Problem Set 1

All parts are due on February 18th, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 1-1. [18 points] **Asymptotic behavior of functions**

For each group of functions, arrange the functions in the group in increasing order of growth. That is, arrange them as a sequence $f_1, f_2, \ldots$ such that $f_1 = O(f_2), f_2 = O(f_3), f_3 = O(f_4), \ldots$. For each group, add a short explanation to explain your ordering.

(a) [6 points] **Group 1:**

$$\begin{aligned} f_1 &= 4n^4 \\ f_2 &= \log\left(4n^{n^4}\right) \\ f_3 &= (\log n)^{4\log(4n)} \\ f_4 &= (\log n)^4 \\ f_5 &= (\log\log n)^4 \end{aligned}$$

Solution: The correct ordering is: f_5, f_4, f_1, f_2, f_3. Here is the justification for the ordering:

1. $f_5 = O(f_4)$ since we have that $\log\log n = O(\log n)$.
2. $f_4 = O(f_1)$ since $f_1 = O(n^4)$ and $\log n = O(n)$, we have that $\log^4 n = O(n^4)$.
3. $f_1 = O(f_2)$ since $f_2 = n^4 \log 4n = n^4(\log n + \log 4) = O(n^4 \log n)$.
4. $f_2 = n^4(\log n + \log 4) = O(n^4 \log n)$. Taking logs, we get that $\log(f_2) = \log\left(n^4 \log n + \log 4\right) \leq 4\log n + \log(\log n + \log 4) = \Theta(\log n)$. Also, $\log(f_3) = \log\left((\log n)^{4\log(4n)}\right) = 4\log(4n)\log\log n = \Theta(\log n \log\log n)$. As the logarithm of f_3 grows asymptotically faster than the logarithm of f_2, $f_2 = O(f_3)$.

(b) [6 points] **Group 2:**

$$f_1 = 4^{4^n}$$
$$f_2 = 4^{4^{n+1}}$$
$$f_3 = 5^{4^n}$$
$$f_4 = 5^{4n}$$
$$f_5 = 4^{5n}$$

Solution: The correct ordering is f_4, f_5, f_1, f_3, f_2. Here is the justification for the ordering:

1. $f_4 = O(f_5)$. Note that $f_4 = 625^n$ and $f_5 = 1024^n$ and thus $\lim_{n\to\infty} \left(\frac{625}{1024}\right)^n = 0$.
2. $f_5 = O(f_1)$. Note that 4^{5n} will grow slower asymptotically than 4^{4^n}, since $5n$ grows strictly slower asymptotically than 4^n.
3. $f_1 = O(f_3)$ since $\lim_{n\to\infty}(4/5)^{4^n} = 0$.
4. $f_3 = O(f_2)$. Note that $\log(f_3) = \log(5^{4^n}) = 4^n \log(5)$ and $\log(f_2) = \log(4^{4^{n+1}}) = 4^{n+1}\log 4$. Since the logarithms of f_2 and f_3 are not within an additive constant of each other, $f_3 = O(f_2)$.

(c) [6 points] **Group 3:**

$$f_1 = \binom{n}{4}$$
$$f_2 = \binom{n}{n/4}$$
$$f_3 = 4n!$$
$$f_4 = 4^{n/4}$$
$$f_5 = (n/4)^{n/4}$$

Solution: The correct ordering is f_1, f_4, f_2, f_5, f_3. Here is the justification for the ordering:

1. $f_1 = O(f_4)$ since $f_1 = O(n^4)$ and $n^4 = O(4^{n/4})$ by definition.
2. $f_4 = O(f_2)$ since $\binom{n}{n/4} = \frac{n\cdot(n-1)\cdot(n-2)\cdots(n-n/4+1)}{(n/4)\cdot(n/4-1)\cdots 1} > 4^{n/4}$.
3. $f_2 = O(f_5)$ by the following reasoning. Since $f_5 = \frac{n^{n/4}}{4^{n/4}}$ and $(\frac{n}{4})! > 4^{n/4}$ for sufficiently large n, $f_5 = \frac{n^{n/4}}{4^{n/4}} > \frac{n^{n/4}}{(\frac{n}{4})!}$. Furthermore, $n^{n/4} > \frac{n!}{(\frac{3n}{4})!}$ and, so, $\frac{n^{n/4}}{(\frac{n}{4})!} > \frac{n!}{(\frac{3n}{4})!(\frac{n}{4})!} = f_2$. To conclude, $f_2 = O(f_5)$.
4. $f_5 = O(f_3)$ since $4n! > \frac{n!}{(\frac{3n}{4})!} > (n/4)^{n/4}$.

