Lecture 04: Repeated Squaring
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 \(g^i = \overbrace{g \circ g \circ \cdots \circ g}^{i\text{-times}}\).

For example, the group \((\mathbb{Z}_7^*, \times)\) is generated by 3 but not 2.
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 $i \in \{0, 1, \ldots, p - 1\}$, we are interested in computing $g^i \in \mathbb{Z}_p^*$. 

Repeated Squaring
Exp (i):

1. \( \text{prod} = e \)  
2. For index in the range \( \{1, \ldots, i\} \):  
   1. \( \text{prod} = \text{prod} \circ g \)  
3. Return \( \text{prod} \)

- Note that this algorithm runs the inner loop \( i \) times. The number \( i \) can take values \( \{0, 1, \ldots, p - 2\} \). For example, if \( i \geq 2^{500} \) 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.
Second Attempt I

Exp (i):

1. If \( i = 0 \): Return \( e \)
2. If \( i \) is even:
   1. \( \alpha = \text{Exp}(i/2) \)
   2. Return \( \alpha \circ \alpha \)
3. If \( i \) is odd:
   1. \( \alpha = \text{Exp}((i - 1)/2) \)
   2. Return \( \alpha \circ \alpha \circ g \)

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 efficient algorithm because it runs in time \( O(\log i) \)
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).
The code shall look as follows

**Exp (i):**

1. If $i = 0$: Return $e$
2. $j \gg 1$
3. If $(i \& 1) == 0$:  
   1. $\alpha = \text{Exp}(j)$  
   2. Return $\alpha \circ \alpha$
4. else:  
   1. $\alpha = \text{Exp}(j)$  
   2. Return $\alpha \circ \alpha \circ g$
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.**

1. For index in the set $\{0, 1, \ldots, t - 1\}$:
   1. If $\text{index} == 0$: $\alpha_{\text{index}} = g$ and $c_{\text{index}} = 1$
   2. Else: $\alpha_{\text{index}} = \alpha_{\text{index}-1} \circ \alpha_{\text{index}-1}$ and $c_{\text{index}} = (c_{\text{index}-1} \ll 1)$

- Note that $\alpha_{\text{index}} = g^{2^\text{index}}$, for all $\text{index} \in \{0, 1, \ldots, t - 1\}$
- Further, note that $c_{\text{index}} = 2^\text{index}$, for all $\text{index} \in \{0, 1, \ldots, t - 1\}$
We shall use the preprocessed data to compute the exponentiation

\[ \text{Exp} \left( i \right): \]

1. prod = e
2. For index in the set \{0, 1, \ldots, t - 1\}:
   - If \( i < c_{\text{index}} \): Break
   - If \( i \& c_{\text{index}} \neq 0 \): \( \text{prod} = \text{prod} \circ \alpha_{\text{index}} \)
3. Return prod

Note that the test “the \((1 + \text{index})\)-th bit in the binary representation of \( i \) is 1” is identical to the test \( i \& c_{\text{index}} \neq 0 \)

If this test passes, then \( \text{prod} \) is multiplied by \( \alpha_{\text{index}} = g^{2^{\text{index}}} \)

Prove: This approach correctly calculates \( g^i \)

Note that the runtime is \( O(\log i) \) (that is, the algorithm is efficient)