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 Tuesday, September 28 at 11:59PM.

Name: Milo H. Knowles

Collaborators: Gillian Belton, Magnus Johnson, Wala'a Alkhanaizi

Part A

Problem 1-1.

(a) In order of increasing growth:

$$[(\log_2 n^2)^2, n, n\log n, n^2, 2^n]$$

First,

$$(\log n^2)^2 = 4(\log n)^2$$

which grows more slowly than n because generally the logarithm of a number is less than the number itself. This also explains why $n\log n$ is before n^2. An expression with a growing variable in the exponent will be bigger than an expression with a constant exponent, so $O(2^n)$ is the largest.

(b) In order of increasing growth:

$$[f_1, f_6, f_3, f_5, f_2, f_4]$$

First, some simplifications:

$$f_1 = \log((\log n)^3) = 3log(\log n)$$

$$f_5 = \log(3(n^{n^3})) = n^3\log(3n)$$

f_1 has the smallest order of growth because it is a log of a log. f_6 is the next smallest because it has a constant exponent. f_3 is raised to the power $\log n$ which grows more slowly than the n^3 term of f_5. f_2 has an exponent that grows with n, but f_4 is biggest because it has two layers of exponents, one of which is constant, and one of which grows with n.

(c) In order of increasing growth:

$$[f_5, f_1, f_2, f_4, f_3]$$

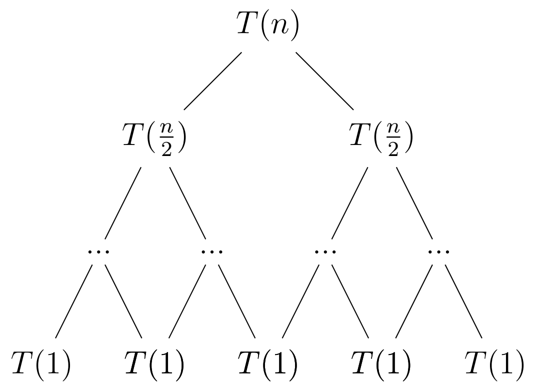
If we realize that $f_1 = 4^{3n} = 2^{6n}$, then the exponents can be easily compared. Ranking the exponents from smallest to largest will give the correct order for the functions in this group.

$$5n < 6n < n^4 < 3^n < 3^{n+1}$$

Problem 1-2.

(a) Solving Recurrences

1. $T(n) = 8T(\frac{n}{3}) + n^2 = O(n^2)$
2. $T(n) = 10T(\frac{n}{3}) + n^2 = O(n^{\log_3 10})$
3. $T(n) = 2T(\frac{n}{2}) + n$
   Using the Master Theorem, $T(n) = O(n \log n)$
   Using the recursion tree method:



The $2T(\frac{n}{2})$ term is the split time.
The n term represents the merge time.

At the 0th level of the tree, $T(n)$ has to merge 2 leaves, each of size $\frac{n}{2}$. Therefore it takes n time to combine on that level. At each level, the time to combine is still n, because there are twice as many leaves, but each leaf has half the size. There are $\log n$ levels of the tree, and each level has a combine time of n, so $T(n) = O(n \log n)$.

4. $T(n) = T(\frac{n}{2}) + O(n)$
   Expanding the recurrence:
   $T(n) = T(\frac{n}{2}) + O(n)$

$T(n) = T(\frac{n}{2}) + cn$
$T(n) = T(\frac{n}{4}) + c\frac{n}{2}cn$
$T(n) = T(\frac{n}{8}) + c\frac{n}{4} + c\frac{n}{2} + cn$
...
$T(n) = T(1) + cn(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^i})$
This sum has a finite value, so $T(n) = O(n)$.

(b) Setting up recurrences:

1. To naively calculate the n^{th} factorial number:
   $T(n \leq 1) = O(1)$
   $T(n) = T(n-1) + O(1)$

   Solution:

$$T(n) = T(n-1) + O(1)$$
$$T(n) = T(n-2) + O(1) + O(1)$$
$$T(n) = T(n-3) + O(1) + O(1) + O(1)$$
$$T(n) = n \cdot O(1)$$
$$T(n) = O(n)$$

2. To naively calculate the n^{th} Fibonacci number:
   $T(n \leq 1) = O(1)$
   $T(n) = T(n-1) + T(n-2) + O(1)$

Problem 1-3.

