UNIVERSIDADE
BEIRA INTERIOR

Certified Programming in the heavy presence of pointers

The case for Union-Find

Jean-Christophe Filliâtre, Mário Pereira, Simão Melo de Sousa

A contextual Introduction

context

a bug is a disaster waiting to happen

security perspective: attack entry, security breach, information leakage, etc.

critical systems perspective: failure, dammage, (mission, life, etc.) loss, etc.

business perspective: disturbance, costs, loss of trust/business, etc.

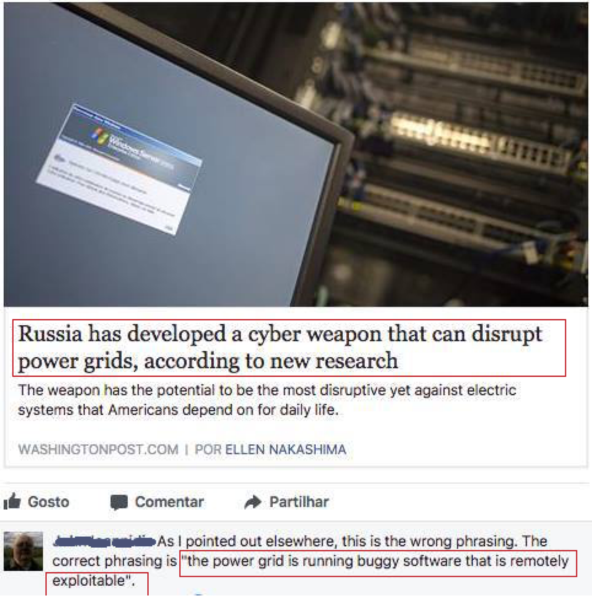
business example

Facebook development team motto:

Do not fail in front of the client, fail in-house

$\Longrightarrow$ heavy corporate investment in Programming (Language) Science and Technologies: Reason, Hack, React, infer, flow, webassembly (contributor) etc...

another example



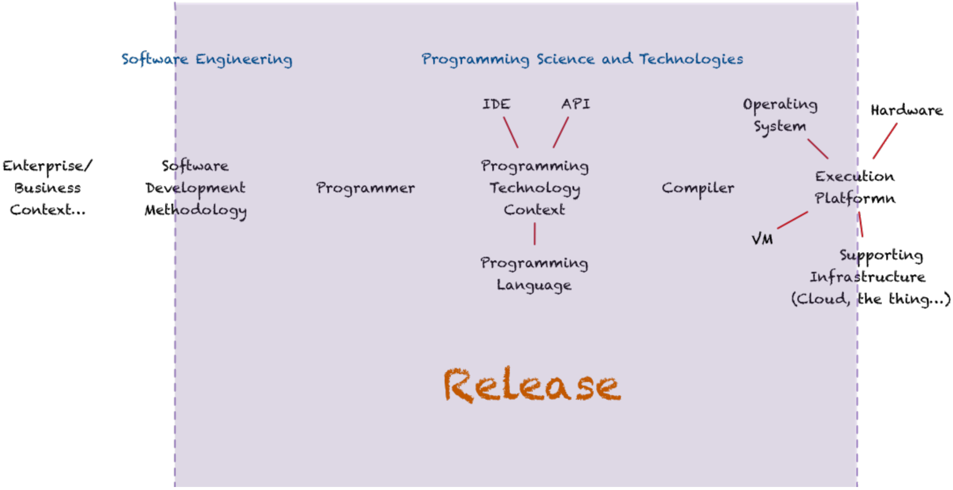
Russia has developed a cyber weapon that can disrupt power grids, according to new research

The weapon has the potential to be the most disruptive yet against electric systems that Americans depend on for daily life.

