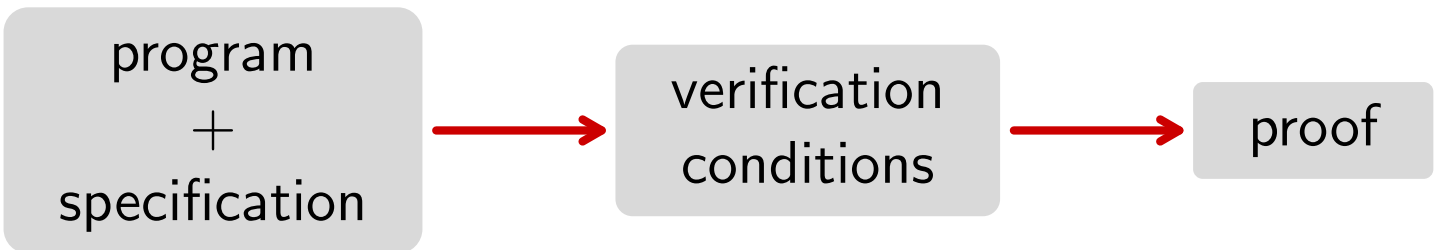
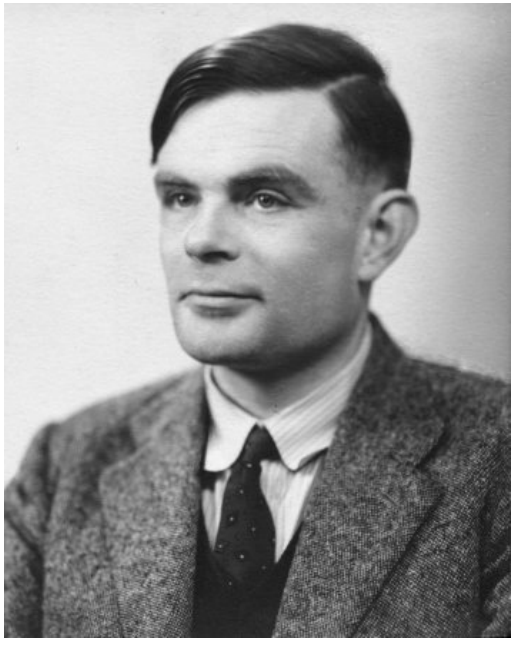
Deductive Program Verification

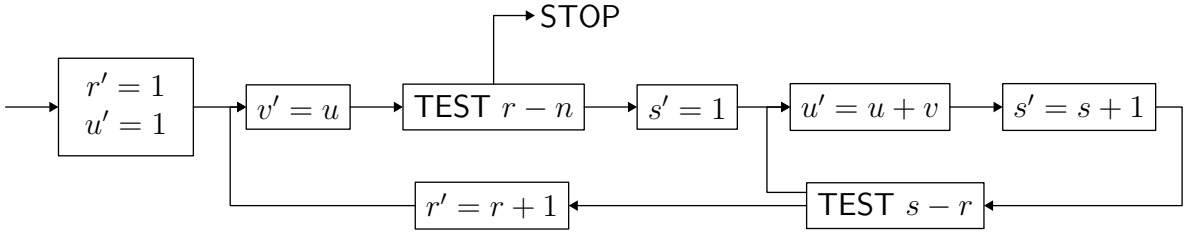
Jean-Christophe Filliâtre
CNRS

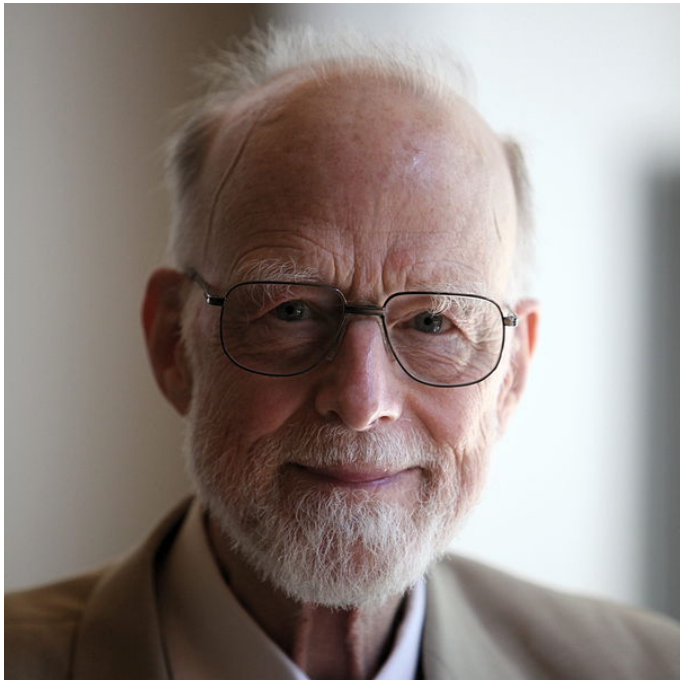
Programming Languages Mentoring Workshop 2013


