Problem Set 5

Both theory and programming questions are due **Monday, October 31** at **11:59PM**. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions.

We will provide the solutions to the problem set 10 hours after the problem set is due. You will have to read the solutions, and write a brief **grading explanation** to help your grader understand your write-up. You will need to submit the grading explanation by **Thursday, November 3rd, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 5-1. [40 points] **The Knight's Shield**

The optimized circuit verifier that you developed on your Amdtel internship was a huge success and got you on a sure track to landing a sweet offer. You also got transferred to a research group that is working on the *Knight's Shield (KS)*[1], a high-stakes project to develop a massive multi-core chip aimed at the exploding secure cloud computing market.

The KS chip packs 16,384 cores in a die that's the same size as a regular CPU die. However, each core is very small, and can only do arithmetic operations using 8-bit or 16-bit unsigned integers (see Table 1). Encryption algorithms typically use 2,048-bit integers, so the KS chip will ship with software that supports arithmetic on large integers. Your job is to help the KS team assess the efficiency of their software.

Operation	R1 size	R2 size	Result size	Result
`ZERO`			8 / 16	0 (zero)
`ONE`			8 / 16	1 (one)
`LSB R1`	16		8	R1 % 256 (least significant byte)
`MSB R1`	16		8	R1 / 256 (most significant byte)
`WORD R1`	8		16	R1 (expanded to 16-bits)
`ADD R1, R2`	8 / 16	8 / 16	16	$\texttt{R1} + \texttt{R2}$
`SUB R1, R2`	8 / 16	8 / 16	16	$\texttt{R1} - \texttt{R2} \mod 65536$
`MUL R1, R2`	8	8	16	$\texttt{R1} \cdot \texttt{R2}$
`DIV R1, R2`	16	8	8	$\texttt{R1} \div \texttt{R2} \mod 256$
`MOD R1, R2`	16	8	8	$\texttt{R1} \,\%\, \texttt{R2}$
`AND R1, R2`	8 / 16	8 / 16	8 / 16	R1 & R2 (bitwise AND)
`OR R1, R2`	8 / 16	8 / 16	8 / 16	R1 ‖ R2 (bitwise OR)
`XOR R1, R2`	8 / 16	8 / 16	8 / 16	R1 ^ R2 (bitwise XOR)

Table 1: Arithmetic operations supported by the KS chip. All sizes are in bits.

[1]The code name is Amdtel confidential information. Please refrain from leaking to TechCrunch.

The KS library supports arbitrarily large base-256 numbers. The base was chosen such that each digit is a byte, and two digits make up a 16-bit number. Numbers are stored as a little-endian sequence of bytes (the first byte of a number is the least significant digit, for example 65534 = 0xFFFE would be stored as [0xFE, 0xFF]). For the rest of the problem, assume all the input numbers have N digits.

Consider the following algorithm for computing $A + B$, assuming both inputs have N digits.

```
ADD(A, B, N)
1  C = ZERO(N + 1) // ZERO(k) creates a k-digit number, with all digits set to 0s.
2  carry = 0
3  for i = 1 to N
4       digit = WORD(A[i]) + WORD(B[i]) + WORD(carry)
5       C[i] = LSB(digit)
6       carry = MSB(digit)
7  C[N + 1] = carry
8  return C
```

(a) [1 point] What is the running time of ADD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N)$, because of the **for** loop on line 3.

(b) [1 point] What is the size of ADD's output?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N)$, because it returns an $(N + 1)$-digit number.

(c) [1 point] ADD's output size suggests an easy lower bound for the subroutine. Does the running time of ADD match this lower bound?

1. Yes
2. No

Solution: Yes. The running time matches the output size, so the algorithm must be optimal.

Consider the following brute-force algorithm for computing $A \cdot B$, assuming both inputs have N digits.

```
MULTIPLY(A, B, N)
1  C = ZERO(2N)
2  for i = 1 to N
3       carry = 0
4       for j = 1 to N
5            digit = A[i] · B[j] + WORD(C[i + j − 1]) + WORD(carry)
6            C[i + j − 1] = LSB(digit)
7            carry = MSB(digit)
8       C[i + N] = carry
9  return C
```

(d) [1 point] What is the running time of MULTIPLY?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^2)$, because of the nested **for** loops on lines 2 and 4.

(e) [1 point] What is the size of MULTIPLY's output?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$

6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N)$, because it returns a $2N$-digit number.

