MATH 449, HOMEWORK 3

DUE SEPTEMBER 29, 2017

Part I. Theory

Problem 1. Prove the following statements about the order of growth of sequences, expressed using the big-$\mathcal{O}$ notation:

- **a.** If f is a polynomial of degree d, then $f(n) = \mathcal{O}(n^d)$. (This says that the order of growth depends only on the polynomial's leading term.)
- **b.** For every $d \in \mathbb{N}$, $n^d = \mathcal{O}(e^n)$. (This says that exponential growth is faster than polynomial growth of any order.) Hint: Use the infinite-series definition of e^n.

Problem 2. Show that the product of two $n \times n$ upper triangular matrices is again an upper triangular matrix. Hint: Split the sum $\sum_{k=1}^{n} u_{ik} v_{kj}$ into $\sum_{k=1}^{j} u_{ik} v_{kj} + \sum_{k=j+1}^{n} u_{ik} v_{kj}$.

Problem 3. Given an $n \times n$ upper triangular matrix

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn}, \end{pmatrix}$$

show that $\det U = u_{11} u_{22} \cdots u_{nn}$, i.e., that the determinant is just the product of the diagonal entries. Hint: Use induction on n and expansion by minors.

Part II. Programming

These programming exercises will introduce you NumPy *arrays*, which are used to represent vectors and matrices. You can do these exercises by entering commands directly in the IPython console in Anaconda. Before you start, run the command `from pylab import *` in order to import all the necessary tools for working with arrays.

Problem 4. Define `x = [1,2,3]` and `y = array([4,5,6])`. Here, `x` is an ordinary Python *list*, while `y` is a NumPy *array*. What is the output of the following commands?

- **a.** `x + x`
- **b.** `y + y`
- **c.** `x + y`

d. `x * 3`
e. `y * 3`
f. `x * y`

There are several ways to reference the entries of a NumPy array `a` of length n:

- *single indices*: If $i = 0, \ldots, n-1$ is nonnegative, then `a[i]` refers to the ith entry of `a`. (Like C/C++/Java, but unlike MATLAB, Python indexing starts at 0, not 1!) On the other hand, if $i = -n, \ldots, -1$ is a *negative* integer, then `a[i]` means the same thing as `a[n+i]`. For example `a[0]` is the first entry of `a`, and `a[-1]` is the last entry.
- *multiple indices*: We can access several elements using a list (or array) containing multiple indices. For example, if `i = [1,3,2]`, then `a[i]` is a new array with entries `a[1]`, `a[3]`, and `a[2]`. This can also be done directly by writing `a[[1,3,2]]`. (Note the double brackets.)
- *slicing*: "Slicing" is a special way to access a range of multiple indices using the colon symbol '`:`'. (It should be especially familiar to former MATLAB users.)

  If i and j are single indices, then `a[i:j]` contains the range of entries beginning with (and including) `a[i]`, up to (and excluding) `a[j]`. For example, `a[1:4]` contains the entries `a[1]`, `a[2]`, and `a[3]`.

  If i and/or j is omitted, then the range is assumed to go all the way to the beginning/end of the array. For example, `a[:3]` contains `a[0]`, `a[1]`, and `a[2]`. Similarly, `a[2:]` contains the consecutive entries of `a` beginning with `a[2]`, `a[3]`, etc. As a special case, `a[:]` is just `a`, since the range includes all indices.

  Finally, `a[i:j:k]` contains the entries beginning with (and including) `a[i]`, up to (and excluding) `a[j]`, taking steps of size k. If k is omitted, then it is assumed to be 1: the expressions `a[i:j:1]`, `a[i:j:]`, and `a[i:j]` give the same result. The step size k can also be negative, in which case the range goes backwards. For example, `a[0:6:2]` contains `a[0]`, `a[2]`, and `a[4]`, while `a[6:0:-2]` contains `a[6]`, `a[4]`, and `a[2]`. The expression `a[::2]` gives the entries of `a` with even indices.

(See `http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html` for many, many more details.)

Problem 5. Assume that `a` is a NumPy array. Write the expression that gives each of the following:

a. The array containing the first and last entries of `a`.
b. The array containing the entries of `a` in reverse order.
c. The array containing the entries of `a` with odd indices, in order.

In addition to vectors, arrays can also represent matrices. As we have seen, a vector can be constructed by passing a *list of entries* to the `array()`

constructor. To construct a matrix, we instead pass a *list of rows* to the `array()` constructor. Each row is itself a list of entries, so this is a nested list of lists. For example, `A = array([[1,2],[3,4]])` corresponds to the matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, i.e., the matrix whose rows are $(1, 2)$ and $(3, 4)$.

The (i, j)th entry of a matrix A can be obtained either as `A[i][j]` (i.e., the jth entry of the ith row of A) or as `A[i,j]`. Multiple indices and slicing can be used for both rows and columns, just as they were for vector arrays in the previous problem. (Again, remember: indices start at 0, not 1!)

Problem 6.

a. Let `A = array([[1,2],[3,4]])` and `I = eye(2)`. (The command `eye(n)` gives the $n \times n$ identity matrix. The name of this function is a pun that comes from MATLAB: the word *eye* sounds like I.) What do `A*I` and `A*A` do? What about `dot(A,I)` and `dot(A,A)`? (Note: For vector arrays, the function `dot()` takes the dot product.)

b. Write a multiple-index expression that gives the array of diagonal entries `A[0,0]` and `A[1,1]`. (Do not use the function `diag()`—which, for future reference, is very useful!)

c. Write a slicing expression that gives the first column of A.