(a) Peak squares in unbalanced arrays.

1. If we know the color of $A[0][k]$ and $A[n-1][k]$ and the colors of the four corners of A, then we can find a smaller sized unbalanced sub-array of A. To do this, we must consider two cases.
   - If the colors at $A[0][k]$ and $A[n-1][k]$ are **the same**, then we choose the corners from the side of A with different colors (this could be either the left corners or right corners). Now we have a sub-array with exactly one corner that is a different color.
   - If the colors at $A[0][k]$ and $A[n-1][k]$ are **different**, then we choose the corners from the side of A with the matching colors. Now we have a sub-array with exactly one corner that is a different color.
2. If we choose $k = \frac{n}{2}$, then we can divide the array A into an unbalanced sub-array with width $\frac{n}{2}$ and height n. We can then divide in half vertically, producing a square, unbalanced sub-array with width $\frac{n}{2}$ and height $\frac{n}{2}$. If we repeat this process recursively, we will eventually have a 2x2 unbalanced subarray, which is necessarily a peak square.

3. $T(n) = T(\frac{n}{2}) + O(1)$
   At each step, the algorithm reduces the size of the array to $\frac{1}{2}$ it's previous size. Each recursive step takes $O(1)$ time, and there will be $\log_2 n$ recursive steps required to reach a 2x2 array. Therefore, the algorithm takes $O(\log n)$ time.

(b) 1. First, note that $A[0]$ and $A[n-1]$ are an adjacent pair of array entries and $A[k]$ and $A[k+1]$ are another adjacent pair of array entries. If we find the maximum entry x of these 4 entries, we also know that there is an adjacent entry (either before or after in the array) that is necessarily smaller than x. Now consider the 2 remaining entries (we eliminate x and the adjacent entry from our original four). For example, if our $x = A[k]$, then the adjacent entry is $A[k+1]$, so we are left with $A[0]$ and $A[-1]$. Let y be the entry from these two remaining entries that is closest to x. Then, on the circular sub-array A[x:y] inclusive, we are guaranteed to find a peak.

2. If we choose k such that $k = \frac{n}{2}$, we can divide our search space in half during the divide step. We are guaranteed to find a peak on the new search space, which has $\frac{n}{2}$. The divide step takes $O(1)$ time, because we are just finding the maximum of four array entries during this step. Once we have a subarray with size $n \leq 4$, we will find the maximum entry directly. There will be order $\log_2 n$ divide steps to reduce the array to this size. Therefore, this algorithm will find a peak in $O(\log n)$ time.

Part B

Problem 1-4.

(a) *Submit your implementation on alg.csail.mit.edu*

(b) *Submit your implementation on alg.csail.mit.edu*

(c) *Submit your implementation on alg.csail.mit.edu*

(d) Below is a list of the constants used in runtime analysis for parts (b) and (c):

- n, the total number of articles
- m, the total number of words in each article (assumed to be the same for every article)
- k, the number of relevant articles to return
- q, the number of distinct terms in the query (for part (c))
- w, the number of unique words in the corpus (no repeats)
- l, the total number of words (with repeats) in the query (for part (c))

The implementation of **get-relevant-articles-tf-idf** from **part(b)** is outlined below:

