an introduction to Deductive Program Verification

Jean-Christophe Filliâtre
CNRS

Mathematical Summer in Paris
July 16, 2018

https://msp.math.ens.fr/

Software is hard. – Don Knuth

why?

- wrong interpretation of specifications
- coding in a hurry
- incompatible changes
- software = complex artifact
- etc.

a famous example: *binary search*

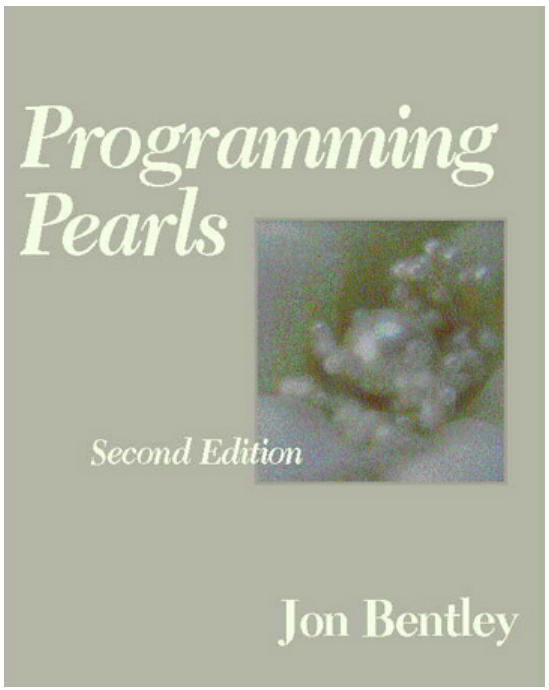
given a sorted array of integer, e.g.

-7	-1	2	2	42	987	1729

decide whether a given integer belongs to it

a famous example: *binary search*

first publication in 1946
first publication without bug in 1962



Jon Bentley. Programming Pearls. 1986.

Writing correct programs

the challenge of binary search

and yet...

in 2006, a bug was found in Java standard library's *binary search*

> Joshua Bloch, Google Research Blog
> *"Nearly All Binary Searches and Mergesorts are Broken"*

it had been there for 9 years

```
...
int mid = (low + high) / 2;
int midVal = a[mid];
...
```

may exceed the capacity of type `int`
then provokes an access out of array bounds

a possible fix

```
int mid = low + (high - low) / 2;
```

better programming languages

- better syntax
  (e.g. avoid considering `DO 17 I = 1. 10` as an assignment)
- more typing
  (e.g. avoid confusion between meters and yards)
- more warnings from the compiler
  (e.g. do not forget some cases)
- etc.

systematic and rigorous test is another, complementary answer

but test is

- costly
- sometimes difficult to perform
- and incomplete (except in some rare cases)

formal methods propose a mathematical approach to software correctness

there are several aspects

- what we compute
- how we compute it
- why it is correct to compute it this way

the code is only one aspect ("how") and nothing else

"what" and "why" are not part of the code

there are informal requirements, comments, web pages, drawings, research articles, etc.

- how: 2 lines of C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- how: 2 lines of C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- what: 15,000 decimals of π

- why: lot of maths, including

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2\, 2^{i+1}}{(2i+1)!}$$

formal methods propose a rigorous approach to programming, where we manipulate

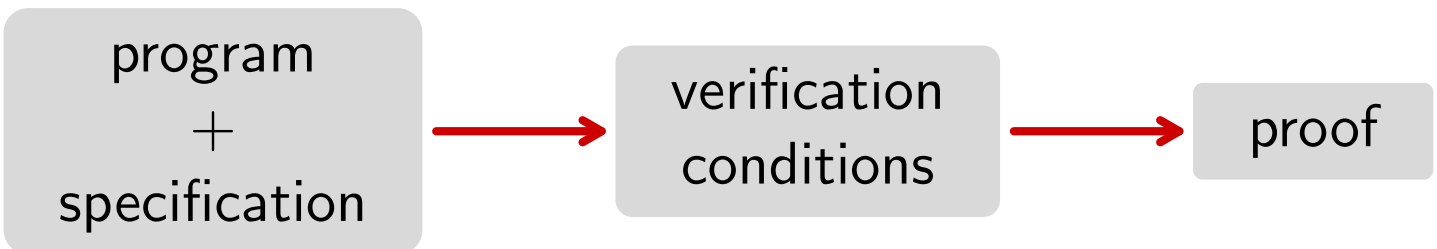
- a specification written in some mathematical language
- a proof that the program satisfies this specification

