolean
    mustPickSomething) {
    if (list.size() == 1)
        return mustPickSomething ? target == list.get(0) : target == list.get
           (0) || target == 0;
    List<Integer> theRest = list.subList(1, list.size());
    boolean canMakeSubsetUsingFirstEntry = nonEmptySubsetSum(theRest, target -
       list.get(0), false);
    boolean canMakeSubsetWithoutFirstEntry = nonEmptySubsetSum(theRest, target,
        mustPickSomething);
    return canMakeSubsetUsingFirstEntry || canMakeSubsetWithoutFirstEntry;
}

/**
 * Given a list of doubles, is there a (possibly empty) subset of doubles whose
     product is in the range between the
 * give lower and upper bound (bounds included)?
 * The product of an empty set is considered to be 1.
 * The answer is no if the value given by the parameter upperBound is strictly
    less than the parameter lowerBound.
 * Solve this problem without using any loops.
 * @return true if such a subset exists
 */
public static boolean subsetProduct(List<Double> list, double lowerBound,
   double upperBound) {
    if (list.isEmpty())
        return lowerBound <= 1 && 1 <= upperBound;
    Double first = list.get(0);
    List<Double> restOfTheList = list.subList(1, list.size());
    boolean canMakeSubsetUsingFirstElement =
            first > 0 ? subsetProduct(restOfTheList, lowerBound / first,
               upperBound / first) :
                    first < 0 ? subsetProduct(restOfTheList, upperBound / first
                       , lowerBound / first) :
                            false;
    boolean canMakeSubsetWithoutFirstElement =
            subsetProduct(restOfTheList, lowerBound, upperBound);
    return canMakeSubsetUsingFirstElement || canMakeSubsetWithoutFirstElement;
}
```

```
/**
 * Given a list of non-negative integers and a positive upperBound, what is the
     maximum value of any subsets of the
 * set that is less than or equal to the specified upperBound?
 *
 * For example
 * {6, 7, 8}, 18 returns 15
 * {}, 10 returns 0
 *
 * @param list
 * @param upperBound
 * @return
 */
public static int maxSubsetSum(List<Integer> list, int upperBound) {
    if (list.isEmpty())
        return 0;
    List<Integer> restOfTheList = list.subList(1, list.size());
    if (list.get(0) <= upperBound) {
        int bestSumWithFirstElement = maxSubsetSum(restOfTheList, upperBound -
            list.get(0)) + list.get(0);
        int bestSumWithoutFirstElement = maxSubsetSum(restOfTheList, upperBound
            );
        return Integer.max(bestSumWithFirstElement, bestSumWithoutFirstElement)
            ;
    } else {
        return maxSubsetSum(restOfTheList, upperBound);
    }
}
```

2.

(a) Implement a solution which uses the sorted dictionary ADT. You may give your solution in code or in pseudocode.

Solution. Here is a Java solution built from scratch. We use a Java NavigableMap interface, which is a SortedMap interface which has, in addition to a firstKey and lastKey, an iterator which traverses the map in key order. The NavigableMap maps a page count to a page. To handle the possibility that more than one page may have the same key, we let the value be a set of all pages with the same count. An alternative and more natural solution is to use a SortedMultiMap which implements this idea (one is available in the Guava library).

In addition, we need a regular map which maps pages to counts, and thus reverses the NavigableMap. This allows us to locate the key of the NavigableMap which maps to a particular page without having to traverse the entire data structure.

```
import java.util.*;

public class KCommonSortedMap implements KCommon {
    private NavigableMap<Integer, Set<String>> sortedCounts;
    private Map<String, Integer> counts;

    public KCommonSortedMap() {
        sortedCounts = new TreeMap<>();
        counts = new HashMap<>();
    }

    @Override
    public void add(Entry p) {
        int count = counts.containsKey(p.page) ? counts.get(p.page) + 1 : 1;
        counts.put(p.page, count);
        removeFromSortedCounts(count - 1, p.page);
        addToSortedCounts(count, p.page);
    }

    private void removeFromSortedCounts(int count, String page) {
        if (count < 0)
            return;
        Set<String> oldSet = sortedCounts.get(count);
        oldSet.remove(page);
        if (oldSet.isEmpty())
            sortedCounts.remove(count);
    }

    private void addToSortedCounts(int count, String page) {
        if (!sortedCounts.containsKey(count)) {
            sortedCounts.put(count, new HashSet<>());
        }
        sortedCounts.get(count).add(page);
    }

