Extended Introduction to Computer Science CS1001.py

Chapter F Lecture 12

Topics in Number Theory: Integer Exponentiation

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University

Fall Semester 2023-24

http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

מבנה ונושאי הקורס (ייתכנו שינויים)

פרק	נושאים מתוכננים
A. יסודות פייתון	• תכנות בסיסי: טיפוסי ערכים, משתנים, משפטי תנאי, לולאות, פונקציות, מודל הזיכרון • נושאים נוספים: דקדוקים פורמליים ותהליך הפירוש של פייתון, פונקציות למבדא, ופונקציות סדר גבוה, אקראיות ושימושיה, סוגי שגיאות (תחביר, זמן ריצה), סגנון תכנות "נכון"
B. ייצוג טיפוסי מידע	• ייצוג שלמים בשיטה הבינארית • ייצוג מספרים עם נקודה עשרונית בשיטת floating point • ייצוג תווים (ASCII, Unicode)
C. אלגוריתמים בסיסיים וסיבוכיות	• חיפוש בינארי, מיון בחירה, מיזוג רשימות ממוינות • סיבוכיות ו- O notation
D. חישוב נומרי	• מציאת שורש של פונקציה ממשית רציפה בשיטת החציה ~~בעבר: שיטת ניוטון-רפסון~~, חישוב נגזרות ואינטגרלים, קירוב ל π
E. רקורסיה	• עצרת, פיבונאצ'י, חיפוש בינארי, מיון מהיר, מיון מיזוג, ממואיזציה, דוגמאות נוספות
F. נושאים בתורת המספרים	• העלאה בחזקה טבעית בשיטת Iterated squaring • בדיקת ראשוניות הסתברותית (המשפט הקטן של פרמה) • פרוטוקול Diffie-Hellman להחלפת מפתח סודי • מחלק משותף מקסימלי (GCD)
G. תכנות מונחה עצמים (OOP) ומבני נתונים	• מחלקות, שדות ומתודות • רשימות מקושרות והשוואה לרשימות של פייתון • עצי חיפוש בינאריים • טבלאות hash • זרמים (streams) ופונקציות גנרטור
H. טקסט	• אלגוריתם CYK ~~בעבר: אלגוריתם קארפ-רבין~~ • דחיסת האפמן, דחיסת למפל זיו
I. ייצוג ועיבוד תמונה	• ייצוג תמונה דיגיטלית, ניקוי רעש (ממוצע וחציון מקומי), נושאים נוספים לפי הזמן
J. קודים לגילוי ולתיקון שגיאות	• ספרת ביקורת, קוד חזרה, ביט זוגיות, מרחק האמינג, קוד האמינג

YOU ARE HERE

Topics in Number Theory: Plan

1. Exponentiation of integers – this lecture

2. Primality testing (using Fermat's "little theorem")

3. Diffie-Helman secret key exchange

4. Euclid's GCD (greatest common divisor)

Integer Exponentiation: Plan

1. Exponentiation of integers (a^b)

   - Naive algorithm (inefficient)

   - Iterated squaring algorithm (efficient)

   - Modular exponentiation ($a^b\ \%c$)

Integer Exponentiation

- Problem definition:
  - Input: two integers a, b where $b \geq 0$
  - Output: a^b

- As you know, Python can do this:

```
>>> print(17**20)
2390724356851513248471153
```

- But we do not settle for a “Python can do it” solution. We want to explore this ourselves, develop an efficient algorithm and analyze it

Naïve Integer Exponentiation

- The naïve method:
  - Compute successive powers $a^0, a^1, a^2, a^3, \ldots., a^b$.

- Time complexity?
  - For simplicity, we count the number of multiplications needed, ignoring the size of the numbers multiplied, which increases throughout the process
  - Justification (beyond simplicity): this will still allow us to compare this naïve solution to the improved one, soon to be presented

  - We need b multiplications. What is the size of b?
  - If b has n bits, namely $2^{n-1} \leq b < 2^n$, this is $\Theta(2^n)$ multiplications.

  - So this solution has exponential time complexity as a function of n, the size in bits of the exponent.

Naïve Integer Exponentiation

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

- If $n = 60$ bits long, say $b = 2^{60} - 17$, such procedure takes $2^{60} - 17 = 1844674407370955 1599$ multiplications.