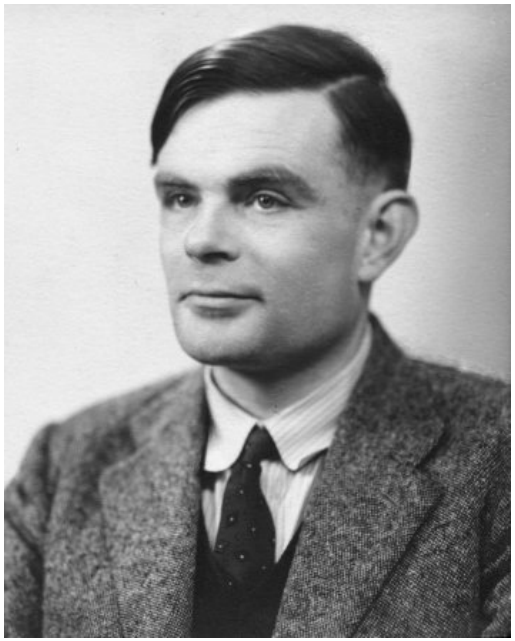
what do we intend to prove?

- safety: the program does not crash
  - no illegal access to memory
  - no illegal operation, such as division by zero
  - termination
- functional correctness
  - the program does what it is supposed to do

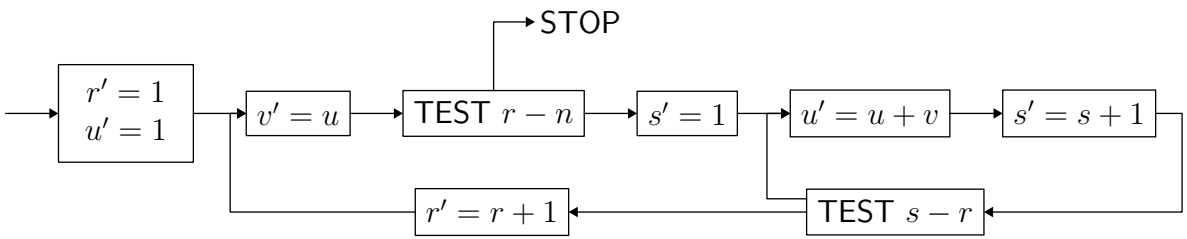
model checking, abstract interpretation, etc.