Problem 1-2. [18 points] **Recurrences**

(a) [12 points] **Solving recurrences:**

Give solutions to the following. In all cases, c is a positive real-valued constant, and n is a nonnegative integer. For simplicity, you may ignore roundoffs in your explanations.

1. $T(n) = c$, for $n \leq 1$, and $T(n) = T(n-1) + c$, for $n > 1$.

   Solution: $T(n) = c(n-1)$. Therefore, $T(n) = \Theta(n)$.
2. $T(n) = c$, for $n \leq 1$, and $T(n) = T(n-1) + cn$, for $n > 1$.

   Solution: $T(n) = c(1 + 2 + 3 + \cdots + n) = c\frac{(n+1)n}{2}$. Therefore, $T(n) = \Theta(n^2)$.
3. $T(n) = c$, for $n \leq 1$, and $T(n) = T(n/2) + c$, for $n > 1$.

   Solution: By using the recursion tree method (you can also use the Master Theorem), the height of the tree is $\Theta(\log_2 n)$ and the amount of work done in each level is c. The total time spent in the recursion tree is $\Theta(c \log_2 n)$. Therefore, $T(n) = \Theta(\log n)$.
4. $T(n) = c$, for $n \leq 1$, and $T(n) = 2T(n/2) + c$, for $n > 1$.

   Solution: By using the recursion tree method, each node of the tree contains two children. Each node performs c work. The height of the tree is $\Theta(\log_2(n))$ or $k \log_2 n$ for some constant k. Therefore, $T(n) = c(2^0 + 2^1 + 2^2 + \cdots + 2^{k \log_2(n)}) = 2cn + c2^k$ and $T(n) = \Theta(n)$.
5. $T(n) = c$, for $n \leq 1$, and $T(n) = 2T(n/2) + cn$, for $n > 1$.

   Solution: The height of the recursion tree is $\Theta(\log_2 n)$ or $k \log_2 n$ for some constant k. We use the same recursion tree as in (a) 4, except now the recursion gives $T(n) = c(2^0(n) + 2^1(n/2) + 2^2(n/4) + \cdots + \cdots + 2^{k \log_2(n)}(n/2^{k \log_2(n)})) = n \log_2 n$. Therefore, $T(n) = \Theta(n \log n)$.
6. $T(n) = c$, for $n \leq 1$, and $T(n) = 3T(n/2) + cn$, for $n > 1$.

   Solution:
   We can set up the following recurrence:

   $T(n) = cn + 3(cn/2) + 3^2(cn/4) + \ldots$
   $= \sum_{i=0}^{\log_2 n} 3^i(cn/2^i)$
   $= \sum_{i=0}^{\log n} (\frac{3}{2})^i cn$
   $= cn(\frac{3^{\log_2 2n}}{n} - 2)$
   $= \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$.

   Therefore, $T(n) = \Theta(n^{\log_2 3})$.

(b) [6 points] **Setting up recurrences:** Write recurrences describing the time complexity of the following algorithms. You don't need to solve the recurrences.

1. Suppose we are given a list of integers that are sorted in nondecreasing order. Using binary search, determine whether a given integer appears in the list.

   Solution: The recursion used for binary search is $T(n) = T(n/2) + 1$ with base case $T(1) = 1$ since we divide the list in half at each level and recurse on only one side. At each level, we perform 1 comparison. By what we saw in (a) 2, $T(n) = \Theta(\log n)$.

2. Suppose we are given an $n \times n$ matrix, with each row and column sorted in nondecreasing order. For example, the matrix below is sorted in this way:

$$\begin{bmatrix} 1 & 4 & 7 & 9 & 12 & 14 & 23 \\ 2 & 4 & 8 & 10 & 15 & 17 & 26 \\ 6 & 8 & 8 & 11 & 64 & 70 & 80 \\ 12 & 13 & 14 & 15 & 65 & 71 & 80 \\ 14 & 22 & 33 & 34 & 65 & 72 & 80 \\ 25 & 26 & 39 & 48 & 66 & 73 & 80 \\ 35 & 36 & 45 & 55 & 70 & 80 & 81 \end{bmatrix}$$

   Determine whether a given integer appears in any matrix that is constructed under these conditions. Solve this problem by *writing and setting up a recurrence*[1].

