Lecture 5 Part B (+6): Integer Exponentiation
Lecture 5B-6A: Plan

• Integer exponentiation:
  • Naive algorithm (inefficient).
  • Iterated squaring algorithm (efficient).
  • Modular exponentiation.
Integer Exponentiation

• How do we compute $a^b$, where $a, b$ are both integers, $b \geq 0$?

```python
>>> a = 17
>>> for b in range(0, 20):
    print(a**b)
1
17
289
4913
83521
1419857
24137569
410338673
6975757441
118587876497
2015993900449
34271896307633
582622237229761
9904578032905937
168377826559400929
2862423051509815793
48661191875666868481
827240261886336764177
14063084452067724991009
239072435685151324847153
```
Integer Exponentiation: Naive Method

• The naive method: Compute successive powers $a, a^2, a^3, \ldots, a^b$.
• Starting with $a^0 = 1$, this takes $b$ multiplications.

• For example, if $b$ is 20 bits long, say $b = 2^{20} - 17$, such procedure takes $b = 2^{20} - 17 = 1048559$ multiplications.

• If $b$ is 1000 bits long, say $b = 2^{1000} - 17$, such procedure takes $b = 2^{1000} - 17$ multiplications.

In decimal, $2^{1000} - 17$ is

\[
1071508607186267320948425049060001810561404811705533607443750
3883703510511249361224931983788156958581275946729175531468251
8714528569231404359845775746985748039345677748242309854210746
0506237114187795418215304647498358194126739876755916554394607
7062914571196477686542167660429831652624386837205668069359
\]

Thus, for a 1000 bit long input, such a computation is completely infeasible.
def naive_power(a, b):
    """computes a**b using all successive powers
    assumes b is a nonnegative integer  """
    result = 1  # a**0
    for i in range(0, b):  # b iterations
        result = result * a
    return result

>>> naive_power(3, 0)
1
>>> naive_power(3, 2)
9
>>> naive_power(3, 10)
59049
>>> naive_power(3, 100)
515377520732011331036461129765621272702107522001
>>> naive_power(3, -10)
1

Take a look at the code and see if you understand it, and specifically why raising 3 to -10 returned 1.
More Efficient Integer Exponentiation: A Concrete Example

• Suppose we want to compute $a^{67}$.

• if $b$ is odd: $a^b = a^{b-1} \cdot a$

• else $a^b = (a^{b/2})^2$

We have 6 squaring. Each takes just one multiplication.

• Plus we had 2 additional multiplications by $a$.

• All in all, we need just $6 + 2 = 8$ multiplications. Way better than the $67$ multiplications of the naive method.

$$a^{67} = a^{66} \cdot a = (a^{33})^2 \cdot a = (a^{32} \cdot a)^2 \cdot a = ((a^{16})^2 \cdot a)^2 \cdot a = ((a^8)^2 \cdot a)^2 \cdot a = ((a^4)^2 \cdot a)^2 \cdot a = ((a^2)^2 \cdot a)^2 \cdot a = (a^2)^2 \cdot a = a^2 \cdot a$$
def power1(a, b):
    """ computes a**b using iterated squaring 
    assumes b is a nonnegative integer """
    result = 1
    while b > 0:  # b has more digits
        if b % 2 == 1:  # b is odd
            result = result * a
            b = b - 1
        else:
            a = a * a
            b = b // 2
    return result
Integer Exponentiation: Iterated Squaring, running the Python Code

Let us now run this on a few cases:

```python
>>> power1(3, 4)
81
>>> power1(5, 5)
3125
>>> power1(2, 10)
1024
>>> power1(2, 30)
1073741824
>>> power1(2, 100)
1267650600228229401496703205376
>>> power1(2, -100)
1
```
Proving **Correctness** using Loop Invariants

We can prove the correctness of the function **power1**, by showing a loop invariant – a condition that holds each time we are about to check the loop condition.