- A computer capable of 10^{10} multiplication per second would still need over 58 years to complete the computation!

- So this exponential-time complexity solution is completely infeasible even for moderate size numbers (with merely a few tens of bits).

Naïve Integer Exponentiation in Python

```
def naive_power(a,b):
    """ Computes a**b using all successive powers.
        Assume a,b are integers, b>=0 """
    result = 1  # a**0
    for i in range(0,b):       # b iterations
        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.

Iterated Squaring (A concrete example first)

- Suppose we want to compute a^{67}.
- if b is odd: $a^b = a^{b-1} \cdot a$

  else $a^b = a^{b/2} \cdot a^{b/2} = \left(a^{b/2}\right)^2$

- Number of multiplications?
- We have 6 squaring, each takes just a single multiplication: $a^n = \left(a^{n/2}\right) \cdot \left(a^{n/2}\right)$
- Plus we have 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.

$$
\begin{aligned}
a^{67} &= a^{66} \cdot a \\
&= (a^{33})^2 \cdot a \\
&= (a^{32} \cdot a)^2 \cdot a \\
&= \left((a^{16})^2 \cdot a\right)^2 \cdot a \\
&= \left(\left((a^{8})^2\right)^2 \cdot a\right)^2 \cdot a \\
&= \left(\left(\left((a^{4})^2\right)^2\right)^2 \cdot a\right)^2 \cdot a \\
&= \left(\left(\left(\left((a^{2})^2\right)^2\right)^2\right)^2 \cdot a\right)^2 \cdot a \\
&= \left(\left(\left(\left(\left((a)^{2}\right)^2\right)^2\right)^2\right)^2 \cdot a\right)^2 \cdot a
\end{aligned}
$$

Iterated Squaring, Recursive Code - Take 1

```
def power_rec1(a,b):
    ''' Computes a**b using iterated squaring, recursively.
        Assume a,b are integers, b>=0  '''

    if b==0:
        return 1

    if b%2 == 1: # b is odd
        return power_rec1(a, b-1) * a
    else:
        return power_rec1(a, b//2) * power_rec1(a, b//2)
```

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

else $a^b = a^{b/2} \cdot a^{b/2} = \left(a^{b/2}\right)^2$

- This implementation calls for improvements in efficiency. Why?

Iterated Squaring, Recursive Code - Take 2

```
def power_rec2(a,b):
    ''' Computes a**b using iterated squaring, recursively.
        Assume a,b are integers, b>=0  '''

    if b==0:
        return 1

    if b%2 == 1: # b is odd
        return power_rec2(a, b-1) * a
    else:
        res = power_rec2(a, b//2)
        return res*res
```

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

else $a^b = a^{b/2} \cdot a^{b/2} = \left(a^{b/2}\right)^2$

- We could further improve style, recalling that when b is even:

$$a^b = \left(a^{b/2}\right)^2 = (a^2)^{b/2}$$

Iterated Squaring, Recursive Code - Take 3

```
def power_rec3(a,b):
    ''' Computes a**b using iterated squaring, recursively.
        Assume a,b are integers, b>=0  '''

    if b==0:
        return 1

    if b%2 == 1: # b is odd
        return power_rec3(a, b-1) * a
    else:
        return power_rec3(a*a, b//2)
```

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

else $a^b = a^{b/2} \cdot a^{b/2} = \left(a^{b/2}\right)^2 = (a^2)^{b/2}$

- Note: following a recursive call in which b is odd, must come a call in which it is even.
- What are the worst and best cases here?

Iterated Squaring, Recursive Code - Take 3 – Time Complexity

- For both worst and best cases, recursion tree is a "chain" of depth $O(logb) = O(n)$.

- As we already mentioned, this is not the real time complexity of the function, since we ignored the time complexity of the arithmetical operations in each call.
  - We have a subtraction (`b-1`), which takes $O(n)$ for an n-bit number.

  - We have a floor division by 2 (`b//2`), which also takes $O(n)$ for an n-bit number (note that this is an exceptions, as division normally takes $O(n^2)$, like multiplication).

  - However, as we already mentioned, the multiplications at each step involve numbers of increasing sizes, and we do not analyze this here (maybe HW).

Iterated Squaring, Iterative Code

