Power-MOD problem

Input: A positive integer \( n \geq 2 \), an integer \( a \in \{0, 1, \ldots, n-1\} \).
Output: \( a^{n-1} \mod n \).

Example: \( n = 7, \ a = 4 \). \( 4^6 \mod 7 = 4096 \mod 7 = 1. \)

```
Slow-PowerMOD \( (a, n) \):
   answer \leftarrow 1
   for i \leftarrow 1 \ to \ n-1 \ do
      answer \leftarrow answer \ast a
   return \ answer \mod n
```

Since the for-loop executes \( n-1 \) times, even without investigating the body of the for-loop, we know that the running time is \( \Omega(n) \) (i.e., asymptotically bounded below by a linear function in \( n \)).

We now show that this is an extremely inefficient algorithm. What is the input size?

\[
\begin{align*}
n & \rightarrow \Theta(\log n) \text{ bits} \\
a & \rightarrow \Theta(\log n) \text{ bits since } a \in \{0, 2, \ldots, n-1\}. \\
\text{Total} & \rightarrow \Theta(\log n) \text{ bits}
\end{align*}
\]

(basically between \( \lceil \log_2 n \rceil \) and \( 2 \lceil \log_2 n \rceil \))

The running time of \( \Omega(n) = \Omega(2^{\log n}) \). If we let \( m = \text{input size} \), then \( m \leq 2 \log n \Rightarrow \log n \geq \frac{m}{2} \). Hence, running time \( = \Omega(2^{\frac{m}{2}}) \).

Note: This is exponential in the input size.
In CS there is a general consensus that

efficient \equiv \text{polynomial time in input size}

inefficient \equiv \text{exponential time in input size}

So \text{SLOW-POWERMOD} is inefficient.

\underline{\text{Example}}: Say \text{n} has 300 digits
\quad \text{a} has 200 digits

\Rightarrow \text{Input is described by 500 characters - very small relative to the gigabytes of input that is common now.}

But even for this very small input, number of multiplications is at least \(10^{290}\). At one multiplication per \(10^{-9}\) nanosecond (\(10^{-9}\) seconds), this still takes \(10^{290}\) seconds, which is much much longer than the age of the universe.  

\underline{\text{Divide-and-Conquer}}

\underline{\text{Goal: To design an algorithm that runs in polynomial time in m (i.e., O(m) or O(m^2) or O(m^3), etc.). In other words, we want an algorithm that runs in O(log n) or O(log^2 n) or O(log^3 n) time, etc., since m = O(log n).}
So we want an algorithm that runs in poly-logarithmic time in n.

**IDEA**
- If \( n \) is even, \( a^n = (a^{n/2}) \cdot (a^{n/2}) \).
  
  So we can compute \( a^{n/2} \), save it in a variable temp & return temp * temp.

- If \( n \) is odd, \( a^n = (a^{(n-1)/2}) \cdot (a^{(n-1)/2}) \cdot a \)
  
  So we can compute \( a^{(n-1)/2} \), save it in a variable temp & return temp * temp * a.

**EXAMPLE:**

\[
\begin{align*}
    a &\rightarrow a^2 \\
    a^2 &\rightarrow a^5 \\
    a^5 &\rightarrow a^{10} \\
    a^{10} &\rightarrow a^{20} \\
    a^{20} &\rightarrow a^{50} \\
\end{align*}
\]

\[\{5 \text{ multiplications, instead of } 19 \}
\]
FASTER-POWER \((a, n)\)

if \(n = 0\)
    return 1

if \(n = 1\)
    return a

if \(n \) is even
    temp \(\leftarrow\) FASTER-POWER \((a, n/2)\)
    return temp * temp

if \(n \) is odd
    temp \(\leftarrow\) FASTER-POWER \((a, (n-1)/2)\)
    return temp * temp * a

FASTER-POWERMOD \((a, n)\)
return FASTER-POWER \((a, n-1)\) mod n.