(f) [1 point] MULTIPLY's output size suggests an easy lower bound for the subroutine. Does the running time of MULTIPLY match this lower bound?

1. Yes
2. No

Solution: No. In fact, we do not know a multiplication algorithm that runs in $\Theta(N)$ time, and we don't have a proof for a better lower bound than $\Theta(N)$ for multiplication.

Consider the following brute-force algorithm for computing $A \div B$ and $A \mod B$, assuming both inputs have N digits. The algorithm uses a procedure COPY(A, N) that creates a copy of an N-digit number A, using $\Theta(N)$ time.

```
DIVMOD(A, B, N)
 1  Q = ZERO(N) // quotient
 2  R = COPY(A, N) // remainder
 3  S_0 = COPY(B, N) // S_i = B · 2^i
 4  i = 0
 5  repeat
 6      i = i + 1
 7      S_i = ADD(S_i-1, S_i-1, N)
 8  until S_i[N + 1] > 0 or CMP(S_i, A, N) == GREATER
 9  for j = i − 1 downto 0
10      Q = ADD(Q, Q, N)
11      if CMP(R, S_j, N) != SMALLER
12          R = SUBTRACT(R, S_j, N)
13          Q[0] = Q[0]||1 // Faster version of Q = Q + 1
14  return (Q, R)
```

(g) [1 point] CMP(A, B, N) returns GREATER if $A > B$, EQUAL if $A = B$, and SMALLER if $A < B$, assuming both A and B are N-digit numbers. What is the running time for an optimal CMP implementation?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$

5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N)$ because, in the worst case, it has to look at every digit in both numbers. The following pseudo-code runs in $\Theta(N)$ time.

GREATEREQUAL(A, B, N)

```
1  for i = 1 to N
2      if A[i] ≠ B[i]
3          if A[i] > B[i]
4              return GREATER
5          else return SMALLER
6  return EQUAL
```

(h) [1 point] SUBTRACT(A, B, N) computes $A - B$, assuming A and B are N-digit numbers. What is the running time for an optimal SUBTRACT implementation?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N)$. The pseudo-code, run-time analysis, and optimality argument are very similar to those for ADD discussed above.

(i) [1 point] What is the running time of DIVMOD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^2)$. The **Repeat** ... **Until** loop starting on line 5 can increase i up to $8N$, which is achieved using the extreme values $B = 1$ and $A = 2^{8N} - 1$. So, both the **Repeat** ... **Until** loop and the **for** loop starting on line 9 run $\Theta(N)$ times. Each loop makes a constant number of calls to ADD, SUBTRACT and CMP, all of which have $\Theta(N)$ running time when given N-digit long inputs. Therefore, the total running time is $\Theta(N) \cdot \Theta(N) = \Theta(N^2)$

The KS library does not use the DIVMOD implementation above. Instead, it uses Newton's method to implement DIV(A, B, N) which computes the division quotient $A \div B$, assuming both inputs have N digits. DIV relies on the subroutines defined above. For example, it uses MULTIPLY to perform large-number multiplication and ADD for large-number addition. MOD(A, B, N) is implemented using the identity $A \mod B = A - (A \div B) \cdot B$.

(j) [2 points] How many times does DIV call MULTIPLY?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(\log N)$ times, because it converges quadratically, according to the Lecture 12 notes.

(k) [2 points] What is the running time of MOD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^2)$.

Although MULTIPLY is called $\Theta(\log N)$ times in DIV, the operand sizes are different each time, so the total running time for DIV is $\Theta(N^2)$. See the Lecture 12 notes for details.

MOD performs an extra multiplication and subtraction, so it takes $\Theta(N^2)$ running time in addition to the running time of DIV. The total running time of MOD is $\Theta(N^2 + N^2) = \Theta(N^2)$

Consider the following brute-force algorithm for computing $B^E \mod M$, assuming all the input numbers have N digits.

```
POWMOD(B, E, M, N)
 1  R = ONE(N) // result
 2  X = COPY(B, N) // multiplier
 3  for i = 1 to N
 4      mask = 1
 5      for bit = 1 to 8
 6          if E[i] & mask != 0
 7              R = MOD(MULTIPLY(R, X, N), M, 2N)
 8          X = MOD(MULTIPLY(X, X, N), M, 2N)
 9          mask = LSB(mask · 2)
10  return R
```

(l) [2 points] What is the running time for POWMOD?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^3)$. The **for** loop on line 3 runs N times, and the inner **for** loop on line 7 runs 8 times, so MULTIPLY and MOD on lines 7 and 8 are called $\Theta(N)$ times. The intermediate multiplication results are always reduced modulo N, so R and X will always have $\Theta(N)$ digits, which means that each MOD and MULTIPLY call takes $\Theta(N^2)$ time.

