Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 13, 2016
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld — 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:**

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

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

1. $f_5 = O(f_2)$ since any polylogarithmic function grows slower than any polynomial function.
2. $f_2 = O(f_3)$ because of the extra log factor
3. $f_3 = O(f_1)$ because logs grow slower than any polynomial.
4. $f_1 = O(f_4)$ because polynomial grows slower than exponential

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

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

Solution: The correct ordering is: $f_1, f_6, f_3, f_5, f_2, f_4$. Here is the justification for the ordering:

1. $f_1 = O(f_6)$ since $\log\log n = O((\log\log n)^3)$ by definition (logarithmic grows more slowly than polylogarithmic).
2. $f_6 = O(f_3)$ since $f_3 = 3^{\log n} = n^{\log 3}$, and every positive polynomial function grows faster than any polylogarithmic function.
3. $f_3 = O(f_5)$ since $\log 3 < 3$, so f_5 is polynomially larger.
4. $f_5 = O(f_2)$ since $f_2 = (\log n)^{\log 27n^3} = (27n^3)^{\log\log n}$, and the n^3 raised to a $\log\log n$ grows faster than just n^3.
5. $f_2 = O(f_4)$ since $f_4 = n^{3^{\log n}} = n^{n^{\log 3}}$, which grows faster than n raised to the $\log\log n$

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

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

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

1. $f_5 = O(f_1)$ since $f_5 = 32^n$ and $f_1 = 64^n$.
2. $f_1 = O(f_2)$ since we when we take logs of both functions, we get $\log f_1 = 3n\log 4$ and $\log f_2 = n^4\log 2$.
3. $f_2 = O(f_4)$ since when we take logs of both functions, we get $\log f_2 = n^4\log 2$ and $\log f_4 = 3^n\log 3$, and exponential functions grow faster than polynomial functions.

4. $f_4 = O(f_3)$ since when we take logs of both functions, we get $\log f_4 = 3^n \log 3$ and $\log f_3 = 3^{n+1} \log 2$.

6 points for correct sequence 3 points for sequence with a pair of functions swapped incorrectly 0 points otherwise

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

(a) [12 points] **Solving recurrences:**
Give solutions to the following.

1. $T(n) = 8T(\frac{n}{3}) + n^2$

   Solution: $T(n) = \Theta(n^2)$ by the Master Theorem
2. $T(n) = 10T(\frac{n}{3}) + n^2$

   Solution: $T(n) = \Theta(n^{\log_3 10})$ by the Master Theorem
3. $T(n) = 2T(\frac{n}{2}) + n$ using both Master Theorem and recursion tree method

   Solution: $T(n) = \Theta(n \log n)$ b the Master Theorem or recursion tree
4. $T(n) = T(n/2) + O(n)$ by expanding out the recurrence

   Solution: $T(n) = O(n)$ by seeing that 1 + 1/2 + 1/4 + ... is a series that adds to a constant number

(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.

   Solution: $T(n) = T(n-1) + O(1)$. Solved, this is $T(n) = \Theta(n)$.
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).

   Solution: $T(n) = T(n-1) + T(n-2) + O(1)$

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.

   Solution: Without loss of generality, we can assume that all corners of A other than $A[n-1][n-1]$ are red. By case analysis, it is straightforward to check that we can always find a smaller unbalanced sub-array of A whose corners contain $A[0][k]$ and $A[n-1][k]$ (see Figure **??**).
   Note that we can define the divide step similarly on the rows by looking at the color of $A[k][0]$ and $A[k][n-1]$.

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.

   Solution: Using the divide method described above and setting $k = \lfloor (n+1)/2 \rfloor$, by checking the color of $A[0][\lfloor (n+1)/2 \rfloor]$ and $A[n-1][\lfloor (n+1)/2 \rfloor]$, we obtain an unbalanced sub-array of size at most $n \times \lceil n/2 \rceil$. Applying the divide method one step further (this time, we check the first and last cells in the middle row of the reduced unbalanced array constructed at the end of the first iteration), by the same argument, we can find an unbalanced sub-array of size at most $\lceil n/2 \rceil \times \lceil n/2 \rceil$.
   The sub-array that we are considering in each step is always unbalanced with at least two rows and two columns; if the number of columns (similarly rows) becomes less than three, we only perform divide step on rows. Moreover at each step either the number of rows or the number of columns of the unbalanced sub-array reduces by a factor of 2. Hence we finally get to a 2×2 unbalanced sub-array which itself is a peak square.

3. [5 points] Write down the exact recurrence relation for runtime of the algorithm and prove that the running time of your algorithm is $O(\log n)$.