   Solution: When you search for an element in the $n \times n$ matrix, you can divide the matrix into 4 matrices and compare the middle element of the matrix (i.e. in position $(\lfloor n/2 \rfloor, \lfloor n/2 \rfloor)$) with the target element. If the target element is less than the middle element, then eliminate the submatrix containing elements $[\lfloor n/2 \rfloor, n] \times [\lfloor n/2 \rfloor, n]$ and recurse on the remaining three submatrices. Otherwise, if the element is greater than the middle element, eliminate the submatrix containing elements $[0, \lfloor n/2 \rfloor] \times [0, \lfloor n/2 \rfloor]$ and recurse on the three remaining submatrices. Repeat this procedure recursively on all submatrices until either all elements have been compared or the element is found.

   This procedure results in the recursive definition: $T(1) = 1$ and $T(n) = 3T(n/2) + 1$ where n is the length of a side of the matrix. By using the recursive tree method (or by Case 1 of the Master Theorem), we find that the solution to the recurrence is $T(n) = \Theta(n^{\log_2 3})$.

   This question can allow several answers, including some that may result in a better runtime. Full credit will be given for any correct, polynomial time methods.

[1] Note that there are other ways to solve this problem that does not require setting up a recurrence, but we specifically want you to set up a recurrence to solve this problem.

Problem 1-3. [24 points] **Maximizing stock gain:**

Here is a new version of the stock gain problem introduced in Lecture 1. Now we are given a finite sequence $A[0, .., n-1]$ of positive integer prices, with $n \geq 4$. The problem is to determine the maximum profit achievable by a sequence of four "Buys" and "Sells" for a particular stock. The four transactions should be "Buy", "Sell", "Buy", and "Sell", in that order, that is, we are buying and selling the same stock twice.[2]

Formally, the problem is to find the maximum value of the expression $(A[i_2] - A[i_1]) + (A[i_4] - A[i_3])$ where $0 \leq i_1 \leq i_2 \leq i_3 \leq i_4 \leq n-1$. Note that you can perform multiple transactions on the same day.

(a) [8 points] **Brute-force algorithm:** Describe (in words *and* pseudocode) a naive, brute-force algorithm that takes A as input and outputs the maximum profit. Analyze its runtime cost.

Solution: One possible brute force approach would be to consider all nondecreasing sequences of length 4 which would include two buy and two sell dates (allowing some of the dates being the same). There are at most n^4 such sequences. Thus, the runtime of this algorithm is $O(n^4)$ to check all profits from all possible n^4 sequences.

Pseudocode:

BRUTE-FORCE(A)

```
1  profit = 0
2  for i ← 0 to length[A] − 1
3          for j ← i to length[A] − 1
4                  for k ← j to length[A] − 1
5                          for l ← k to length[A] − 1
6                                  if i, j, k, l < length[A]
7                                          profit = max(profit, A[l] − A[k] + A[j] − A[i])
8  return profit
```

(b) [16 points] **Much better algorithm:** Describe (in words *and* pseudocode) an $O(n)$ time algorithm for this problem. Analyze its runtime cost.

Solution: The solution builds on the $O(n)$ time stock buying algorithm described in lecture for the case of a single buy/sell operation.

Let $G^*[j_0]$ for a given $0 \leq j_0 \leq n-1$ be the best profit achievable by a single buy/sell operation if we commit to selling on day j_0. In the lecture, we showed how to compute all values $G^*[j_0]$ in $O(n)$ time.

[2] This problem assumes that the data is provided ahead of time, and we can choose the best dates with full knowledge of the prices. This is clearly not realistic for buying and selling actual stock, but may be useful for analyzing decisions after the fact.

Now, for a given day $0 \leq i \leq n-1$, let us denote by $PBS[i]$ the best profit one can achieve by a single buy/sell operation that is concluded at day i at the latest. (So, the best profit overall from a single buy/sell operation is $PBS[n-1]$.)

