PS5

June 26, 2025

Problem 5-1

(a)

The loop has N iterations and each command in the loop is $\Theta(1)$ therefore ADD(A, B, N) is $\Theta(N)$.

(b)

The size of the ADD is $N + 1$ therefore $\Theta(N)$.

(c)

Yes, the output size suggest a lower bound to the subroutine, which implies that the subroutine is optimal.

(d)

MULTIPLY has a nested loop and each command takes $\Theta(1)$, therefore the running time is $\Theta(N)$

(e)

The size of MULTIPLY's output is $2N$, therefore $O(N)$.

(f)

No, MULTIPLY's output size doesn't suggest any easy lower bound for the subroutine.

(g)

CMP's optimal (worst case) running time is $\Theta(N)$ since comparison of N digists requires N $\Theta(1)$ comparisons.

(h)

Since we can encode negative numbers in terms of positive numbers, we have SUBSTRACT $(A, B) =$ ADD $(A, -B)$. As a result, SUBSTRACT has the same optimal running time of ADD, $O(N)$.

(i)

The running time is $\Theta\left(N^2\right)$ since in the worst case $A = 2^{8N} - 1$ and $B = 1$ therefore, in the worst case $i = 1, ..., \log(A) = i = 1, ..., O(8N)$. The ADD function in the loops is $\Theta(N)$ giving a quadratic running time.

(j)

Newton's method has a converges quadratically (doubling the number of digits of precision in each iteration). As a result, DIV is $\Theta(\log N)$.

(k)

Since the number of digits change in DIV, the total running time is $O\left(N^2\right)$ even though MULTIPLY is called only $\Theta(\log N)$ times. Moreover, an additional multipication and division is performed. the multipication running time is also $\Theta\left(N^2\right)$, therefore MOD is $\Theta\left(N^2\right)$.

(l)

POWMOD has a running time of $\Theta\left(N^3\right)$, since the for loop runs N times and MOD is $\Theta\left(N^2\right)$

(m)

Utilizing the Karatuba algorithm, MULTIPLY runs in $\Theta\left(N^{\log_2 3}\right)$.

(n)

MOD includes a call to DIV of two N digit numbers and multipication of two $\Theta(N)$ digit numbers. Applying the Karatuba algorithm for multipication DIV has the same running time as multipication $\Theta\left(N^{\log_2 3}\right)$. Therefore, in total, MOD is $\Theta\left(N^{\log_2 3} + N^{\log_2 3}\right) = \Theta\left(N^{\log_2 3}\right)$.

(o)

POWMOD iterates N times on MOD, therefore its running time is $\Theta\left(N \cdot N^{\log_2 3}\right) = \Theta\left(N^{\log_2 3+1}\right) = \Theta\left(N^{\log_2 3+\log_2 2}\right) = \Theta\left(N^{\log_2 6}\right)$.

(p)

KTHROOT(A, K, N) computes the $\lfloor A^{1/k} \rfloor$ using binary search, assuming that A and K are both N- digit numbers.

We can use POWMOD to evuate $x^k \mod (M)$ where $M = 2^N - 1$ is the length of A in each recursion step of the binary search.

Since A is a N-digit number it is at most $M = 2^N - 1$, as a result there we require $\log(M) = \Theta\left(\log\left(2^N\right)\right) = \Theta(N)$ recursion steps. Overall the running time is $O\left(N \cdot N^{\log_2 6}\right) = O\left(N^{\log_2 3+2}\right)$

Pseudo code (without proper indentations):

```
def KTHROOT($A, K, N$):
M = $2^{N-1}$
low = 0, high = $M$
for $i = 1$ until $M + 1$:
mid = low + (high - low)//2
midPowK = POWMOD(mid, $K, M, N$)
if midPowK == $A$:
return mid
if midPowK < $A$:
high = mid
else:
low = mid
```

Problem 5-2

(a)

The decription function $D(n)$ involves evaluating $c^d \mod (m)$. This can be achieved with a single call of POWMOD(c, d, m, N) which has running time of $\Theta\left(N^{\log_2 6}\right)$.

(b)

A picture of $R \cdot C$ pixels has $3RC$ bytes (that are RGB coordinates). We break the bytes into groups of $N - 1$ groups of bytes. There are $\left\lceil \frac{3RC}{N-1} \right\rceil = \Theta\left(\frac{3RC}{N}\right)$ such groups.

