Brian Loughran
Programming Assignment #3 Analysis

README:

Python version: 3.6.2 (Python 2/3 compatible)

Input File: SignalInput.txt

Input File Format: To run the unweaving algorithm included in this package, the algorithm needs to know the given signal (s), and both of the origin signals (x, y). Each of these values is given as a string on each line in the input file. The first line of the input file is the signal s. The second line is x, and the third line is y. An example of an input file using the values of s, x, and y from the problem statement is a follows:

100010101
101
0

Execution: The __main__() file in the directory is UnweaveMain.py. The code will read the values in the input file SignalInput.txt and write the result to the output file UnweaveOutput.txt. Since UnweaveMain.py reads its inputs from SignalInput.txt, there are no arguments needed to be given to UnweaveMain.py, so you can just run it from the command line or you favorite IDE with no inputs or environment variables.

The other files included in the directory are KnownSignal.py and PossibleSolution.py. These two functions are just helper classes for UnweaveMai.py. KnownSignal.py is a data structure used to store the given signals x and y. PossibleSolution.py is another data structure that stores one solution considered by the unweaving algorithm.

Output: The output file stores the result of the unweaving operation at the bottom, as well as verbose debugging information. Some debugging information includes echoing the input given in the input file, and for each index in s, every permuted solution considered, the subset of permuted solutions that matched the input signal s, and the subset of those solutions which were not considered duplicates of each other. If there is a single solution being considered at a given index in s, it will note that and echo the solution through that index to output. If the unweaving is a success it will indicate success as well as the pattern used to get the unwoven solution. If the solution is a failure, either because there is no possible unweaving solution or if there were not enough instances of x or y to designate a full match, the output will indicate that. Example input and output files are included in the package for convenience.

ANALYSIS: Unweaving of Babel

Algorithm:

This is fairly obviously a dynamic programming problem. Thus we should start with a Bellman Equation for the given problem. One for the given problem is shown below:

Brian Loughran
Programming Assignment #3 Analysis

$$possSols(x, y, s[:n]) = \{$$
$$isValid(x, y, s[:n-1].append(x.next),$$
$$isValid(x, y, s[:n-1].append(y.next)\}$$

All this is saying is that we can iterate through each index in s and check if the solution at the previous index works if the next unwoven character is in x and if the solution at the previous index works if the next unwoven character is in y. Checking both and maintaining a list of the possible solutions will ensure that we are not having a greedy bias toward the solution being in x or y, and will allow us to track all possible solutions. If we do this until n is the length of s, then we will have iterated through all the characters in s, and we will have either a set of unweaving solutions or no solutions. If there is a set of solutions, then any of them can be a valid unweaving solution, and if there are no solutions, then there is no possible unweaving solution no matter which combination of x and y we choose.

For example, given s = 0110..., x=010, y=11, we can walk through the first few iterations of the above process. s_index is used to denote the index in s that we are analyzing, s[index] is the value of s at that index, x.next is the next value in x for a given solution, y.next is the next value in y for a given solution, possSols(s[:index]) is the list of possible solutions at a given index, and possSols(s[:index-1]) is the list of possible solutions at the previous index. The rows of x.next and y.next correspond to the next value for each based on the given possible solution (since the next value in x or y could be different depending on the different solution patterns considered.

s_index	s[index]	possSols(s[:index-1])	x.next	y.next	possSols(s[:index])
1	0	-	0	1	[x]
2	1	[x]	1	1	[x, x] [x, y]
3	1	[x, x] [x, y]	0 1	1 1	[x, x, y] [x, y, x] [x, y, y]
4	0	[x, x, y] [x, y, x] [x, y, y]	0 0 1	1 1 1	[x, x, x, y] [x, y, x, x]

Table 1: Example iterations of an unweaving algorithm

Note that this process assumes that the 0 index of s lines up with the 0 index of x and y. While there are ways to generalize this assumption, for simplicity and implementation sake we carry this assumption through the implementation.

