Problem Set 1

All parts are due Thursday, February 20 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. 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. [15 points] **Asymptotic Behavior of Functions**

For each group of functions, sort the functions in increasing order of asymptotic (big O) growth. If some have the same asymptotic growth, be sure to indicate that. $\log$ means base 2.

(a) [5 points] **Group 1:**

$$\begin{aligned} f_1(n) &= 3.3n \\ f_2(n) &= n^{3.3} \\ f_3(n) &= n^{0.9}\log^3 n \\ f_4(n) &= (\log n)^{3\log n} \\ f_5(n) &= (3.3)^n \end{aligned}$$

(b) [5 points] **Group 2:**

$$\begin{aligned} f_1(n) &= 3^{3^{n+1}} \\ f_2(n) &= 9^{9^n} \\ f_3(n) &= 3^{3^n} \\ f_4(n) &= \binom{n}{n/3} \\ f_5(n) &= 9^{9n} \end{aligned}$$

(c) [5 points] **Group 3:**

$$\begin{aligned} f_1(n) &= 3^n \\ f_2(n) &= n^9 \cdot 3^{n/3} \\ f_3(n) &= n^{n^{1/9}} \\ f_4(n) &= n^{n/3} \\ f_5(n) &= (n/3)^{n/3} \end{aligned}$$

Problem 1-2. [15 points] **Recurrences**

For each of the following recurrence relations, choose the correct asymptotic solution. In each case, c is a constant.

(a) [3 points] What is the asymptotic expression for $T(x)$ defined as follows?

$$\begin{array}{rcll} T(x) & = & c & \text{for } x \leq 1, \text{ and} \\ T(x) & = & c + T(x-1) & \text{for } x \geq 2. \end{array}$$

1. $\Theta(\log x)$.
2. $\Theta(x)$.
3. $\Theta(x \log x)$.
4. $\Theta(x^2)$.
5. $\Theta(2^x)$.

(b) [3 points] What is the asymptotic expression for $T(x)$ defined as follows?

$$\begin{array}{rcll} T(x) & = & c & \text{for } x \leq 1, \text{ and} \\ T(x) & = & x + 2T(x/2) & \text{for } x \geq 2. \end{array}$$

1. $\Theta(\log x)$.
2. $\Theta(x)$.
3. $\Theta(x \log x)$.
4. $\Theta(x^2)$.
5. $\Theta(2^x)$.

(c) [3 points] What is the asymptotic expression for $T(x, y)$ defined as follows?

$$\begin{array}{rcll} T(x,y) & = & \Theta(x) & \text{for any } x, \text{ and } y \leq 1, \text{ and} \\ T(x,y) & = & \Theta(x) + T(x, y/2) & \text{for any } x, \text{ and } y \geq 2. \end{array}$$

1. $\Theta(\log y)$.
2. $\Theta(x)$.
3. $\Theta(x \log y)$.
4. $\Theta(y \log x)$.
5. $\Theta(\log x \log y)$.
6. $\Theta(xy)$.

(d) [3 points] What is the asymptotic expression for $T(x, y)$ defined as follows?

$$\begin{array}{rcll} T(x,y) & = & \Theta(\log x) & \text{for any } x, \text{ and } y \leq 1, \text{ and} \\ T(x,y) & = & \Theta(\log x) + T(x, y/2) & \text{for any } x, \text{ and } y \geq 2. \end{array}$$

1. $\Theta(\log y)$.

2. $\Theta(x)$.
3. $\Theta(x \log y)$.
4. $\Theta(y \log x)$.
5. $\Theta(\log x \log y)$.
6. $\Theta(xy)$.

(e) [3 points] What is the asymptotic expression for $T(x, y)$ defined as follows? Here, S is an auxiliary function, used to help define T.

$$\begin{array}{lcll} T(x,y) & = & \Theta(x) & \text{for any } x \text{ and } y \leq 1, \\ T(x,y) & = & \Theta(x) + S(x, y/2) & \text{for any } x \text{ and } y \geq 2, \end{array}$$

$$\begin{array}{lcll} S(x,y) & = & \Theta(y) & \text{for any } y \text{ and } x \leq 1, \text{ and} \\ S(x,y) & = & \Theta(y) + T(x/2, y) & \text{for any } y \text{ and } x \geq 2. \end{array}$$

1. $\Theta(\log x + \log y)$.
2. $\Theta(\log x \log y)$.
3. $\Theta(x + y)$.
4. $\Theta(xy)$.
5. $\Theta(x \log y + y \log x)$.

