Lecture 06: Repeated Squaring

Repeated Squaring

・通 と ・ ヨ と ・

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

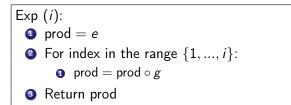
• We define
$$g^i = \overbrace{g \circ g \circ \cdots \circ g}^{i}$$

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

◆□▶ ◆□▶ ◆□▶ ◆□▶

Motivation of Efficient Algorithm to Compute Exponentiation

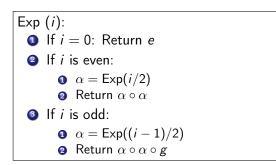
- Suppose *p* is a prime number that is represented using 1000-bits
- Note that the number p is in the range [2⁹⁹⁹, 2¹⁰⁰⁰). We shall summarize this by stating that p is roughly (in the order of) 2¹⁰⁰⁰.
- Suppose we are interested to work on the field (\mathbb{Z}_p^*, \times) with generator g
- Given input $i \in \{0, 1, \dots, p-1\}$, we are interested in computing $g^i \in \mathbb{Z}_p^*$



- Note that this algorithm runs the inner loop *i* times. The number *i* can take values {0, 1, ..., *p* − 2}. For example, if *i* ≥ 2⁵⁰⁰ 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

▲御▶ ▲ 国▶ ▲ 国▶

Second Attempt I



 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)

▲御▶ ▲屋▶ ▲屋▶

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)

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) $\alpha = \text{Exp}(j)$ c) Return $\alpha \circ \alpha$ c) else: c) $\alpha = \text{Exp}(j)$ c) Return $\alpha \circ \alpha \circ g$

- 4 回 ト - 4 回 ト - 4 回 ト

• 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
$$\{0, 1, ..., t - 1\}$$
:

• 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 $\in \{0, 1, \dots, t-1\}$

イロト 不得 トイヨト イヨト 三日

Final Attempt II

We shall use the preprocessed data to compute the exponentiation

Exp (i):
a prod = e
For index in the set
$$\{0, 1, \dots, t-1\}$$
:
a If $(i < c_{index})$: Break
b If $(i\&c_{index}) \neq 0$: prod = prod $\circ \alpha_{index}$
c Return prod

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

向下 イヨト イヨト

- Let us consider a problem that shall use all the facts we studied about groups and fields in the last two lectures. There are multiple solutions with varying degree of complexities
- Compute

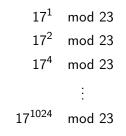
 $17^{2020} \mod 23$

Repeated Squaring

伺 ト イヨ ト イヨト

Solution 1.

• We can use repeated squaring directly to compute



- Write 2020 in binary and compute 17²⁰²⁰ mod 23 using the values computed above
- Although this is a correct and a tractable way to compute this value, it is computationally intensive and prone to errors (without a calculator)

Solution 2.

- In homework you will prove that $x^p = x \mod p$, where p is a prime and x is any integer
- You can use this fact to simplify the computation of 17²⁰²⁰ mod 23 as follows

$$17^{2020} \mod 23 = 17^{23} \cdot 17^{1997} \mod 23$$

$$= (17^{23})^2 \cdot 17^{1974} \mod 23$$

$$\vdots$$

$$= (17^{23})^{87} \cdot 17^{19} \mod 23$$

$$= (17)^{87} \cdot 17^{19} \mod 23, \qquad \text{using } x^p = x \mod p$$

$$= 17^{106} \mod 23$$

$$= (17)^4 \cdot 17^{14} \mod 23, \qquad \text{using } x^p = x \mod p$$

$$= 17^{18} \mod 23$$

• This final expression can be computed using the repeated squaring technique

▲御▶ ▲理▶ ▲理▶

Solution 3.

- In homework you will prove that x^{p-1} = 1 mod p, where p is a prime and x is any integer NOT divisible by p (there are also alternate proofs of this statement by considering the size of the subgroup of (Z^{*}_p, ×) that is generated by x)
- So, we can compute the expression as follows

• BTW, in general you can conclude that

$$x^n = x^n \mod p - 1 \mod p$$
,

for any integer n and any integer x that is not divisible by p
Now you can compute 17¹⁸ mod 23 result using repeated squaring technique

$$17^{1} = 17 \mod 23$$

 $17^{2} = 13 \mod 23$
 $17^{4} = 8 \mod 23$
 $17^{8} = 18 \mod 23$
 $17^{16} = 2 \mod 23$

• Now, we have

$$17^{18} = 17^{16+2} \mod 23$$

= 17^{16} \cdot 17^2 \cdot mod 23
= 2 \cdot 13 \cdot mod 23
= 3 \cdot mod 23

• Therefore, we conclude that

$$17^{2020} = 17^{18} = 3 \mod 23.$$

That is our answer!

・ロト ・ 日 ・ ・ ヨ ・ ・ ヨ ・