program
+
specification
verification
conditions
proof

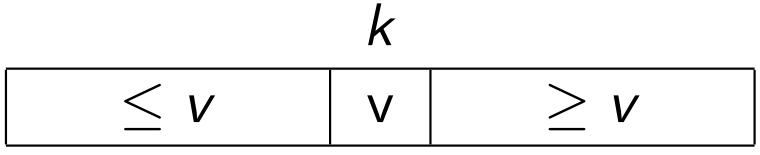


A. M. Turing. Checking a large routine. 1949.





Tony Hoare.
Proof of a program: FIND.
Commun. ACM, 1971.



which programs? which specs?

programs

- pseudo code / mainstream languages / DSL
- small / large

specs

- safety, i.e. the program does not crash
- absence of arithmetic overflow
- complex behavioral property, e.g. “sorts an array”

which logic?

- too rich: we won't be able to automate proofs
- too poor: we can't model programming languages and we can't specify programs

typically, a compromise

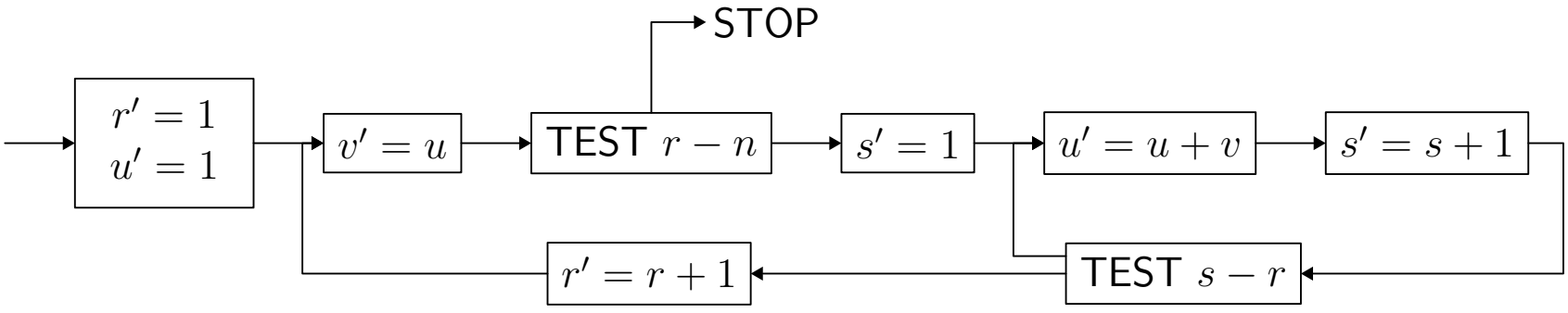
- e.g. first-order logic + equality + arithmetic

what about proofs?