Assume the KS library swaps out the brute-force MULTIPLY with an implementation of Karatsuba's algorithm.

(m) [1 point] What will the running time for MULTIPLY be after the optimization?

1. $\Theta(1)$

2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^{\log_2 3})$, the running time of Karatsuba's algorithm.

(n) [2 points] What will the running time for MOD be after the optimization?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^{\log_2 3})$, for the same reason that MOD takes $\Theta(N^2)$ time when MULTIPLY is implemented using the brute-force algorithm.

(o) [2 points] What will the running time for POWMOD be after the optimization?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^{\log_2 6})$. As argued earlier, POWMOD calls MULTIPLY and MOD $\Theta(N)$ times with $\Theta(N)$-digit long arguments. So the total running time is $\Theta(N) \cdot \Theta(N^{\log_2 3}) = \Theta(N^{1+\log_2 3}) = \Theta(N^{\log_2(2\cdot 3)}) = \Theta(N^{\log_2 6})$.

(p) [20 points] Write pseudo-code for KTHROOT(A, K, N), which computes $\lfloor \sqrt[K]{A} \rfloor$ using binary search, assuming that A and K are both N-digit numbers. The running time for KTHROOT(A, K, N) should be $\Theta(N^{2+\log_2 3})$.

Solution: The pseudo-code below implements a binary search for the answer.

```
KTHROOT(A, K, N)
 1  T_0 = ONE(N) // T_i = 2^i
 2  i = 0
 3  repeat
 4      i = i + 1
 5      T_i = ADD(T_{i-1}, T_{i-1}, N)
 6  until T_i[N + 1] > 0 or POWEXCEEDS(T_i, K, A, N)
 7  R = ZERO(N)
 8  for j = i − 1 downto 0
 9      R' = ADD(R, T_j)
10      unless POWEXCEEDS(R', K, A, N)
11          R = R'
12  return R
```

The pseudo-code uses a helper subroutine POWEXCEEDS(B, E, A, N), which returns TRUE iff $B^E > A$, assuming all input numbers are N-digits long. This solution takes a non-standard approach to implementing binary search that is worth taking a look at.

The regular binary search implementation requires a $[low, high]$ range that contains the answer, and uses a guess $g = \lfloor \frac{low+high}{2} \rfloor$ to narrow the range down to a single number. In contrast, this implementation assumes that the answer is non-negative $low = 0$, and does not require a $high$ upper bound on the answer. The **repeat**-**until** loop on lines 3-8 computes the upper bound by generating powers of i (T_i is computed to hold 2^i inside the loop) and stopping when the answer is smaller than T_i, which means that we can use $high = T_i$ and start the typical binary search algorithm on the $[0, T_i]$ interval. If the answer has n bits, the **repeat**-**until** loop runs for n iterations, so having to compute the upper bound doesn't change the asymptotic running time of binary search.

Line 7 initializes R, which is going to be returned as the result of the binary search. R stands for "result", but is actually the low variable in regulary binary search. This is because we are asked to compute the floor of the k^{th} root of A. The **for** loop on lines 8-11 implements the binary search, by successively computing the bits of R, from the highest bit to the lowest. The variables map to the classical binary search implementation as follows: low is R, $high$ is $R + 2T_j - 1$, and the guess g is $R + T_j$. This holds before the loop because $R = 0$, and we know that the answer is smaller than T_i, so it has to be $\leq T_i - 1 = 2T_{i-1} - 1 = 2T_j - 1$. Convince yourself that the invariant holds throughout the loop, using the fact that $T_i = 2T_{i-1}$.

This variant of binary search is interesting because the successive values of the guess g only differ by one bit, and it can be implemented solely using bitwise operations (shifting, and bitwise OR), whereas the classical binary search implementation requires addition. Note that the division by two in classical binary search can be implemented using a right shift, so the only difference is the need for addition (the $+$ operator).

