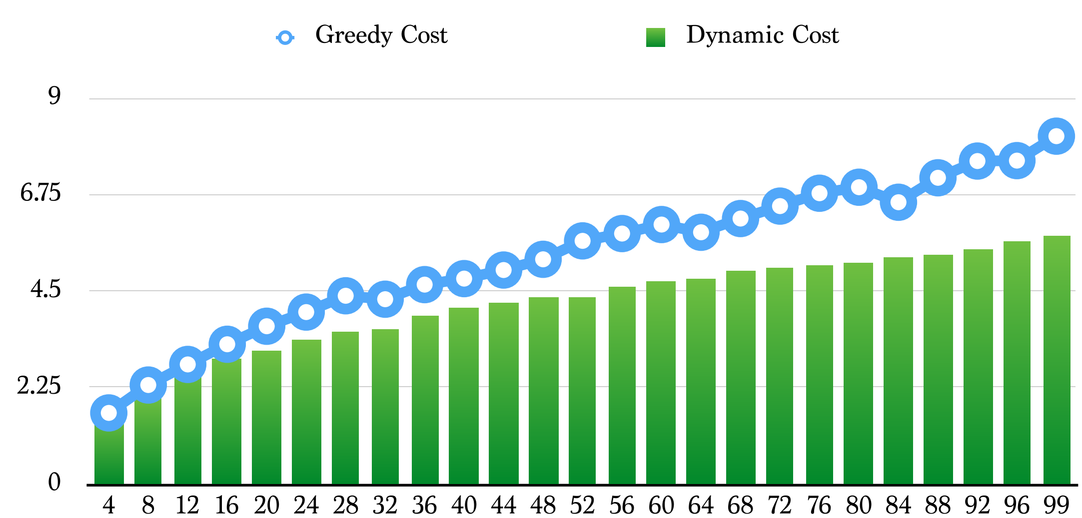
Julian Weiss julian.weiss@rochester.edu

CSC 172 PROJECT — THE SCIENCE OF DATA STRUCTURES
Dynamic Programming

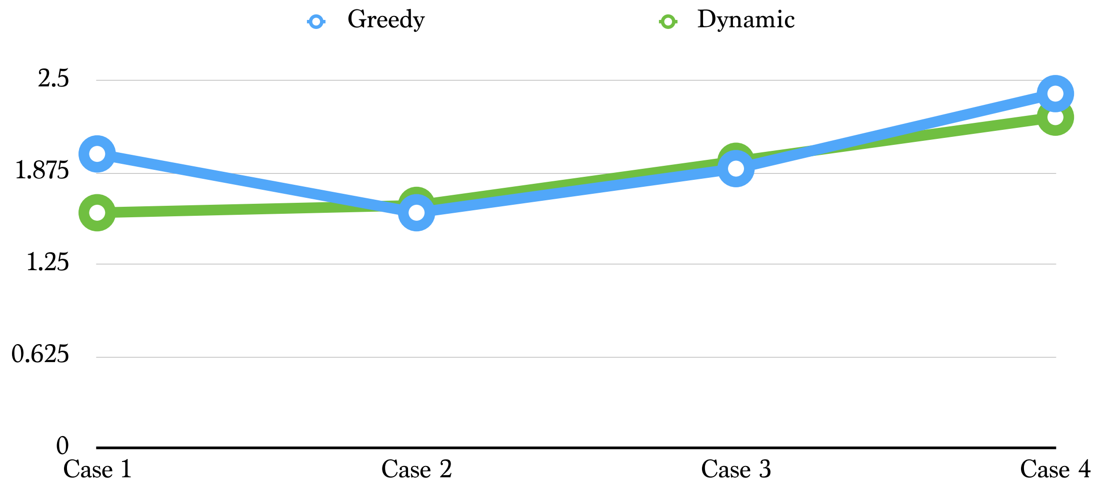
Dynamic Programming was an intense, analytical project, that required an immense amount of comprehension at every step. Unlike previous assignments, this exercise required a very detailed understanding of the data structures and algorithms utilized in the process, rather than graphics or IO. My submission was organized as follows:

Driver.java	controls the usage of all parts (labelled), including Greedy, Dynamic, and probability assessment.
AssociatedProb.java	is a simple data structure, used for convenience, containing a probability (integer, given * 10) and key (given letter, sometimes arbitrary)
Node.java	the core data structure for both Greedy and Dynamic optimizations, contains pointers to parent Node, both children Nodes, data (*AssociatedProb*), and array index. Not all components are used in each process; most are required for Dynamic.
DynamicSolution.java	handles most of the legwork for the Dynamic BST, contains a copy of the given probability set (*ArrayList<AssociatedProb>*) as well as the dynamic table (*Node[][]*).

Diving right in, my hypothesis for part three (first bullet) is: around <u>22-distribution sized sets</u>, the greedy solution begins to clearly diverge from the dynamic, which, although still comparable, only diverge more and more as time goes on. I arrived at this proposition by averaging the greedy and dynamic costs for each size set (each of random probabilities) that were generated from the 1-100 normalizing loop (which ran itself 5 times, and I re-tested for an hour or so). I've also decided to analyze the data in the graph below, where it's clear that the greedy cost's slope is quite a bit greater than the dynamic's, and becomes overtly noticeable around 40 (although it's visible from ~20):



Using four ratio-based distribution tables to demonstrate the cost differences between the greedy and dynamic algorithms with particular uniformities, the average results (and what the submitted code examples performs) were:



Case One Uniform (all 1/N)	**Case Two** Sharply Biased (hump at 1)	**Case Three** Slightly Biased (hump at N/4)	**Case Four** Symmetrical (max at N/2)
Greedy tree **failed** to optimize for any list **N > 2** with each key/probability 1/N. All generated trees have only left children. Dynamic produced completely balanced trees, and at a better cost.	Greedy tree still **fails for N > 2,** generating all left children from the max, while Dynamic generates a balanced tree, at similar costs.	This time Greedy generated decent trees for **N < 5, otherwise it failed** to properly weight. Dynamic, as always, generated a nice tree, but not at a better cost.	Greedy **failed** for **N > 7,** improperly weighting the keys (exaggerating left-branches), terribly generating a tree in comparison to the optimal. Dynamic worked as well as it always did, this time also at a better cost.

The long-and-short of this is: **Greedy rarely generates optimal trees**, but for small *N* values, it can succeed with an even smaller cost than Dynamic. Most of this could be linked to how the Greedy algorithm sorts the probability list before generating a tree (otherwise the root can never be guaranteed to be the highest probability, as is required for BSTs). To create other test cases, a "triangle" function might need to be defined, as these values were determined and analyzed manually. Also, if the Greedy function is ever to create valid BSTs for higher *N* values, the sorting block (below) would have to be altered:

```
public static Node greedySolution(ArrayList<AssociatedProb> givenAPs){
   AssociatedProb[] sortedAPs = givenAPs.toArray(new AssociatedProb[givenAPs.size()]);
   Arrays.sort(sortedAPs, new Comparator<AssociatedProb>(){
      @Override
      public int compare(AssociatedProb ap1, AssociatedProb ap2){
         return ap1.prob - ap2.prob;
      }
   });

   Stack<AssociatedProb> treeStack = new Stack<AssociatedProb>();
   for(AssociatedProb ap : sortedAPs)
      treeStack.push(ap);
   return constructBST(new Node(treeStack.pop()), treeStack);
}//end method
```