CS 5 Nightly Wrapup

College Canceled

Claremont (The Student Life): The administrators of Harvey Mudd College announced today that the entire institution had been canceled. Classes will terminate immediately.

"We realized that there is a much better economic model," explained President G. Reedy. We will continue to accept students, and the tuition will remain the same. After four years of paying tuition, the students will be awarded a degree, just as in previous years. The only difference will be that we won't hold classes. That will give the students more time for the pursuits they love, like video gaming, dancing, partying, and setting things on fire, without harming their chances of getting a lucrative job after they get their degree."

When asked what the faculty would be doing, President Reedy smiled. "That's the best part!" he exclaimed. "We'll finally be rid of the pesky critters."

No penguins could be reached for comment.

Reminders of Countability

We know that programs are countable...

…and even simple functions are uncountable…

…so there must be more functions than programs…

…and therefore there are functions that can’t be computed!

Reminders of Countability

We know that programs are countable...

…and even simple functions are uncountable…

What Can’t’ Be Computed?

But are all the uncomputable functions as boring as *f(N)* = *x*?

Show me something interesting!

Measuring the “Complexity” of Data

Andrei Kolmogorov
1903-1987

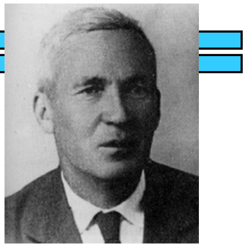
10^{5000}

versus

156234103423479583941807 45…2123975

5001 digits long

Measuring the “Complexity” of Data



10^{5000} = 1000000000000000000000…0000000000000

5001 digits long

Program takes no arguments!

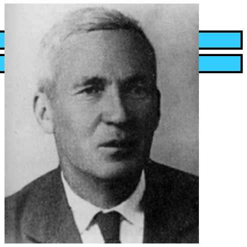
```
def a():
    return 1000000000000000000000…000000000000
```

Program returns desired number and halts!

Total length: 5017

I sorta think we can do ***much*** better!

Measuring the “Complexity” of Data



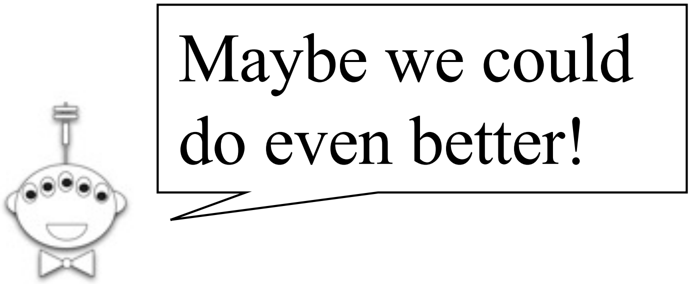
10^{5000}

Program takes no arguments!

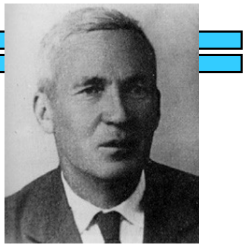
```
def a():
    result = "1"
    for d in range(0, 5000):
        result += "0"
    return int(result)
```

Program returns desired number and halts!

Total length: 100



Measuring the “Complexity” of Data



$$\underbrace{\text{15623410342347958394180745…2123975}}_{\text{5001 digits long}}$$

Program takes no arguments!

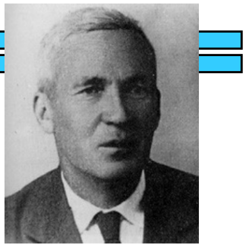
```
def a():
    return 156234103423474 5…2123975
```

Program returns desired number and halts!

I sorta doubt we can do much better!

