Numerical computations

Topic 6

Dr. Douglas Stebila

Department of Computing and Software
McMaster University

Spring/Summer 2020

Table of Contents

Table of Contents

Base 10 Decimal representation

2	0	5	3
10^3	10^2	10^1	10^0

$$2*10^3+0*10^2+5*10^1+3*10^0 = (((((2*10)+0)*10)+5)*10)+3 = 2053$$

Base 2 Decimal representation

1	1	0	1
2^3	2^2	2^1	2^0

$$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 8 + 4 + 1 (in base 10) = 13 (in base 10)$$

Question

What is the decimal (base 10) representation of the following binary (base 2) number?
011011

1. 11
2. 27
3. 54
4. 59
5. Don't know

Question

What is the decimal (base 10) representation of the following binary (base 2) number?
011011

1. 11
2. 27
3. 54
4. 59
5. Don't know

Converting from binary to decimal

Basic approach: Compute the sum of the relevant powers of 2.

$$b = b_{n-1} b_{n-2} b_{n-3} \ldots b_0$$

$$d = 2^{n-1} b_{n-1} + 2^{n-2} b_{n-2} + \cdots + 2^1 b_1 + 2^0 b_0$$

Slightly tricky in Python because string indexing starts at b[0] but we want to think of that as b[n-1].

Converting from binary to decimal

Compute the sum of the relevant powers of 2.

$$
\begin{aligned}
b &= b_0 b_1 b_2 \ldots b_{n-1} \\
d &= 2^{n-1} b_0 + 2^{n-2} b_1 + \cdots + 2^1 b_{n-2} + 2^0 b_{n-1} \\
&= \sum_{i=0}^{n-1} 2^{n-1-i} b_i
\end{aligned}
$$

```
def binaryToDec(b: "string of 0s and 1s"):
  d = 0
  for i in range(0, len(b)):
    if b[i] == "1":
      d += 2**(len(b)-1-i)
  return d
```

Converting from binary to decimal

Starting with the leftmost digit, the accumulator is multiplied by the base and the next digit is added (Horner's Rule)

```
def binaryToDec(b: "string of 0s and 1s"):
    d = 0
    for i in range(len(b)):
        d = 2*d + int(b[i])
    return d
```

Converting from decimal to binary

For converting a number into a series a digits, the remainder of the division with the base gives the least digit; this is repeated with the quotient, until the number fits into a single digit

2053 % 10 = 3

2053 // 10 = 205

205 % 10 = 5

205 // 10 = 20

20 % 10 = 0

20 // 10 = 2

2 % 10 = 2

2 // 10 = 0

⇒ stop

13 % 2 = 1

13 // 2 = 6

6 % 2 = 0

6 // 2 = 3

3 % 2 = 1

3 // 2 = 1

1 % 2 = 1

1 // 2 = 0

⇒ stop

13 decimal = 1101 binary

Question

What is the binary (base 2) representation of the following decimal (base 10) number?
41

1. 1001
2. 100101
3. 101001
4. 0101001
5. Don't know

Question

What is the binary (base 2) representation of the following decimal (base 10) number?
41

1. 1001
2. 100101
3. 101001
4. 0101001
5. Don't know

Converting from decimal to binary

```
def decToBinary(d):
  b = ""
  while d > 0:
    if d % 2 == 0:
      b = "0" + b
    else:
      b = "1" + b
    d = d // 2
  return b
```

Converting from decimal to binary

```
def decToBinary(d):
  b = ""
  while d > 0:
    b = str(d % 2) + b
    d = d // 2
  return b
```

Other bases

Octal

- Octal = base 8
- Digits from 0 up to 7
- Example:
- 135 base 8 = $1*8^2 + 3*8^1 + 5*8^0$
  = 93 base 10

Hexadecimal

- Hexadecimal = base 16
- Digits from 0 up to 15
  - Represent "digits" 10, 11, 12, 13, 14, 15 as A, B, C, D, E, F
- Example:
- 2A base 16 = $2*16^1 + 10*16^0$
  = 42