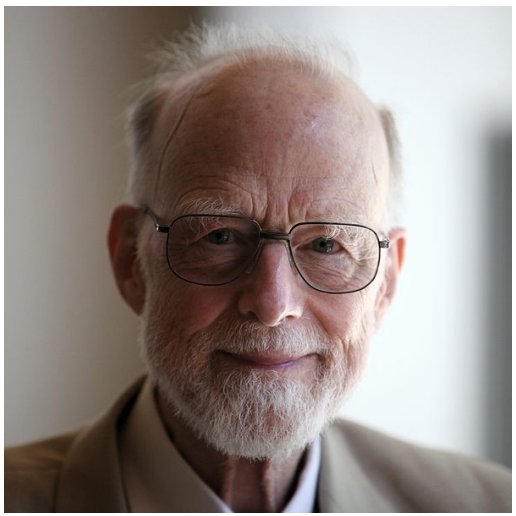
this lecture: deductive verification





A. M. Turing. Checking a large routine. 1949.

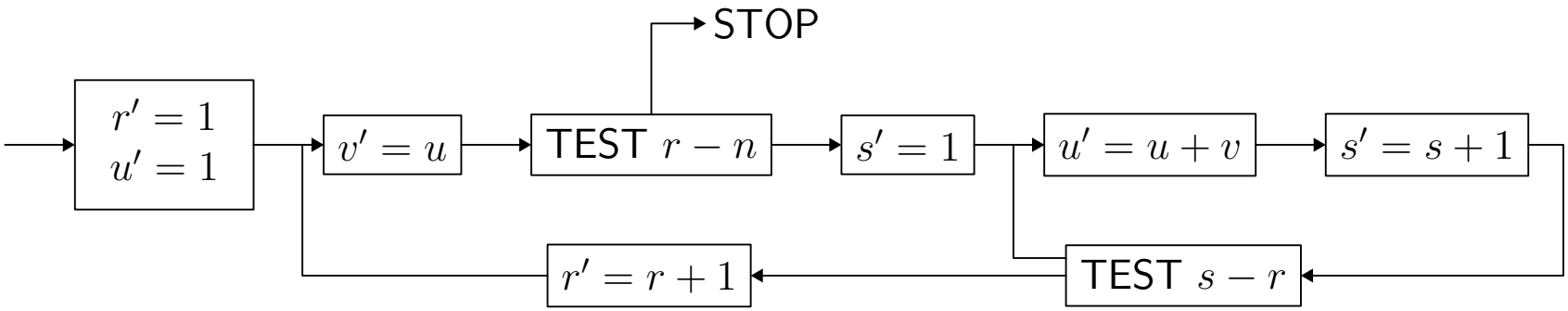




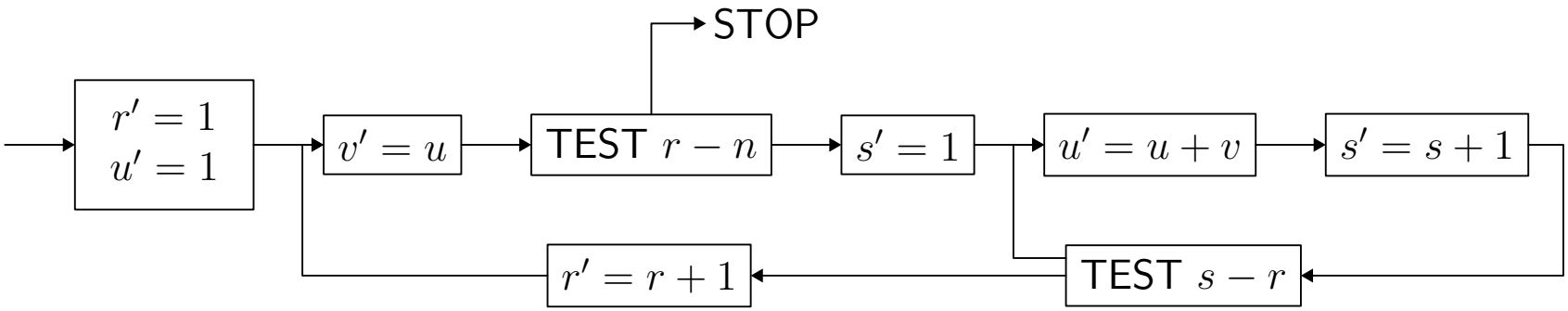
Tony Hoare.

An Axiomatic Basis for Computer Programming.
1969.

checking a large routine (Turing, 1949)

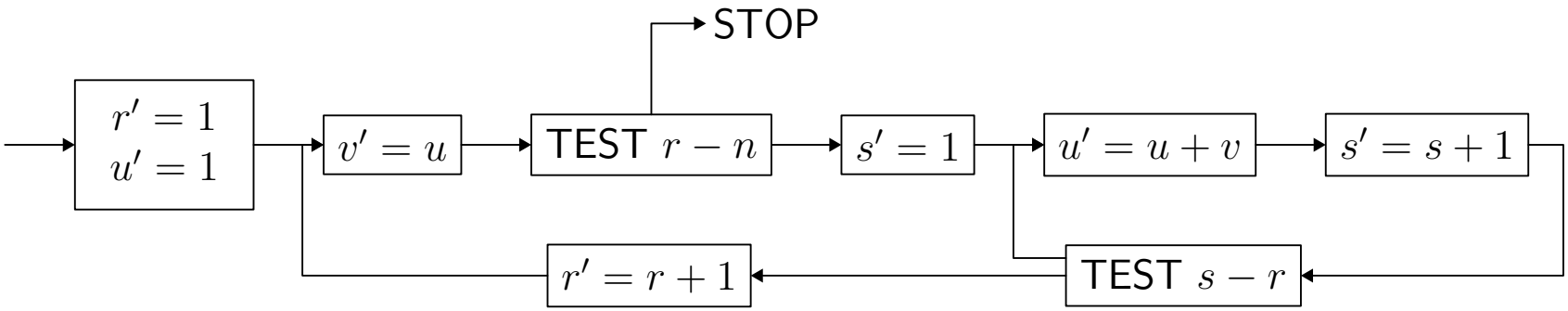


checking a large routine (Turing, 1949)



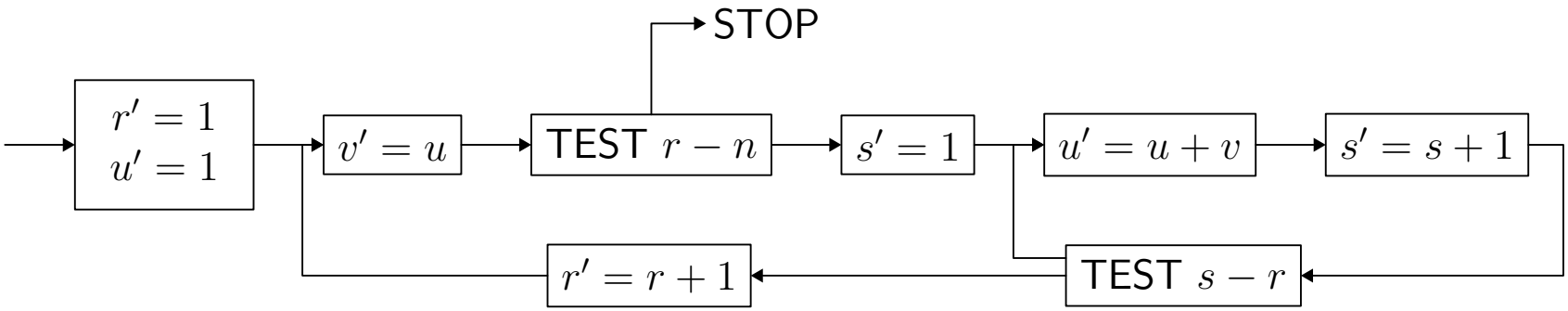
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$

checking a large routine (Turing, 1949)



precondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$
postcondition $\{u = n!\}$

checking a large routine (Turing, 1949)



