# Lecture 04: Repeated Squaring

Repeated Squaring

★課▶ ★注▶ ★注▶

- Let  $(G, \circ)$  be a group with generator g
- We define  $g^0 = e$ , where  $r \in G$  is the identity element of G

• We define 
$$g^i = \overbrace{g \circ g \circ \cdots \circ g}^{i}$$

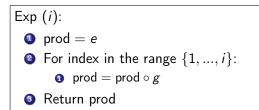
 $\bullet$  For example, the group  $(\mathbb{Z}_7^*,\times)$  is generated by 3 but not 2

→ 御 → → 注 → → 注 →

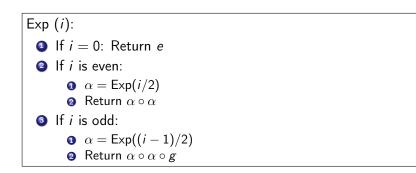
# Motivation of Efficient Algorithm to Compute Exponentiation

- Suppose *p* is a prime number that is represented using 1000-bits
- Note that the number p is in the range  $[2^{999}, 2^{1000})$ . We shall summarize this by stating that p is roughly (in the order of)  $2^{1000}$ .
- Suppose we are interested to work on the field  $(\mathbb{Z}_p^*, \times)$  with generator g
- Given input  $i \in \{0, 1, \dots, p-1\}$ , we are interested in computing  $g^i \in \mathbb{Z}_p^*$

・ロト ・ 日 ・ ・ 日 ・ ・ 日 ・



- Note that this algorithm runs the inner loop *i* times. The number *i* can take values {0, 1, ..., *p* − 2}. For example, if *i* ≥ 2<sup>500</sup> then the algorithm will run the inner loop more than the number of atoms in the universe. Effectively, the algorithm is useless
- The algorithm takes O(i) run-time. The size of the input *i* is log *i*. So, this algorithm is an exponential time algorithm



 Note that the argument to Exp becomes smaller by one-bit in recursive call. So, the algorithm performs (at most) 1000 recursive call. This is an <u>efficient</u> algorithm because it runs in time O(log i)

▲圖 ▶ ▲ 国 ▶ ▲ 国 ▶ …

#### A Few Optimizations.

- Testing whether *i* is even or not can be performed by computing *i*&1 (here, & is the bit-wise and of the binary representation of *i* and 1
- Computing (i/2) when *i* is even, or computing (i-1)/2 when *i* is odd can be achieved by  $i \gg 1$  (that is, right-shift the binary representation of *i* by one position)

(日本) (日本) (日本)

## Second Attempt III

The code shall look as follows

Exp (i): a) If i = 0: Return eb)  $j \gg 1$ c) If (i&1) == 0: c)  $\alpha = \operatorname{Exp}(j)$ c) Return  $\alpha \circ \alpha$ c) else: c)  $\alpha = \operatorname{Exp}(j)$ c) Return  $\alpha \circ \alpha \circ g$ 

#### Repeated Squaring

▲圖 ▶ ▲ 臣 ▶ ▲ 臣 ▶

• The algorithm makes recursive calls. Can we further optimize and avoid recursive function calls? That is, can we unroll the recursion into a for loop? In the following code, we assume that we represent the prime p using *t*-bits. For example, we were considering t = 1000 in the ongoing example.

We perform a preprocessing step to compute the following global variables.

#### **Global Preprocessing.**

• For index in the set 
$$\{0, 1, \ldots, t-1\}$$
:

• If index == 0: 
$$\alpha_{index} = g$$
 and  $c_{index} = 1$ 

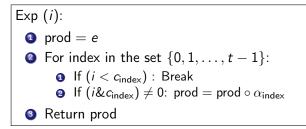
**2** Else: 
$$\alpha_{index} = \alpha_{index-1} \circ \alpha_{index-1}$$
 and  $c_{index} = (c_{index-1} \ll 1)$ 

- Note that  $lpha_{\mathsf{index}} = g^{2^{\mathsf{index}}}$ , for all  $\mathsf{index} \in \{0, 1, \dots, t-1\}$
- Further, note that  $c_{index} = 2^{index}$ , for all index  $\in \{0, 1, \dots, t-1\}$

(비) (종) (종) (종)

### Final Attempt II

We shall use the preprocessed data to compute the exponentiation



- Note that the test "the (1 + index)-th bit in the binary representation of *i* is 1" is identical to the test  $(i\&c_{index}) \neq 0$
- $\bullet\,$  If this test passes, then prod is multiplied by  $\alpha_{\rm index}={g^{2^{\rm index}}}$
- Prove: This approach correctly calculates g<sup>i</sup>
- Note that the runtime is  $O(\log i)$  (that is, the algorithm is efficient)

▲ 同 ▶ → 目 ▶ → ● ▶ →