Anu Varshneya

GitHub: https://github.com/anuvarsh

Homework 2

|  |  |
| --- | --- |
| 1.  (0.1) | a) f = Θ(g)  b) f = Ω(g)  c) f = O(g)  d) f = Θ(g)  e) f = Θ(g)  f) f = O(g)  g) f = O(g)  h) f = O(g)  i) f = O(g)  j) f = Ω(g)  k) f = O(g)  l) f = O(g)  m) f = O(g)  n) f = Θ(g)  o) f = O(g)  p) f = O(g)  q) f = Ω(g) |
| 2.  (0.4) | a) ( ) = X \* Y  ( ) = X \* ( ) = X2 \* Y  ( ) = Xn \* Y  Every entry of XY is the addition of the 2 products of the entries of the original matrices. Therefore, ever entry can be computed in 2 multiplications and 1 addition. So the entire matrix can be calculated with 8 multiplications and 4 additions.  b) For the sake of example, n = 2k. To calculate X2^k, you can calculate Y = X2^k-1, and then square Y to get Y2 = X2^k. If this is done recursively, it is similar to repeated squaring, where it would result in X2, X4, X8, … , X2^k = Xn. At each squaring, the exponent of X is doubled from the previous exponent of X, so it must take k = log(n) matrix multiplications to compute Xn. |
| 3.  (1.2) | bits = log2(N+1), decimal = log10(N+1)  =  log2(N+1) = log2(10) \* log10(N+1)  log2(N+1) = ~4 \* log10(N+1)  bits = ~4 \* decimal  For very large numbers, the ratio of the two lengths is approximately 4. |
| 4.  (1.4) | Assume:  n! = O(nn) and n! = Ω((n/2)(n/2))  So, ((n/2)(n/2) < n! < nn  (n/2)log(n/2) < log(n!) < nlog(n)  (1/2)nlog(n/2) < log(n!) < nlog(n)  (1/2)n(log(n) – log(2)) < log(n!) < nlog(n)  Because (1/2) and log(2) are constants, they can be disregarded, so:  nlog(n) < log(n!) < nlog(n)  Therefore,  log(n!) = Θ(nlog(n)) |
| 5.  (1.11) | 41536 – 94824 is divisible by 35. |
| 6.  (1.13) | 6123,456 – 530,000 is a multiple of 31. |
| 7.  (1.16) | Assume b = 15.  By repeated squaring:  a\*a = a2  a2 \* a2 = a4  a4 \* a4 = a8  a15 = a \* a2 \* a4 \* a8  Total = 6 multiplications.  “Other” method:  a\*a\*a = a3  a3 \* a3 = a6  a6 \* a6 = a12  a15 = a3 \* a12  Total = 5 multiplications. |
| 8.  (1.25) | 2125 mod 127 = 64 |
| 9.  (1.33) | # 9-lcm.py in hw2  def lcm(x, y):  if (x > y):  greater = x  else:  greater = y  while(True):  if((greater % x == 0) and (greater % y == 0)):  lcm = greater  break  greater += 1  return lcm |
| 10.  (1.35d) | We can’t immediately base a primarity test on this rule because as N gets larger, the complexity becomes O(nn), so it would take too long for an average computer to compute the answer. Furthermore, this theorem can guarantee that all prime numbers will be indicated as prime, but it may also include false positives. This means that a number that isn’t actually prime may be considered a prime according to the theorem. |
| 11.  (1.39) | # 11-modexp.py in hw2  # to run, use complex\_mod\_exp(a, b, c, p) where a^(b^c) mod p, and p is a prime number  def mod\_exp(a, b, p):  if (b == 1):  return a % p  x = modexp(a, b>>1, p)  x = (x \* x) % p  if (b & 1 == 1):  x = (x \* a) % p  return x  def exp(b, c):  if (c == 1):  return b  z = exp(b, c>>1)  z = z \* z  if (c & 1 == 1):  z = z \* b  return z  def complex\_mod\_exp(a, b, c, p):  pow1 = exp(b, c)  res = modexp(a, pow1, p)  return res |