WASHINGTONPOST.COM | POR ELLEN NAKASHIMA

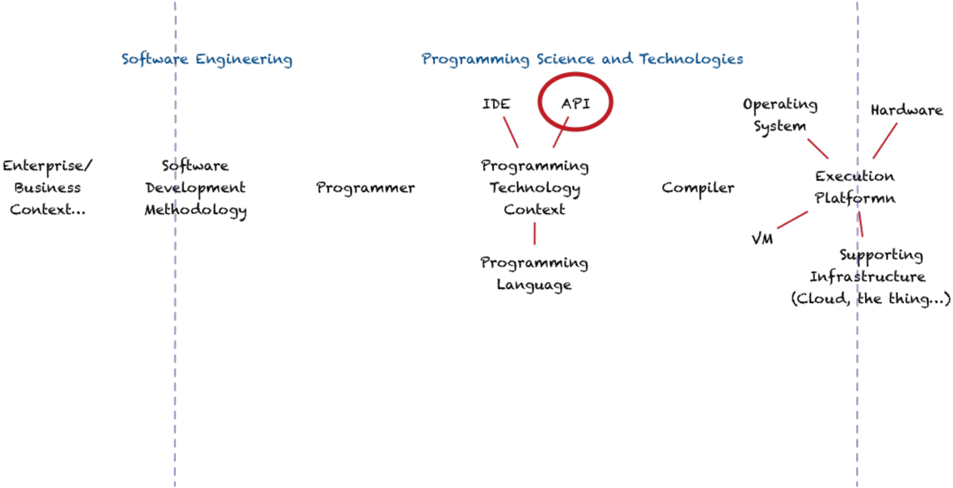
[redacted] As I pointed out elsewhere, this is the wrong phrasing. The correct phrasing is "the power grid is running buggy software that is remotely exploitable".

software development context



the software lifecycle is complex

in this talk



Software Engineering

Programming Science and Technologies

IDE

API

Operating System

Hardware

Enterprise/ Business Context...

Software Development Methodology

Programmer

Programming Technology Context

Compiler

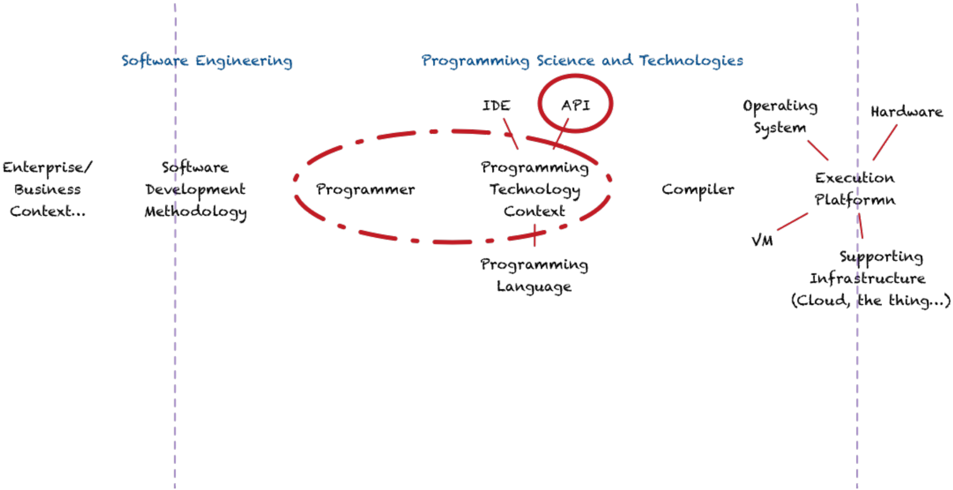
Execution Platformn

VM

Programming Language

Supporting Infrastructure (Cloud, the thing...)

but first:


Software Engineering
Programming Science and Technologies
Enterprise/ Business Context...
Software Development Methodology
Programmer
IDE
API
Programming Technology Context
Programming Language
Compiler
Operating System
Hardware
Execution Platformn
VM
Supporting Infrastructure (Cloud, the thing...)

Some facts about programming activities

programming environments are always the **result of design choices** (made by its creators)

there is no known silver bullet, then it's always a (maybe outdated) **compromise** (security, efficiency, high level/low level, automatic/manual memory management etc.)

Some facts about programming activities

- **code reuse** (central to the OO paradigm success), boilerplates/ code recipes, *stackoverflow* style of programming, the `import everything` paradigm, etc.

- relation between the programmer and its programming environment:

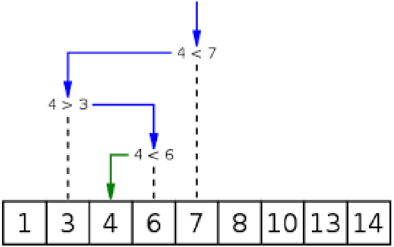
  from the programming environment perspective, it is **assumed** that the programmer **knows** the programming environment, **its limitations, its compromises** and subsequent coding practices, and **masters what he's doing**

  from the programmers perspective, it is **assumed** that the offered programming facilities **do exactly** what they're supposed to do, as efficient as possible, as simple as possible and do not introduce unspecified behavior

how do we know that these assumptions hold?

binary search: first publication in 1946 ...

