Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld

September 13, 2016
Problem Set 1

Problem Set 1

All parts are due on September 27, 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 0:**

$$\begin{aligned} f_1 &= n^2 \\ f_2 &= n \\ f_3 &= n \log n \\ f_4 &= 2^n \\ f_5 &= (\log n^2)^2 \end{aligned}$$

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

$$\begin{aligned} f_1 &= \log\left((\log n)^3\right) \\ f_2 &= (\log n)^{3 \log 3n} \\ f_3 &= 3^{\log n} \\ f_4 &= n^{3^{\log n}} \\ f_5 &= \log\left(3n^{n^3}\right) \\ f_6 &= (\log \log n)^3 \end{aligned}$$

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

$$f_1 = 4^{3n}$$
$$f_2 = 2^{n^4}$$
$$f_3 = 2^{3^{n+1}}$$
$$f_4 = 3^{3^n}$$
$$f_5 = 2^{5n}$$

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

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

Give solutions to the following.

1. $T(n) = 8T(\frac{n}{3}) + n^2$
2. $T(n) = 10T(\frac{n}{3}) + n^2$
3. $T(n) = 2T(\frac{n}{2}) + n$ using both Master Theorem and recursion tree method
4. $T(n) = T(n/2) + O(n)$ by expanding out the recurrence

(b) [6 points] **Setting up recurrences:**

1. What is the recurrence relation for the the time to naively calculate (using T-notation) the n^{th} factorial number? Assume multiplication between two numbers takes constant time. Then solve this recurrence.
2. What is the recurrence relation for naively calculating the n^{th} fibonacci number? Assume addition between two numbers takes constant time. (*Note*: You might already know how, but we'll talk about how to calculate the n^{th} fibonacci number more efficiently in a later class).

Problem 1-3. [24 points] **Balances and Extremes**

(a) [14 points] **Peak squares in unbalanced arrays**

Consider an $n \times n$ 2d-array A whose cells are colored either blue or red. A 2d-array A is called *unbalanced* if exactly one of its corner cells ($A[0][0], A[n-1][0], A[0][n-1], A[n-1][n-1]$) has a different color from the rest. A 2×2 square in A is a *peak square* if exactly one of the cells in the 2×2 sub-array has a different color from the rest. The goal of this problem is to design an efficient algorithm that finds a peak square in unbalanced 2d-arrays; another way of thinking about the problem is to find a 2×2 unbalanced 2d-array in an $n \times n$ unbalanced 2d-array.

1. [4 points] Suppose A is an unbalanced 2d-array. By knowing the color of $A[0][k]$ and $A[n-1][k]$ (for $0 < k < n-1$), show that we can find a smaller size unbalanced sub-array of A.
2. [5 points] Using the divide routine suggested in part 1 (with careful choice of k), design an algorithm that finds a peak square of unbalanced 2d-arrays.
3. [5 points] Write down the exact recurrence relation for runtime of the algorithm and prove that the running time of your algorithm is $O(\log n)$.

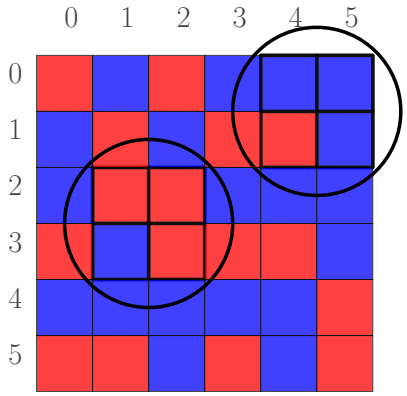

Figure 1: The input is an unbalanced 6×6 2d-array. Two peak squares are shown in this example.

(b) [10 points] **Peak in circular arrays**

A *circular array* is an array whose first and last cells are neighbors. More precisely, in a circular array A of size n, $A[0]$ is a neighbor of $A[n-1]$. The goal of this problem is to modify the "peak finding" algorithm taught in class for 1d-array so that it find a peak of a circular array in $O(\log n)$.