    @Override
    public List<String> common(int k) {
        List<String> result = new ArrayList<>();
        for (int count : sortedCounts.descendingKeySet()) {
            for (String page : sortedCounts.get(count)) {
                result.add(page);
                if (result.size() == k)
                    return result;
            }
        }
        return result;
    }
}
```

•

(b) Assume that the sorted dictionary is implemented with a balanced binary search tree. At the point in the log file at which r distinct pages have been encountered, how long does it take to make a call to add()? How long does it take to make a call to common()?

Solution. A call to add() consists of one update to the Map (which is O(1) with a hash table), and at most one insertion and one deletion and a search to the NavigableMap, which is $O(\log r)$ with a balanced binary search tree. Therefore add() is $O(\log r)$.

Traversing a balanced binary search tree, which is done through a single call to max() and successive calls to predecessor(), and conveniently implemented through an iterator in the NavigableMap interface, can be done in $O(\log r)$ per operation. Therfore, common(k) takes $O(k \log r)$ •

(c) Implement a solution which uses the (non-sorted) dictionary ADT.

Solution. Here is a Java solution.

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class KCommonMap implements KCommon {
    private Map<String, Integer> counts;

    public KCommonMap() {
        counts = new HashMap<>();
    }

    @Override
    public void add(Entry p) {
        int count = counts.containsKey(p.page) ? counts.get(p.page) + 1: 1;
        counts.put(p.page, count);
    }

    @Override
    public List<String> common(int k) {
        return counts.entrySet().stream()
                .sorted((i, j) -> -i.getValue().compareTo(j.getValue()))
                .limit(k)
                .map(Map.Entry::getKey)
                .collect(Collectors.toList());
    }
}
```

•

(d) Assume that the dictionary is implemented with a hash table (a dictionary implementation in which all operations take O(1) time). At the point in the log file at which r distinct pages have been encountered, how long does it take to make a call to add()? How long does it take to make a call to common()?

Solution. add() takes constant time, and common(k) sorts r elements and takes the k last elements, which takes $O(r \log r)$ time. •

(e) Under what conditions is the solution in (a)-(b) preferable to the solution in (c)-(d)? Justify your anwer. (Hint: there are various things to consider. How often does one method need to be called over the other for the balanced binary search tree solution to be considered the more efficient solution? How likely is it to encounter the same page again in the log?)

Solution. Since add() and common(k) cost about the same in the balanced binary search tree implementation, whereas common(k) is much more expensive than add() in the hash table implementation, the balanced binary search tree is more efficient when there are frequent calls to common(k) relative to the number of add(). Specifically, if common(k) is called about as frequently as add(k), say add() and common() calls alternate, then the first implementation is more efficient. N successive add, followed common calls will run in $O(N \log r)$ for the balanced BST implementation, but in $O(Nr \log r)$ for the hash table implementation.

On the other hand, if the calls to common() are relatively infrequent, then the hash table is the better solution. For example, if we have N calls to add() followed by a single call to common(), the balanced BST implementation will run in $O((N+1)\log r) = O(N \log r)$ whereas the hash table implementation will run in $O(N + r \log r)$. The exact point at which the hash table is the better solution is when N satisfies

$$N + r \log r \geq N \log r$$

which is when

$$N \geq \frac{r \log r}{\log r - 1} = \Omega(r).$$

Another aspect to this tradeoff is the size of the data structure relative to the number of add() calls. If the data structure stays bounded in size (say because the same pages reappear throughout the log), then the calls to add() in a balanced BST implementation are not that costly compared to the constant cost of hash table add(). Therefore, even with relatively few calls to common(k), the balanced BST implementation is in this case just as efficient as the hash table. •

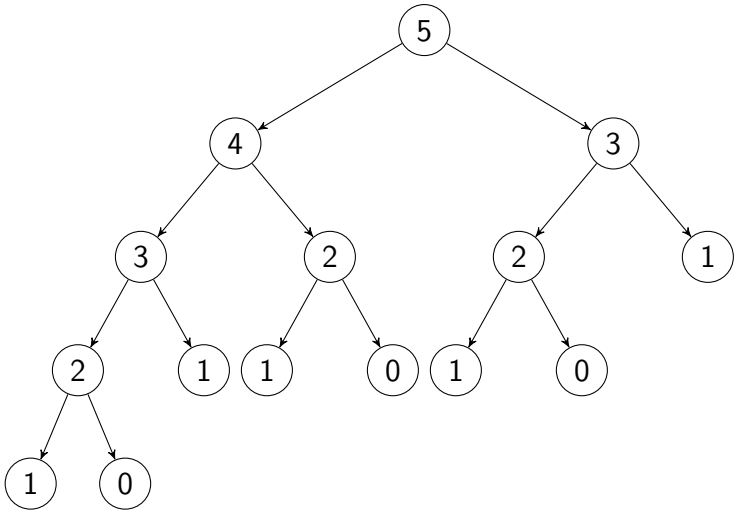
3. Recall that the Fibonacci numbers are defined as follows:

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{else} \end{cases}$$

Consider the following naive implementation for computing the n-th Fibonacci number.

```
int fib(n) {
    if (n == 0 or n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```

(a) Draw the recursion tree corresponding to a call to fib(5)



Solution. •

(b) Possibly with the help of this tree, explain why the naive implementation is not an efficient implementation. In other words, where is there potential savings?

Solution. Several nodes are computed more than once: specifically, fib(2) is recomputed 3 times and fib(3) recomputed twice. •

(c) Propose an algorithm in code or pseudocode to compute the n-th Fibonacci number in $\Theta(n)$ time.

Solution. Here's a recursive implementation that runs in $\Theta(n)$:

```
int fib(n) {
    int fibOfIMinus2 = 0;
    int fibOfIMinus1 = 0;
    int fibOfI = 1;
    for (int i = 1; i <= n; i++) {
        fibOfIMinus2 = fibOfMinus1;
        fibOfIMinus1 = fibOfI;
        fibOfI = fibOfMinus2 + fibOfIMinus1;
    }
    return fibOfI;
}
```

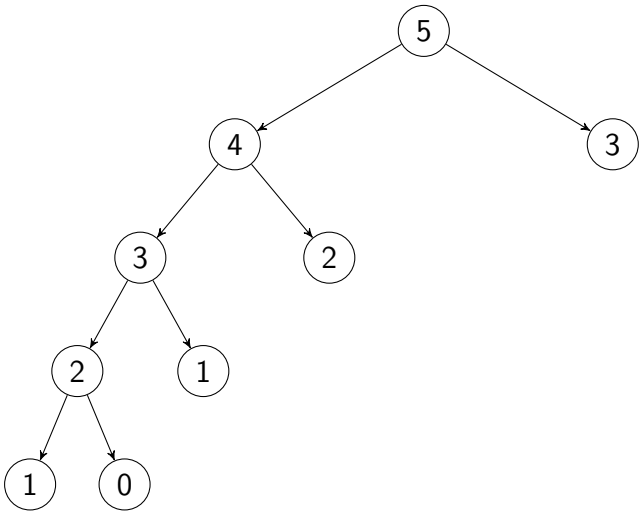
Since this solution is non-recursive, the recursion tree for fib(5) is a single node

•

(d) Propose a second solution in which you start with the naive implementation above, and add to it memoization code to store previously computed values in a dictionary. (Help: review the lecture notes on memoization.) Draw a recursion tree corresponding to a call to your second solution to compute the 5-th Fibonacci number.

Solution. Here is a memoized version of the naive solution

```
Map<Integer, Integer> fibCache = new HashMap<>() // stored in a global
   variable, as a private field in a class, or in a clojure
int fib(n) {
    if (fibCache.containsKey(n))
        return fibCache.get(n);
    int fibn;
    if (n == 0 or n == 1)
        fibn = 1;
    fibn = fib(n - 1) + fib(n - 2);
    fibCache.put(n, fibn);
    return fibn;
}
```

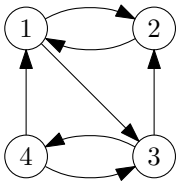


The memoized solution is also $\Theta(n)$. •

4. Recall that an adjacency matrix is a matrix whose rows and columns are indexed by the vertices, with a 1 in row i and column j if there is an edge from vertex i to vertex j.

(a) Draw the graph whose adjacency matrix is below, using circles for the vertices and arrows for the edges. Label each vertex with the number of its row and column.

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Solution. 

•

(b) An adjacency list is an array indexed by the vertices. Each array cell points to a linked list of neighbor of that vertex). How many linked list nodes would be needed to represent the graph from part (a)?

Solution. There are 7 linked list nodes. •

(c) In the dictionary representation, a graph is a dictionary whose keys are vertices and whose values are lists of the outgoing neighbors of each vertex. Read the documentation for the Java Map interface and write Java code that produces the dictionary representation for the same graph. In this code, use the number i to represent the vertex whose adjacencies are given by the i-th row of the matrix. Use the Java Collections idiom for instantiating a collections:

```
InterfaceType<T> dataStructure = new ImplementationType<>();
```

For example, for instantiating a list of integers:

```
List<Integer> list = new ArrayList<>();
```

You may use any valid implementation of the Map interface to instantiate the map, but you may only use methods of the Map interface to interact with it.

Solution.

```
Map<Integer,List<Integer>> graph = new HashMap<>();
graph.put(1, Arrays.asList(2, 3));
graph.put(2, Arrays.asList(1));
graph.put(3, Arrays.asList(2, 4));
graph.put(4, Arrays.asList(1, 3));
```

A call to

```
System.out.println(graph);
```

returns

```
{1=[2, 3], 2=[1], 3=[2, 4], 4=[1, 3]}
```

•

5. Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

(a) The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

Solution. An adjacency list will use 10,000 + 20,000 = 30,000 units of space whereas an adjacency matrix will use $(10{,}000)^2 = 100{,}000{,}000$ units of space, so I would use the adjacency list. •

(b) The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

Solution. An adjacency list will use 10,000 + 20,000,000 $\simeq$ 20,000,000 units of space whereas an adjacency matrix will use $(10{,}000)^2 = 100{,}000{,}000$ units of space, so I would still use the adjacency list (because I want to use as little space as possible). •

(c) You need to answer the query areAdjacent as fast as possible, no matter how much space you use.

Solution. Each list of neighbors in the adjacency list has length at most 10,000, so a query areAdjacent requires a scan through a list that is this long in the worse case. In contrast, the adjacency matrix makes constant time areAdjacent queries. So I would use the adjacency matrix. •