Total length: 5017

What is the Complexity Of…?



```
def f():return ...
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ←

Python has at least 15 bytes of "overhead"

kc(1000000000) = 20 = 15 + 5 ⇨ 10**9

(1 followed by 9 0's)

kc(100…000) = 10**100 -> 22 This is called a *googol*

(1 followed by 100 0's)

kc(999…999) = int('9'*100) => 23.

(100 9's)

kc(100…000) = 10**10**100 26 This is a *googolplex*

(1 followed by a googol 0's)

kc(1010…) = 26

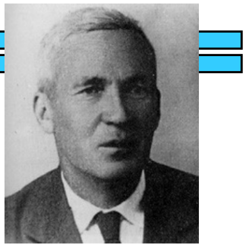
(10 a billion times…try using a string)

kc(314159265…) =

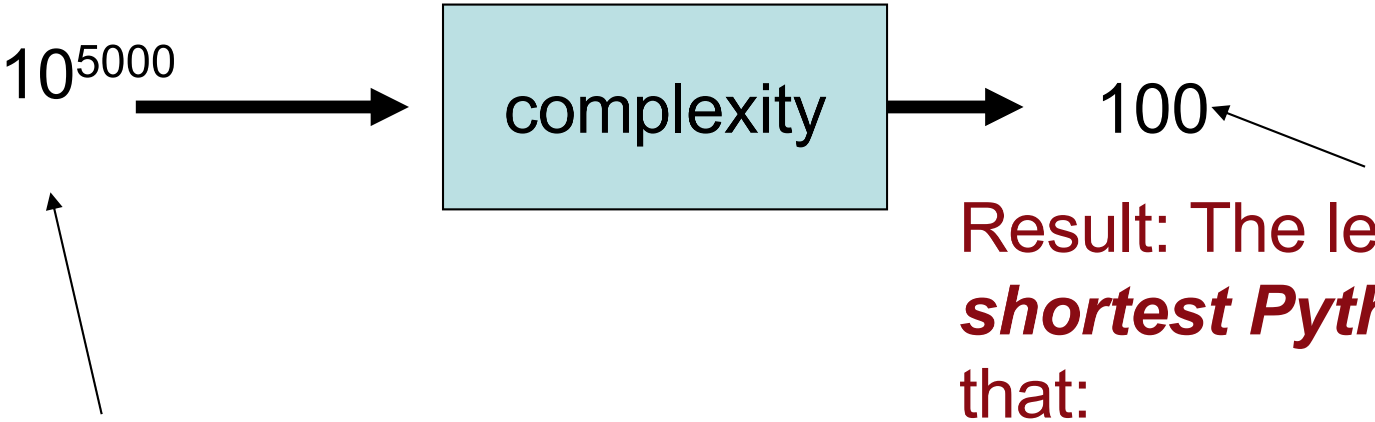
(2 billion digits of pi)

Worksheet!

Measuring the “Complexity” of Data



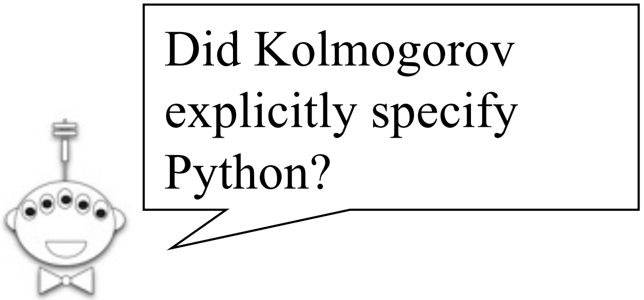
Objective…



Result: The length of the ***shortest Python program*** that:

- -Takes no arguments
- -Runs
- -Returns the integer n

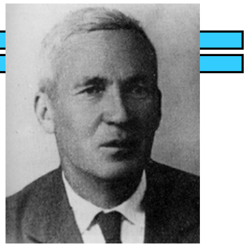
Argument: An integer n



length 100

```
def a():
    result = "1"
    for d in range(5000):
        result += "0"
    return int(result)
```

Measuring the “Complexity” of Data

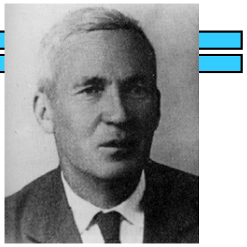


We will show that **`complexity`** is uncomputable

Specifically, we will show that *any* implementation of **`complexity`** must *necessarily* contain a bug:

There is at least one number for which it will return the wrong answer!

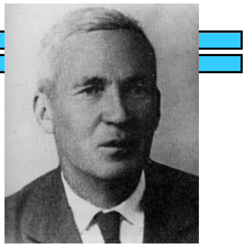
Measuring the “Complexity” of Data



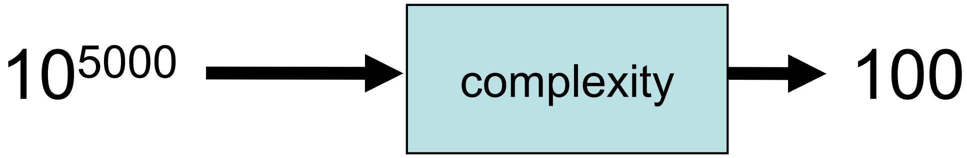
Our key insight:

For any value *k,* there is a number *n* whose complexity is greater than *k* (why?)

Measuring the “Complexity” of Data



By Way of Contradiction (“BWOC”), assume we have a “Complexity” function…