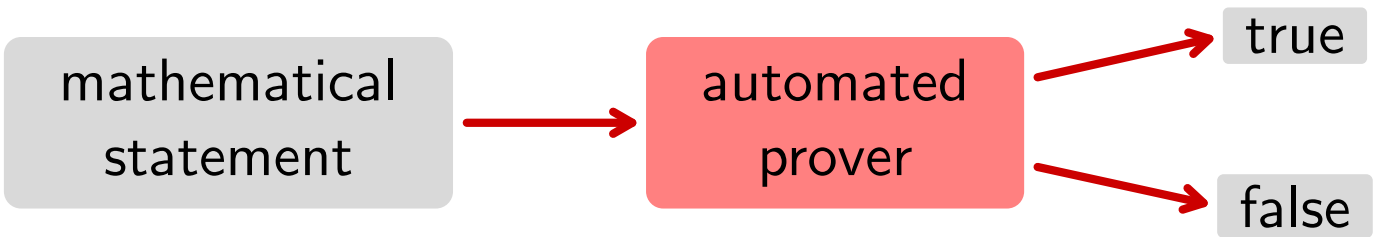
precondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do` invariant $\{u = r!\}$
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do` invariant $\{u = s \times r!\}$
    $u \leftarrow u + v$
postcondition $\{u = n!\}$

```
forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = n!) /\
  (0 <= n - 1 ->
    1 = 0! /\
    (forall u:int.
      (forall r:int. 0 <= r /\ r <= n - 1 -> u = r! ->
        (1 > r -> u = (r + 1)!) /\
        (1 <= r ->
          u = 1 * r! /\
          (forall u1:int.
            (forall s:int. 1 <= s /\ s <= r -> u1 = s * r! ->
              (forall u2:int.
                 u2 = u1 + u -> u2 = (s + 1) * r!)) /\
            (u1 = (r + 1) * r! -> u1 = (r + 1)!)))) /\
      (u = ((n - 1) + 1)! -> u = n!)))
```

what do we do with this mathematical statement?

we could perform a manual proof (as Turing and Hoare did)
but it is long, tedious, and error-prone

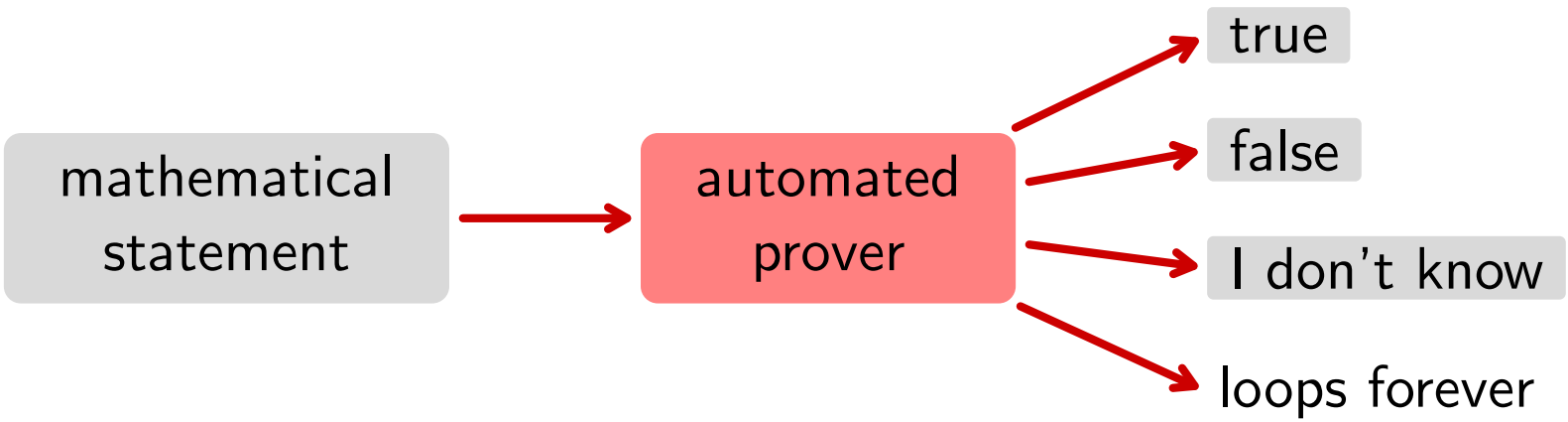
so we turn to tools that mechanize mathematical reasoning


mathematical statement
automated prover
true
false

it is not possible to implement such a program
(Turing/Church, 1936, from Gödel)