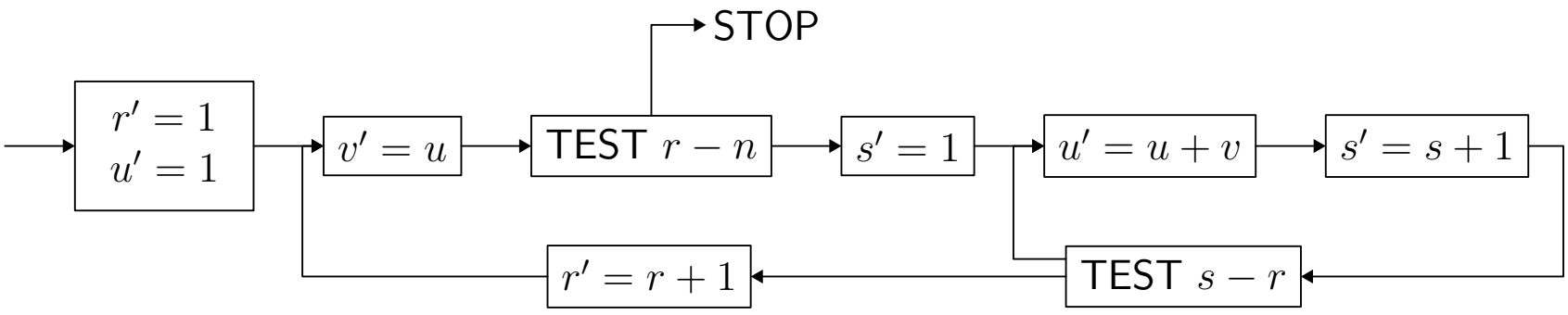
a gift: theorem provers

- proof assistants: Coq, PVS, Isabelle, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- SMT solvers: CVC3, Z3, Yices, Alt-Ergo, etc.
- dedicated provers

checking a large routine (Turing, 1949)

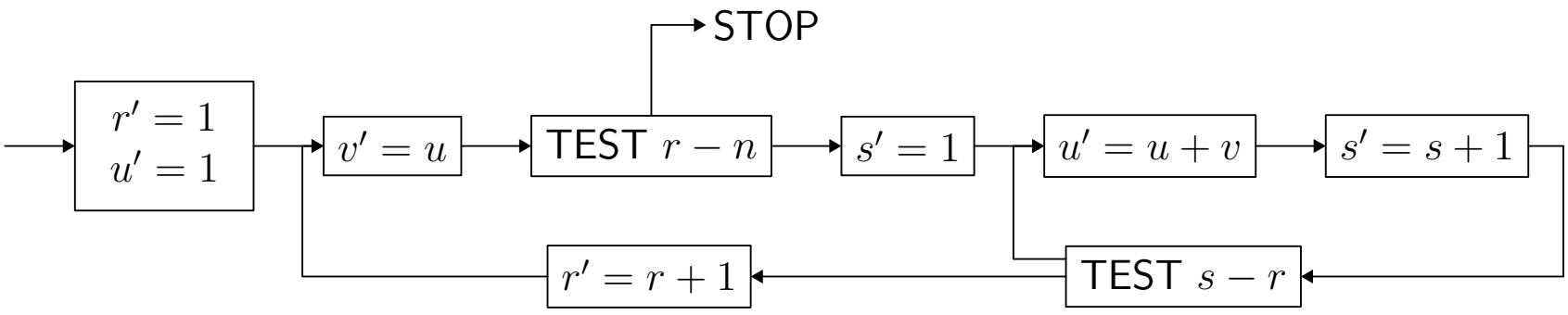


checking a large routine (Turing, 1949)



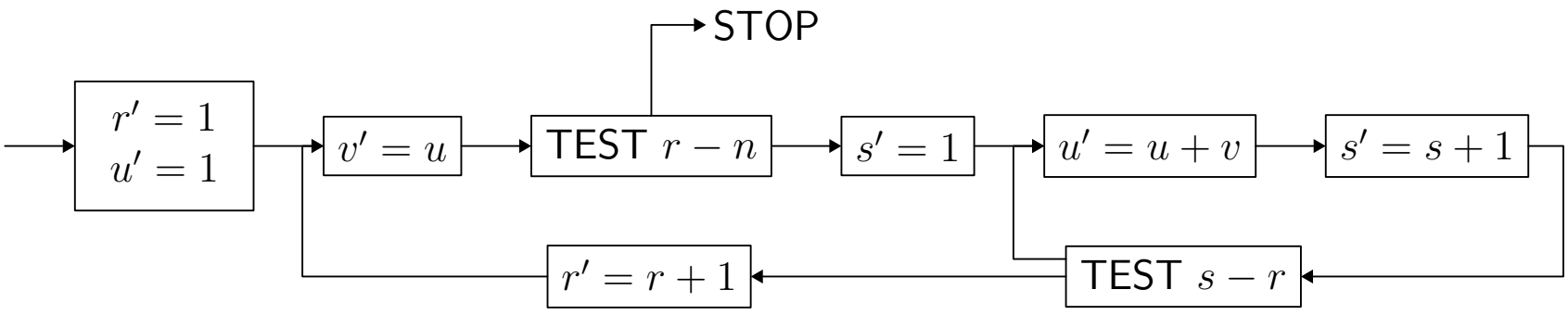
$u \leftarrow 1$
for $r = 0$ to $n - 1$ do
  $v \leftarrow u$
  for $s = 1$ to r do
    $u \leftarrow u + v$