One thing of note is that our list of possible solutions can quickly expand in size. Given s=11111111…, x=11, and y=111, for example, the number of possible solutions would expand at a rate of 2^n. Handling this many solutions would take exponential runtime, so clearly we must do something to limit the number of possible solutions that we can have at a time. The way that I did this was to introduce the idea of removing duplicates. One way to do this is to examine the final index of x and y. If the final index of x and y is the same for two solutions, then we say those solutions are duplicates of each other. For example, for each of the two solutions given at s_index=4 in the above example, we note that the final x

index will be 0 and the final y index will be 1 for both solutions. These are duplicate solutions for all intents and purposes. In the above example, s_index=5 will have x.next=0 and y.next=1 for both cases, thus while we are eliminating a possible solution, we are not eliminating any paths to the solution that might give a different outcome than the duplicate path. Another way of looking at what we are doing with removing duplicates is via a tree where each level in the tree is an index in s, and each node in the tree is the solution being considered:

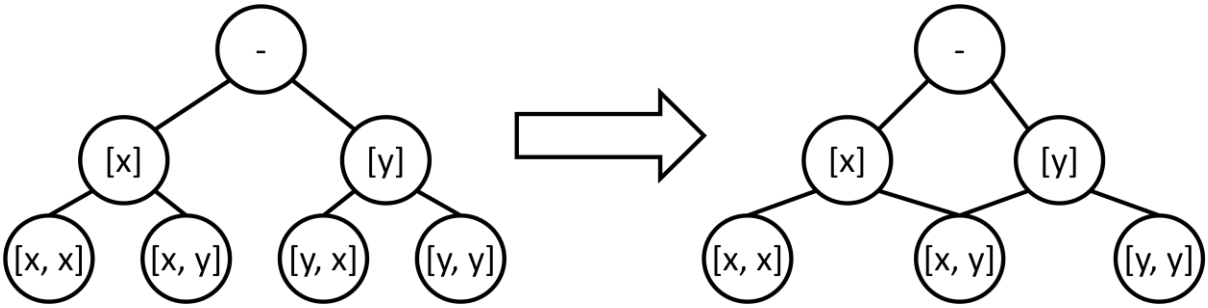


Figure 1: Tree diagram to show duplicate removal

We note that the values which have the same final index in x and the same final index in y are combined into one solution. As the tree on the left continues to increment the index of s, we are presented with 2^n possible solutions. However, as the tree on the right increments the index of s, we are presented with just one more possible solution, meaning that when we reach the final index in s, we will have just len(s)+1 solutions. If the length of the repeating signals x and y are short, then we can have no more than len(x)*len(y) possible solutions for a given index. Removing duplicates in this way will allow us to run this algorithm in polynomial time rather than exponential time. More detail into the total runtime will be discussed in the runtime section.

Now that we have the basic building blocks of a process, we can write a formalized algorithm. The algorithm looks like:

1. For each index in s:
    a. Get the next set of possible solutions based on the previous set.
    b. Remove any solutions that are not correct.
    c. Remove any duplicate solutions.
    d. If there are no solutions remaining, there is no possible unweaving, return False.
2. If you iterate over each index in s there must be at least one solution remaining. Any solution will produce a correct unweaving, thus we can return something other than False to indicate success.

This is fairly straightforward. This algorithm will indicate whether there is any possible unweaving of s into x and y, and it will do so in polynomial time. By storing the breakdown of the signal s into x and y as we have done in Table 1, we can also specify whether we assumed a particular index was a character in x or a character in y. This is a valid algorithm because all possible, non-duplicate breakdowns of the signal are considered, and we are not greedily matching on x or y.