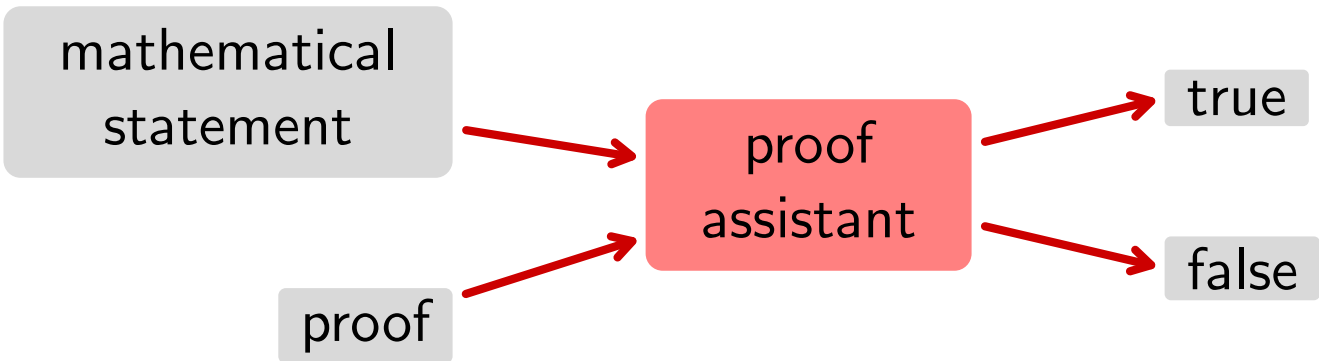
full employment theorem for mathematicians

Kurt Gödel



examples: Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

if we only intend to check a proof, we can do it



examples: Coq, Isabelle, PVS, HOL Light, etc.

Georges Gonthier, using Coq

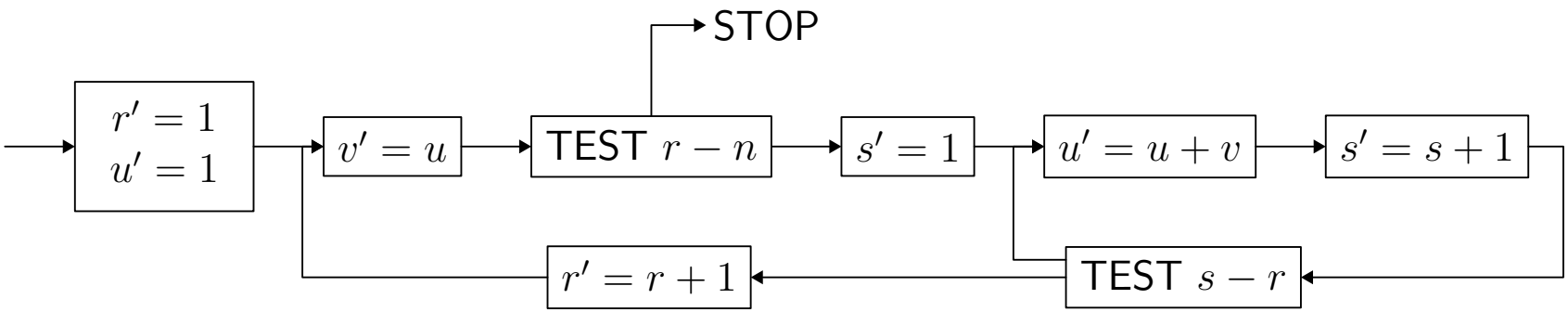
- the four color theorem
- Feit–Thompson theorem

Thomas Hales, using HOL Light

- Kepler conjecture

let's verify a program

Turing's routine



$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$

termination

in programming, we have

- loops
  the program execution may return to a previous point
- recursion
  a function can be defined with self-references

$x \leftarrow$ *a positive integer*
`while` $x \neq 1$ `do`
  `if` x *is even* `then`
    $x \leftarrow x/2$
  `else`
    $x \leftarrow 3x + 1$

$$f(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ f(f(n+11)) & \text{otherwise} \end{cases}$$

$$f(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ f(f(n+11)) & \text{otherwise} \end{cases}$$

it's called McCarthy's 91 function

you can prove either

partial correctness
if the precondition holds
and if the program terminates
then its postcondition holds

or

total correctness
if the precondition holds
then the program terminates
and its postcondition holds

partial correctness is a rather weak property,
since non-termination can turn your whole proof into something meaningless

bad news: we cannot check automatically whether a program terminates or not

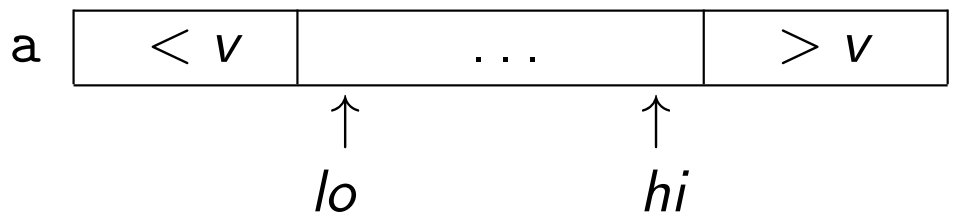
we have to provide hints, such as an upper bound on the number of steps before termination

let's prove that McCarthy's 91 function terminates

$$f(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ f(f(n + 11)) & \text{otherwise} \end{cases}$$