   Solution: To write the recurrence relation for the runtime of the described algorithm, we can define the time function either as $T(n, n)$ or $T(n^2)$. The first function denotes the running time of the proposed algorithm to find a peak square in a 2d-array of size $n \times n$ and the second definition denotes the running time of

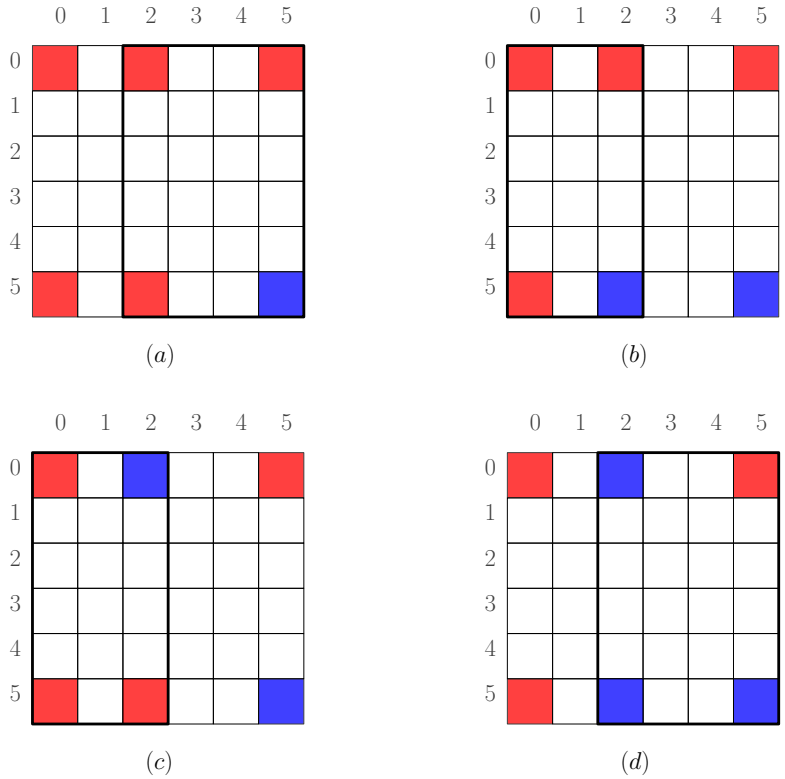

Figure 1: Four different cases based on color of $A[0][k]$ and $A[n-1][k]$. Note that in all four cases, there exit a smaller unbalanced sub-array whose corners include $A[0][k]$ and $A[n-1][k]$.

the algorithm to find a peak square in a 2d-array with n^2 cells. You can use both definitions. Here we write the recurrence relation using the second definition.

$$T(n^2) = c + T(\frac{n^2}{2})$$
$$T(n^2) = c + c + T(\frac{n^2}{4})$$
$$\vdots$$
$$T(n^2) = c + c + \cdots + c + T(\frac{n^2}{2^i})$$

Thus, by setting $i = (\log^2 n) - 2 = 2\log n - 2$, $T(n^2) = c \cdot (2\log n - 2) + T(4) = O(\log n)$.

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

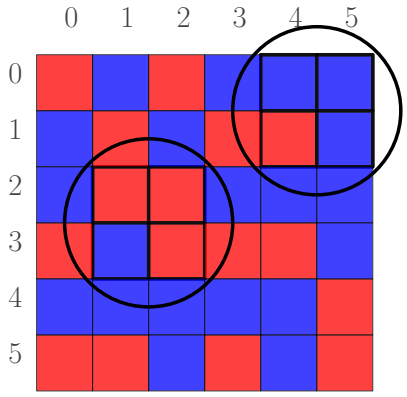

Figure 2: The input is an unbalanced 6×6 2d-array. Two peak squares are shown in this example.

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?

   Solution: We consider four different cases depending on which cell among $A[0]$, $A[k]$, $A[k+1]$ and $A[n-1]$ obtains the maximum value.
   - $A[0] > A[k], A[0] > A[k+1], A[0] > A[n-1]$: A must have a peak in the range 0 to k. Thus a peak of A will be in the circular sub-array $A[0, ..., k]$.
   - $A[k] > A[0], A[k] > A[k+1], A[k] > A[n-1]$: A must have a peak in the range 0 to k. Thus a peak of A will be in the circular sub-array $A[0, ..., k]$.
   - $A[k+1] > A[0], A[k+1] > A[k], A[k+1] > A[n-1]$: A must have a peak in the range $k+1$ to $n-1$. Thus a peak of A will be in the circular sub-array $A[k+1, ..., n-1]$.
   - $A[n-1] > A[0], A[n-1] > A[k], A[n-1] > A[k+1]$: A must have a peak in the range $k+1$ to $n-1$. Thus a peak of A will be in the circular sub-array $A[k+1, ..., n-1]$.