Table of Contents

Base 10 Decimal representation

2	0	.	5	3
10^1	10^0		2^{-1}	2^{-2}

$$2 * 10^1 + 0 * 10^0 + 5 * 10^{-1} + 3 * 10^{-2} = 20.53$$

Base 2 Binary representation

1	1	.	0	1
2^1	2^0		2^{-1}	2^{-2}

$$1*2^1+1*2^0+0*2^{-1}+1*2^{-2} = 2+0+0.25(inbase10) = 2.25(inbase10)$$

Representing rational numbers in binary

How can we represent 0.1?

.	0	0	1
	2^{-1}	2^{-2}	2^{-3}

$= 1/8 = .125$

.	0	0	0	1	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}

$= 1/16 + 1/32 = 3/32 = .09375$

.	0	0	0	1	1	0	0	1	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}

$= 51/512 = .099609375$

Representing rational numbers in binary

How can we represent 0.1?

.	0	0	0	1	1	0	0	1	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}

- Infinitely many binary digits are needed: after the initial digit 0, the digits 0011 keep repeating
- Depending on the base, certain rational numbers cannot be written with finitely many digits, e.g. 1/3 in decimal, 1/10 in binary

Question

Which of $3/4$ and $1/3$ have a finite binary representation?

1. Both
2. Only $3/4$
3. Only $1/3$
4. Neither
5. Don't know

Question

Which of $3/4$ and $1/3$ have a finite binary representation?

1. Both
2. Only $3/4$
3. Only $1/3$
4. Neither
5. Don't know

Real numbers in Python

$3/4 = .11$ base 2

finite binary representation

$1/3 = .01010101\ \ldots$ base 2

no finite binary representation

- Following the IEEE standard, Python uses 52 binary digits, approximately 16 decimal digits.
- Standard output gives only the first 16 decimal digits, even if the conversion from binary results in more digits. Only an approximate value is printed!

```
format(1/3, ".60f")
format(1/10, ".60f")
```

.60f: print floating point number with 60 digits after

Rounding Errors

With a finite number of digits, arithmetic operations will lead to rounding errors. Sometime the error gets cancelled, sometimes not

```
>>> 1/3+1/3+1/3, 1/6+1/6+1/6+1/6+1/6+1/6
>>> format(1/3+1/3+1/3, ".60f"), ...
```

As a consequence, fractional numbers should never be compared for equality:

```
a == b
```

should become

```
abs(b-a) <= epsilon
```

However, epsilon must not be too small!

Floating Point Numbers

A floating point number consists of a fraction (mantissa) with the significant digits and an exponent. For example, for decimal numbers:

$(1.963, 3) = 1.963 * 10^3 = 1963$

Following the IEEE standard, Python stores the fraction with 52 bits in normalized form (leading 1 before .) and the exponent with 11 bits (with range -1022 to 1023):

$(1.f) * 2^e$

The largest positive number is

$(1.11\ldots \text{ base } 2) * 2^{1023} = 1.7976931348623157 * 10^{308}$

The smallest positive number is

$1.0 * 2^{-1022} = 2.2250738585072014 * 10^{-308}$

Floating Point Numbers

Most computers follow the IEEE standard: floating-point computation will have the same results across computers. Several formats exist, with 8 bytes being widely used:

1 bit sign	11 bits exponent	52 bits fraction

- The number of bits of the fraction limits the precision.
- The number of bits in the exponent limits the range.
  Arithmetic operations on float may lead to a loss of significant digits:
-
```
>>> 10000000000000000+1
>>> 10000000000000000.0+1
```
- +, - on float can be risky operations!