I do not think it means what you think it means

```
lo ← 0
hi ← len(a) − 1
while lo ≤ hi do
    m ← lo + (hi − lo)/2
    if a[m] < v
        lo ← m + 1
    else if a[m] > v
        hi ← m − 1
    else
        return m
return -1
```



`def` $binary\_search(a, v)$:
  `requires` ... the array is sorted ...
  `ensures` $0 \leq result < len(a) \wedge a[result] = v$
    $\vee\ result = -1 \wedge \forall i.\ 0 \leq i < len(a) \Rightarrow a[i] \neq v$

this is perfectly fine

but if we write instead

```
def binary_search(a, v):
  requires ... the array is sorted ...
  ensures (0 ≤ result < len(a) ⇒ a[result] = v)
          ∧ (result = −1 ⇒ ∀i. 0 ≤ i < len(a) ⇒ a[i] ≠ v)
```

the program can now return −2 and yet be proved correct

before you do any proof, get the specification right

then have the reader agree with you on the spec

otherwise, the whole proof is a waste of time

ghost code

data and code added to the program
to make the proof simpler

we search the smallest Fibonacci number equal to or greater than n

$a, b \leftarrow 0, 1$
`while` $a < n$ `do`
  $a, b \leftarrow b, a + b$
`return` a

to prove it correct
we may want to introduce a loop invariant as follows

$a, b \leftarrow 0, 1$
`while` $a < n$ `do`
  `invariant` $\exists i.\ i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
`return` a

but proving the existence of i is difficult for theorem provers

instead, we can keep track of the value of i with a ghost variable

$a, b \leftarrow 0, 1$
$i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  $i \leftarrow i + 1$
`return` a

instead of having the theorem prover guessing the right value, we provide it

- ghost code may read regular data but can't modify it
- ghost code cannot modify the control flow of regular code
- regular code does not see ghost data

consequence: ghost code can be removed
without observable modification

$a, b \leftarrow 0, 1$
$i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  $i \leftarrow i + 1$
`return` a

$a, b \leftarrow 0, 1$
`ghost` $i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  `ghost` $i \leftarrow i + 1$
`return` a

$a, b \leftarrow 0, 1$

`while` $a < n$ `do`

$\quad a, b \leftarrow b, a + b$

`return` a

suppose we want to prove that, for all n,

$$n! \geq 1$$

we can make a program that proves it

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`

$\quad f \leftarrow i \times f$

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)!$
  $f \leftarrow i \times f$
`assert` $f = n!$

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)! \wedge f \geq 1$
  $f \leftarrow i \times f$
`assert` $f = n!$
`assert` $n! \geq 1$

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)! \wedge f \geq 1$
  $f \leftarrow i \times f$
`assert` $f = n!$
`assert` $n! \geq 1$

- the whole program is ghost
  (we do not intend to run it)
- we have performed a proof by induction
  (automated theorem provers won't do that by themselves)

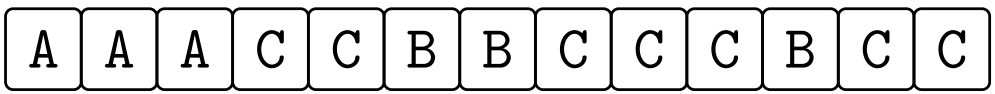
conclusion

- we can verify programs, once and for all
- we have tools to do this
  and in particular theorem provers
- a program can be a proof
- the programming language does not matter
- go see The Princess Bride if you haven't already