The rest of the problems in Part A involve the *peak-finding problem*, which we discussed in Lecture 1. Recall that a *peak* in a matrix is a location with the property that all of its neighbors (north, south, east, and west) have value less than or equal to the value of the peak. A location counts as a peak if it is on the "boundary" of the matrix and all its neighbors inside the matrix have a value that is less than or equal to its own.

See `peak_finding/README.md` for instructions on running implementations of the following algorithm. This will allow you to definitively check if any counterexamples indeed return an incorrect answer for the algorithm in question.

Problem 1-3. [10 points] **Binary Search on Columns:** This algorithm was described in Lecture 1. It operates by binary search on columns. The algorithm chooses a midpoint column, finds a location containing a maximum element in the column, checks to see whether that element is a peak, and if not, searches recursively in one half of the matrix. This algorithm is implemented in the `algorithm_3` function in `peak_finding/algorithms.py`.

(a) [3 points] **Time complexity analysis:** Prove a good upper bound on the running time for this algorithm.

(b) [7 points] **Correctness proof:**

NOTE: This subproblem starts with an explanation which leads into your actual assignment, which is marked with **Your Assignment**.

In general, to prove correctness of an algorithm, we must show two things: "termination" and "safety", which means that the algorithm never actually does anything wrong. Termination means the algorithm ends and returns an answer. Safety means if it does return, it returns a correct answer. Various methods can be used to show each of these. For this algorithm, termination is easy to prove based on the fact that each call of the algorithm either passes control to a recursive call for an instance of the problem with strictly fewer columns or else terminates without any further recursive calls.

Safety is more difficult. A good approach for a recursive algorithm is to use mathematical induction based on the recursion depth. That means we must show two things:

1. (**Base step:**) When the algorithm returns directly, without making any further calls. it returns a correct answer for the given problem instance.
2. (**Inductive step:**) When the algorithm returns after making one or more recursive calls, if the recursive calls all return correct answers for their smaller problem instances, then the main algorithm returns a correct answer for the main problem instance.

For this algorithm in particular, one might (incorrectly) try to argue the following:

1. (**Base step:**) When the algorithm returns directly, without making any further calls. it returns a peak for the given problem instance. This is easy to see, based on the algorithm code.
2. (**Inductive step:**) When the algorithm returns after making a recursive call, if the recursive call returns a peak for its problem instance, then the main algorithm returns a peak for the main problem instance.

Unfortunately, this doesn't quite work. Just knowing that the recursive call returns a peak isn't quite enough.

For example, consider the following matrix:

$$\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

The algorithm chooses the middle column, finds 9 to be the maximum element in the column, and then recursively considers the two-column matrix to the right of the middle column. Suppose that the recursive call returned with the entry in row 2, column 4, which has value 3. That is a peak in the two-column matrix. But it is not a peak in the larger matrix.

It turns out that we need more: that the returned peak is in fact the maximum value in its column. Basically, this algorithm is solving a stronger problem: to return a peak

that is also the maximum value in its column. For this stronger problem statement, the base step is still easy to see. Now let's consider the inductive step.

Your Assignment: Consider any execution of the algorithm that terminates after making a recursive call. Assume that the recursive call returns a peak for its subproblem that is also the maximum value in its column. Prove that the main algorithm returns a peak for the main problem that is also the maximum value in its column. In doing this, you should not need to argue about how the recursive call operates—just use the fact that it returns a "correct" answer with respect to this stronger hypothesis.

Problem 1-4. [10 points] **Binary Search on Columns, With Column Peaks:** This algorithm is similar to the previous algorithm, but instead of finding a maximum element in the chosen column, this algorithm instead finds a peak. This algorithm is implemented in the `algorithm4` function in `peak_finding/algorithms.py`.

