Homework 3

J.B. Morris, Trixie Roque, Khalid Seirafi

04/07/15

1. Python code for bozosort is in GitHub repo ”trixr4kdz/cmsi282/homework3” (Note: all codes for this assignment are in this folder).

# items	Avg. runtime
2	0.0242834091187 ms
3	0.0498423576355 ms
4	0.154537439346 ms
5	1.04509353638 ms
6	7.71605968475 ms
7	53.8139247894 ms
8	405.303764343 ms
9	3.15729027271 s
10	37.2595239639 s

2. Code is in GitHub repo

3. The original message is:
LET US CHANGE OUR TRADITIONAL ATTITUDE TO THE CONSTRUCTION OF PROGRAMS INSTEAD OF IMAGINING THAT OUR MAIN TASK IS TO INSTRUCT A COMPUTER WHAT TO DO LET US CONCENTRATE RATHER ON EXPLAINING TO HUMAN BEINGS WHAT WE WANT A COMPUTER TO DO

Mapping:

Alphabet	Cipher Letter	Alphabet	Cipher Letter
A	O	N	R
B	S	O	W
C	E	P	F
D	K	Q	Y
E	I	R	T
F	X	S	D
G	H	T	Q
H	V	U	L
I	M	V	Z
J	C	W	B
K	P	X	J
L	U	Y	G
M	A	Z	N

4. Using the keyphrase, this table is created:

D	A	R	N	O
T	H	E	C	Y
P	L	S	I/J	Q
U	B	F	G	K
M	V	W	X	Z

So the original message is:

COMPUTER SCIENCE IS NO MORE ABOUT COMPUTERS THAN ASTRONOMY IS ABOUT TELESCOPESS

5. We would need to find what the 2 prime numbers, p and q, are such that $pq = N$. We can use a python program in order to do that since the number is not large enough that factoring will be a problem. Then we would have $p = 822893$, $q = 886969$. Then we would need to find the modulus, M, such that $M = (p-1)(q-1)$. So $M = (p-1)(q-1) = (822892)(886968) = 729878871456$. Then to find the secret key, d, we use the property, $de \equiv 1 \mod M$, to find the inverse mod of e from the public key. We can use a language like Scala which has a built-in function modInverse() to find the d. Then d = BigInt("5").modInverse(BigInt("729878871456")) = 583903097165. Therefore, the private key is $(N, d) = (729880581317, 583903097165)$

6. (1.45)

(a) We would want digital signatures so that we do not have to rely on paper documents and still have a way to ensure that the intended person was the actual sender. Since the digital signatures would be unique to the person (by having a function that takes in their own secret key along with the message) which would simulate how each person has their own unique handwritten signature, there would be no need for the other party to have a paper copy of the file. There is no need to sign a paper by hand and then mail it or bring it in person since the signature would be produced electronically. The verify algorithm would then ensure that a third party isn't producing the documents since that would mean using invalid signatures.

(b) Since d is the inverse mod of e, $de(modN) \equiv 1modN = 1$:
Given $sign(M,d) = M^d(modN)$, (N,e), and M. Since we know that $M^d \equiv M^d(modN)$, anyone who has the signature M^d and the actual message M can verify the signature by simply raising the signature to the power of e and confirming that the original message modN is the result, $(M^d)^e \equiv M(modN)$.
Implementation for this problem, "digi_sign.py", is in GitHub repo.

(c) p = 113, q = 23, e = 13, d = 2085, N = 2599
The mapping of the letters of the name to values in $[0, 1, ..., N-1]$ is random such that:
name_map = {'T':20,'R':168,'I':131,'X':459,'E':2000,'O':765,'Q':1007,'U':313}
so T $\rightarrow 20 = m_1$. Then $sign(m_1, d) = m_1^d modN = 20^{2085}(mod2599) = 2330$
To check: $(m_1^d modN)^e = m_1^1 modN = m_1 modN$ so $(20^{2085} mod2599)^{13} = 20^1 mod2599 = 20$

(d) p = 17, q = 23, e = 17
Then L = (p-1)(q-1) = 352
Then to find the secret key d to raise the message to, we have to make use of the property $de \equiv 1(modL) = 145(mod352)$. So Eve should raise it to the 145th power.