http://projecteuler.net/

verification of an algorithm

given a multiset of N votes

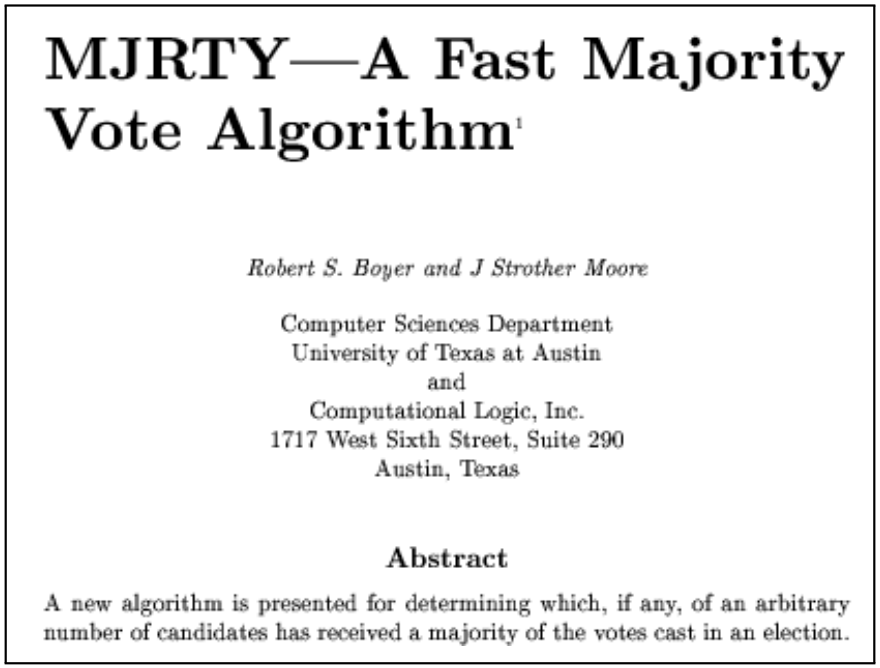


determine the majority, if any

due to Boyer & Moore (1980)

linear time

uses only three variables



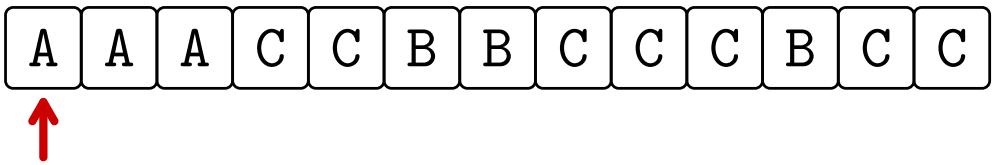
MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

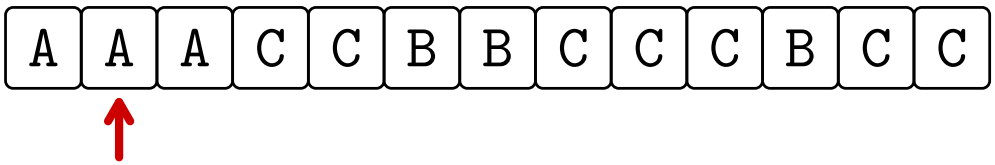
Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

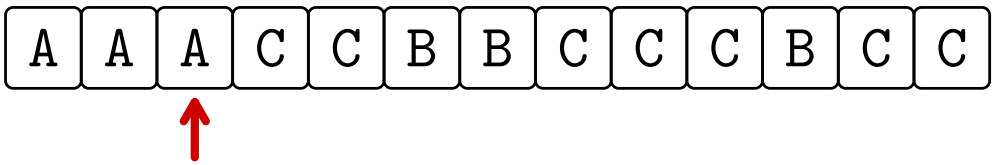
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.