Observe that $PBS[0] = 0$ and $PBS[i] = \max(PBS[i-1], G^*[i])$. Thus, once all the values $G^*[j_0]$ are computed, we can compute all values $PBS[i]$ in $O(n)$ time via the following simple pass through the array A:

```
1  PBS[0] = 0
2  for i ← 1 to n − 1
3          PBS[i] = max(PBS[i − 1], G*[i])
```

Next, we notice that if $\overline{PBS}[i]$, for a given $0 \leq i \leq n-1$, is the maximum profit from a single buy/sell operation that starts *no earlier* than day i then we can also compute all the values $\overline{PBS}[i]$ in $O(n)$ time by exactly the same approach as above. One just need to look at suffix maximas (instead of prefix minimas) and performing all the passes from the end to the beginning of A.

Finally, we can conclude that our final answer: the best overall profit max_gain_2 from a sequence of two sell/buy operations is exactly

$$max\_gain\_2 = \max_{0 \leq i \leq n-1} PBS[i] + \overline{PBS}[i],$$

where the optimal value of i corresponds to a day i such that the first sell/buy operation concludes by day i and the second one does not start before day i. (Note that such day i always exists.) So, max_gain_2 can be computed in $O(n)$ time via another simple pass through A:

```
1  max_gain_2 = 0
2  for i ← 0 to n − 1
3          max_gain_2 = max(PBS[i] + PBS[i], max_gain_2)
```

This results in an overall $O(n)$ time algorithm as desired.

Entire algorithm:

MOST-PROFITABLE-TWO-BUY-SELLS(A)

```
1   PMin[0] = A[0], G*[0] = 0, PBS[0] = 0
2   for i ← 1 to n − 1
3           PMin[i] = min(PMin[i − 1], A[i])
4   for i ← 1 to n − 1
5           G*[i] = max(G*[i − 1], A[i] − PMin[i])
6   for i ← 1 to n − 1
7           PBS[i] = max(PBS[i − 1], G*[i])
8   PMax[n − 1] = A[n − 1], G*'[n − 1] = 0, PBS'[n − 1] = 0
9   for i ← n − 2 to 0
10          PMax[i] = max(PMax[i + 1], A[i])
11  for i ← n − 2 to 0
12          G*'[i] = max(G*'[i + 1], PMax[i] − A[i])
13  for i ← n − 2 to 0
14          PBS'[i] = max(PBS'[i + 1], G*'[i])
15  max_gain_2 = 0
16  for i ← 0 to n − 1
17          max_gain_2 = max(PBS[i] + PBS'[i], max_gain_2)
18          return max_gain_2
```

Alternate Solution:

This essentially follows the basic idea of stock buying described in lecture. As in class, we will compute the prefix minima and use the prefix minima to compute the best gain from a buy and sell at each date. However, in addition to the prefix minima, we will also compute two other prefixes: the prefix buy/sell maxima and the prefix buy/sell/buy maxima.

We maintain four quantities: the three prefix minima and maxima and the final maximal gain. Each of the prefix minima and maxima is maintained as a list. We call these prefix lists $PMin$, $PBSMax$, and $PBSBMax$. max_gain is the variable representing the maximum gain that we want to return at the end. $PMin[j]$ represents the lowest possible "monetary loss" of a single Buy transaction on or before date j (i.e. $PMin[j] = \min_{i \leq j} A[i]$ as in lecture). $PBSMax[j]$ represents the highest possible profit for successive Buy and Sell transactions on or before j. $PBSBMax$ represents the highest possible profit for successive Buy, Sell, and Buy transactions on or before j. Finally, max_gain represents the highest possible profit for successive Buy, Sell, Buy, and Sell transactions up to some date.