Brian Loughran
Programming Assignment #3 Analysis

Psuedo-Code: Unweaving implementation

```
// helper class to store solution info
class PossSol:
  signal                                  // the signal of the proposed solution (e.g. 0110)
  signal_breakdown                        // divide signal into x and y (e.g. [x, x, x, y])
  final_x_index                           // the final x index of the solution
  final_y_index                           // the final y index of the solution

// this is the main unweaving function, unweave takes s, x, and y from the problem statement as input
unweave(s, x, y):
  poss_sols = []                          // initialize possible solutions to empty
  for char in s:                          // iterate over each character in s
    poss_sols = get_next_sols(poss_sols, x, y)    // get the next solutions based on previous solutions
    poss_sols = pop_wrong(poss_sols, char)        // make sure the solution is correct
    poss_sols = remove_duplicates(poss_sols)      // remove duplicates

    if len(poss_sols) == 0:
      return false                        // there is no possible solution, terminate here

  // end for loop
  return poss_sols[0]                     // any solution is correct, just return the first one

def get_next_sols(old_sols, x, y):        // helper function #1
  new_sols = []                           // initialize the new solutions to none
  for sol in old_sols:                    // iterate over each of the old solutions

    // assume the next bit in s is from x
    signal = sol.signal.append(x[sol.final_x_index])      // add next x bit to signal
    breakdown = sol.breakdown.append[x]   // add choice x to breakdown
    x_ind = sol.final_x_index.increment()       // increment x index (be sure to wrap around)
    y_ind = sol.final_y_index             // the y index remains the same if this solution uses x
    solution = PossSol(signal, breakdown, x_ind, y_ind)     // create a solution using PossSol
    new_sols.append(solution)             // add the solution to our list of solutions

    // assume the next bit in s is from y
    signal = sol.signal.append(y[sol.final_y_index])      // add next y bit to signal
    breakdown = sol.breakdown.append[y]   // add choice y to breakdown
    x_ind = sol.final_x_index             // the x index remains the same if this solution uses y
    y_ind = sol.final_y_index.increment()       // increment y index (be sure to wrap around)
    solution = PossSol(signal, breakdown, x_ind, y_ind)     // create a solution using PossSol
    new_sols.append(solution)             // add the solution to our list of solutions
```

```
  // note that we need a base case for if there are no current solutions (like for the first index in s)
  // the base case is effectively identical but without the loop, thus does not need to be laid out here
  return new_sols                                  // return all possible new solutions

def pop_wrong (sols, char):                        // helper function #2
  poss_sols = []                                   // initialize the possible solutions to none
  for sol in sols:                                 // iterate over each possible solution
    if sol.signal[-1] == char                      // check that the last character in the signal matches s
      poss_sols.append(sol)                        // add the correct solution to the solution set
  return poss_sols                                 // return all the correct solutions

// this function uses a hash table to determine if a single solution is duplicate in O(1) time
def remove_duplicates(sols)                        // helper function #3
  recorded_indexes = {}                            // hash table to store final indexes
  poss_sols = []                                   // initialize the possible solutions to none
  for sol in sols:                                 // iterate over each possible solution
    hash = sol.final_x_index + '-' + sol.final_y_index          // create a hash entry for the final index set
    if not recorded_indexes.get(hash):             // check if there was a solution with these indexes yet
      recorded_indexes[hash] = True                // don't allow any more solutions with this index
      poss_sols.append(sol)                        // this solution is not duplicate, add it to solutions
  return poss_sols                                 // return non-duplicate solutions
```

The pseudocode as a whole follows the basic layout of our formal algorithm. We utilize helper functions 1, 2, and 3 to run steps 1a, 1b and 1c respectively to maintain readability of our main unweave() function. Step 1d is accomplished using a simple conditional statement, and step 2 is accomplished using a return statement. As the pseudocode closely follows the algorithm laid out and already proven to be correct, the pseudocode must also be correct.

Runtime: The algorithm given for unweaving a signal s into repeating signals x and y requires first that we iterate over each character in s. For each character in s there m possible solutions to analyze. Steps 1a, 1b and 1c all require O(1) time per possible solution, as there are a finite number of instructions to be run per solution, and thus run in O(m) time for a given m solutions. Step 1d runs in O(1) time, thus for each index in s, we can run in O(m) time, for a total runtime of O(n*m) (where n is the number of characters in s). Step 2 of the algorithm runs in O(1) time, thus has no effect on the algorithmic runtime.

Thus to solve for the total runtime, we must determine an upper bound on the value m. We note in Figure 1 that the maximum number of possible, non-duplicate solutions is O(len(s)+1) = O(n). And since we are iterating n times over each index in s, the algorithm can be bounded by an algorithmic runtime of O(n^2), which is an efficient, polynomial solution. However, for this problem, often the length of signal x and y are far smaller than the length of s as to get a true representative sample of s. For a given len(x) and len(y), there can only be O(len(x)*len(y)) non-duplicate solutions, thus, the algorithm can be reduced to an algorithmic runtime of O(n*len(x)*len(y)) in the case that x and y are much shorter than s,

and in this case the algorithm can be solved in linear time based on the size of s. Thus the algorithm can be bounded by O(n^2) or O(n*len(x)*len(y)) depending on the size of signal x and y, whichever runtime is shorter is the algorithmic runtime of the given algorithm.