```
def Complexity(number):
   # code goes here
   return complexity
```

```
def BFF():
    def     plexity(number):
       #     goes here
       retu     lculated_complexity

    c
    w                               = 50000 + 200:

    return counter
```

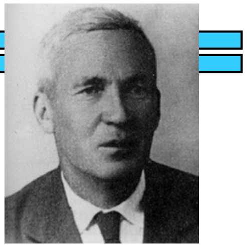
Assume the length of this code is 50000

Bug Finding Function!

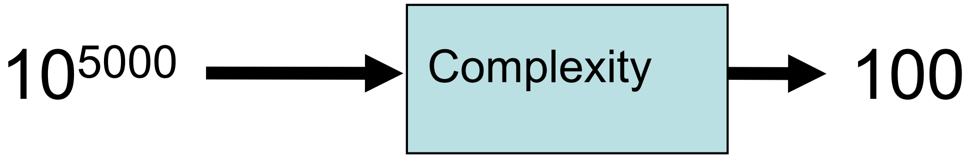
Notice that BFF takes no arguments, returns a number, and halts!

Look at the value returned by BFF. What can you say about this value?

Measuring the "Complexity" of Data



By Way of Contradiction ("BWOC"), assume we have a "Complexity" function…



10^{5000}

Complexity

100

```
def Complexity(number):
   # code goes here
   return complexity
```

```
def BFF():
   def complexity(number):
      # code goes here
      return calculated_complexity

   counter = 0
   while complexity(counter) <= 50000 + 200:
      counter = counter + 1
   return counter
```

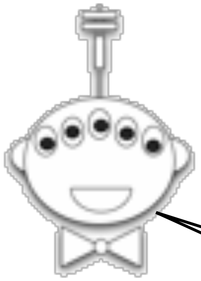
Assume the length of this code is 50000

Notice that BFF takes no arguments, returns a number, and halts!

Look at the value returned by BFF. What can you say about this value?

The Alien’s Life Advice

Chew with your mouth closed.

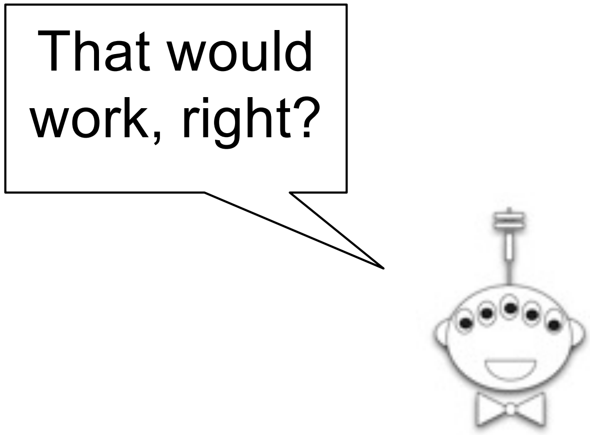


Your parents were right.

Here’s a Way to Do Complexity

How about this?

1. There are countably many programs
2. Order them from shortest to longest
3. Check each in order to see if it returns *n*



The one that we find first is the shortest that can return *n*!

Here’s a Way to Do Complexity

How about this?

```
x = 0
while True:
    x = x + 1
```

1. There are countably many p
2. Order them from shortest to longest
3. Check each in order to see if it returns *x*

The one that we find first is at can return x!

Can’t be done!

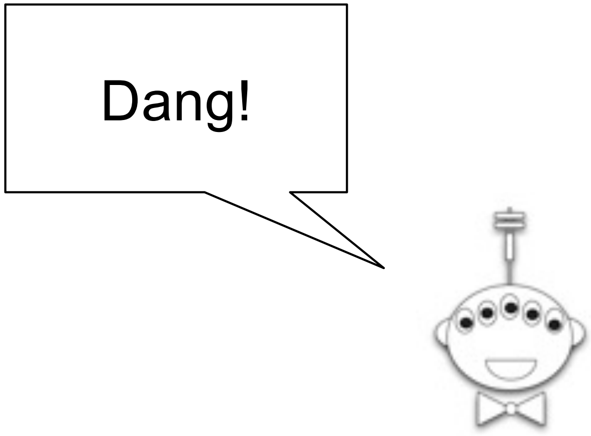
Halt Checking Is Uncomputable

The code for a Python function