We initialize $PMin[0] = A[0]$, $PBSMax[0] = 0$, $PBSBMax[0] = -A[0]$, and $max\_gain = 0$. As in class, we first compute the three types of prefixes:

```
1  PMin[0] = A[0], PBSMax[0] = 0, PBSBMax[0] = −A[0]
2  for j ← 1 to length[A] − 1
3          PMin[j] = min(PMin[j − 1], A[j])
4  for j ← 1 to length[A] − 1
5          PBSMax[j] = max(PBSMax[j − 1], A[j] − PMin[j])
6  for j ← 1 to length[A] − 1
7          PBSBMax[j] = max(PBSBMax[j − 1], PBSMax[j] − A[j])
```

$PMin[j]$ compares the current Buy price with the smallest Buy price before the current date, j, and takes the minimum of the two. $PBSMax[j]$ compares the best Buy-Sell profit *strictly before* the current date j with the profit resulting from a Buy before or on j and a Sell on j, and records the maximum of the two. $PBSBMax[j]$ records the maximum of a Buy-Sell-Buy profit strictly before j and Buy-Sell on or before j plus a Buy on j. After we calculate the prefix minima and maxima, we can calculate the maximum gain resulting from all four transactions:

```
1  max_gain = 0
2  for j ← 1 to length[A] − 1
3          max_gain = max(A[j] + PBSBMax[j], max_gain)
```

As we iterate through A, max_gain contains the best profit that has been seen so far resulting from "Buy", "Sell", "Buy", "Sell" from nondecreasing dates. Therefore, at the end of the iteration through A, max_gain contains the best profit.

Entire Pseudocode:

MOST-PROFITABLE-BUYS-SELLS(A)

```
1  PMin[0] = A[0], PBSMax[0] = 0, PBSBMax[0] = −A[0], max_gain = 0
2  for j ← 1 to length[A] − 1
3          PMin[j] = min(PMin[j − 1], A[j])
4  for j ← 1 to length[A] − 1
5          PBSMax[j] = max(PBSMax[j − 1], A[j] − PMin[j])
6  for j ← 1 to length[A] − 1
7          PBSBMax[j] = max(PBSBMax[j − 1], PBSMax[j] − A[j])
8  for j ← 1 to length[A] − 1
9          max_gain = max(A[j] + PBSBMax[j], max_gain)
10 return max_gain
```

Part B

Problem 1-4. [40 points] **Document distance**

In this problem, we revisit the "Document Distance" problem and algorithms that were introduced in Recitation 2. You will use some of these algorithms, and some variations, to compare text files for the play "Henry IV, Part I" by William Shakespeare, which he wrote in around 1597, to files for three other English comedies:

1. "Henry IV, Part II", by Shakespeare, written at approximately the same time,
2. "The Tempest", by Shakespeare, written around 1610, and
3. "The Pirates of Penzance", by Gilbert and Sullivan, written around 1879.

Files for these plays appear in the plays folder of the problem set. Code for the routines presented in recitation can be found on Stellar.

(a) [10 points] **Comparing word frequencies:** Construct a program for determining the "distance" between two documents, defined to be the angle between two vectors representing the frequencies of occurrence of all words in the two documents. You may use any of the routines provided in recitation, or write your own. Specifically, implement the doc_dist function in docdist.py.

On this problem set, we will not grade you based on the efficiency of your code, as long as it is "reasonable", i.e., it runs within the (generous) bounds we allow. To check whether your code is reasonable, you can run tests.py and check if the profiled output shows that the tests ran under a few seconds.

(b) [10 points] **Comparing frequencies of word pairs:** Modify your program so that, instead of comparing word frequencies, it compares the frequency of occurrence of *consecutive pairs of words*. Here, the order of words matters. In other words, "thou art" would be a different word pair than "art thou". Implement the doc_dist_pairs function in docdist.py.

(c) [10 points] **Comparing frequences of most frequent words:** Now return to considering word frequencies. For better efficiency, we might consider not vectors that count *all* words, but vectors that count just the 50 words that appear most frequently for each file in the comparison. Break ties alphabetically (using default Python string comparison) on the words. Implement the doc_dist_50 function in docdist.py.

(d) [5 points] **Analysis:** Analyze the running time for your file comparison programs in parts (a), (b), and (c) above. As parameters, you should use n, the total number of word occurrences in the two files being compared, and m, the total number of *distinct* words in the two files. (You may assume that you are given the input files as lists of words, and need not analyze the time for producing those lists.) For part (c), you should also use a parameter k representing the number of frequently-occurring words that are being used in the comparison. If you use Python Dictionaries, consider

insertion and lookup to be $O(1)$-time operations, and listing all the elements in a dictionary (iterating on items or keys) to be $O(s)$, where s is the number of items in the dictionary.

