POWER-MOD problem

INPUT: A positive integer $n \geq 2$, an integer $a \in \{0, 1, \ldots, n-1\}$.

OUTPUT: $a^{n-1} \bmod n$.

EXAMPLE: $n = 7$, $a = 4$. $4^6 \bmod 7 = 4096 \bmod 7 = 1$. □

```
SLOW-POWERMOD (a, n):
    answer ← 1
    for i ← 1 to n-1 do
        answer ← answer * 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?

$n \longrightarrow \Theta(\log n)$ bits

$a \longrightarrow O(\log n)$ bits since $a \in \{1, 2, \ldots, n-1\}$.

Total $\longrightarrow \Theta(\log n)$ bits

(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 = input size, then $m \leq 2 \log n \Rightarrow \log n \geq m/2$.

Hence, running time $= \boxed{\Omega(2^{m/2})}$.

Note: This is exponential in the input size.

In CS there is a general consensus that

efficient ≡ polynomial time in input size

inefficient ≡ exponential time in input size

So SLOW-POWERMOD is inefficient.

EXAMPLE: Say n has 300 digits
a has 200 digits

$\Rightarrow$ 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^{299}. At one multiplication per nanosecond (10^{-9} seconds), this still takes 10^{290} seconds, which is much much longer than the age of the universe. □

DIVIDE-AND-CONQUER

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 $m = \Theta(\log n)$.

So we want an algorithm that runs in <u>poly-logarithmic</u> time in n.

<u>IDEA</u>

- 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:

$$\left.\begin{array}{c} a^{20} \\ \uparrow * \\ a^{10} \\ \uparrow * \\ a^{5} \\ \uparrow *,* \\ a^{2} \\ \uparrow * \\ a^{1} \end{array}\right\}$$

5 multiplications instead of 19 multiplications we used in SLOW-POWERMOD

```
FASTER-POWER(a, n)
    if n = 0
        return 1
    if n = 1
        return a
    if n is even
        temp ← FASTER-POWER(a, n/2)
        return temp * temp
    if n is odd
        temp ← 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,

$$M(n) \leq M(\lfloor n/2 \rfloor) + 2 \quad \text{for } n \geq 2$$
$$M(n) = 0 \quad \text{for } n \in \{0, 1\}.$$

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:

$$M(n) \le M(n/2) + 2 \quad \text{for } n \geq 2$$
$$M(n) = 0 \quad \text{for } n \in \{0,1\}.$$

Unrolling this recurrence gives us:

$$\begin{aligned} M(n) &\le M(n/2) + 2 \\ &\le M(n/4) + 2 + 2 \\ &\le M(n/8) + 2 + 2 + 2 \\ &\cdots \\ &\le M(n/2^j) + j \cdot 2 \end{aligned}$$

Setting $j = \log_2 n$, we get

$$M(n) \le M(1) + \log n \cdot 2 = 2 \cdot \log n$$
$$\therefore \quad M(n) = O(\log n).$$

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

Now let $T(n)$ = running time of FASTER-POWER. Clearly, $T(n) = \Omega(M(n))$, but is $T(n) = O(M(n))$? Unfortunately, not - because each multiplication can be quite costly.

Cost of a Multiplication Operation

It is not hard to see that the "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 ~~[illegible]~~ can have $(x+y)$ bits (at most).

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

$a^2 \longrightarrow 2\log n$ bits long

$a^3 \longrightarrow 3\log n$ bits long

etc. Thus, in the code for FASTER-POWER, temp (which contains $a^{n/2}$) is

$$\frac{n}{2} \cdot \log n$$

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:

$(a \cdot b \cdot c) \bmod n = (((a \cdot b) \bmod n) \cdot c) \bmod n$.

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."

```
FAST-POWERMOD (a, n)                    if n-1 = 0
                                           return 1
    if n-1 = 1
        return  a mod n
    if n-1 is even
        temp ← FAST-POWERMOD (a, (n-1)/2)
        return (temp * temp) mod n
    if n-1 is odd
        temp ← FAST-POWERMOD (a, (n-2)/2)
        return (((temp * temp) mod n) * a) mod n
```

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

The overall running time of the function is

(number of multiplications) $\times$ $O(\log^2 n)$ +

(number of mods) $\times$ $O(\log^2 n)$

Note that the # of mod operations = # of multiplication operations. Therefore, the running time is

$$2 \cdot M(n) \cdot \Theta(\log^2 n) = O(\log^3 n).$$

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.