```
def hc(f):
    # Clever stuff here
```

It is *impossible* to write a bug-free function **`hc(f)`** that decides whether **`f`** halts, i.e.,

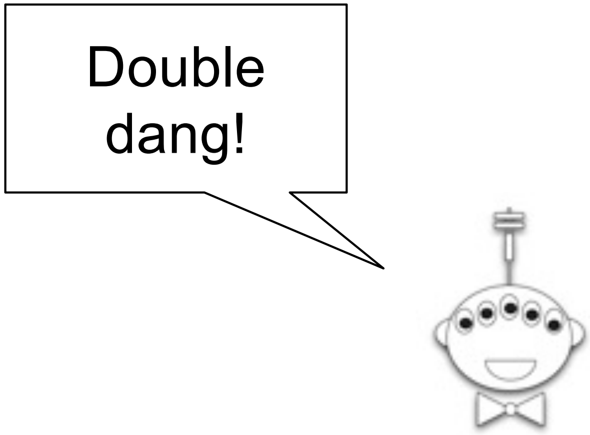
1. Returns True if **`f()`** halts, or
2. Returns False if **`f()`** loops forever



Halt Checking Is Uncomputable

Suppose **`hc(f)`** works for all zero-argument functions **`f`**. Write this zero-argument BFF:

```
def BFF():
    if hc(BFF):
        while True:
            print('Ha!’)
    else:
        return 42
```

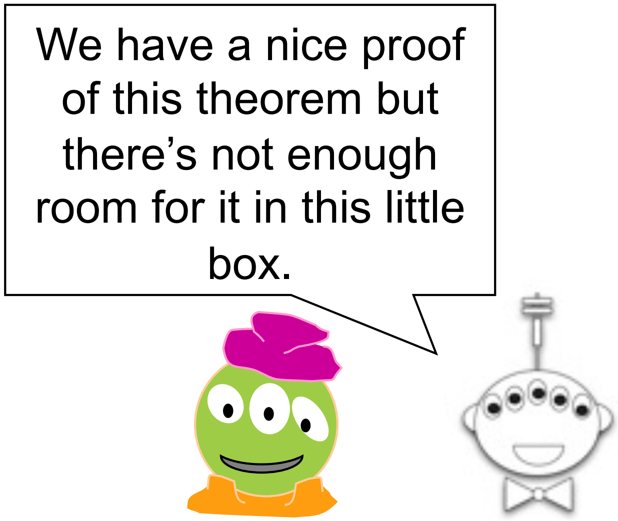


Should **`hc(BFF)`** return True or False?

The Halting Problem and Famous Open Problems

Fermat’s Last Theorem: There exists no integer $n > 2$ s.t. $a^n + b^n = c^n$ for non-zero integers a, b, and c

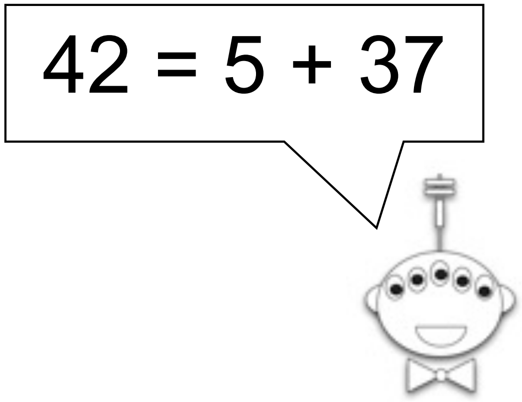
Pierre de Fermat
1601-1665


We have a nice proof of this theorem but there’s not enough room for it in this little box.

The Halting Problem and Famous Open Problems

Goldbach’s Conjecture: Every positive even integer >= 4 can be written as the sum of two primes.

$4 = 2 + 2$
$6 = 3 + 3$
$8 = 3 + 5$
$10 = 3 + 7 = 5 + 5$
$12 = 5 + 7$
$14 = 3 + 11 = 7 + 7$



Verified up to 4 x 10^{18}

The Halting Problem and Famous Open Problems

Goldbach's Conjecture: Every positive even integer >= 4 can be written as the sum of two primes.

$1,000,000 has been offered!

The Halting Problem and Famous Open Problems

Goldbach's Conjecture: Every positive even integer >= 4 can be written as the sum of ~~two~~ primes.

at most 300,000

(Schnilerman, 1939)

Getting from 300,000 down to 2 shouldn' t be so hard!

Using a Halt Checker to Prove or Disprove the Goldbach Conjecture…