(c)

We then encrypt each group. Each encryption involves evaluating $n^e \mod m$, where n is an N digit number. This can be achieved with POWMOD which has a running time of $\Theta\left(N^{\log_2 6}\right)$. Therefore the total running time is $\Theta\left(\frac{3RC}{N} N^{\log_2 6}\right) = \Theta\left(R \cdot C \cdot N^{\log_2 3}\right)$

(d)

We have $0^e \mod m = 0 \mod m$ and $1^e \mod m = 1 \mod , m$. In addition, $m-1 \mod m = m \mod m - 1 \mod m = -1 \mod m$.

Since $a \cdot b \mod m = x \rightarrow a \cdot b = abm + x$

in addition $a = qm + l$, $b = q'm + l'$ therefore $ab = qq'm^2 + qml' + q'ml + ll'$.

$$\rightarrow ab \mod m = x \mod m = ll' \mod m = (a \mod m)(b \mod m) \mod (m)$$

This implies that $a^e \mod m = (a \mod m)^e \mod m$.

Therefore, for odd e, we have $(m-1)^e \mod m = (-1)^e \mod m = -1 \mod m$.

These three are fixed points under RSA

(e)

Using modular arithmetic rules (addition, substraction, multipication and power)

1.

$$\begin{aligned} E(-n) = (-n)^e \mod m \\ = (-n \mod m)^e \mod m \\ = (-1 \mod m)^e (n \mod m)^e \mod m \\ = -1 (n \mod m)^e \mod m = -E(n) \mod m \end{aligned}$$

if e is odd!

2.

$$\begin{aligned} E(n_1) + E(n_2) = n_1^e \mod m + n_2^e \mod m \\ \neq (n_1 + n_2)^e \mod m = E(n_1 + n_2) \mod m \end{aligned}$$

3.

$$\begin{aligned} E(n_1) - E(n_2) = n_1^e \mod m - n_2^e \mod m \\ \neq (n_1 - n_2)^e \mod m = E(n_1 - n_2) \mod m \end{aligned}$$

4.

$$\begin{aligned} E(n_1) \cdot E(n_1) = (n_1^e \mod m)(n_2^e \mod m) \\ = (n_1^e n_2^e \mod m) \mod m \\ = (n_1 n_2)^e \mod m \mod m \\ E(n_1 \cdot n_2) \mod m \end{aligned}$$

5.

Using the result of 4 n_2 times we get

$$E(n_1) \cdots E(n_1) = E(n_1 \cdots n_1) \mod m = E(n_1^{n_2}) \mod m$$

(f)

The following solution to the deterministic encryption have the following outcome:

1. Appending the same long number to each message will still generate the same number when using ASCII. So this will not solve the problem.
2. Appending a random number to each message, will completely modify the number obtained by using ASCII encoding. Therefore, this solution would work.
3. If agreed in advance what are the keys used at each date, the encryption scheme would. As it would make the encryption scheme done seem none deterministic.
4. Using an uncommon encoding, wouldn't fix the problem, since unless the encoding would vary each time the same number each time one sends the same message.
5. Sharring a "secret" key with Goople would still generate the same number each time. One would need to randomize the message, such as in proposition 2 in order to obtain successfull encryption.

Problem 5-3

(a)

The method with the highest CPU usage whose running time not proportionate to its output is "fast_mul". The only method a longer running time is "__add__", which is already has an optimal running time, $O(N)$.

(b)

It is called 93496 times.

(c)

The algorithm implemented in "fast_mul" is the Karatsuba algorithm which has a running time of $O\left(N^{\log_2 3}\right)$.

(d)

"__dif__" uses Newton's method and the Karatsuba algorithm to implenent division. Resulting in the same running time as fast_mul: $O\left(N^{\log_2 3}\right)$.

(e)

1 and 2. 1verdict_32 shows that RSA has fixed points (the mouth remains black) and is deterministic (the rest of the face is same color)

2 and 3. 4verdict_512 doesn't show fixed points for the bigger key size, and still shows the RSA is deterministic.

4 and 5. 5futur_1024 doesn't show that the RSA has fixed points and is deterministic due to the noise in the original image.

(f)

"decrypt" calls "raw_crypt" which calls on "powmod" only once for such an image. Since "decrypt" utilizes a dictionary "chunk_cache" to store the previous in_chunks which are partial strings of the input hex_string. Since all the in_chunks are the same we only call powmod once and than look up the decryption in the dictionary chunk_cache.

(g)

The solution appears in the methods "slow_mul" and "slow_divmod" in Bin_num.py. Which can be tested in rsa_test.py.