checking a large routine (Turing, 1949)



requires $\{n \geq 0\}$
$u \leftarrow 1$
for $r = 0$ to $n - 1$ do
  $v \leftarrow u$
  for $s = 1$ to r do
    $u \leftarrow u + v$
ensures $\{u = fact(n)\}$

checking a large routine (Turing, 1949)



requires $\{n \geq 0\}$
$u \leftarrow 1$
for $r = 0$ to $n - 1$ do invariant $\{u = fact(r)\}$
  $v \leftarrow u$
  for $s = 1$ to r do invariant $\{u = s \times fact(r)\}$
    $u \leftarrow u + v$
ensures $\{u = fact(n)\}$

```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                 u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
              (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
      (u = fact((n - 1) + 1) → u = fact(n))))
```

```
function fact(int) : int
axiom fact0: fact(0) = 1

goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
```

```
function fact(int) : int

axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
goal vc: ∀ n:int. n ≥ 0 →

  (0 ≤ n - 1 →

    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →

        (1 ≤ r →

          (∀ u1:int.

            (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
```

SMT means Satisfiability Modulo Theories

an SMT solver combines

$$\forall + \text{SAT} + \text{Equality} + \text{Arith} + \ldots$$

e.g.

$$\dfrac{\dfrac{\dfrac{n \geq 0 \qquad 0 > n - 1}{n = 0}(\mathit{Arith}) \qquad \dfrac{}{\texttt{fact}(0) = 1}(\mathit{Ax})}{1 = \texttt{fact}(n)}(\mathit{Equality})}$$

computing the verification conditions

a well-known technique: weakest preconditions
(Dijkstra 1971, Barnett/Leino 2005)

yet extracting verification conditions for a realistic programming language is a lot of work

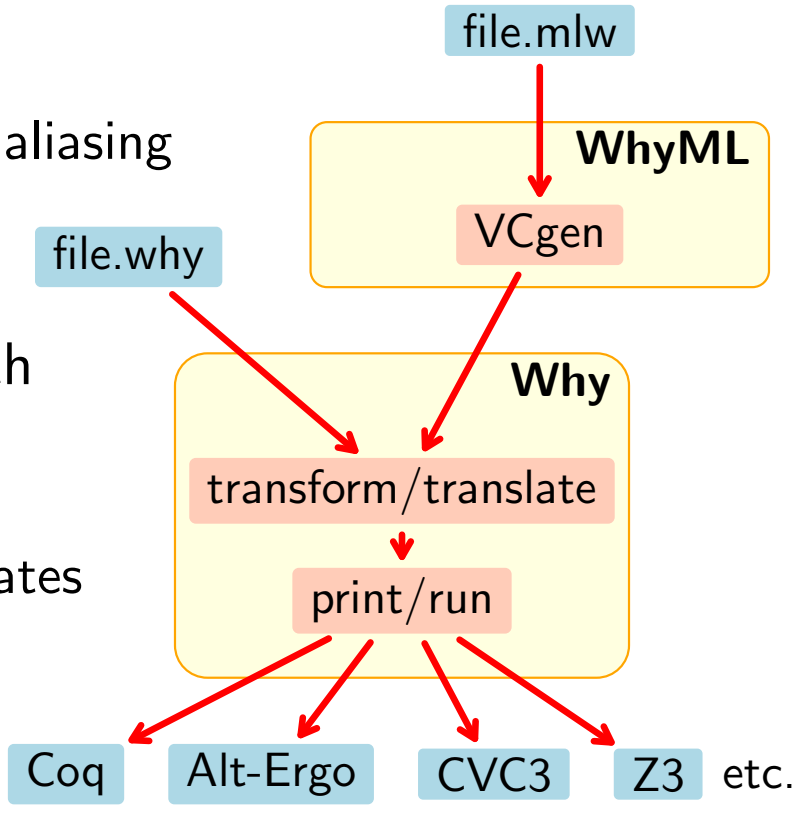
as in a compiler, we rather translate to some intermediate language from which we extract VCs

two examples:

- Boogie (Microsoft Research)
- Why3 (Univ. Paris Sud / Inria)

Why3 in a nutshell

- a programming language, with
  - polymorphism
  - pattern-matching
  - exceptions
  - mutable data structures, but no aliasing
- a polymorphic first-order logic, with
  - algebraic data types
  - recursive definitions
  - inductive and coinductive predicates

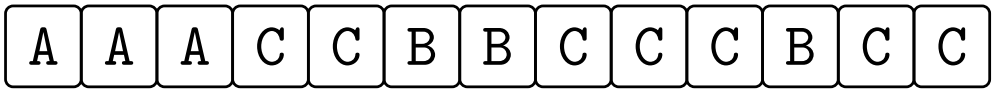


- Java programs: Krakatoa (Marché Paulin Urbain)
- C programs: Jessie plug-in of Frama-C (Marché Moy)
- Ada programs: Hi-Lite (Adacore)
- probabilistic programs (Barthe et al.)
- cryptographic programs (Vieira)

an example of program verification

Boyer-Moore's majority

given a multiset of N votes

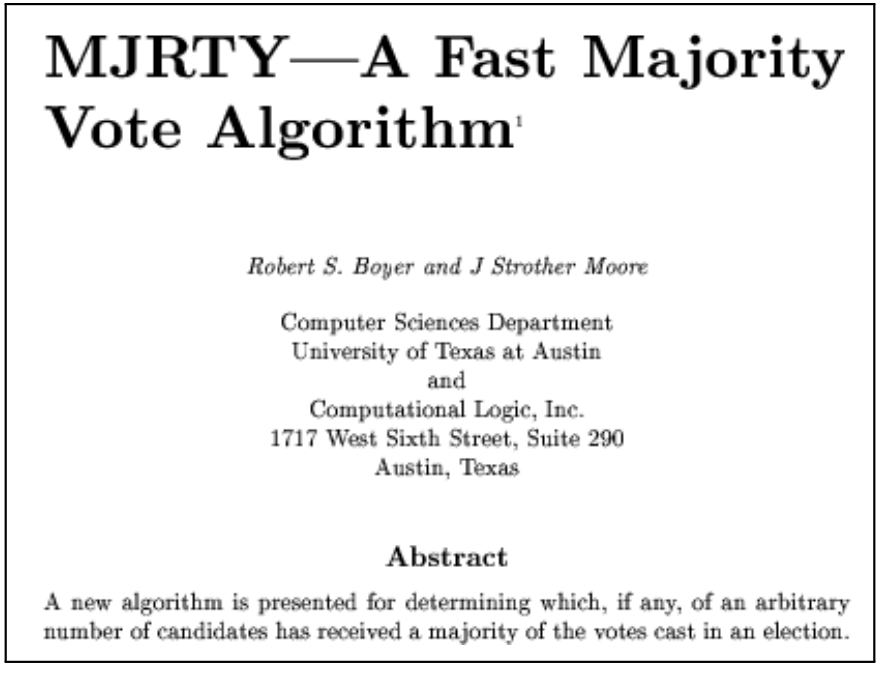


determine the majority, if any

due to Boyer & Moore (1980)

linear time

constant extra space



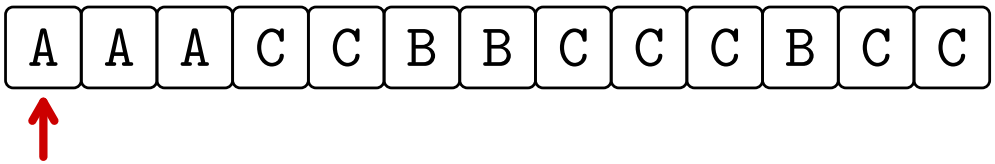
MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

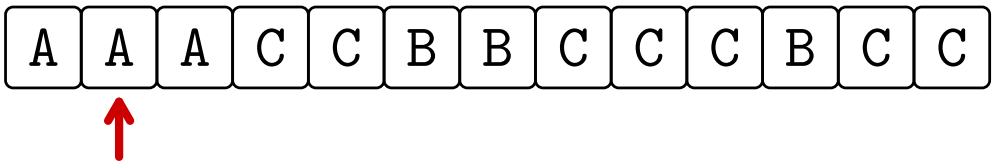
Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