```
def prime_split(n):
    """Takes an EVEN POSITIVE integer argument
    n and returns True if n can be
    written as the sum of two primes and
    False otherwise."""

def goldbach(current):
    while True:
        if not prime_split(current):
            return  # DONE!
        else current = current + 2
```

Consider… `goldbach(4)`

Yowza this is cool!

Who needs chocolate when there are proofs this sweet?

Halt Checking in the Real World

The impossibility of halt checking implies you can't write a program that will understand other programs

So don't waste your time trying!

But wait!
You can write one that understands *some* other programs…

Regular Expressions

10

(10)*

1* | 10*

Examples of three regular expressions and overall "regex" syntax.

A ***regular expression*** is composed of three operations:

• *Kleene* Star	a*	“0 or more a’s”	high precedence
• Concatenation	ab	“a *then* b”	
• Union	a | b	“a *or* b”	low precedence

where a and b can be any character strings—or regular expressions

base case

recursively defined !

Regular Expressions

10

Matches the string 10, which is the language { 10 }
…or L = { w | w is 10 }

(10)*

What strings are in the other two REs' languages?

1* | 10*

A ***regular expression*** is composed of three operations:

• *Kleene* Star	a*	"0 or more a's"
• Concatenation	ab	"a *then* b"
• Union	a | b	"a *or* b"

high precedence

low precedence

where a and b can be any character strings—or regular expressions

base case

recursively defined !

Regular Expressions

Here is a fairly complex regular expression.

What strings are in (and out of) this language?

(01* | 10)*

A ***regular expression*** is composed of three operations:

Kleene Star	a*	"0 or more a's"
Concatenation	ab	"a *then* b"
Union	a | b	"a *or* b"

high precedence

low precedence

where a and b can be any bit strings—or regular expressions

base case

recursively defined !

Try It!

Operator Name	Example	Description
• *Kleene* Star	a*	"0 or more a's"
• Concatenation	ab	"a *then* b"
• Union	a | b	"a *or* b"

Description of a formal language	**Equivalent RE**
L = { w | w contains at least one 0 }	1*0(0|1)*
L = { w | w's second-to-last character is a 1 }	(0|1)*1(0|1)
L = { w | every 1 in w has a 0 after it }	0*(100*)* *or* (0|10)*
L = { w | w's first and last bits are the same }	(1(0|1)*1)|(0(0|1)*0)

How could you implement other operators?

one or more **as**	a+
strings ***not*** matching **11**	**~(11)**
strings ***not*** matching **a**	~a

Try writing these REs in terms of the original three…