Solution:

(a) If you used docdist1.py, docdist2.py, or docdist3.py for your solution, the runtime of your program should be $O(mn + m^2)$. For each word in the wordlist, the program checks whether that word is in the current list of distinct words. This requires $O(m)$ time per word, for a total of $O(mn)$ time to construct a list of distinct words with their frequencies. After constructing lists of distinct words with their frequencies, the program either determines the vector angle using a nested loop or by first sorting the lists using insertion sort. In either case, this would require $O(m^2)$ time.

If you used dictionaries as in docdist4.py for your solution, the runtime of your program should be $O(m + n)$. Lookup and insertion on a dictionary can be done in $O(1)$ time. Since constructing a dictionary of distinct words with their frequencies requires one lookup and one insertion per word in the wordlist, this evaluates to $O(n)$ time. Similarly, since finding the inner product of two dictionaries requires one lookup for each distinct word in each dictionary (iterating over the items or keys of each dictionary) to check whether the word is present in the other dictionary, this evaluates to $O(m)$ time.

(b) The main difference between parts (a) and (b) is that the number of distinct word pairs is different from the number of distinct words. Specifically, there are at most $\min(m^2, n)$ distinct word pairs. Thus, the runtimes change as follows: If you used docdist1.py, docdist2.py, or docdist3.py for your solution, the runtime of your program should be $O(\min(m^2, n)n + \min(m^2, n)^2)$. If you used dictionaries as in docdist4.py for your solution, the runtime of your program should be $O(\min(m^2, n) + n)$.

(c) The main differences between parts (a) and (c) are that we need to determine the k most frequent distinct words in each file, and we now evaluate our inner product based on at most k distinct words.

There are many possible approaches to get the k most frequent distinct words. In general, getting the k most frequent distinct words is faster if done on the list of distinct words with their counts rather than the original word list. Some possibilities of doing so are as follows:

- Sort using insertion sort on word frequency: $O(m^2)$
- Sort using merge sort or some similar sorting algorithm: $O(m \log m)$
- Scan through the words k times, picking the next most frequent word each time: $O(mk)$

If you decided to get k most frequent distinct words from the orignal word list, you still likely used a variant of one of the above methods. However, your runtime

would be in terms of n rather than m.

If we let r be the runtime for getting the k most frequent distinct words, the runtimes for the overall problems change as follows: If you used docdist1.py, docdist2.py, or docdist3.py for your solution, the runtime of your program should be $O(mn + r + k^2)$. If you used dictionaries as in docdist4.py for your solution, the runtime of your program should be $O(n + r + k)$.

(e) [5 points] **Conclusion:** Use your program to determine the angle between your vector representing "Henry IV, Part I" and your vector for each of the other three plays. Repeat this for your functions in parts (a), (b), and (c) above. What can you conclude from these results?

Solution: Should all be within +/- 0.1

a Word distance

"Henry IV, Part I" and "Henry IV, Part II" 0.30

"Henry IV, Part I" and "The Tempest" 0.39

"Henry IV, Part I "and "The Pirates of Penzance" 0.53

b Word distance with pairs

"Henry IV, Part I" and "Henry IV, Part II" 0.91

"Henry IV, Part I" and "The Tempest" 1.10

"Henry IV, Part I" and The "Pirates of Penzance" 1.25

c Word distance with the 50 most common words

"Henry IV, Part I" and "Henry IV, Part II" 0.29

"Henry IV, Part I" and "The Tempest" 0.36

"Henry IV, Part I" and "The Pirates of Penzance" 0.50

"Henry IV, Part I" is most like "Henry IV, Part II", somewhat like "The Tempest", and least like "The Pirates of Penzance". This is all computed in terms of size of the angle between the representing vectors. All the tests seem to demonstrate this pretty well. This is reasonable because "Henry IV, Part I" is more closely related to "Henry IV, Part II" than other works of Shakespeare. The Tempest is also written by Shakespeare, so it is more similar to "Henry IV, Part I" than a work by another author such as with "The Pirates of Penzance".

Using pairs draws more distinctions than using word counts, so "Henry IV, Part I" is computed as being more different from all of the other files (larger angles) than it was from word counts alone. This could be because the order of words and phrasing is more subject to change between works.

Using only 50 words draws fewer distinctions, so the files appear closer (smaller angles). However, they are not much closer—it appears that using only 50 words distinguishes nearly as much as using all the words in "Henry IV, Part I".