- **Base**: We show the condition holds before we enter the loop for the first time.
- **Step**: we show that if the condition holds before entering the loop for the \(i\)-th time, it will hold when we enter the loop for the \((i + 1)\)-th time.
- **Termination**: We also show that the iteration is executed a finite number of times. This implies that the condition will hold when we exit the loop for the last time.
- Finally, we show that if the condition is satisfied when the execution terminates, this implies that the code is indeed correct.

Note that such proof is in fact a proof by induction on the number of times the loop is executed.
Proving **Correctness of power**

- Denote the arguments to the function by $A, B$ (to distinguish from the changing values $a, b$).
- We claim that each time we are about to check the loop condition, the following invariant holds:

\[ \text{result} \cdot a^b = A^B \]

- First, let's check it by adding printing to the code:

```python
...  
while b>0:
    print("result = ",result, \  
" a =", a," b =" ,b, \  
" result*(a**b)=", result*a**b)
    if b%2 == 1:
        ...
```

```python
...  
```

```python
...  
```
Correctness of the Python Code (cont.)

>>> power1(3,11)
result = 1  a = 3  b = 11  result*(a**b)= 177147
result = 3  a = 3  b = 10  result*(a**b)= 177147
result = 3  a = 9  b = 5  result*(a**b)= 177147
result = 27  a = 9  b = 4  result*(a**b)= 177147
result = 27  a = 81  b = 2  result*(a**b)= 177147
result = 27  a = 6561  b = 1 result*(a**b)= 177147
177147

• So at least in this example the condition holds every time!
Proof of Correctness (1)

• Now we want to prove that this is indeed an invariant condition.
• Denote the arguments to the function by $A$, $B$ (to distinguish from the changing values $a$, $b$).
• We claim that each time we are about to check the loop condition, the following condition holds:

$$\text{result} \cdot a^b = A^B$$

• **Base**: The first time we enter the loop, $\text{result}=1$, $a=A$, and $b=B$, so the condition is true.
Proof of Correctness (2)

• **Step:** Now execute the loop body once. The values of the variables change (the new ones are denoted $a'$ and $b'$).

• There are two possibilities:
  
  If $b$ is odd, then
  
  \[
  \begin{align*}
  \text{result}' &= \text{result} \cdot a \\
  b' &= (b - 1) \\
  a' &= a
  \end{align*}
  \]

  So:
  \[
  \text{result}' \cdot (a')^{b'} = \text{result} \cdot a \cdot a^{b-1} = \text{result} \cdot a^b = A^B
  \]

  Substitute the values

  Inductive assumption

So the invariant holds after executing another iteration.

The code:

```python
if b%2 == 1:  # b is odd
    result = result*a
    b = b-1
else:
    a = a*a
    b = b//2
```
Proof of Correctness (3)

- **Step**: Now execute the loop body once. The values of the variables change (the new ones are denoted $a'$ and $b'$).

- There are two possibilities:
  
  **If $b$ is even**, then
  
  $result' = result$
  
  $b' = b/2$
  
  $a' = a^2$
  
  **The code**:
  
  ```python
  if b%2 == 1:  # b is odd
    result = result*a
    b = b-1
  else:
    a = a*a
    b = b//2
  ```

  So: $result' \cdot (a')^{b'} = result \cdot (a^2)^{b/2} = result \cdot (a)^b = A^B$

  **Substitute the values**

  **Inductive assumption**
Proof of Correctness (4)

• So in both cases, the invariant condition remains true after each execution of the loop body.

• **Termination:** the loop terminates, because \( b \) is reduced in each execution of the loop body.

• When the loop terminates, \( b = 0 \) (why?)

So: \( A^B = \text{result} \cdot a^b = \text{result} \cdot a^0 = \text{result} \), as desired.

QED
Correctness of Code – Remarks

• In general, it is not easy to design correct code. It is even harder to prove that a given piece of code is correct (namely it meets its specifications).

• In the course, we may see a couple more examples of program correctness, using the same technique of loop invariants. (You will not be expected to prove correctness in this way.)