Is there an equivalent RE to this one ***that avoids the nested * operators***?

(01* | 10)*

Extra: can *every* RE avoid nested *'s ?

Try It!

Operator Name	Example	Description
• *Kleene* Star	a*	"0 or more a's"
• Concatenation	ab	"a *then* b"
• Union	a | b	"a *or* b"

Description of a formal language	**Equivalent RE**
L = { w | w contains at least one 0 }	1*0(0|1)*
L = { w | w's second-to-last character is a 1 }	
L = { w | every 1 in w has a 0 after it }	
L = { w | w's first and last bits are the same }	

How could you implement other operators?

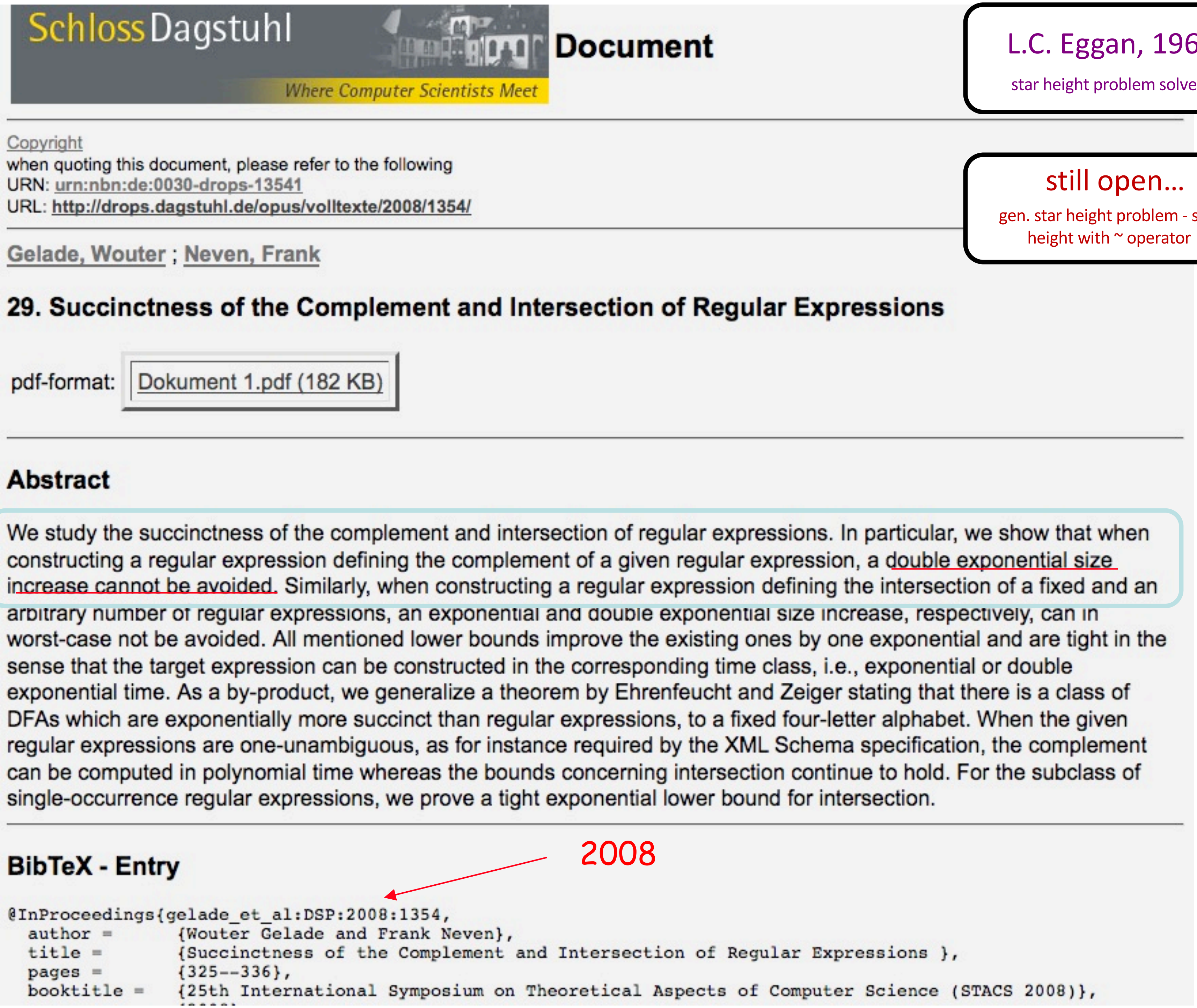
one or more **as**	a+
strings ***not*** matching **11**	**~(11)**
strings ***not*** matching **a**	~a

Try writing these REs in terms of the original three…

Is there an equivalent RE to this one ***that avoids the nested * operators***?

(01* | 10)*

Extra: can *every* RE avoid nested *'s ?



Document

Copyright
when quoting this document, please refer to the following
URN: urn:nbn:de:0030-drops-13541
URL: http://drops.dagstuhl.de/opus/volltexte/2008/1354/

Gelade, Wouter ; Neven, Frank

29. Succinctness of the Complement and Intersection of Regular Expressions

pdf-format: Dokument 1.pdf (182 KB)

Abstract

We study the succinctness of the complement and intersection of regular expressions. In particular, we show that when constructing a regular expression defining the complement of a given regular expression, a double exponential size increase cannot be avoided. Similarly, when constructing a regular expression defining the intersection of a fixed and an arbitrary number of regular expressions, an exponential and double exponential size increase, respectively, can in worst-case not be avoided. All mentioned lower bounds improve the existing ones by one exponential and are tight in the sense that the target expression can be constructed in the corresponding time class, i.e., exponential or double exponential time. As a by-product, we generalize a theorem by Ehrenfeucht and Zeiger stating that there is a class of DFAs which are exponentially more succinct than regular expressions, to a fixed four-letter alphabet. When the given regular expressions are one-unambiguous, as for instance required by the XML Schema specification, the complement can be computed in polynomial time whereas the bounds concerning intersection continue to hold. For the subclass of single-occurrence regular expressions, we prove a tight exponential lower bound for intersection.

BibTeX - Entry