Alternatives to floating point numbers

- What if we really want exact arithmetic?
- "Multi-precision arithmetic"
- Two approaches in Python:
  - Decimal
  - Fraction

Decimal Fixed Point Numbers

Arithmetic with the Python decimal library will produce the same errors as calculations by hand; by default, 28 fractional digits are kept; the precision can be changed:

```
from decimal import Decimal

Decimal(1)/Decimal(3)
Decimal(1)/Decimal(10)

Decimal('0.1')+Decimal('0.2')==Decimal('0.3')
```

Decimal Fixed Point Numbers

Why not use decimal all the time?
Because it is much slower than floating point operations.

- CPUs have floating point operations built in.
- But not arbitrary decimal precision.

```
a = 0.1
b = 57
%time a / b
```

```
from decimal import Decimal
a = Decimal("0.1")
b = Decimal(57)
%time a / b
```

%time is a "magic" function built in to Jupyter notebooks for timing

Rational Numbers

The Python library fractions stores rational numbers a/b with numerator a and denominator b as a pair (a, b). This makes calculations with +, -, *, / precise.

```
>>> from fractions import Fraction
>>> Fraction(1)/Fraction(3) == Fraction(1, 3)
```

Conversion between float, Decimal, Fraction reveals differences in representation:

```
>>> Decimal("0.1")
>>> Fraction(1)/Fraction(10)
>>> format(0.1, ".60f")
```

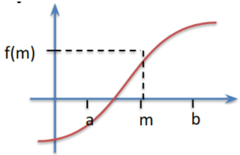
What to do about float

1. Avoid float, use integer instead: compute with rather $, mm rather than m
2. Use decimal or fractions standard library instead: slower, particularly for very large/small numbers
3. Use a library for interval arithmetic with float: gives safe lower and upper bounds, takes twice as much memory/time
4. Use a problem-specific library with or check literature for algorithms with known numerical properties (e.g. solving differential equations)
5. Use symbolic computation as in computer algebra systems instead of a programming language
6. When using float, never compare for equality; check result within a tolerance

Table of Contents

Formulating a problem

- What is the x-intersect of a function f?
- An x-intersect of function f is an x such that $f(x) = 0$.



Suppose f is monotonically increasing on [a, b] and: $f(a) \leq 0$, $f(b) \geq 0$
To determine the x-intersect with precision $e > 0$:

1. as long as $b - a > e$, do 2. – 4.
2. calculate $m = (a + b)/2$
3. if $f(m) \leq 0$, set a to m, or
4. if $f(m) \geq 0$, set b to m

The result is (a, b) such that $f(a) \leq 0$, $f(b) \geq 0$, $b - a \leq e$

Computing the x-intersect

Input:

- a range [a, b]
- a function f that is monotonically increasing on [a, b] such that $f(a) \leq 0$ and $f(b) \geq 0$
- a precision $e > 0$:

Instructions:

1. As long as $b - a >$, do steps 2–4
2. Calculate $m = (a + b)/2$
3. If $f(m) \leq 0$, set a to m
4. Otherwise, if $f(m) \geq 0$, set b to m

Output:

- Values (a, b) such that $f(a) \leq 0$, $f(b) \geq 0$, and $b - a \leq e$

Question: can we compute the exact x-intersect by taking $e = 0$?

1. Yes
2. No

No. If a and b are rational and the x-intersect is irrational, then it will never be reached.

Approximate x-intersect

- In Python, functions can be passed as arguments
- So we'll create a generic x_intersect function to which we can pass any function f

```
def x_intersect(f, a, b, eps):
  while b-a > eps:
    m = (a+b)/2
    if f(m) <= 0:
      a = m
    else:
      b = m
  return a, b
```

Approximate x-intersect

```
def f1(x):
    return x*x-4

def f2(x):
    return x*x-2

type(f1)

x_intersect(f1, 0, 100, 1e-8)
x_intersect(f2, 0, 100, 1e-8)

x_intersect(f1, 0, 100, 1e-15)
x_intersect(f1, 0, 100, 1e-16)

x_intersect(lambda x: x**3-17, 0, 100, 1e-8)
```

