The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework08.html`

1. Suppose that the coins of Combinatoria come in the denominations $d_1, \ldots d_k$. where $d_1 = 1$ and the other values d_i values for a set of distinct integers greater than 1. Given an integer, $n > 0$, the problem of making change is to find the fewest number of Combinatorian coins whose values sum to n.

   (a) Give an instance of the making-change problem for which it is suboptimal to use the standard greedy algorithm, which repeatedly chooses a highest-valued coin whose value is at most n until the sum of chosen coins equal n.

   Solution. Suppose that the denominations are $d_1 = 1, d_2 = 3, d_3 = 4$. Then to make change for $n = 6$, the greedy strategy will return three coins, namely $(4, 1, 1)$, even though this could have been done with only two coins of denomination 3. •

   (b) Describe an efficient algorithm for solving the problem of making change. What is the running time of your algorithm?

   Solution. A common strategy to solve this problem is to use dynamic programming. However, this is not something we've covered yet in this class. The intended solution was to use backtracking to get a brute-force solution and to improve the efficiency with memoization, as hinted in the title description of this problem.

   The key idea is that unlike the greedy strategy which chooses the largest coin denomination less than the value we're making change for, the brute-force strategy tries every possible coin denomination less than the value, and picks the one which results in the fewest number of coins needed to make change for the remainder of the value.

   Finally, (just because we can) we add additional logic to reconstruct a minimum list of coins for the change. Note that this is the exact same technique as the one used to reconstruct a shortest path in Dijkstra with a dictionary that keeps track of the choice made at every step.

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MakeChange {

   private static final Map<Integer, Integer> cacheForOptimalCoinChoice = new HashMap<>();
   private static final Map<Integer, Integer> cacheForOptimalNumberOfCoins = new HashMap<>();

   private static Integer optimalNumberOfCoins(List<Integer> denominations, int value) {
      if (cacheForOptimalNumberOfCoins.containsKey(value))
         return cacheForOptimalNumberOfCoins.get(value);

      if (value == 0) return 0;

      int optimalCoinChoice = 1;
      int optimalNumberOfCoins = value; // one way to make change is entirely with pennies

      for (int coinToTry : denominations) {
         if (coinToTry <= value) {
            int optimalAmountOfChangeForRemainder = optimalNumberOfCoins(denominations, value - coinToTry);
            if (optimalAmountOfChangeForRemainder + 1 < optimalNumberOfCoins) {
               optimalCoinChoice = coinToTry;
               optimalNumberOfCoins = optimalAmountOfChangeForRemainder + 1;
            }
         }
      }

      cacheForOptimalCoinChoice.put(value, optimalCoinChoice);
      cacheForOptimalNumberOfCoins.put(value, optimalNumberOfCoins);
      return optimalNumberOfCoins;
   }

   private static List<Integer> makeChange(List<Integer> denominations, int value) {
      optimalNumberOfCoins(denominations, value);

      List<Integer> change = new ArrayList<>();
      while (value > 0) {
         int coin = cacheForOptimalCoinChoice.get(value);
         change.add(coin);
         value -= coin;
      }
      return change;
   }