Analysis

Let \(M(n)\) denote the number of multiplications performed by FASTER-POWER. Then,

\[
M(n) \leq M\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2 \quad \text{for } n \geq 2
\]

\[
M(n) = 0 \quad \text{for } n \in \{0, 1\}.
\]

It does not change the asymptotic behavior of \(M(n)\) if we drop the floor in the right hand side of the recurrence.
So we work with the recurrence:

\[ M(n) \leq M\left(\frac{n}{2}\right) + 2 \quad \text{for } n \geq 2 \]
\[ M(n) = 0 \quad \text{for } n \in \{0, 1\}. \]

Unrolling this recurrence gives us:

\[ M(n) \leq M\left(\frac{n}{2}\right) + 2 \]
\[ \leq M\left(\frac{n}{4}\right) + 2 + 2 \]
\[ \leq M\left(\frac{n}{8}\right) + 2 + 2 + 2 \]
\[ \cdots \]
\[ \leq M\left(\frac{n}{2^j}\right) + j \cdot 2 \]

Setting \( j = \log_2 n \), we get:

\[ M(n) \leq M(1) + \log n \cdot 2 = 2 \cdot \log n \]

\[ \therefore M(n) = \Theta(\log n) \]

It is similarly easy to show that \( M(n) = \Omega(\log n) \), implying that \( M(n) = \Theta(\log n) \).

Now let \( T(n) \) be the running time of \text{FASTER-POWER}.

Clearly, \( T(n) = \Omega(M(n)) \), but is \( T(n) = \Theta(M(n)) \)? Unfortunately, not – because each multiplication can be quite costly.

\textbf{Cost of a Multiplication Operation}

It is not hard to see that the \textbf{"Elementary School" Multiplication algorithm} can multiply
A number that is $x$ bits long and a number that is $y$ bits long in time $\Theta(x \cdot y)$.

Also note that the result obtained by multiplying an $x$-bit number and a $y$-bit number can have $(x+y)$ bits (at most).

So if we assume that $a$ is $\log n$ bits long, then

\[
a^2 \rightarrow 2\log n \text{ bits long}
\]
\[
a^3 \rightarrow 3\log n \text{ bits long}
\]

etc. Thus, in the code for `FASTER-POWER`,

\[
\text{temp (which contains } a^{\log_2 n} \text{) is}
\]

\[
\frac{n \cdot \log n}{2}
\]

bits long. Therefore, temp * temp takes $\Theta(n^2 \log^2 n)$ time. Recall that we felt earlier that $\Omega(n)$ is too inefficient. Here we see that a single multiplication could take even more time. So we need to improve the speed of multiplications & for this we need to control the size of intermediate results of our computation. To do this we use the following

**FACT:**

\[
(a \cdot b \cdot c) \mod n = ((a \cdot b) \mod n) \cdot c \mod n.
\]
This means that we don’t have to wait to compute $a^{n-1}$ before taking the mod. We can take the mod after each multiplication, thus keep all intermediate results “small.”

\[
\text{FAST-POWERMOD}(a, n) \rightarrow \text{if } n - 1 = 0 \rightarrow \text{return } 1
\]

\[
\text{if } n - 1 = 1 \rightarrow \text{return } a \mod n
\]

\[
\text{if } n - 1 \text{ is even}
\]

\[
\text{temp } \leftarrow \text{FAST-POWERMOD}(a, (n-1)/2)
\]

\[
\text{return } ((\text{temp } \times \text{temp}) \mod n)
\]

\[
\text{if } n - 1 \text{ is odd}
\]

\[
\text{temp } \leftarrow \text{FAST-POWERMOD}(a, (n-2)/2)
\]

\[
\text{return } ((\text{temp } \times \text{temp}) \mod n) \times a \mod n
\]

In the above function every multiplication (and every \mod) operation takes two log₂n-bit numbers as arguments. Hence, every multiplication (and \mod) takes $O(\log^2 n)$ time.

The overall running time of the function is

\[
\text{(number of multiplications) } \times O(\log^2 n) + \text{(number of mods) } \times O(\log^2 n)
\]

Note that the \# of mod operations = \# of multiplication operations. Therefore, the running
time is \( 2 \cdot M(n) \cdot O(\log^2 n) = O(\log^3 n) \).

Since the input size is \( \Theta(\log n) \), this algorithm runs in polynomial time (\( O(m^3) \) time, where \( m = \) input size) in the input size.