Approximate square root

Integer a is an approximate square root of n if

$$a^2 \leq n < (a+1)^2$$

One way to compute the square root is by linear search (exhaustive search):

```
def linearSqrt(n):
    a = 0
    while (a+1)*(a+1) <= n:
        a = a+1
    return a
```

Approximate square root

```
def linearSqrt(n):
    a = 0
    while (a+1)*(a+1) <= n:
        a = a+1
    return a
```

Trace for input n = 27:

```
>>> linearSqrt(27)
5
```

Statement	a
A	0
B	1
B	2
B	3
B	4
B	5

Approximate square root

```
>>> linearSqrt(27)
5
```

Question: For $n \geq 0$, how often is B executed?

1. n
2. n + 1
3. $\sqrt{n}$
4. n^2
5. $(n+1)^2$

Approximate square root

```
>>> linearSqrt(27)
5
```

B is executed exactly as many times as whatever the output of the function is.
linearSqrt(4) $= 2 \rightarrow 2 times$
linearSqrt(8) $= 2 \rightarrow 2 times$
linearSqrt(9) $= 3 \rightarrow 3 times$
linearSqrt(10) $= 3 \rightarrow 3 times$
Hence, A is executed $\sqrt{n}$ times ($\sqrt{}$ here means integer square root)

Binary Search of Integer Square Root

Suppose we have a, b such that the root is between a and b:

$$0 \leq a < b \quad and \quad a^2 \leq n < b^2$$

We repeatedly replace either a or b by (a+b)//2 such that above holds, until a+1 == b; then a is the approximate square root

```
while a+1 != b:
    c = (a+b)//2          # C
    if c*c <= n: a = c    # D
    else: b = c           # E
```

How do we find suitable initial values for a and b?

Trace for n = 27, a = 0, b = 8:

Statement	a	b	c
C	0	8	4
D	4	8	4
D	4	8	6
E	4	6	6
C	4	6	5
D	5	6	5

Binary Search of Integer Square Root

For a we take 0. For b, the smallest value satisfying $0 \leq a < b$ is 1, which we multiply by 2 until b satisfies $a^2 \leq n < b2$

```
def binarySqrt(n):
  a = 0
  b = 1
  while b*b <= n:
    b = 2*b
  while a+1 != b:
    c = (a+b)//2
    if c*c <= n: a = c
    else: b = c
  return a
```

How often are the loop bodies executed?

Binary Search of Integer Square Root

The first loop sets b $= 2^k > \sqrt{n}$ after k executions. The second loop halves the interval b - a at every execution, until a+1 = b, hence also takes k executions.

```
def binarySqrt(n):
  a = 0
  b = 1                  # A
  while b*b <= n:
    b = 2*b              # B
  while a+1 != b:
    c = (a+b)//2         # C
    if c*c <= n: a = c   # D
    else: b = c          # E
  return a
```

Trace for n = 27:

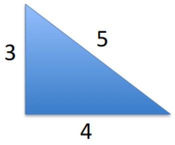
Statement	a	b	c
A	0	1	
B	0	2	
B	0	4	
B	0	8	
C	0	8	4
...	...	...	...

Hence each loop takes k $= \log_2 b$ executions, so k $\approx \log_2 \sqrt{n}$

Pythagorean Triples

A triple(a, b, c) of integers is Pythagorean if $a^2 + b^2 = c^2$

A simple way to find such triples is by brute force: enumerate all values of a, b, c and check if they form a Pythagorean Triple




```
def printPythagoreanTriples1(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      for c in range(1, n+1):
        if a**2+b**2 == c**2:
          print(a, b, c)
```

Pythagorean Triples

Question: For n≥0, how often is A executed?

1. 3n
2. 3(n+1)
3. $3n^2$
4. n^3
5. $(n+1)^3$

```
def printPythagoreanTriples1(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      for c in range(1, n+1):
        if a**2+b**2 == c**2:    # A
          print(a, b, c)
```

Pythagorean Triples

Question: For n≥0, how often is A executed?