   public static void main(String[] args) {
      List<Integer> denoms = java.util.Arrays.asList(1, 3, 4);
      int value = 6;
      System.out.printf("denominations:␣%s,␣value:␣%d,␣change:␣%s", denoms, value, makeChange(denoms, value));
   }
}
```

To discuss the analysis of this algorithm, let n be the value for which we're making change for and let k be the number of coin denominations. Since this is a recursive algorithm, the strategy is going to be add up the amount of work done in all recursive calls.

Notice first that we used memoization. This means that the recursive calls do $O(k)$ work the first time it is called with a particular value for the parameter `value`, but does $O(1)$ work on all subsequent calls on that particular value.

Notice also that the set of values on which recursive calls are made is the set of the values between 0 and the original value, the one on which the first recursive call was made. This is because starting from the initial value, calls are made to smaller values, the smallest jump being by a difference of 1 (for the penny). The decrementing in value stops when the value of 0 is reached.

Finally, notice that there can be at most k calls on each of these distinct values. This is because each call is made from some other larger value when that larger value tried a particular coin denomination to do a recursive call on.

Now, let's put all these observations together. We can break all the recursive calls into two kinds:

- those that do $O(k)$ work. There is going to be one call per distinct value since it's the first call to a value that does the real work. These n calls contribute $O(nk)$.
- those that do $O(1)$ work. There is going to be at most $k-1$ per distinct value since these are the subsequent calls, which can just grab the memoized result. These $(k-1)n$ calls contribute $O((k-1)n \cdot 1) = O(kn)$.

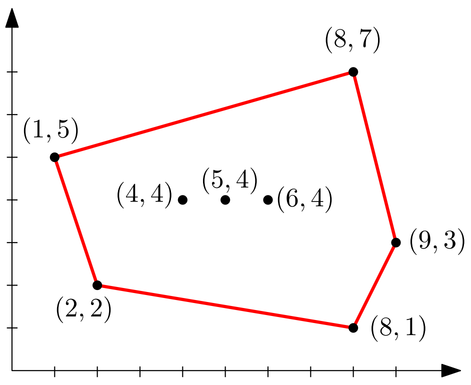
Therefore, this algorithm has running time $O(nk)$.

Final thought, memoization has the effect of pruning the recursion tree, as you may recall from the previous homework exercise which had you draw the fibonacci recursion tree when memoization was applied. Without memoization, we are dealing with a full recursion tree, with n levels, a branching factor of k and $O(k)$ work per call, for a total running time of $\sum_{i=0}^{n} k^i \cdot k = \Theta(k^n)$. •

2. (a) Draw the convex hull of the following set of points:

$$\{(2,2),(4,4),(6,4),(8,1),(8,7),(9,3),(1,5),(5,4)\}.$$

Solution.



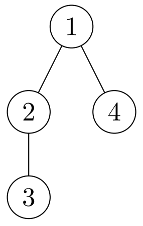
•

(b) Suppose that we run Graham's scan on this set of points. Describe the sequence of pushes and pops that are applied to the stack. Your answer should have the format: push (x1, y1), push (x4, y4), pop, push (x3, y3)...

Solution. push (8,1), push (9,3), push (8,7), push (6,4), pop, push (5,4), pop, push (4,4), pop, push (1,5), push (2,2) •

3. (a) Draw a graph, a designated starting vertex in your graph, and a sequence of the vertices that can be reached from the starting vertex, with the property that the sequence you give could have been generated by a depth first search, but not by a breadth first search.

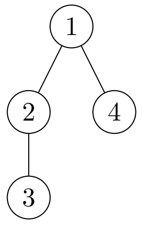
Solution. In the following graph, if 1 is the starting vertex, DFS could generate the sequence 1-2-3-4, but BFS could not.



•

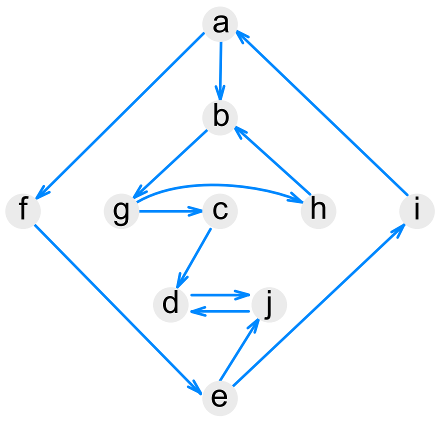
(b) Draw a graph, a designated starting vertex in your graph, and a sequence of the vertices that can be reached from the starting vertex, with the property that the sequence you give could have been generated by a breadth first search, but not by a depth first search.

Solution. In the same graph, if 1 is the starting vertex, BFS could generate the sequence 1-2-4-3, but DFS could not.

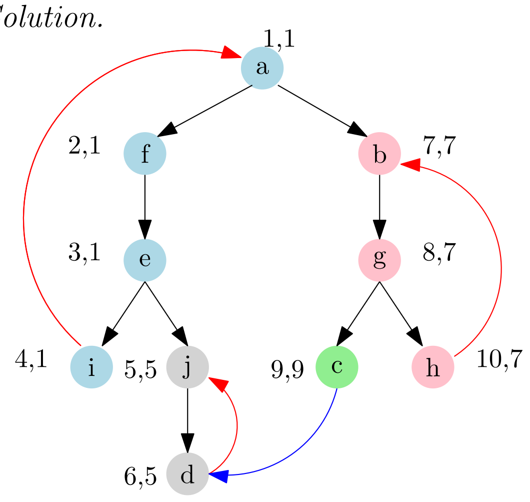


•

4. For the graph below, draw a possible depth-first search forest. Label each edge by whether it is a tree edge, a backwards edge, a forward edge, or a cross edge. Additionally, label each vertex with the lowlink number that would be computed by Tarjan's strongly connected components algorithm, and list the strongly connected components.



Solution.



The numbers next to each vertex correspond to the index, followed by the lowlink.

The components are:
a, f, e, i
j,d
b,g,h
c

Edges in black are tree edges, those in red are backwards edges, the one in blue is a cross edge. There are no forward edges. •