• However, in most cases you will have to rely on your understanding, intuition, test cases, and informative prints to convince yourselves that the code you wrote is indeed hopefully correct.

• Finally, we remark that software and hardware verification are major issues in the corresponding industries. Elective courses on these topics are being offered at TAU (and elsewhere).
Iterated Squaring, Improvements

• Note that we could discard the two lines struck through below.

```python
def power1(a,b):
    result = 1
    while b>0:       # b has more digits
        if b%2 == 1: # b is odd
            result = result*a
            b = b-1
        else:
            a = a*a
            b = b//2
    return result
```

• This is because after an iteration in which $b$ is odd, comes an iteration in which it is even.

• Plus recall that $b//2$ rounds down the result.
A Different View of Iterated Squaring

• The resulting implementation provides a different, and interesting interpretation of our algorithm, which will be explained now.

• The new interpretation relates to $b$'s representation in binary.

• First, note that $b//2$ actually discards the least significant bit (LSB) of $b$.

```python
def power2(a, b):
    result = 1
    while b > 0:  # b has more digits
        if b % 2 == 1:  # b is odd
            result = result * a
        a = a * a
        b = b // 2    # discard b's rightmost bit
    return result
```
The Binary Interpretation: A Concrete Example

• Suppose we want to compute $a^{67}$.

• We can represent 67 as a sum of powers of 2 (this representation is unique, and corresponds to the binary representation of 67 (1000011), that is $67 = 64 + 2 + 1$.

• We compute the terms $a^{2^i}$: $a^2, a^4, a^8, a^{16}, a^{32}, a^{64}$. Each additional squaring takes just one multiplication (e.g. $a^{64} = a^{32} \cdot a^{32}$). So overall, computing all these six exponents takes just 6 multiplications.

• We note that $a^{67} = a^{64+2+1} = a^{64} \cdot a^2 \cdot a^1$. So to compute $a^{67}$ takes 2 additional multiplications.

• All in all, we need just $6 + 2 = 8$ multiplications. Way better than the 67 multiplications of the naive method.
The Binary Interpretation: general observations (1)

• Let \( b \) be an \( n \)-bit integer, namely \( 2^{n-1} \leq b < 2^n \).

• So instead of starting with \( a^0 \) and computing all successive powers of \( a \), namely \( a, a^2, a^3, \ldots, a^b \), we can use just successive powers of two of \( a \) smaller or equal to \( b \), namely \( a, a^2, a^4, a^8, \ldots, a^{2^{n-1}} \).

To compute them, observe that \( a^{2i+1} = (a^{2i})^2 \).

• We can combine \( a, a^2, a^4, a^8, \ldots, a^{2^{n-1}} \) to compute the desired power, \( a^b \) (next slide).
The Binary Interpretation:

general observations (2)

- We can combine $a, a^2, a^4, a^8 \ldots, a^{2^{n-1}}$ to compute the desired power, $a^b$.

- Let $b = \sum_{i=0}^{n-1} (b_i \cdot 2^i)$.

  The $b_i$'s are simply the bits in the binary representation of $b$ ($b = b_{n-1} \ldots b_2 b_1 b_0$).

- Then $a^b = a^{\sum_{i=0}^{n-1} b_i \cdot 2^i} = \prod_{i=0}^{n-1} \left(a^{b_i \cdot 2^i}\right) = \prod_{i=0}^{n-1} \left(a^{2^i}\right)^{b_i}$

  \[ a^{c+d} = a^c \cdot a^d \quad a^{c\cdot d} = (a^c)^d \]

- In fact, note that we should accumulate only those powers of $a$ that correspond to bits in $b$ with value 1.
Number of Multiplication Analysis

• Let $b$ be an $n$-bit integer, namely $2^{n-1} \leq b < 2^n$.