1. 3n
2. 3(n+1)
3. $3n^2$
4. n^3
5. $(n+1)^3$

Each of a, b, c take n different values, in all combinations, so A is executed on n*n*n = n^3 combinations in total

```
def printPythagoreanTriples1(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      for c in range(1, n+1):
        if a**2+b**2 == c**2:    # A
          print(a, b, c)
```

Pythagorean Triples, improved

Rather than going over all values of c, we calculate c from a, b and check if it is an integer

```
def printPythagoreanTriples2(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:  # A
        print(a, b, c)
```

Pythagorean Triples, improved

Question: For $n \geq 0$, how often is A executed?

1. $2n$
2. $2(n+1)$
3. n^2
4. $3n^2$
5. $(n+1)^2$

```
def printPythagoreanTriples2(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:   # A
        print(a, b, c)
```

Pythagorean Triples, improved

Question: For n≥0, how often is A executed?

1. 2n
2. 2(n+1)
3. n^2
4. $3n^2$
5. $(n+1)^2$

Both a and b take n different values in all combinations, so A is executed on n*n = n^2 combinations in total

```
def printPythagoreanTriples2(n):
  for a in range(1, n+1):
    for b in range(1, n+1):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:   # A
        print(a, b, c)
```

Pythagorean Triples, improved

Both (3, 4, 5) and (4, 3, 5) are printed, which is unnecessary. We can restrict a, b, c such that $0 < a \leq b < c \leq n$

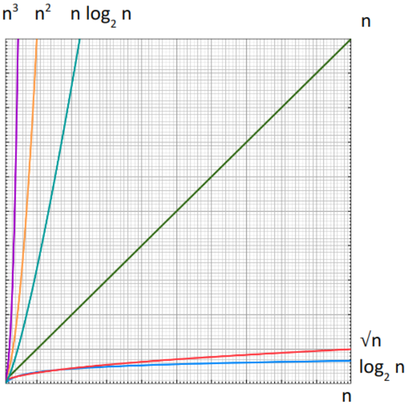
```
def printPythagoreanTriples(n):
  for a in range(1, n):
    for b in range(a, n):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        print(a, b, c)
```

Pythagorean Triples, improved

If a = 1, then b takes n-1 values, if a = 2, then n-2 values, etc. In total $(n-1)+(n-2)+\ldots+1 = n(n-1)/2 = n^2/2-n/2$ Compared to the previous version, A is executed only half as often.

```
def printPythagoreanTriples(n):
  for a in range(1, n):
    for b in range(a, n):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        print(a, b, c)
```

Execution Time of Programs



If we know how many steps a program takes depending on the input, we can use this to predict the execution time

This can be done without knowing details of the processor and compilation to machine language!

Predicting execution time v1

```
def countPythagoreanTriples1(n):
  k = 0
  for a in range(1, n+1):
    for b in range(1, n+1):
      for c in range(1, n+1):
        if a**2+b**2 == c**2:  # A
          k += 1  # Count them rather than print
                  # them for more reliable timing
                  # measurements
  return k
```

Question: Suppose it takes t sec for n = 200. How many seconds should it take for n = 400?

1. 2t
2. 4t
3. 8t
4. t^2
5. t^3

Predicting execution time v1

```
def countPythagoreanTriples1(n):
  k = 0
  for a in range(1, n+1):
    for b in range(1, n+1):
      for c in range(1, n+1):
        if a**2+b**2 == c**2:  # A
          k += 1  # Count them rather than print
                  # them for more reliable timing
                  # measurements
  return k
```

Each of a, b, c take n different values, in all combinations, so A is executed on $n*n*n = n^3$ combinations in total

For $n = 200$: 200^3 executions of A

For $n = 400$: 400^3 executions of A

$(400^3/200^3) = (400/200)^3 = 2^3 = 8$

For $n = 400$: 8 times longer, 8 t sec

Predicting execution time v2

```
def countPythagoreanTriples2(n):
  k = 0
  for a in range(1, n+1):
    for b in range(1, n+1):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        k += 1
  return k
```

Question: Suppose it takes t sec for n = 200. How many seconds should it take for n = 400?