The DIVMOD pseudo-code given in this problemset follows the same binary search strategy for computing the division quotient. Take another look at the pseudo-code

and convince yourself that this is the case. The first result of Googling for *double division* should help visualize the divsion algorithm.

```
POWEXCEEDS(B, E, A, N)
 1  A2 = ZERO(2N) // A, zero-extended to 2N digits
 2  for i = 1 to N
 3      A2[i] = A[i]
 4  ones = 0 // number of set bits in E
 5  for i = 1 to N
 6      mask = 1
 7      for bit = 1 to 8
 8          if E[i] & mask != 0
 9              ones = ones + 1
10          mask = LSB(mask · 2)
11  R = ONE(N) // result
12  X = COPY(B, N) // multiplier
13  for i = 1 to N
14      mask = 1
15      for bit = 1 to 8
16          if E[i] & mask != 0
17              R = MULTIPLY(R, X, N)
18              if CMP(R, A2, 2N) == GREATER
19                  return TRUE
                // ones is the number of 1s that haven't yet contributed to R
20              ones = ones − 1
21              if ones == 0
22                  return FALSE // R = B^E ≤ A
23          X = MULTIPLY(X, X, N)
24          if CMP(X, A2, 2N) == GREATER
25              return TRUE
26          mask = LSB(mask · 2)
27  return FALSE
```

The POWEXCEEDS implementation is similar to that of POWMOD. Instead of calling MOD, lines 18 and 24 return FALSE if the intermediate result is greater than A, so a modular reduction would occur. If R ever gets greater than A, we know that the final result will also be greater than A, so it's safe to return FALSE. However, X is not always multiplied by R, so we must take some precaution. This implementation counts the 1 bits in the exponent, and returns as soon as R receives the result of the multiplication corresponding to the most significant exponent bit that is set to 1. This means that when we square X, we know the value will eventually be multiplied into R. Therefore, it is also safe to return FALSE when an intermediate X value exceeds A.

Because of all the precautions above, all the intermediate results in POWEXCEEDS are at most $2N$ digits long, so the subroutine has the same running time as POWMOD, $\Theta(N^{\log_2 6})$.

NTHROOT performs a binary search, which may use up to $8N$ trials. So the total running time of NTHROOT is $\Theta(N \cdot N^{\log_2 6}) = \Theta(N^{1+\log_2 6}) = \Theta(N^{2+\log_2 3})$

Problem 5-2. [18 points] **RSA Public-Key Encryption**

The RSA (Rivest-Shamir-Adelman) public-key cryptosystem is a cornerstone of Internet security. It provides the "S" (security) in the HTTPS sessions used for e-commerce and cloud services that handle private information, such as e-mail. RSA secures SSH sessions (used to connect to Athena, for example), and MIT certificates used to log into Stellar. You figure that the KS chip must perform RSA efficiently, since RSA plays such an important role in cloud security. This problem will acquaint you with the encryption and decryption algorithms in RSA.

RSA works as follows. Each user generates two large random primes p and q, and sets his public modulus $m = p \cdot q$. The user then chooses a small number[2] e that is co-prime with $(p-1)(q-1)$, and computes $d = e^{-1} \mod (p-1)(q-1)$. The user announces his public key (e, m) to the world, and keeps d private. In order to send an encrypted message to our user, another user would encode the message as a number smaller than n, and encrypt it as $c = E(n) = n^e \mod m$. Our user would decode the message using $D(c) = c^d \mod m$. Assume that keys can be generated reasonably fast and that $D(E(n)) = n$, for all but a negligible fraction of values of n.

(a) [1 point] What is the running time of an implementation of $D(n)$ that uses the KS library in Problem 1, with the optimized version of MULTIPLY (Karatsuba's algorithm), assuming that n, d and m are N-byte numbers?

1. $\Theta(1)$
2. $\Theta(\log N)$
3. $\Theta(N)$
4. $\Theta(N^2)$
5. $\Theta(N^2 \log N)$
6. $\Theta(N^{\log_2 3})$
7. $\Theta(N^{\log_2 6})$
8. $\Theta(N^3)$

Solution: $\Theta(N^{\log_2 6})$, because $D(n)$ can be computed using a single call to POWMOD.