```
@InProceedings{gelade_et_al:DSP:2008:1354,
  author =      {Wouter Gelade and Frank Neven},
  title =       {Succinctness of the Complement and Intersection of Regular Expressions },
  pages =       {325--336},
  booktitle =   {25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)},
```

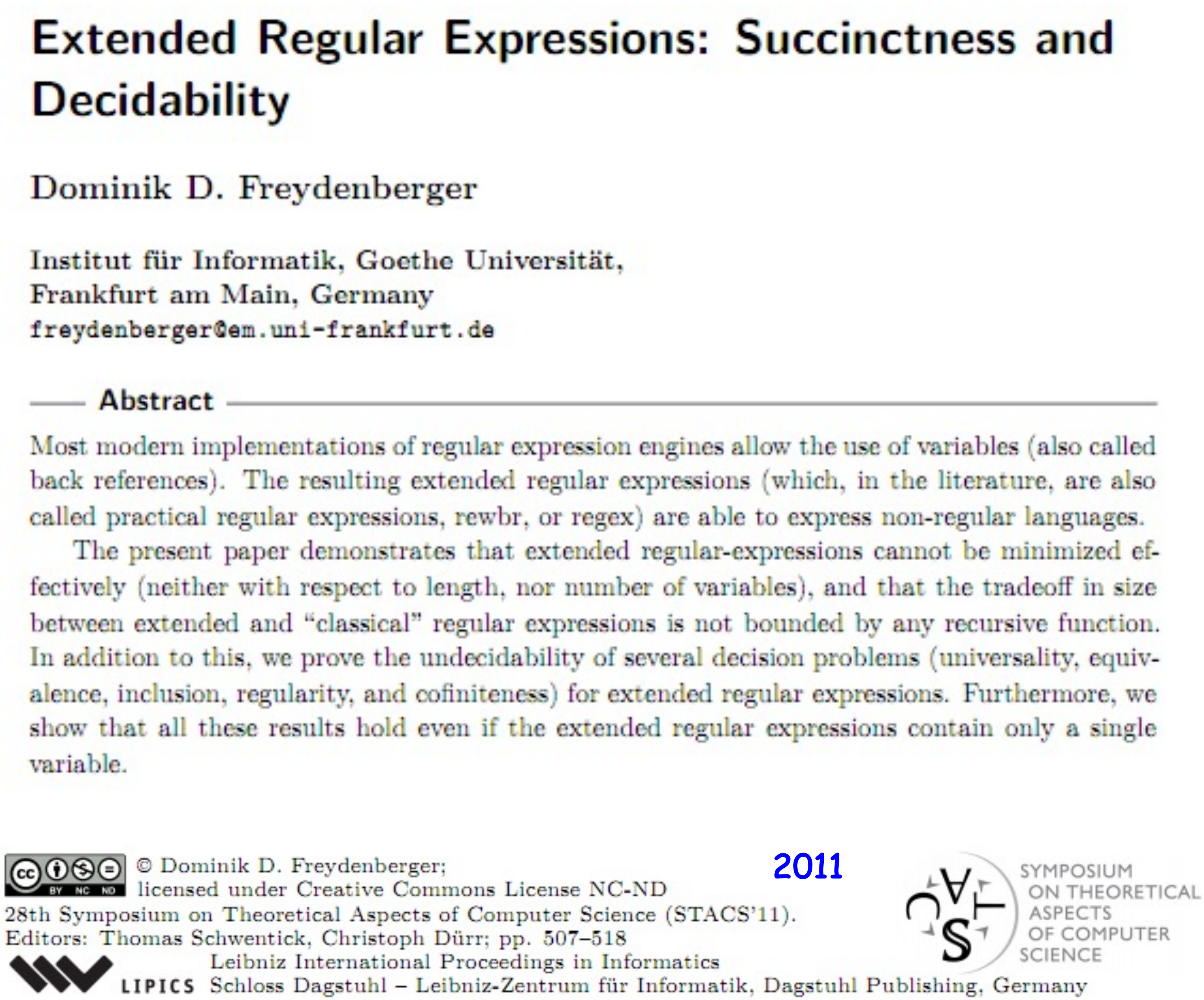
L.C. Eggan, 1963

star height problem solved

still open…

gen. star height problem - star height with ~ operator

2008



Extended Regular Expressions: Succinctness and Decidability

Dominik D. Freydenberger

Institut für Informatik, Goethe Universität,
Frankfurt am Main, Germany
freydenberger@em.uni-frankfurt.de

Abstract

Most modern implementations of regular expression engines allow the use of variables (also called back references). The resulting extended regular expressions (which, in the literature, are also called practical regular expressions, rewbr, or regex) are able to express non-regular languages.

The present paper demonstrates that extended regular-expressions cannot be minimized effectively (neither with respect to length, nor number of variables), and that the tradeoff in size between extended and "classical" regular expressions is not bounded by any recursive function. In addition to this, we prove the undecidability of several decision problems (universality, equivalence, inclusion, regularity, and cofiniteness) for extended regular expressions. Furthermore, we show that all these results hold even if the extended regular expressions contain only a single variable.

2011

28th Symposium on Theoretical Aspects of Computer Science (STACS'11).
Editors: Thomas Schwentick, Christoph Dürr; pp. 507–518
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

SYMPOSIUM
ON THEORETICAL
ASPECTS
OF COMPUTER
SCIENCE

REs in Practice

Almost all languages have an RE library…

Unix's **egrep** does a line-by-line search for a regex:

