Anu Varshneya

GitHub: https://github.com/anuvarsh

Homework 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. hw3/Bozosort.py  import random  def is\_sorted(numbers):  for i in range(1, len(numbers)):  if numbers[i-1] > numbers[i]:  return False  return True  def bozosort(numbers):  while not is\_sorted(numbers):  index\_1, index\_2 = random.randint(0, len(numbers)-1), random.randint(0, len(numbers)-1)  numbers[index\_1], numbers[index\_2] = numbers[index\_2], numbers[index\_1]  (all times in seconds)   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | List Size | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | **Average** | | 2 | 0.000116 | 0.000063 | 0.000106 | 0.000097 | 0.000071 | **0.000091** | | 3 | 0.000347 | 0.000150 | 0.000178 | 0.000152 | 0.000026 | **0.000171** | | 4 | 0.000298 | 0.000032 | 0.000196 | 0.000024 | 0.000320 | **0.000174** | | 5 | 0.002025 | 0.002348 | 0.000323 | 0.002598 | 0.000667 | **0.001592** | | 6 | 0.004018 | 0.001374 | 0.016966 | 0.004070 | 0.004450 | **0.006176** | | 7 | 0.049771 | 0.001385 | 0.045764 | 0.062408 | 0.035294 | **0.038924** | | 8 | 0.036734 | 0.043729 | 0.728373 | 0.151329 | 0.312168 | **0.254467** | | 9 | 0.013107 | 2.822606 | 4.836567 | 1.576887 | 1.052645 | **2.060362** | | 10 | 1.271833 | 56.325354 | 12.183629 | 38.464283 | 4.456985 | **22.540417** | |
| 2. hw3/Vigenere.py  # Understood and adapted from http://rosettacode.org/wiki/Vigen%C3%A8re\_cipher#Python  from itertools import starmap, cycle  def encrypt(message, key):  # convert to uppercase.  # strip out non-alpha characters.  # create autokey  message = filter(lambda x: x.isalpha(), message.upper())  autokey = key + message  # single letter encrpytion.  def enc(c,k): return chr(((ord(k) - ord('A') + ord(c) - ord('A')) % 26) + ord('A'))  return "".join(starmap(enc, zip(message, autokey)))  def decrypt(message, key):  # single letter decryption.  # create autokey  def dec(c,k): return chr(((ord(c) - ord(k)) % 26) + ord('A'))  keylen = len(key)  msgtext = ""  for i in range(len(message)):  if i < keylen:  k = key[i]  else:  k = msgtext[i-keylen]  msgtext += dec(message[i],k)  return msgtext  if \_\_name\_\_ == "\_\_main\_\_":  text = "LET US CHANGE OUR TRADITIONAL ATTITUDE TO THE CONSTRUCTION OF PROGRAMS"  key = "ALGORITHMS"  encr = encrypt(text, key)  decr = decrypt(encr, key)  print text  print encr  print decr |
| 3.  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  Resolved using *https://www.iam.unibe.ch/~run/frequencies.php* for frequency analysis and *http://quipqiup.com/index.php.* |
| 4. COMPUTER SCIENCE IS NO MORE ABOUT COMPUTERS THAN ASTRONOMY IS ABOUT TELESCOPES.  (scratch work attached to the back of the homework) |
| 5. hw3/RSAKey.py  class RSAKey:  def \_\_init\_\_(self):  self.prime\_list = []  def next\_prime(self):  if not self.prime\_list:  self.scanner = 2  self.prime\_list.append(self.scanner)  return self.scanner  found = False  while not found:  self.scanner += 1  for i in self.prime\_list:  if self.scanner % i == 0:  break  else:  found = True  else:  self.prime\_list.append(self.scanner)  return self.scanner  def a\_factor(self, q):  self.prime\_list = []  mill = 0  found = False  while not found:  p = self.next\_prime()  if p/1000 > mill:  print 'tried up to', p  mill += 1  if (q % p) == 0:  found = True  else:  return p  def egcd(self, a, b):  if a == 0:  return (b, 0, 1)  else:  g, y, x = self.egcd(b % a, a)  return (g, x - (b // a) \* y, y)  def modinv(self, a, m):  gcd, x, y = self.egcd(a, m)  if gcd != 1:  return None # modular inverse does not exist  else:  return x % m  if \_\_name\_\_ == '\_\_main\_\_':  #t1 = tm.time()  RSAobject = RSAKey()  n = 729880581317  #n = 221  #y = x.a\_factor(n)  #print y, n/y  #t2 = tm.time()  #print "took ", int(t2 - t1), " seconds"  a = (822892\*886968)  e = 5  print a  print e  answer = RSAobject.modinv(e,a)  print "d = ", answer  The function next\_prime searches for the next prime number in a list by checking a numbers divisibility with all of the previous prime numbers. If the number is divisible by any previous number, the function end, but if not, then that prime number is added to the end of the list. The a\_factor function finds a prime factor ‘p’ of ‘q’ with use of the next\_prime search function. Simply by finding the first prime factor, we can find the corresponding factor that makes up ‘q.’ In this case, we were lucky because the corresponding factor for ‘p’ was also prime, so no additional work was required**. Therefore the ‘p’ and ‘q’ that make up N in the public key are 822893 and 886969**. Because the private key ‘d’ is the mod inverse of e relative to (p-1)(q-1), we can easily solve for the private key. This was easily solved with the extended Euclidean algorithm. (http://en.wikibooks.org/wiki/Algorithm\_Implementation/Mathematics/Extended\_Euclidean\_algorithm)  Private Key = **583903097165** |