You're thinking of using RSA to encrypt important sensitive images, such as last night's picture of you doing a Keg stand. Formally, a picture has $R \times C$ pixels (R rows, C columns), and each pixel is represented as 3 bytes that are RGB color space coordinates[3]. The RSA key is (e, m), where m is an N-byte number. An inefficient encryption method would process each row of pixel data as follows:

1.Break the $3C$ bytes of pixel data into groups of $N-1$ bytes

2.Pad the last group with 0 bytes up to $N-1$ bytes

[2] 65,537 is a popular choice nowadays

[3] see `http://en.wikipedia.org/wiki/RGB_color_space`

3.Encrypt each group of $N-1$ bytes to obtain an N-byte output

4.Concatenate the N-byte outputs

(b) [1 point] How many calls to the RSA encryption function $E(n)$ are necessary to encrypt an $R \times C$-pixel image?

1. $\Theta(1)$
2. $\Theta(RC)$
3. $\Theta(\frac{RC}{N})$
4. $\Theta(\frac{RN}{C})$
5. $\Theta(\frac{CN}{R})$

Solution: The exact number is $R\lceil\frac{C}{N-1}\rceil$, which is $\Theta(\frac{RC}{N})$.

(c) [1 point] What is the running time for decrypting an $R \times C$-pixel image that was encrypted using the method above, using the KS library in Problem 1, with the optimized version of MULTIPLY (Karatsuba's algorithm)?

1. $\Theta(N)$
2. $\Theta(N^2)$
3. $\Theta(N^2 \log N)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^{\log_2 6})$
6. $\Theta(RCN)$
7. $\Theta(RCN^2)$
8. $\Theta(RCN^2 \log N)$
9. $\Theta(RCN^{\log_2 3})$
10. $\Theta(RCN^{\log_2 6})$
11. $\Theta(RN)$
12. $\Theta(RN^2)$
13. $\Theta(RN^2 \log N)$
14. $\Theta(RN^{\log_2 3})$
15. $\Theta(RN^{\log_2 6})$

Solution: $\Theta(RCN^{\log_2 3})$

The running time of one decryption operation is $\Theta(N^{\log_2 6})$, and decrypting the entire image requires $\Theta(\frac{RC}{N})$ operations, so the total running time is $\Theta(\frac{RC \cdot N^{\log_2 6}}{N}) = \Theta(\frac{RC \cdot N^{1+\log_2 3}}{N}) = \Theta(RCN^{\log_2 3})$.

(d) [5 points] A fixed point under RSA is a number n such that $E(n) \equiv n \mod m$, so RSA does not encrypt the number at all. Which of the following numbers are fixed points under RSA? (True / False)

1. 0
2. 1
3. 2
4. 3
5. $m - 2$
6. $m - 1$

Solution: 0, 1, and $m - 1$ are fixed points.

$0^e = 0$ and $1^e = 1$ for any value of $e > 0$. $m - 1 \equiv -1 \mod m$, and we know that $(-1)^e = -1$ if e is odd. The other choices can be eliminated by choosing $e = 3$ and $m = 35 = 5 \cdot 7$. $2^3 \equiv 8 \mod 35$, $3^3 \equiv 27 \mod 35$, and $(35-2)^3 \equiv (-2)^3 \equiv -8 \equiv 27 \mod 35$.

(e) [5 points] What other weaknesses does the RSA algorithm have? (True / False)

1. $E(-n) \equiv -E(n) \mod m$
2. $E(n_1) + E(n_2) \equiv E(n_1 + n_2) \mod m$
3. $E(n_1) - E(n_2) \equiv E(n_1 - n_2) \mod m$
4. $E(n_1) \cdot E(n_2) \equiv E(n_1 \cdot n_2) \mod m$
5. $E(n_1)^{n_2} \equiv E(n_1^{n_2}) \mod m$

Solution: $E(-n) \equiv -E(n) \mod m$, $E(n_1) \cdot E(n_2) \equiv E(n_1 \cdot n_2) \mod m$, and $E(n_1)^{n_2} \equiv E(n_1^{n_2}) \mod m$

The positive answers can be proven by using algebra on the definition of $E(n)$. $E(n_1 \cdot n_2) \equiv (n_1 \cdot n_2)^e \equiv (n_1^e) \cdot (n_2^e) \equiv E(n_1) \cdot E(n_2) \mod m$. Also, $E(-n) \equiv (-n)^e \equiv -(n^e)$, because e is odd. Last, $E(n_1^{n_2}) \equiv (n_1^{n_2})^e \equiv n_1^{(n_2 \cdot e)} \equiv (n_1^e)^(n_2) \equiv E(n_1)^{n_2}$.