```
def power1(a,b):
    """ Computes a**b using iterated squaring.
        Assume a,b are integers, b>=0 """
    result = 1
    while b>0:
        if b%2 == 1:
            result *= a
            b = b-1
        else:
            a = a*a
            b = b//2
    return result
```

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

else $a^b = a^{b/2} \cdot a^{b/2} = \left(a^{b/2}\right)^2$
$= (a^2)^{b/2}$

Iterated Squaring: Executions

Let us now run this on a few cases:

```
>>> power1(3 ,4)
81
>>> power1(5, 5)
3125
>>> power1(2, 10)
1024
>>> power1(2, 30)
1073741824
>>> power1(2, 100)
1267650600228229401496703205376
>>> power1(2, -100)
1
```

Correctness using Loop Invariant

- A loop invariant is some value that remains unchanged between iterations.

- We claim that each time we are about to check the loop condition, the following invariant holds:

$$result \cdot a^{b} = {a_0}^{b_0}$$

where a_0, b_0 are the initial values of a, b (functions arguments).

- This loop invariant can be proven by induction on the iteration number (complete proof in the appendix).

- When the loop terminates, $b = 0$.
  Conclusion: when the loop terminates, $result \cdot a^{0} = result = {a_0}^{b_0}$
  as desired.

Loop Invariant

- We can easily check this by adding prints to the code:

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

```
>>> 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 indeed holds every time!

Comic Relief *

* אנו מזמינים אתכם לשלוח לנו הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Iterated Squaring: Simplifying the Code

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

```
def power1(a,b):
    result = 1
    while b>0:
        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 following an iteration in which *b* is odd, must come 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.