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 $O(\log n)$.

   Solution: Our algorithm for finding peak in circular arrays is a divide-and-conquer approach which is very similar to the algorithm taught in class for finding peak in 1d-arrays. Using the result of previous part and setting $k = \lfloor n/2 \rfloor$, by looking at value of $A[0], A[k], A[k+1]$ and $A[n-1]$, we can find a circular sub-array of A (of size half of the size of A) that contains a peak of A. Moreover, the

stop rule for our recursion is the following: if the size of the circular array is less than 4, return the maximum of $A[0], A[1]$ and $A[2]$ (which is trivially a peak). Let $T(n)$ denote the running time of our algorithm over a circular array of size n. Then, we have the following recursion relation for T:

$$
\begin{aligned}
T(n) &= 4c + T(n/2) \quad \%c \text{ is a constant} \\
T(n) &= 4c + 4c + T(n/4) \\
&\vdots \\
T(n) &= 4c + 4c + \cdots + 4c + T(n/2^i) \quad \% \text{ after } i \text{ iterations}
\end{aligned}
$$

Thus $T(n) = i \cdot 4c + T(n/2^i)$ where $2^i \leq n/3$. Setting $i = \log(n/3)$ and $T(3) = 4$, we have $T(n) = O(\log n)$.

“

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!

Solution:

(a) See staff code.

(b) See staff code.

(c) See staff code.

(d) The following analyses assume the dictionary method to calculate TF and IDF scores.

To get relevant articles, it takes $O(mn)$ to scan through all the words, $O(n \lg n)$ to sort the values (using mergesort or Python's default sort), and $O(k)$ to slice and return the top k articles. Thus the running time is $O(mn + n \lg n + k)$.

Finding a search query involves $O(qn)$ to find the TF and IDF scores for each word in the query for each document, $O(n \lg n)$ to sort the n articles' scores, and $O(k)$ to slice and return the top k values. Thus the running time is $O(qn+n \lg n+k)$.

In both of these, the k term isn't really necessary, because it's safe to assume that $k = O(n)$. In addition, if you implemented insertion sort, the sorting would take $O(n^2)$ instead.

Note: Several students decided to not include the $n \lg n$ term, under the assumption that it's much smaller than the mn term. However, no information was given about the relationship between m and n!

(e) 1. For an example of TF-IDF being better, consider the following corpus:

- $d_1 =$ "the cat the mat the rat the bat"
- $d_2 =$ "the adorable dog is running"
- $d_3 =$ "look at the dog go"

Using document distance, the most relevant articles for d_2 are $[d_1, d_3]$ in that order, and similarly the most relevant articles for d_3 are $[d_1, d_2]$. However, we'd expect that the two documents related to dogs have higher relevance – because of the occurences of "the" in the first document, it has much higher weighting than it needs to. Using TF-IDF scoring instead fixes this problem.

For an example of document distance performing better, consider the following corpus:

- $d_1 =$ "i love all dogs"
- $d_2 =$ "dogs are great"
- $d_3 =$ "who let the dogs out"

In this case, using TF-IDF shows that none of these articles have similarity; this is because their only shared word, "dogs", has a IDF score of zero. With document distance, all of these documents have some similarity.

One way to address this could be to make the IDF score positive, but small, for words that appear in every document – this way, they aren't entirely ignored. For example, setting $\text{idf}(t) = \ln\left(1 + \frac{n}{i\text{df}(t)}\right)$ is a simple way to address this.

2. When searching for a query, TF-IDF scoring is biased towards longer documents because the term frequencies are higher. One way to address this problem is by normalizing frequencies, i.e. dividing each frequency by the length of the document.

3. The top three related articles for "Apple" using document distance are:
   - Apple Inc. (score 0.44880829036)
   - Banana (score 0.451814653477)
   - Tomato (score 0.455796896735)

   The top three related articles for "Apple" using TF-IDF are:
   - Apple Inc. (score 1.24535507606)
   - Macintosh (score 1.4112032101)
   - Pear (score 1.46202665011)

   It seems like TF-IDF performs worse, because it has two tech-related articles (Apple Inc. and Macintosh) while document distance only has one. This is likely because the IDF of words like "Apple" are really high, which prioritizes them over fruit-related articles. It's also correlated with the large length of the Apple Inc. article. The problem here is that the same word is used in different contexts: one way to address this could be to group together words that have a similar meaning and figure out the overall context of an articlear. For example, if we know that "Apple" is a fruit-related article, then give higher weighting to other fruit-related articles like "Pear." Better and more sophisticated methods might involve ML and NLP to figure out the main topics of an article.
   Several students assumed that "Apple" was a search query instead of an article – if the numbers was right and there was a good justification, students were given full credit.