The negative answers can be proven by counter-example. $E(1) + E(1) = 2 \neq 8 = 2^3 = E(1+1)$ for $n_1 = n_2 = 1$, $e = 3$ and $m = 35$. Similarly, $E(2) - E(1) = 2^3 - 1 = 7 \neq 1 = E(2-1)$.

(f) [5 points] Amdtel plans to use RSA encryption to secretly tell Gopple when its latest smartphone CPU is ready to ship. Amdtel will send one message every day to Gopple, using Gopple's public key (e_G, m_G). The message will be NO (the number 20079 when using ASCII), until the day the CPU is ready, then the message will change to YES (the number 5858675 when using ASCII). You pointed out to your manager that this security scheme is broken, because an attacker could look at the encrypted messages, and know that the CPU is ready when the daily encrypted message changes. This is a problem of deterministic encryption. If $E(20079)$ always takes the same value, an attacker can distinguish $E(20079)$ from $E(5858675)$. How can the problem of deterministic encryption be fixed? (True / False)

1. Append the same long number (the equivalent of a string such as 'XXXPADDINGXXX') to each message, so the messages are bigger.

2. Append a random number to each message. All random numbers will have the same size, so the receiver can recognize and discard them.
3. Use a different encryption key to encrypt each message, and use Gopple's public exponent and modulus to encrypt the decryption key for each message.
4. Use an uncommon encoding, such as UTF-7, so that the attacker will not know the contents of the original messages.
5. Share a "secret" key with Gopple, so that the attacker can't use the knowledge on Gopple's public exponent and modulus.

Solution: Appending a random number and using per-message encryption keys both work, because they make the encryption output non-deterministic. All the other proposals don't change the fact that a message will always look the same after encryption, which is what the attacker uses to detect the change from NO to YES.

Problem 5-3. [42 points] **Image Decryption**

Your manager wants to show off the power of the Knight's Shield chip by decrypting a live video stream directly using the RSA public-key crypto-system. RSA is quite resource-intensive, so most systems only use it to encrypt the key of a faster algorithm. Decrypting live video would be an impressive technical feat!

Unfortunately, the performance of the KS chip on RSA decryption doesn't come even close to what's needed for streaming video. The hardware engineers said the chip definitely has enough computing power, and blamed the problem on the RSA implementation. Your new manager has heard about your algorithmic chops, and has high hopes that you'll get the project back on track. The software engineers suggested that you benchmark the software using images because, after all, video is just a sequence of frames.

The code is in the `rsa` directory in the zip file for this problem set.

(a) [2 points] Run the code under the python profiler with the command below, and identify the method inside `bignum.py` that is most suitable for optimization. Look at the methods that take up the most CPU time, and choose the first method whose running time isn't proportional to the size of its output.

```
python -m cProfile -s time rsa.py < tests/1verdict_32.in
```

Warning: the command above can take 1-10 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly. If you have installed PyPy successfully, you should replace `python` with `pypy` in the command above for a 2-10x speed improvement.

What is the name of the method with the highest CPU usage?

Solution: `fast_mul`

The first line in the profiler output points to `__add__`, but the addition algorithm is optimal ($O(N)$ running time, $O(N)$ output size). The next line points to `fast_mul`.

PyPy's output may have `__init__` and `normalize` above `fast_mul`. These methods are also implemented using optimal algorithms, so they are not good answers for the question.

(b) [1 point] How many times is the method called?

Solution: The second line in the profiler output indicates that `fast_mul` is called 93496 times.

(c) [1 point] The troublesome method is implementing a familiar arithmetic operation. What is the tightest asymptotic bound for the worst-case running time of the method that contains the bottleneck? Express your answer in terms of N, the number of digits in the input numbers.

1. $\Theta(N)$.
2. $\Theta(N \log n)$

3. $\Theta(N \log^2 n)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^2)$
6. $\Theta(N^{\log_2 7})$
7. $\Theta(N^3)$

Solution: `__mul__` implements multiplication using Karatsuba's algorithm. The running time for this algorithm is $\Theta(N^{\log_2 3})$.

(d) [1 point] What is the tightest asymptotic bound for the worst-case running time of division? Express your answer in terms of N, the number of digits in the input numbers.

