The homework assignment is available at:
http://www.jennylam.cc/courses/146-s17/homework12.html

1. Consider the following problem:

   input: a sequence of n items with weights [w1, w2, ... wn] and dollar amounts [v1, v2, ..., vn], and an integer max_weight output: the maximum dollar amount that can be achieved by selecting a subset of these items whose total weight is less than or equal to maxi_weight Assume that there is at least one item, and that all weights an dollar amounts are non-negative. Unlike in the make-change problem, once an item is chosen, it may not be "re-chosen" again.

   (a) Give a recursive solution for this problem. In other words, express this problem in terms of subproblems. Hint: review the maximum subset sum problem

   Solution. In the base case, we have no items to try, in which case the maximum dollar amount that can be achieved is 0.

   The solution consists of trying out a solution which includes the last item (as long as its weight is less than max_weight), and one which does not include the last item •

   (b) Give a dynamic programming solution to this problem in Java (include the solution as a java file).

   Solution. In the following code, we present a brute-force implementation (bfKnapsack) followed by the dynamic programming implementation (dpKnapsack).

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

public class Knapsack {
   public static void main(String[] args) {
      List<Integer> weights = Arrays.asList(1, 4, 2, 5);
      List<Integer> values = Arrays.asList(3, 7, 2, 6);
      int maxWeight = 5;
      System.out.println(dpKnapsack(weights, values, maxWeight));
   }

   private static int bfKnapsack(List<Integer> weights, List<Integer> values,
                                              int maxWeight) {
      int n = weights.size();
      if (n == 0)
         return 0;

      int bestValueExcludingLastItem = bfKnapsack(weights.subList(0, n-1),
                                       values.subList(0, n-1), maxWeight);

      if (weights.get(n-1) <= maxWeight) {
         int bestValueIncludingLastItem = bfKnapsack(weights.subList(0, n - 1),
                  values.subList(0, n - 1), maxWeight - weights.get(n - 1))
               + values.get(n - 1);
         return Math.max(bestValueExcludingLastItem, bestValueIncludingLastItem);
      }
      return bestValueExcludingLastItem;
   }
```

```
    private static int dpKnapsack(List<Integer> weights, List<Integer> values,
                                  int maxWeight) {
       int n = weights.size();

       // knapsack[i][w] = subproblem with items 0 (inclusive)
       // up to i (exclusive) and maxWeight w.
       int[][] knapsack = new int[n+1][maxWeight+1];

       for (int i = 1; i < n+1; i++) {
          for (int w = 0; w < maxWeight+1; w++) {
             if (weights.get(i-1) <= w)
                knapsack[i][w] = Math.max(knapsack[i-1][w],
                   knapsack[i-1][w-weights.get(i-1)] + values.get(i-1));
             else
                knapsack[i][w] = knapsack[i-1][w];
          }
       }
       return knapsack[n][maxWeight];
    }
}
```

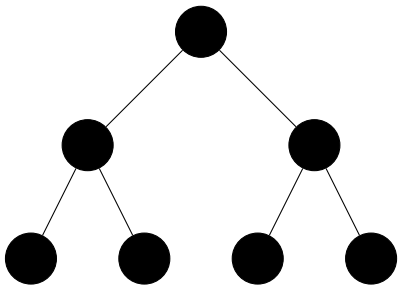
•

(c) What is the time complexity and space complexity of this solution? Justify your answer.

Solution. The time complexity is dominated by the two nested for-loops and is $O(nw)$ where n is the number of items to choose from and w is max_weight. The space complexity is dominated by the size of the table, which is $O(nw)$. •

2. (a) Is there a (min-)heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order?

Solution. Yes, the following one:



•

(b) How about an in-order traversal?

Solution. In an inorder traversal, a parent is always listed after its left child. But in a min-heap, a parent is always smaller than its left child. So the parent, which is smaller, is listed after its left child, which is bigger. So an inorder traversal of a min-heap is not sorted (in increasing order). •

(c) How about a post order traversal?

Solution. No, for the very same reason as for inorder traversal. For post-order traversal, you can also replace the word "left" by "right" in the previous argument, and the argument will still be true. •

3. (a) Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (Hint: Execute the usual binary search tree insert and then perform rotations to restore the min-heap order property.)

Solution. Do a regular binary tree insertion, using the item's key to guide in the descent. The inserted node is at a leaf. Assign it a random priority value. Now simulate a sift-up operation using the priority of the node, but instead of actually exchanging the nodes if the priority of the node is less than its parent's (which is what you would do in an actual heap), perform a rotation, which has the benefit of preserving the BST property. Specifically, if the nodes need to be exchanged and the current node is a left child of its parent, perform a right rotation, and vice versa if it's a right child:

```
def treap_insert(item, root):
    node = bst_insert(item, root)
    node.priority = random()
    p = node.parent
    while p is not None and node.priority >= p.priority:
        if node is p.left:
             node.rotate_right()
        else:
            node.rotate_left()
        node, p = node.parent, p.parent
     return node
```

•

(b) Show that the expected running time of TREAP-INSERT is $O(\log n)$.

Solution. As mentioned in the description of this problem, the whole point of this invention of a treap is that augmenting a BST with randomly assigned priority values and preserving the heap property on these priorities results in a tree with the same structure as having inserted values in the order given by the priority. And as pointed out in class, inserting values in a random order leads to a tree of height $O(\log n)$ on expectation.

Now for the analysis of treap-insert(): since the height of the tree is $O(\log n)$ on expectation, the BST-insert() operation and the "sift-up" operation also take $O(\log n)$ on expectation, hence so does the treap-insert() operation. •

4. (based on Goodrich & Tamassia R-8.7) Suppose that the deterministic quicksort algorithm described in class is executed on a sequence with duplicate elements. What happens in the partition step when all the elements of the input sequence are equal?

(a) How many comparisons are done on quicksort if the entire array is the same number?

Solution. We are assuming the version of quicksort in which the partitioning scheme produces two subarrays plus the pivot, and one subarray contains elements less than or equal the pivot, and the other contains elements that are strictly greater than to the pivot. Then the partitioning will be unbalanced: quicksort will do $n-1$ comparisons, and recurse into the lower subarray of size $n-1$. Therefore, the total amount of work will be $\Theta(n^2)$. •

(b) Describe a modified version of quicksort in which arrays of same number run in $n-1$ comparisons. Hint: partition the array into three parts rather than two.

Solution.

```
public static void threeWayQuicksort(int[] a, int i, int j) {
   if (j - i <= 1)
      return;
   List<Integer> pivotRange = threeWayPartition(a, i, j);
   threeWayQuicksort(a, i, pivotRange.get(0));
   threeWayQuicksort(a, pivotRange.get(1), j);
}

public static List<Integer> threeWayPartition(int[] a, int i, int j) {
   int k = i;
   int l = i;
   int m = j;
   int pivot = a[i];

   // invariant:
   // | < pivot | = pivot | unsorted | > pivot |
   // i.........k.........l..........m.........j
   while (l < m) {
      if (a[l] < pivot) {
         swap(a, k, l);
         k++; l++;
      } else if (a[l] == pivot) {
         l++;
      } else if (a[m-1] > pivot) {
         m--;
      } else {
         swap(a, l, m-1);
         m--;
      }
   }
   return Arrays.asList(k, m);
}
```

•