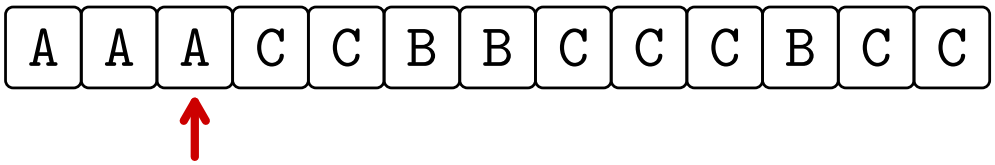
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.



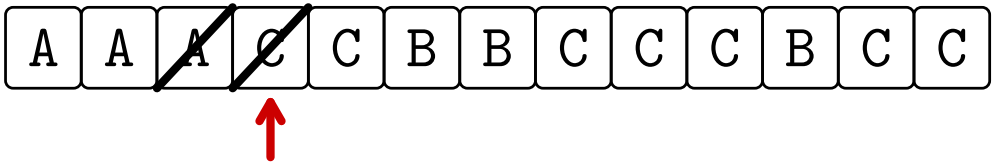

```
cand = A
k    = 1
```



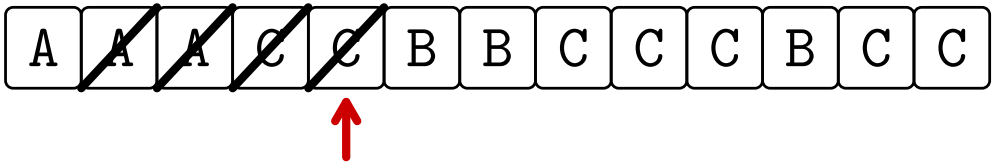

```
cand = A
k    = 2
```



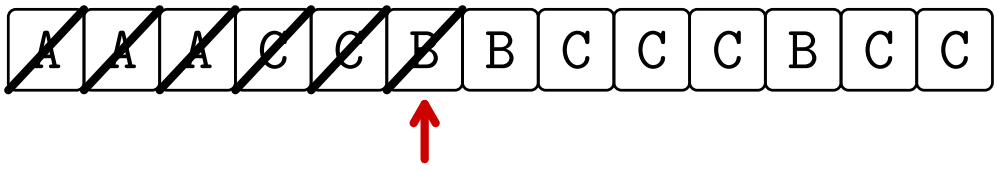

```
cand = A
k    = 3
```



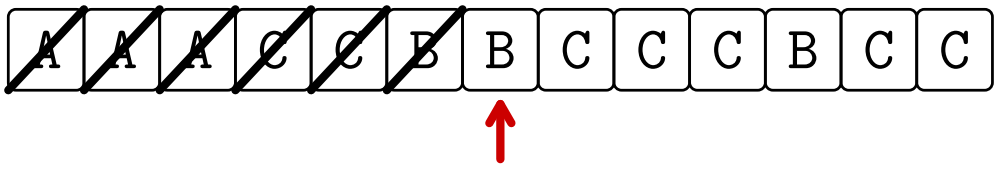

```
cand = A
k    = 2
```



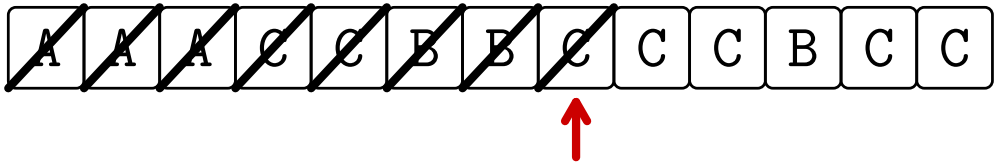

```
cand = A
k    = 1
```



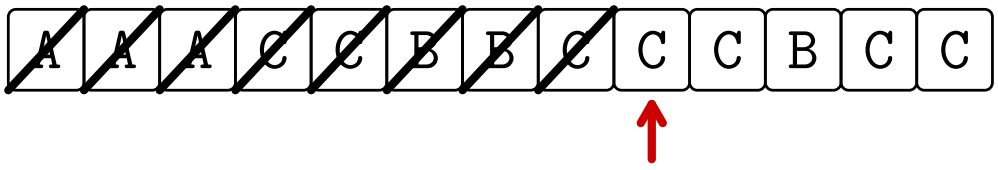

```
cand = A
k    = 0
```



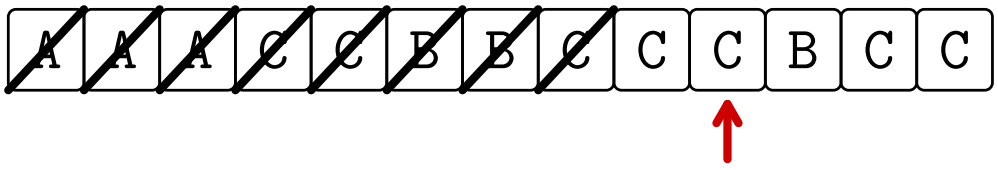

```
cand = B
k    = 1
```



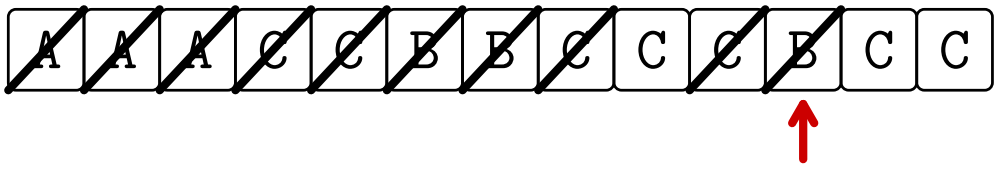

```
cand = B
k    = 0
```



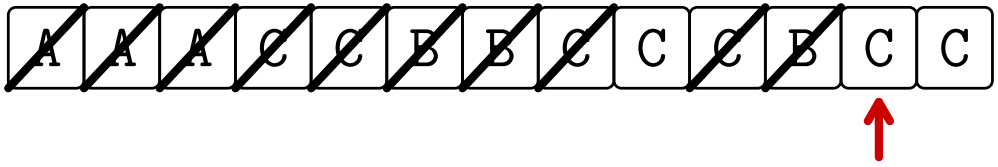

```
cand = C
k    = 1
```



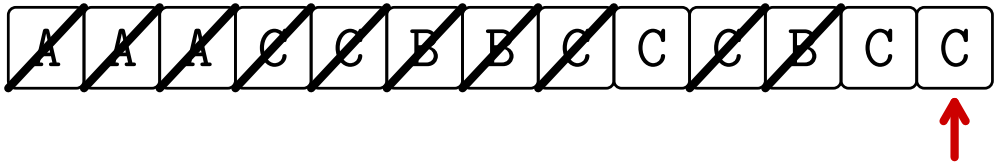

```
cand = C
k    = 2
```



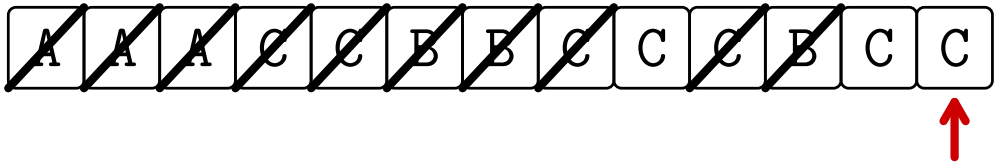

```
cand = C
k    = 1
```




```
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: $7 > 13/2$)

Why3 code

```
let mjrty (a: array candidate) : candidate =
  let n = length a in
  let cand = ref a[0] in let k = ref 0 in
  for i = 0 to n-1 do
    if !k = 0 then begin cand := a[i]; k := 1 end
    else if !cand = a[i] then incr k else decr k
  done;
  if !k = 0 then raise Not_found;
  try
    if 2 * !k > n then raise Found; k := 0;
    for i = 0 to n-1 do
      if a[i] = !cand then begin
        incr k; if 2 * !k > n then raise Found
      end
    done;
    raise Not_found
  with Found →
    !cand
  end
```

demo

still a lot to do

to simultaneously

- increase proof automation
  - e.g. automatic induction
- enrich the specification logic
  - e.g. higher-order logic
- support more programming constructs
  - e.g. continuations, coroutines, higher-order functions