1. $\Theta(N)$.
2. $\Theta(N \log n)$
3. $\Theta(N \log^2 n)$
4. $\Theta(N^{\log_2 3})$
5. $\Theta(N^2)$
6. $\Theta(N^{\log_2 7})$
7. $\Theta(N^3)$

Solution: `__div__` uses `__divmod__`, which implements division using Newton's algorithm. The asymptotic running time is the same as the running time of the underlying multiplication algorithm which is $\Theta(N^{\log_2 3})$ in this case.

We have implemented a visualizer for your image decryption output, to help you debug your code. The visualizer will also come in handy for answering the question below. To use the visualizer, first produce a trace.

```
TRACE=jsonp python rsa.py < tests/1verdict_32.in > trace.jsonp
```

On Windows, use the following command instead.

```
rsa_jsonp.bat < tests/1verdict_32.in > trace.jsonp
```

Then use Google Chrome to open `visualizer/bin/visualizer.html`

(e) [6 points] The test cases that we supply highlight the problems of RSA that we discussed above. Which of the following is true? (True / False)

1. Test `1verdict_32` shows that RSA has fixed points.
2. Test `1verdict_32` shows that RSA is deterministic.
3. Test `2logo_32` shows that RSA has fixed points.
4. Test `2logo_32` shows that RSA is deterministic.
5. Test `5future_1024` shows that RSA has fixed points.
6. Test `5future_1024` shows that RSA is deterministic.

Solution: Test `1verdict_32` shows both fixed points (the black eyes and mouth remain black in the encrypted image) and determinism (the rest of the face looks the same, so we can guess it's the same color). In `4verdict_512`, the bigger key size hides the fixed points.

Test `2logo_32` shows fixed points. Although the colors are off, the encrypted image clearly represents the MIT logo.

Test `5future_1024` does not shows fixed points or deterministic encryption, because of the noise in the source image. The encrypted image looks like white noise.

(f) [1 point] Read the code in `rsa.py`. Given a decrypted image of $R \times C$ pixels (R rows, C columns), where all the pixels are white (all the image data bytes are 255), how many times will `powmod` be called during the decryption operation in `decrypt_image`?

1. $\Theta(1)$
2. $\Theta(RC)$
3. $\Theta(\frac{RC}{N})$
4. $\Theta(\frac{RN}{C})$
5. $\Theta(\frac{CN}{R})$

Solution: $\Theta(1)$. `RsaKey` uses a dictionary to cache decryption results, so `powmod` is called at most twice: once for a "chunk" of pixels inside a row, and once for a "chunk" of pixels at the end of the row, which would be padded with 0s.

(g) [30 points] The multiplication and division operations in `big_num.py` are implemented using asymptotically efficient algorithms that we have discussed in class. However, the sizes of the numbers involved in RSA for typical key sizes aren't suitable for complex algorithms with high constant factors. Add new methods to `BigNum` implementing multiplication and division using straight-forward algorithms with low constant factors, and modify the main multiplication and division methods to use the simple algorithms if at least one of the inputs has 64 digits (bytes) or less. Please note that you are not allowed to import any additional Python libraries and our test will check this.

The KS software testing team has put together a few tests to help you check your code's correctness and speed. `big_num_test.py` contains unit tests with small inputs for all `BigNum` public methods. `rsa_test.py` runs the image decryption code on the test cases in the `tests/` directory.

You can use the following command to run all the image decryption tests.

```
python rsa_test.py
```

To work on a single test case, run the simulator on the test case with the following command.

```
python rsa.py < tests/1verdict_32.in > out
```

Then compare your output with the correct output for the test case.

```
diff out tests/1verdict_32.gold
```

For Windows, use `fc` to compare files.

```
fc out tests/1verdict_32.gold
```

While debugging your code, you should open a new Terminal window (Command Prompt in Windows), and set the `KS_DEBUG` environment variable (`export KS_DEBUG=true`; on Windows, use `set KS_DEBUG=true`) to use a slower version of our code that has more consistency checks.

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 90 seconds on any reasonably recent computer using PyPy, or less than 600 seconds when using CPython), upload your modified `big_num.py` to the course submission site. Our automated grading code will use our versions of `test_rsa.py`, `rsa.py` and `ks_primitives.py` / `ks_primitives_unchecked.py`, so please do not modify these files.

Solution: The solution archive on the course Web site contains the staff's solution and secret test cases.