```
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 LSB
    return result
```

- The new interpretation relates to b's representation in binary.
- Note that `b//2` actually discards the least significant bit (LSB) of b.

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 $67 = 64 + 2 + 1$.
- Our algorithm computes the terms $a^{(2^i)}$: $a^2,\ a^4,\ a^8,\ a^{16},\ a^{32},\ a^{64}$
- And uses (some of) them to compute $a^{67} = a^{64+2+1} = a^{64} \cdot a^2 \cdot a^1$
- In fact, note that the algorithm uses only those powers of a that correspond to bits in b with value 1:

$$a^{67} = a^{1\cdot 64 + 0\cdot 32 + 0\cdot 16 + 0\cdot 8 + 0\cdot 4 + 1\cdot 2 + 1\cdot 1}$$

$$= a^{1\cdot 64} \cdot \cancel{a^{0\cdot 32}} \cdot \cancel{a^{0\cdot 16}} \cdot \cancel{a^{0\cdot 8}} \cdot \cancel{a^{0\cdot 4}} \cdot a^{1\cdot 2} \cdot a^{1\cdot 1}$$

The Binary Interpretation:

Generalization of the previous example

- Let b be an n-bit non-negative integer
  - namely $2^{n-1} \leq b < 2^n$
  - In particular $b = (b_{n-1} \dots b_2 b_1 b_0)_2 = \sum_{i=0}^{n-1}(\, b_i \cdot 2^i)$
    - e.g. $b = 67_{10} = 1000011_2$

- Compute: $a^2,\ \ a^4,\ \ a^8 \dots, a^{(2^{n-1})}$ (no need for $a^{(2^n)} > a^b$)

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

$a^{x+y} = a^x \cdot a^y$

The Binary Interpretation:

Complexity

- Let b be an n-bit non-negative integer
  - namely $2^{n-1} \leq b < 2^n$
  - In particular $b = (b_{n-1} \dots b_2 b_1 b_0)_2 = \sum_{i=0}^{n-1} (\, b_i \cdot 2^i)$
    - e.g. $b = 67_{10} = 1000011_2$

- Compute: $a^2, \ \ a^4, \ a^8 \dots, a^{(2^{n-1})}$ (no need for $a^{(2^n)} > a^b$)
  - requires $n - 1$ multiplications

- Then $a^b = a^{\sum_{i=0}^{n-1} \, b_i \cdot 2^i} = \prod_{i=0}^{n-1} \left(a^{b_i \cdot 2^i} \right) = \prod_{b_i = 1} \left(a^{2^i} \right)$
  - requires at most $n - 1$ multiplications
    - why at most? and how many at least?

Complexity Summary: Naïve vs. Iterated Squaring

- Given two integers a, b, where $b \geq 0$ and 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

$O(2^n)$ multiplications

$O(n)$ 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 bit operations are required. This simplifies the analysis but also may deviate significantly from "the truth".

Python Implementation - Remarks

- While the abstract iterated squaring algorithm performs at most $2(n-1)$ multiplications,

  our Python code of power2 may perform up to $2n$ multiplications (where are the 2 additional ones hiding?)

- This difference is negligible, and can be eliminated by adding appropriate conditions to the code (which we avoided, to keep the code simple).

Time Measurements for Naive Squaring vs. Iterated Squaring

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

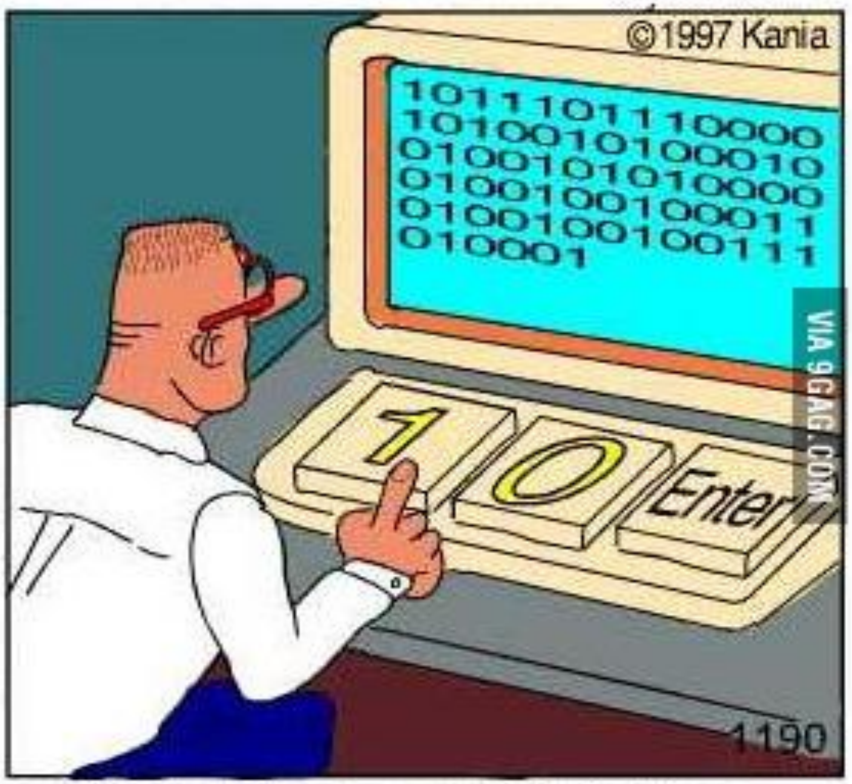
```
>>> elapsed("naive_power(3, 2*10**5)")
2.244201
>>> elapsed("power2(3, 2*10**5)")
0.03179299999999996

>>> elapsed("naive_power(3, 10**6)")
57.69631299999996
>>> elapsed("power2(3, 10**6)")
0.3366879999999952

>>> elapsed("naive_power(3, 2*10**6)")
205.5677550000003
>>> elapsed("power2(3, 2*10**6)")
1.0069569999999999
```

Iterated squaring wins (big time)!

Comic Relief *



Real programmers code in binary.

* אנו מזמינים אתכם לשלוח לנו הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Modular Exponentiation

$$a^b (mod\ c)$$

Huge Numbers

- Using iterated squaring, we can compute a^b for, say,
  $b = 2^{100} - 17 = 1267650600228229401496703205359$.
  This will take no more 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 $a = 2$ the result of the exponentiation above is $2^{100} - 16$ bits long! For $a > 2$ the result is even larger!!
  - 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 $a, b \geq 0, c \geq 2$ are all integers. This is the remainder of a^b when divided by c.
  - In Python, this can be expressed as $(a ** b) \% c$.

Does Modular Exponentiation Have Any Uses?

Applications using modular exponentiation directly (partial list):

- Randomized primality testing
- Diffie Hellman secret key exchange
- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC)

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

Modular Exponentiation

- We should still be a bit careful. Computing a^b first, and only then taking the remainder mod c, is not going to help at all.

- Instead, we compute all the squares $mod\ c$, namely:
  - $a^1\ mod\ c,\quad a^2\ mod\ c,\ \ a^4\ mod\ c\ \ldots$

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

$$(a \cdot b) mod\ c \quad = \quad ((a\ mod\ c) \cdot (b\ mod\ c))\ mod\ c$$

- This way, intermediate results never exceed $(c-1)^2$, eliminating the problem of huge numbers.

Code for Modular Exponentiation

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