Post-Processing: One other part of the algorithm not mentioned above was to ensure that we get at least a full match of x and a full match of y in signal s. This is a simple process, and requires only a quick step on the post-processing end of the algorithm, thus was not included above for readability. The easiest way to do this is to check the signal_breakdown in PossSol to see how many instances of x and y there were in the solution breakdown and compare that to the length of signal x and length of signal y. If there were more instances than the length for both x and y, then the solution was valid. If not, the solution was not. Pseudocode is included below:

Post-Processing Pseudocode:

```
def full_match(signal_breakdown, x, y):        // function to see if we have a full match of x and y
  xs, ys = 0                                   // initialize the number of x's and y's to 0
  for char in signal_breakdown:                // iterate over each character in the solution
    if char == 'x':
      xs += 1                                  // increment xs for instance of x
    if char == 'y':
      ys += 1                                  // increment ys for instance of y
  if xs >= len(x) and ys >= len(y)             // ensure there are enough x's and y's for a full match
    return True
  return False
```

This code will return True if there are enough x's and y's to get a full match for each. If not, this code will return False and we can indicate the lack of a full match to the user along with the proposed solution.

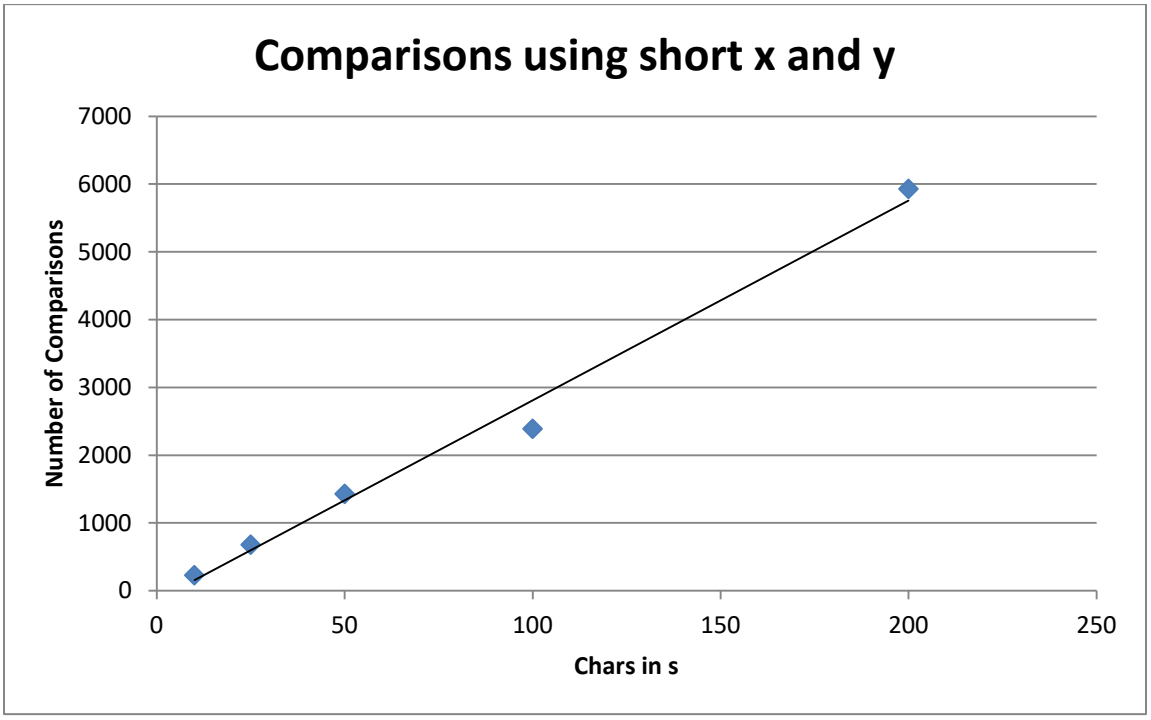
Post-Processing Runtime: The algorithm for ensuring the full match of x and y iterates over each character in the solution (which has the same number of characters as are in s) and performs a finite set of operations to each in O(1) time. Thus the total runtime of the post-processing algorithm is O(n), and does not affect the total runtime of the unweaving algorithm.

Empirical Measurements:

Empirical measurements in this section are taken not by measuring the runtime of each algorithm, as that is prone to error. Measurements are made by measuring the number of comparisons made by each sorting algorithm. A "comparison" is made whenever a possible solution is processed, either by being permuted from a previous solution, being compared to s, or being considered as a duplicate as removed. Thus, in this fashion, some possible solutions may have as many as three comparisons for a given index in s. However, doing it this way will give us accurate results on the order of work being done by the algorithm.