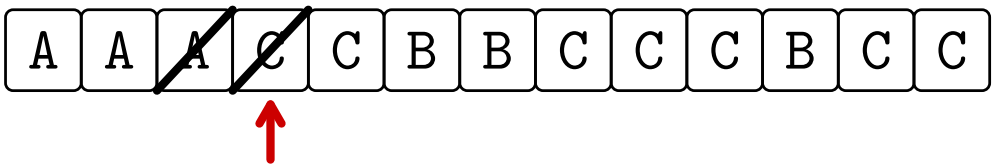

```
cand = A
k    = 1
```



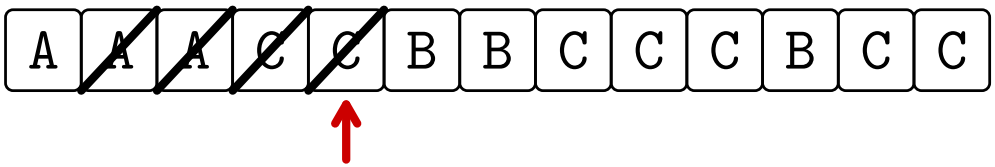

```
cand = A
k    = 2
```



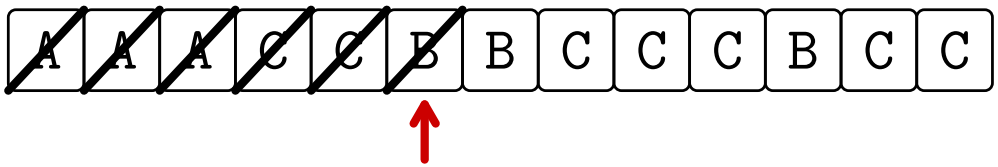

```
cand = A
k    = 3
```



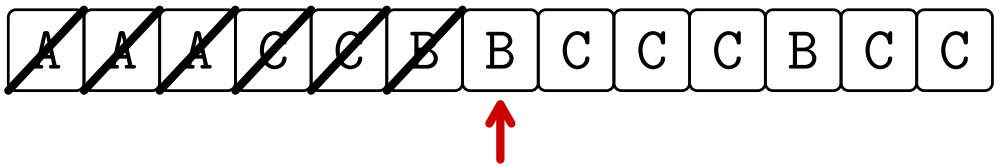

```
cand = A
k    = 2
```



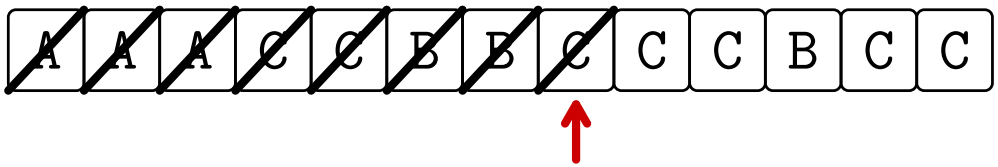

```
cand = A
k    = 1
```



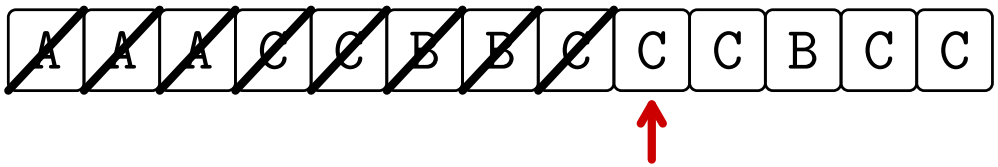

```
cand = A
k    = 0
```



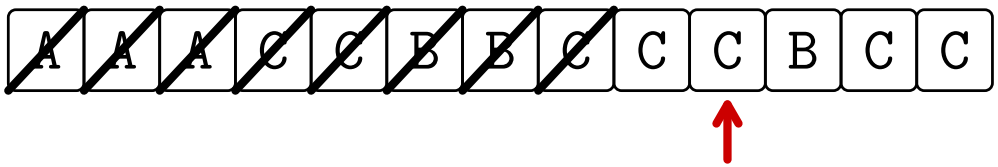

```
cand = B
k    = 1
```



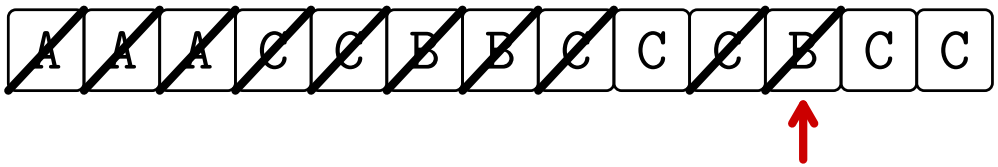

```
cand = B
k    = 0
```



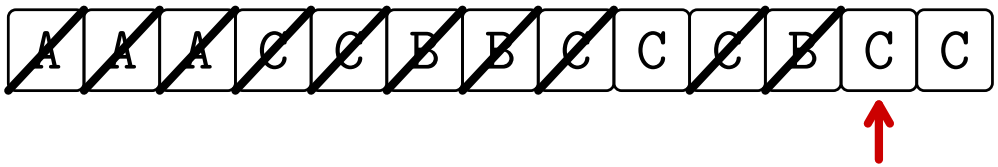

```
cand = C
k    = 1
```



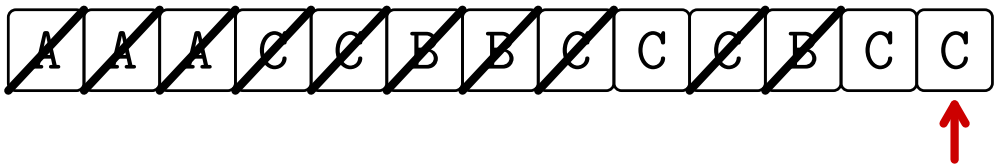

```
cand = C
k    = 2
```



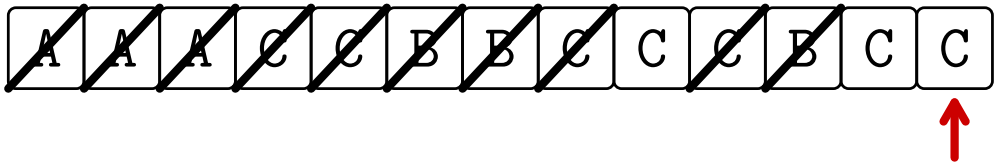

```
cand = C
k    = 1
```




```
cand = C
k    = 2
```




```
cand = C
k    = 3
```




```
cand = C
k    = 3
```

then we check if C indeed has majority, with a second pass
(in that case, it has: $7 > 13/2$)