1. [4 points] Suppose we know the values of $A[0], A[k], A[k+1]$ and $A[n-1]$ for $1 < k < n-1$. Considering the maximum value among the four cells, in which (circular) sub-array would you be guaranteed to find a peak?
2. [6 points] Using the divide routine suggested in previous part, design a divide-and-conquer algorithm that returns a peak of circular arrays of size n in $O(\log n)$.

Part B

Problem 1-4. [40 points] **6006LE**

The 6.006 staff believes that there's great room for improvement in the search engine market. To address this problem, they'd like to create a hip new search engine named 6006LE (coincidentally pronounced "google"). But they need your help!

To create the search engine, we will use variations of the "Document Distance" problem that was introduced in class. To test the search engine, we will use a selection of Wikipedia articles, whose contents are contained (as text files) in the problem set folder. For your convenience, we have provided some code to parse the articles.

To check if your code is correct, we have provided some tests in tests.py. There are **not** the same tests the autograder will use! Note that 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 make sure it completes within a few seconds.

Note: The autograder will be running your code with Python 3! Make sure to run the tests with Python 3 as well.

(a) [10 points] **Related articles:** When browsing a particular article, we would like to recommend relevant articles to the user, *i.e.*, those with the least "distance" from that article.

Assume we have a set of n documents $d_1, d_2, \ldots d_n$. Given a term t and document d_i, define the *term frequency* $\text{tf}(t, d_i)$ as the frequency of term t in document d_i.

As discussed in recitation, the "distance" between two articles is the angle, in radians, between the vectors of $\text{tf}(t, d)$ values for all terms in the two articles.

Construct a program that, given an article title, returns the k articles with the least distance from that article. Specifically, implement the get_relevant_articles_doc_dist function in search_engine.py.

Hint: Write a helper function to find the angle between two vectors – you can use it for the next part as well.

Note: While you may look through the staff-provided code to learn different ways of implementing document distance, you should implement it on your own.

(b) [10 points] **Weighing by inverse document frequency:** The 6006LE team thinks they can do better. Instead of simply using a vector of term frequencies, they'd like to weigh terms by their "inverse document frequency" (IDF).

Given a term t and document d_i, define the *inverse document frequency* $\text{idf}(t) = \ln \frac{n}{\text{df}(t)}$, where $\text{df}(t)$ is the number of documents that contain term t. If no documents contain t, then $\text{idf}(t) = 0$.

Therefore, instead of using a vector of term frequencies, we use a vector of "term frequency – inverse document frequency" (TF-IDF) scores, defined as $\text{tfidf}(t, d_i) = \text{tf}(t, d_i) \cdot \text{idf}(t)$.

The "distance" between two articles is now defined as the angle between the vectors of TF-IDF scores. Implement the get_relevant_articles_tf_idf function in search_engine.py, which should return the k articles most relevant to a given article.

(c) [10 points] **Search for an article:** Now, for the most important part: searching! Given a search query, construct a function to return the k articles most relevant to that query. Here, relevance is defined as the sum of the TF-IDF scores for each *distinct* term in the query. Specifically, implement the search function in search_engine.py.

(d) [2 points] **Runtime analysis:** Analyze the runtime for your implementations of parts (b) and (c) above. Include the analysis in your problem set write-up.

You may use the following parameters:

- n, the total number of files
- m, the total number of words in each file (for the sake of analysis, we can assume every file has the same size)
- k, the number of relevant articles to return
- q, the number of distinct terms in the query (for part (c))

(e) [8 points] **Conclusion:** We have two methods of scoring, with and without inverse document frequency. The 6006LE team wants to analyze how these methods perform, and figure out where improvements can be made. Answer the following questions in your problem set write-up:

1. For each method, provide a simple example of where that method returns a more intuitive result, and explain why this happens. How could you modify the TF-IDF calculation to address this? An example consists of some number of "documents" (these can be short strings), and a document whose list of relevant articles makes sense when using one method, but not the other.
2. How does the length of the document have an impact when getting results for a search query? How could you modify the TF-IDF calculation to address this?
3. Let's say we *don't* want the "Apple Inc." article to show up as a related article for "Apple", because fruit-related articles are more relevant. What are the top three related articles (and their scores) for "Apple", using each method? Which method do you think performs better, and why? How could you modify the TF-IDF calculation to address this? This is open-ended – be creative!