1. 2t
2. 4t
3. 8t
4. t^2
5. t^3

Predicting execution time v2

```
def countPythagoreanTriples2(n):
  k = 0
  for a in range(1, n+1):
    for b in range(1, n+1):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        k += 1
  return k
```

Both a and b take n different values in all combinations, so A is executed on n*n = n^2 combinations in total

For n = 200: 200^2 executions of A

For n = 400: 400^2 executions of A

For n = 400: 4 times longer, 4 t sec (ignoring runtime of binarySqrt)

Predicting execution time v3

```
def countPythagoreanTriples(n):
  k = 0
  for a in range(1, n):
    for b in range(a, n):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        k += 1
  return k
```

Question: Suppose it takes t sec for n = 200. How many seconds should it take for n = 400?

1. 2t
2. 4t
3. 8t
4. t^2
5. t^3

Predicting execution time v3

```
def countPythagoreanTriples(n):
  k = 0
  for a in range(1, n):
    for b in range(a, n):
      c2 = a*a+b*b
      c = binarySqrt(c2)
      if c <= n and c2 == c*c:    # A
        k += 1
  return k
```

If a = 1, then b takes n-1 values, if a = 2, then n-2 values, etc. In total (n-1)+(n-2)+...+1 = n(n-1)/2 = n^2/2-n/2

For large n, n/2 is negligible, so we just focus on the difference from the squared term: $((800^2/2)/(400^2/2)) = 4$

For n = 800: 4 times longer, 4 t sec, when ignoring binarySqrt

Table of Contents

Solutions of $ax^2+bx+c = 0$

$$x_{1/2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

```
import math
def quadraticEquationSolution(a, b, c):
    d = math.sqrt(b*b-4*a*c)
    return (-b+d)/(2*a), (-b-d)/(2*a)
```

```
>>> quadraticEquationSolution(1, -3, -4)
(4.0, -1.0)
>>> quadraticEquationSolution(1, -2e8, 1)
(200000000.0, 0.0)

# 0 is not a solution of
# x^2 - 200000000x + 1 = 0
```

Solutions of $ax^2+bx+c = 0$

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
import math
def quadraticEquationSolution(a, b, c):
    d = math.sqrt(b*b-4*a*c)
    return (-b+d)/(2*a), (-b-d)/(2*a)
>>> quadraticEquationSolution(1, -2e8, 1)
(200000000.0, 0.0)
```

```
b*b-4*a*c
= (-2e8)*(-2e8)
 -4*1*1
= 4e16-4
= 4e16
# Loss of significant digits
```

```
# Therefore,
d = sqrt(4e16) = 2e8
# and:
(-b+d)/(2*a)
= (-(-2e8)+2e8)/(2*1)
= 2e8
# and:
(-b-d)/(2*a)
= (-(-2e8)-2e8)/(2*1)
= 0
```

Solutions of $ax^2+bx+c = 0$

How can we avoid such errors?

- Use decimal for exact precision $\rightarrow$ Slower
- Use an alternative formula that is more numerically stable and doesn't lead to large intermediate values

Solutions of $ax^2+bx+c = 0$

Solutions of the quadratic equation are related by Vieta's formula:

$$x_1 x_2 = \frac{c}{a}$$

Given the solution with larger absolute value, the smaller can be computed using Vieta, avoiding the loss of significant digits

```
def quadraticEquationSolutionPlus(a, b, c):
    d = math.sqrt(b*b-4*a*c)
    x1 = -(b+d)/(2*a) if b>=0 else (d-b)/(2*a)
    x2 = c/(x1*a)
    return x1, x2

quadraticEquationSolutionPlus(1, -2e8, 1)
```

http://en.wikipedia.org/wiki/Quadratic_equation#Vieta.27s_formulas