```
def modpower(a,b,c):
    """ computes a**b modulo c,
        using iterated squaring
    """
    result = 1
    while b>0:
        if b%2 == 1:
            result = (result * a) % c
        a = (a*a) % c
        b = b//2

    return result
```

Code for Modular Exponentiation

- A few test cases:

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

>>> modpower(2, 2**100-17, 5**100)
7763470113346743895580721708565743044722675708816681629524142921320613


>>> modpower(17, 2**1000+3**500, 5**100+2)
1119887451125159802119138842145903567973956282356934957211106448264630
```

Built In Modular Exponentiation: pow(a,b,c)

- Guido van Rossum (Python "father") has not waited for our code, and Python has a built in function, pow(a,b,c), for efficiently computing a^b mod c.

```
>>> modpower(17, 2**1000+3**500, 5**100+2) \
        - pow(17, 2**1000+3**500, 5**100+2)
0
```

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

```
>>> elapsed("modpower(17, 2**1000+3**500, 5**100+2)", number=1000)
2.28089400000046
>>> elapsed("pow(17, 2**1000+3**500, 5**100+2)", number=1000)
0.745319999999924
```

- So our code is about 3 times slower than Python's buit-in pow.

Modular Exponentiation: Time Complexity Analysis

- Suppose a, b, c are all n-bit long integers, $b \geq 0$ and $c \geq 2$.

- To compute $a^b\ mod\ c$ using iterated squaring we need at most $2(n-1) = O(n)$ multiplications, each followed immediately by a modulo operation

  - Since Intermediate multiplicands never exceed c, each multiplication takes $O(n^2)$ bit operations (using elementary school multiplication as we saw earlier in the course).
  - Each product is smaller than c^2, which has at most $2n$ bits, and so computing the remainder of such product modulo c takes another $O(n^2)$ bit operations (using long division, also studied in elementary school, but we did not see it in this course).

- All by all, computing $a^b\ mod\ c$ takes $O(n^3)$ bit operations.

Appendix

Proving Correctness using Induction

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.

Proof of Correctness: Base

- Now we want to prove that this is indeed an invariant condition.
- We claim that each time we are about to check the loop condition, the following condition holds:

$$result \cdot a^b = {a_0}^{b_0}$$

- Base: The first time we enter the loop

  $result = 1, a = a_0$, and $b = b_0$

  so the condition is true.

Proof of Correctness: Step (odd b)

- Step: Now execute the loop body. The values of the variables change (the new ones are denoted a' , b' and $result'$).
- There are two possibilities:

If b is odd, then

$$result' = result \cdot a$$
$$b' = (b - 1)$$
$$a' = a$$

unchanged

```
if b%2 == 1: # b is odd
     result = result*a
     b = b-1
else:
     a = a*a
     b = b//2
```

So: $result' \cdot (a')^{b'} = result \cdot a \cdot a^{b-1} = result \cdot a^b = a_0^{b_0}$

Substitute the values

Inductive assumption

Proof of Correctness: Step (even b)

If b is even, then

$result' = result$ unchanged

$b' = b/2$

$a' = a^2$

```
if b%2 == 1: # b is odd
      result = result*a
      b = b-1
else:
      a = a*a
      b = b//2
```

So: $result' \cdot (a')^{b'} = result \cdot (a^2)^{b/2} = result \cdot (a)^b = a_0^{b_0}$

Substitute the values

Inductive assumption

- So in both cases, the loop invariant indeed holds after each execution of the loop body.

Proof of Correctness: Termination

- Termination: the loop must terminate, because b is reduced in each execution of the loop body by at least 1.
- We just proved this loop invariant holds: $result \cdot a^b = {a_0}^{b_0}$
- When the loop terminates, $b = 0$ (why?)

  So: $result \cdot a^b = result \cdot a^0 = result = a_0^{b_0}$

  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.
- 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 write is ~~indeed~~ hopefully correct.
- Finally, we remark that software and hardware verification are major issues in the corresponding industries (and academia). Elective courses on these topics are being offered at TAU (and elsewhere).