(a) [3 points] **Time complexity analysis:** Prove a good upper bound on the running time for this algorithm.

(b) [7 points] Show that this algorithm is incorrect by defining a particular counterexample matrix, running the code on that counterexample and showing that the result is not a peak. The matrix must be of size no larger than 7 by 7 and different than the one presented in lecture.

Problem 1-5. [10 points] **Binary Search, Alternating Columns and Rows:** This algorithm alternates binary searching on columns and rows. NOTE: within each middle row/column it checks, it looks for the max. This algorithm is implemented in the `algorithm5` function in `peak_finding/algorithms.py`.

(a) [3 points] **Time complexity analysis:** Prove a good upper bound on the running time for this algorithm.

(b) [7 points] Is this algorithm correct or incorrect? If it is correct, give a proof. If it is incorrect, give a counterexample matrix as in the previous problem.

Problem 1-6. [10 points] **Greedy Algorithm:** This algorithm starts in the upper left location (location (1,1)) of the matrix. It executes an iterative loop, at each stage testing whether the current location is a peak. If it is, the algorithm returns. If not, the algorithm moves on to a larger neighbor.

(a) [5 points] Write your own pseudocode code for this algorithm.

(b) [3 points] **Time complexity analysis:** Prove a good upper bound on the running time for your algorithm.

(c) [2 points] Prove that your algorithm is correct. Prove both termination and "safety", that is, that any returned answer must be correct.

Part B

Problem 1-7. [30 points] **Alternative Document Distance**

In Lecture 2, we described the calculation of the "Document Distance" between two documents D_1 and D_2. It was calculated as the angle between the vectors representing the number of occurrences of all words in the documents, where the number of dimensions for the vector is simply the number of distinct words that appear anywhere in either document.

The file `document_distance/docdist.py` outlines a simple Document Distance algorithm. It works by "parsing" each document into a list of words **or** pairs of words, sorting the list alphabetically (keeping all repeated dat), and condensing the sorted list so that all the occurrences of the same word w are represented by a pair $(w, count)$. Then it compares the condensed lists for D_1 and D_2 by calculating the angle between the vectors corresponding to the condensed lists.

We have provided a partial implementation of this algorithm, in `docdist.py`. The next two subproblems involve implementing the remaining parts.

Note: you may use only lists and tuples in your implementation - no dictionaries.

See `document_distance/README.md` for how to run your implementation.

(a) [10 points] Implement the `get_counts` function. Specifically, it is given a sorted list of data which may contain duplicates. The elements are all either words or tuples, as in Part (c). It must return a list of (datum, count) pairs such that each datum in the input list appears in exactly one pair, and its corresponding count is the number of times the datum appeared in the input list. Additionally, the returned list of pairs must be sorted with respect to their datum components.

For example, given, `["is", "is", "it", "it", "what"]`, the function should return `[("is", 2), ("it", 2), ("what", 1)]`.

(b) [10 points] Implement the `get_inner_product` function. Here, a sorted list of (datum, count) pairs is represented by a vector in which each datum corresponds to a different dimension and the value of the vector in that dimension is the count associated with the corresponding datum in the given list. Given two sorted lists of (datum, count) pairs, one for each file, your function should return the inner product of their vectors.

For example, given
`[("is", 2), ("it", 2), ("what", 1)]` and
`[("indeed", 1), ("is", 1), ("it", 1)]`
your function should return 4.

$$0 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 + 0 \cdot 1 = 4$$

(c) [10 points] One might think that simply comparing the number of occurrences of each word might not be a very reliable test of document similarity. As an extreme example,

we could compare a document D_1 with another document D_2 that is obtained by randomly permuting the words in D_1. These would be regarded as identical according to this simple test, but they are certainly quite different. So let's consider a new test: instead of comparing the documents based on number of occurrences of each word, let's consider the number of occurrences of each pair of consecutive words.

Implement the `get_pairs` function in `docdist.py` which, given a list of words in their order of appearance in the document, produces a list of pairs of consecutive words.

For example, `["the", "fox", "says"]` should return `[("the", "fox"), ("fox", "says")]`.