7. (1.46)

(a) $sign(M,d) = M^d(modN)$ which is what Eve would get if Bob agreed to sign anything from Eve. Then since Eve would have Bob's public key (N,e) and e is the inverse mod of d, she can just raise M^d to the power of e then $modN$ the result to get M. In other words, since $de \equiv 1(modL)$ for some number L and X is the signature:

$$\begin{aligned} X &= M^d modN \\ X^e modN &= MmodN \\ &= M \end{aligned}$$

(b) Suppose Eve chose some random number r such that r is unlikely to be text, then she has to find out if $gcd(N, r) = 1$ so that she can find the modular inverse of r. If $gcd(N, r)$ happens to not be 1, then N must be divisible by r since there are only 2 numbers that can divide N which will reveal p and q to Eve. If N and r are relatively prime, then Eve can find some number s such that $s \equiv r^{-1} modN$. She can encrypt this s using Bob's public key. Then she can take Alice's encrypted message to Bob, call it x^e, and multiply it with s^e such that $x^e s^e \equiv (xs)^e modN$. She can then ask Bob to sign this and Eve will get $(xs)^{ed} modN = (xs) modN$. Then all Eve has to do to get x is to calculate $xss \equiv xsr^{-1} \equiv modN$.

8. (2.4) (a) Recurrence Relation A: $T(n) = 5T(\frac{n}{2}) + n$. So using the Master Theorem yields

$$O(n^{log_2 5}) = O(n^{2.32})$$

(b) Recurrence Relation B: $T(n) = 2T(n-1) + 1$. We can't use the Master Theorem since there is no b so we use repeated substitution which yields:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T((n-1)-1)+1) = 2^2T(n-2) + 3 \\ &= 2(2^2T(n-2)-1)+3)+1 = 2^3T(n-3) + (2*3) + 1 \\ &... \\ &... \\ &= 2^{n-2}T(n-(n-2)-1) + 2(n-2) + 1 = 2n - 2T(1) + 2n - 3 \\ &= 2^nT(1) + 2n \end{aligned}$$

so the complexity for B is $O(2^n)$

(c) Recurrence Relation C: $T(n) = 9T(\frac{n}{3}) + n^2$. So since $log_b a = log_3 9 = 2 = d$, using the Master Theorem yields

$$O(n^{log_3 9}) = O(n^2 logn)$$

I will pick Algorithm C since its complexity is less than both A and B

9. (2.12) Recurrence relation: $T(n) = 2(\frac{n}{2}) + 1$

   Then running the program, we get n - 1 lines

   Looking at the long run, we get $\theta(n)$

10. (2.23) Please look at code titled "Problem_223.py" in GitHub repo

    (a) Look at the function "majority()" in python code

    (b) Yes; look at function "majority_linear (w/ prune)" in python code

11. (3.2)

    (a) Tree edges: A → B, B → C, C → D, D → H, H → G, G → F, F → E

    Forward edges: A → F, B → E

    Back edges: D → B, E → D, E → G, F → G

    Let node = (num_pre_vertices, num_post_vertices):

    Then A = (1, 16), B = (2, 15), C = (3, 14), D = (4, 13), H = (5, 12), G = (6, 11), F = (7, 10), E = (8, 9)

12. (3.8) (a) We can model this problem using a depth-first search tree. A depth-first tree is a connected graph with no circuits and that visits portions of a graph from a given vertex; if it is not possible to move to a node at a lower level, we can backtrack to another portion and visit those nodes. In this problem, we want to explore the states we will have by transferring the contents of one container to another. The root node will be the initial state of the containers (in this case, we define state as the amount of water in each) (i.e. [(10,0),(7,7),(4,4)] for the 10- , 7-, 4- pint containers respectively). The children of the parent nodes will be the states of the container after the contents of a specific container have been changed (containers can be filled, emptied, or contents be transferred). The last node created from the tree (the leaf) will have the state that we want the containers to be in so that one of the containers has the amount we want (that is, 2 pints). A question we can ask for a depth-first search tree is how many steps can we make in order to get to the desired state.

    How many different paths can we take in order to get to the goal state.

    (b)

    (c) Python implementation "water_problem.py" is in GitHub repo (Note: this program is taken from https://github.com/jamescooke/water-pouring-python)