```
egrep    'hh'
egrep    'y.*y'
egrep    '(xq|hq)'        /usr/share/dict/words
egrep    '^y.*y$'
```

symbol for ***start of a line***

symbol for any character—a shortcut for **(a|b|c|…|z|0|1|…|9|…)**

symbol for ***end of a line***

good for crosswords !

	vowel		**U**		**vowel**	

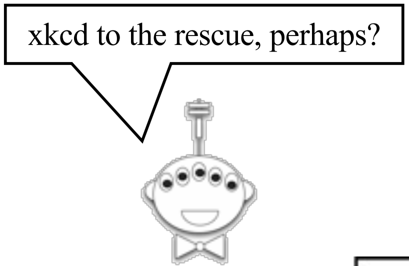
with first and last the same?

not always obvious …

```
egrep '^(0|1(01*0)*1)(0|1(01*0)*1)*$' binStr
```

knuth: ~cs60/egrep

egrep –f regexFile matchingStringFile

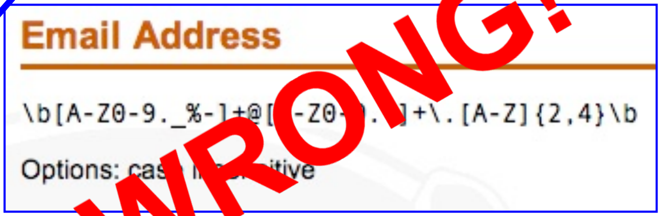


REs to the Rescue!

PERL

practical extraction and report language

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP



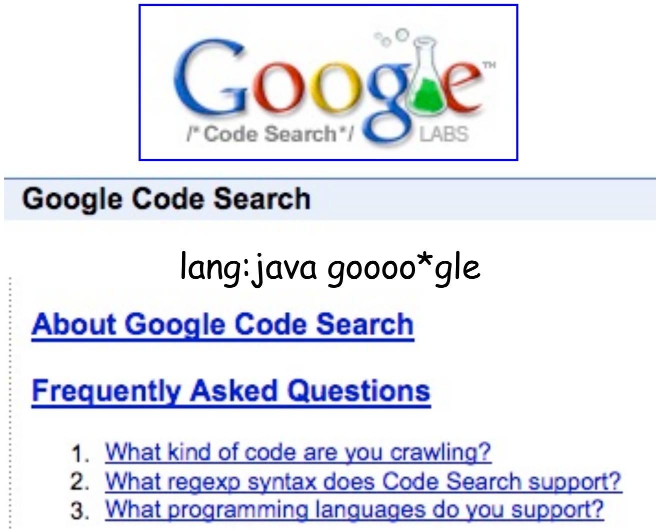
Email Address

```
\b[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

Options: case insensitive

WRONG!

www.regular-expressions.info/regexbuddy/email.html



Google Code Search

lang:java goooo*gle

About Google Code Search

Frequently Asked Questions

1. What kind of code are you crawling?
2. What regexp syntax does Code Search support?
3. What programming languages do you support?

But how does regular expression matching actually ***work...*** ?