| 6.  a) Digital signatures help verify that a message was not altered by a third party during communication, and also helps verify the identity of the sender.  b) 6b-RSA.py  # a^b mod p  def mod\_exp(a, b, p):  if (b == 1):  return a % p  x = mod\_exp(a, b>>1, p)  x = (x \* x) % p  if (b & 1 == 1):  x = (x \* a) % p  return x  def verify(n, e, s, m):  x = mod\_exp(s, e, n)  return x == m  if \_\_name\_\_ == "\_\_main\_\_":  #p = 11  #q = 5  n = 55  e = 3  d = 27  m = 12  s = mod\_exp(m, d, n)  print verify(n, e, s, m)  This function returns true.  c) 6c-RSA.py  # a^b mod p  def mod\_exp(a, b, p):  if (b == 1):  return a % p  x = mod\_exp(a, b>>1, p)  x = (x \* x) % p  if (b & 1 == 1):  x = (x \* a) % p  return x  def verify(n, e, s, m):  x = mod\_exp(s, e, n)  return x == m  if \_\_name\_\_ == "\_\_main\_\_":  #p = 11  #q = 13  n = 143  e = 7  d = 103  m = "anu"  s = [0, 0, 0]  for i in range(len(m)):  s[i] = mod\_exp(ord(m[i]), d, n) #the mapping is done by convering the character to integers with ord()  print s[i]  for i in s:  print chr(mod\_exp(i, e, n)) #this proves that (m1)^e = m mod n, and converts the integer back to a character for comparing convenience  #print verify(n, e, s, m)  d) 391 = 17 \* 23, so p = 17 and q = 23. (p-1)(q-1) = 16\*22 = 352. The mod inverse of e relative to 352 is 145. Therefore, she should raise her message by 145. This is true because 145\*17 = 2465 = 1 mod 352. |
| 7.  a) When Eve intercepts the encrypted message, she has Me mod N. If she asks Bob to sign it with his private key, Eve would get (Me)d mod N = M because of RSA. From there, it would be simple to solve for d.  b) Even if the messages were chosen randomly, even can pick a coprime to N, c, and ask Bob to sign Me \* ce mod N. This equals (Mc)ed mod N = Mc mod N. Using the Extended Euclidean, Eve could find c-1 mod N and ultimately find M. |
| 8.  a) This is an example of the Master theorem where a = 5, b = 2, and d = 1. Therefore the running time is O(nlogb(a)) = O(n2.33)  b) In this case, T(n) = 2T(n-1) + A. The running time is therefore O(2n).  c) This is another example of the Master theorem where a = 9, b = 3, and d = 2. Therefore, the running time is O(n2 log n).  When comparing the running times, it is evident that the best option is a) because the running time is the most efficient. |
| 9. Because each function call prints one line and then calls the same function on an input half its size, the number of times “still going” is printed is T(n) = 2T(n/2) + 1. By the master theorem, this means that T(n) = Θ(n). |
| 10.  a) If A has a majority element, that element must be the majority element in A1, A2, or both. To determine whether or not there is a majority element in A, recursively determine the majority elements of A1 and A2, and check if those majority elements are the majority element of A. The running time for this process would be T(n) = 2T(n/2) + O(n) = O(n log n).  b) The hint explains that we should create pairs of 2 with the elements in A to get n/2 pairs. If the two elements in the pair are different, then both are discarded, but if they are the same then we keep one of them. After this process, there is a max of n/2 elements left because at least half of them would be discarded (this would only happen in the case that all of the elements are the same). If there is a majority element among the remaining elements, then A has a majority element that appears at least n/4 times. Therefore, the majority element would have had to been paired with itself in at least n/4 of the pairs during the procedure, proving that A has at least n/2 copies of the majority element. The running time of this algorithm would be T(n) = T(n/2) + O(n) = O(n). |
| 11. |
| 12.  a) Given the constraints of the problem, we can model this as a directed graph where each node contains 3 numbers: (S10, S7, S4). Each of the three numbers would indicate how many pints of water are in the given container at that particular vertex. At any given time, the amount of water in each contain must be between 0 and n where n is the max amount of pints that the container can carry. An edge between two nodes can exist if the two nodes different in only 2 coordinates (because only two containers can be involved in a water exchange at a time), and if one of the containers either becomes empty or becomes full as a result of the exchange. The question that needs to be answered as a result of this is whether there is a path that exists between the node that reads (0, 7, 4) and (\_, \_, 2) or (\_, 2, \_).  b) The algorithm that should be used in order to solve this problem is a depth first search algorithm. This is because this algorithm visits every node in a graph, as it restarts its search in a new unvisited node when all of the nodes directly attached to the initial node are observed. However, an extra line of code would need to be added that ends the search and returns “YES” when one of the two appropriate answer nodes appears in the search, or returns “NO” if all options are exhausted and no path is found. |