• $a^b = a^{\sum_{i=0}^{n-1} b_i \cdot 2^i} = \prod_{i=0}^{n-1} (a^{b_i \cdot 2^i}) = \prod_{i=0}^{n-1} (a^{2^i})^{b_i}$

• How many multiplications are needed?

$\prod_{i=0}^{n-1} (a^{2^i})^{b_i}$ has $n$ terms and thus at most $n - 1$ multiplications between them.

  • why at most?
  • and how many at least?

Computing the $a^{2^i}$'s requires additional $n - 1$ multiplications (recall that squaring requires just one multiplication).
Naïve vs. Iterated Squaring

Given two integers $a, b$, where $b \geq 0$ the size of $b$ is $n$ bits, namely $2^{n-1} \leq b < 2^n$:

- The Naïve algorithm takes $b$ multiplications, which is between $2^{n-1}$ and $2^n - 1$.
- Iterated squaring takes between $n - 1$ and $2(n - 1)$ multiplications.

So naïve is exponentially slower than iterated squaring!

Remark: We counted just “multiplications” here, and ignored the size of numbers being multiplied, and how many basic operations this requires. This simplifies the analysis and right now does not deviate too much from “the truth”.

23
Python Implementation - Remarks

- While the abstract iterated squaring algorithm performs at most $2(n - 1)$ multiplications, the Python code of `power2` may perform 2 additional ones (where are they hiding?), so most $2n$ multiplications. Try it on $b = 3$.

- This difference is not very significant, and can be eliminated by adding appropriate conditions to the code (which we avoid, to keep the code simple).
The Two Types of Time Complexity Analysis

1) Mathematical analysis:
   • Analyzing the number of operations exactly (like we did)
   • Analyzing the number of operations approximately (up to constants) and asymptotically (we will do this a lot in the future).
   (Caveat: When faced with a concrete task on a specific problem size, you may be far away from “the asymptotic”.)

2) Direct measurements of the actual running time:
   • For direct measurements, we will use either
     • the time package and the time.clock() function, or
     • the timeit package and the timeit.timeit() function.
   • Both have some deficiencies, yet are highly useful for our needs.
Direct Time Measurement, Using 
\texttt{time.clock()}

• The function \texttt{elapsed} measures the CPU time taken to execute the given expression (given as a \texttt{string}). Returns a result in \texttt{seconds}. Note that the code first imports the \texttt{time} module.

```python
import time        # imports the Python time module

def elapsed(expression, number=1):
    ''' computes elapsed time for executing code number times (default is 1 time). expression should be a string representing a Python expression.'''
    t1 = time.clock()
    for i in range(number):
        eval(expression)   # eval invokes the interpreter
    t2 = time.clock()
    return t2-t1
```
Direct Time Measurement, Using time.clock()

• Put elapsed.py at the same folder with function you want to measure their running time
• Open elapsed.py, and hit the F5 button or choose run => run module

Examples:
>>> elapsed("sum(range(10**7))")
0.33300399999999897
>>> elapsed("sum(range(10**8))")
3.362785999999998
>>> elapsed("sum(range(10**9))")
34.02992000000004
Reality Show: Naive Squaring vs. Iterated Squaring

Actual Running Time Analysis:
We'll measure the time needed (in seconds) for computing 3 raised to the powers $2 \cdot 10^5$, $10^6$, $2 \cdot 10^6$ using the two algorithms.

```python
>>> from power import *
>>> elapsed("naive_power(3, 200000)")
2.244201
>>> elapsed("power(3, 200000)")
0.03179299999999996

>>> elapsed("naive_power(3, 1000000)")
57.696312999999996
>>> elapsed("power(3, 1000000)")
0.3366879999999952

>>> elapsed("naive_power(3, 2000000)")
205.56775500000003
>>> elapsed("power(3, 2000000)")
1.0069569999999999
```

Iterated squaring wins (big time)!
Comment about time.clock()

- The python documentation for time.clock() states that it is
  
  **Deprecated** since version 3.3: The behaviour of this function depends on the platform: use perf_counter() or process_time() instead, depending on your requirements, to have a well defined behaviour.