Brian Loughran
Programming Assignment #3 Analysis

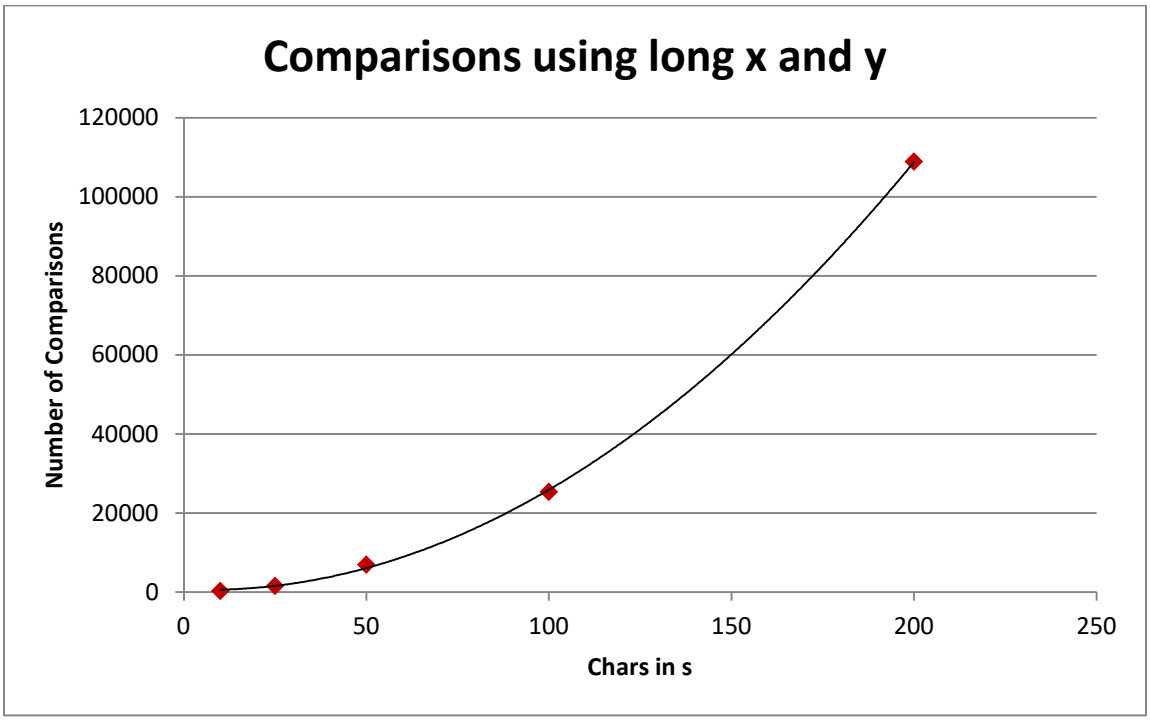
Two sets of comparisons are measured here, one set using very short strings for x and y, and another using long strings for x and y. The goal of doing two sets of comparisons was to isolate the two predicted runtimes of the algorithm based on the lengths of x and y. To ensure the worst possible scenario (e.g. the maximum number of comparisons) for each of the two cases, strings of repeating 1's were used for each scenario, as that would give the algorithm the most trouble eliminating duplicates and incorrect solutions. Below are the results of using short strings for x and y:



Graph 1: Number of comparisons using short strings for x and y

As expected, the algorithm runs in linear time in relation to the number of characters in s. This is as predicted, since using short strings should give us an algorithmic runtime of O(len(x)*len(y)*n).

Next we can analyze the number of comparisons when using relatively long strings for x and y (for reference, "long" string for x and y are for any values such that len(x)*len(y) > n). The results are graphed below:



Graph 2: Number of comparisons using long strings for x and y

Again, the results match up with the predicted runtime for long signals x and y with O(n^2) runtime. This is due to the number of possible solutions increasing at a linear rate as we progressively increment the index in s as discussed in the runtime section.

Thus our empirical data matches with our theoretical runtimes, giving us a decent indication that the theoretical runtimes are correct. You can see the raw data for each of the charts in the appendix for reference, or view in EmpericalData.xlsx.

Appendix (Raw Runtime Data):

n	comparisons (small x, y)	comparisons (long x, y)
10	230	285
25	680	1569
50	1430	6942
100	2390	25350
200	5930	108888