1. For each article, build a term-frequency dictionary. This takes $\theta(nm)$ time, because the code iterates over exactly n articles, and m words in each article.
2. Next, the function builds a document-frequency dictionary, storing each word in the corpus as a key, and the number of articles that contain that word as a value. This takes $\theta(nm)$ time, because the code iterates over n articles, and m words in each article.
3. For each *u*nique term in the corpus, the inverse-document-frequency is computed and added to a new **corpusIDFDict** dictionary. This takes $\theta(w)$ time, because the code only considers unique words, without repeats.
4. For each article in the corpus an angle is computed between that article and the query article. $\theta(n-1)$ comparisons must be done, because the query article is not compared to itself. The helper function **computeAngleBetweenWFIDFDicts** is used to compute the angle between two WFIDF dictionaries. It runs in $\theta(5m)$ time.
   - The helper function **computeAngleBetweenWFIDFDicts** is used to compute the angle between two WFIDF dictionaries. It runs in $\theta(5m)$ time.
   - $\theta(m)$ to convert one term-frequency dictionary to a TFIDF dictionary.
   - $\theta(m)$ to convert the other term-frequency dictionary to a TFIDF dictionary.
   - $\theta(m)$ to compute the dot product of the two TFIDF dictionary values.
   - $\theta(m)$ to compute the magnitude of one TFIDF dictionary.
   - $\theta(m)$ to compute the magnitude of the other TFIDF dictionary.
5. The angle between every article and the query article is stored in a list. There are $n-1$ elements in the list, so it will take $O(n \cdot \log_2 n)$ time to sort this list.
6. To return the k closest articles from the sorted list, it will take $\theta(k)$ time.

The implementation of **search** from **part(c)** is outlined below:

1. For each article build a term-frequency dictionary. This takes $\theta(nm)$ time, because the code iterates over exactly n articles, and m words in each article.
2. Next, the function builds a document-frequency dictionary, storing each word in the corpus as a key, and the number of articles that contain that word as a value. This takes $\theta(nm)$ time, because the code iterates over n articles, and m words in each article.
3. For each *u*nique term in the corpus, the inverse-document-frequency is computed and added to a new **corpusIDFDict** dictionary. This takes $\theta(w)$ time, because the code only considers unique words, without repeats.
4. Parse the user's query and remove repeated words. This takes $\theta(l)$ time.
5. For each article, add up the TFIDF scores for each distinct word in the query to get a total score. This takes $\theta(nq)$ time, because there are n articles and q words in the query that must be retrieved from an article's TFIDF dictionary.

6. It takes $O(n \cdot \log_2 n)$ time to sort the list of n articles by their TFIDF score.
7. It takes $\theta(k)$ time to take the k best non-negative articles from the sorted list of articles.

(e) Part e

1. Although the **get-relevant-articles-tf-idf** method is clearly superior to the **get-relevant-articles-doc-dist** at returning relevant articles, both methods show cases where they return more intuitive results.
   - The document-distance method for finding relevant articles is advantageous when the corpus of articles is relatively homogenous in terms of topic. If all articles focus on a similar topic, say astronomy, then words like ”planet” and ”telescope” might occur in every article, and therefore be ignored by the TF-IDF method?this is bad for finding relevant documents. However, the document distance would not ignore these words, and do a better job of providing similar articles.
   - In general, TF-IDF is useful for searching through a diverse set of documents to find results that are tailored to the query article. This is because TF-IDF gives common words a low weight, and ignores words that appear in every article (and, the, with, because). It gives high weight to the words in each article that are unique.
2. Generally, longer documents will have higher word frequencies for words that are related to the topic of that article. These document-specific words are also likely to have a high TF-IDF weight, because they are unique to that article. Therefore, they will make a large contribution computed ”relevance” of the document. So long documents will tend to dominate search results. We could modify the TF-IDF calculation so that word-frequencies are normalized based on the total number of distinct words N in an article. For each word w in an article which occurs with frequency F_w, the TF-IDF calculation would look like:

$$TFIDF_w = (F_w \cdot IDF_w)/N$$

3. The top 3 searches for ”Apple” using TF-IDF are:
   ’Apple Inc’, 1.2453550760561019
   ’Macintosh’, 1.4112032100984244
   ’Pear’, 1.4620266501066561

   And the top 3 searches for ”Apple” using document-distance are:
   ’Apple Inc’, 0.4488082903602776
   ’Banana’, 0.4518146534769454
   ’Tomato’, 0.4557968967347264

Here, document-distance performs better, because it returns two fruits, whereas TF-IDF returns only one fruit. To improve TF-IDF, we could remove all instances of the title word, ”Apple,” from the article. This would remove the ambiguity surrounding the word ”apple,” and leave behind words that describe the characteristics of an apple (the fruit). Therefore, a search would find documents that have similar descriptors (other fruit articles), and not just articles the same title.