- Deprecated means: advice not to use it in new code written, but it is not removed from the language, so that old code does not stop functioning.

- The reason – it measures different “time” on different systems: processor time vs. wall-clock time.
Wait a Minute

• Using iterated squaring, we can compute \(a^b\) for any \(a\) and, say, \(b = 2^{100} - 17 \approx 1.26765060228229401496703205359\). This will take less than 200 multiplications, a piece of cake even for an old, faltering machine.

• A piece of cake? Really? 200 multiplications of what size numbers?

• For any integer \(a\) other than 0 or 1, the result of the exponentiation above is over \(2^{99}\) bits long. No machine could generate, manipulate, or store such huge numbers.

• Can anything be done? Not really!

• Unless you are ready to consider a closely related problem:

  • **Modular exponentiation:** Compute \(a^b \mod c\), where \(a, b, c \geq 2\) are all integers. This is the remainder of \(a^b\) when divided by \(c\). In Python, this can be expressed as \((a ** b) \% c\).
Modular Exponentiation

• We should still be a bit careful. Computing \( a^b \) first, and then taking the remainder \( \text{mod } c \), is not going to help at all.

• Instead, we compute all the successive squares \( \text{mod } c \), namely \( a^1 \text{ mod } c, \ a^2 \text{ mod } c, \ a^4 \text{ mod } c \) (and any other power that is needed).

• In fact, following every multiplication, we compute the remainder. We rely on the fact that for all \( a, b, c \) :

\[
((a \text{ mod } c) \cdot (b \text{ mod } c)) \text{ mod } c = (a \cdot b) \text{ mod } c.
\]

• This way, intermediate results never exceed \( c^2 \), eliminating the problem of huge numbers.
Modular Exponentiation in Python

We can easily modify our function, power, to handle modular exponentiation.

```python
def modpower(a, b, c):
    """ computes a**b modulo c, using iterated squaring
    assumes b is a nonnegative integer
    ""
    result = 1
    while b > 0:
        if b % 2 == 1:
            result = (result * a) % c
        a = (a * a) % c
        b = b // 2
    return result
```
Modular Exponentiation in Python

A few test cases:

```python
>>> modpower(2, 10, 100)  # sanity check: \(2^{10} = 1024\)
24

>>> modpower(17, 2**100+3**50, 5**100+2)
3568728177468732193582328510109849308957750682733818418319936978305748

>>> 5**100+2  # the modulus, in case you are curious
7888609052210118054117285652827862296732064351090230047702789306640627

>>> modpower(17, 2**1000+3**500, 5**100+2)
1119887451125159802119138842145903567973956282356934957211106448264630
```
Built In Modular Exponentiation: \texttt{pow}(a,b,c)

- Guido van Rossum has not waited for our code, and Python has a built-in function, \texttt{pow}(a,b,c), for efficiently computing $a^b \mod c$.

  
  
  \begin{verbatim}
  >>> modpower(17, 2**1000+3**500, 5**100+2)\n  - pow(17, 2**1000+3**500, 5**100+2)
  0
  \end{verbatim}

- This is comforting: modpower code and Python pow agree. Phew ...

  
  \begin{verbatim}
  >>> elapsed("modpower(17, 2**1000+3**500, 5**100+2)"")
  0.0026359999999542
  >>> elapsed("modpower(17, 2**1000+3**500, 5**100+2)",
  number=1000)
  2.280894000000046
  >>> elapsed("pow(17, 2**1000+3**500, 5**100+2)",
  number=1000)
  0.7453199999999924
  \end{verbatim}

- So our code is just three times slower than \texttt{pow}.
Does Modular Exponentiation Have Any Uses?

Applications using modular exponentiation directly (partial list):

- Randomized primality testing.
- Diffie Hellman Key Exchange
- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC)

We will discuss the first two topics later in this course, and leave RSA PKC to an (elective) crypto course.