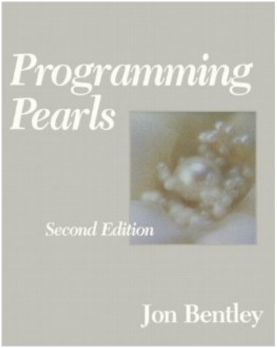
... first **correct** publication in 1962



old but gold

Jon Bentley - Programming Pearls. 1986 (2^{nd} ed. 2000)

(column 4) - writing correct programs
The challenge of binary search



(as seen p.37)

Warning
Boring material ahead
skip to section 4,4
when drowsiness strikes

concise and crystal clear explanation of what is going on...

and yet...

old but gold

in 2006, an embarrassing bug was found in the standard library of Java... in the (binary) search methods

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

... been there for more than nine years!

old but gold

the bug :

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

may surpass the `int` type range: *integer overflow*

and then cause an *array out of bound* error

possible solution:

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

Computer Arithmetics $\neq$ Arithmetics

even with integers

and

it's a good idea to have API/Standard Libraries done right...

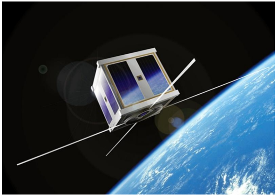
The Vocal Project

VOCAL – a **V**erified **OCA**ml **L**ibrary

a library of **correct-by-construction**, **efficient general-purpose** data structures and algorithms

- priority queues
- hash tables
- sequences
- sets/maps
- resizable arrays
- heap
- ...

- graph algorithms
- sorting
- searching
- string processing
- union-find
- ...



software that could benefit from such a library: Coq, Frama-C, Astrée, SPARK, Infer, Alt-Ergo, Cubicle, EasyCrypt, ProVerif, etc.

which, in turn, are used in avionics, defense, aerospacial, finance, security, hardware, etc.

(LRI/CNRS + Inria + Verimag + TrutInSoft + OCamlPro)

a specification language for OCaml

regular .mli files with

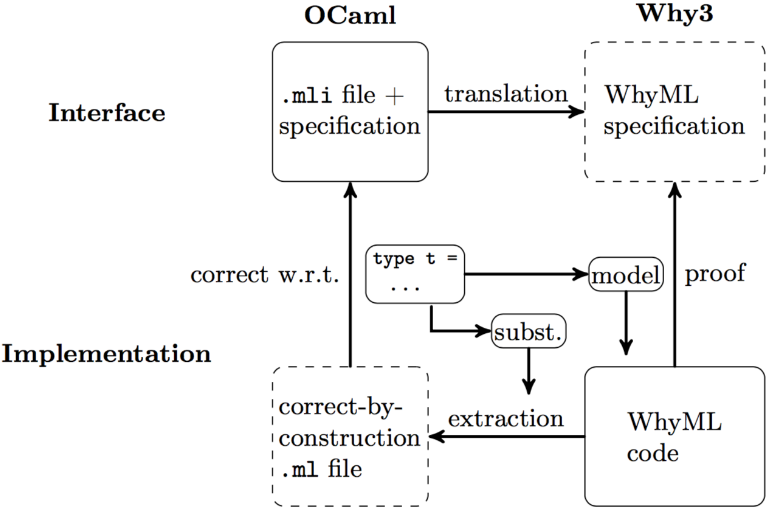
- formal specification in special comments (*à la* JML / ACSL)
- informal comments
- users can ignore formal specs
- simple, mostly first-order logic

regular .ml files

- no spec

three design workflows: why3, COQ, CFML+COQ

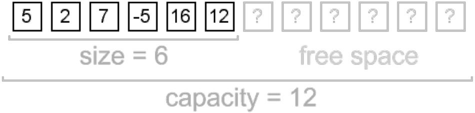
Vocal/Why3 workflow



example — Vector.mli

```
(** resizable arrays *)

type 'a t
(*@ ephemeral *)
(*@ field mutable view: 'a seq *)
(*@ invariant length view ≤ Sys.max_array_length *)
```



example — Vector.mli

```
val resize: 'a t → int → unit
(** [resize a n] sets the length of vector [a] to [n].
    The elements that are no longer part of the vector, if any,
    are internally replaced by the dummy value of vector [a],
    so that they can be garbage collected when possible.
    Raise [Invalid_argument]
             if [n < 0] or [n > Sys.max_array_length] *)
(*@ resize a n
      checks   0 ≤ n ≤  Sys.max_array_length
      modifies a
      ensures  length a.view = n
      ensures  forall i. 0 ≤ i < min (length (old a.view)) n →
                         a.view[i] = (old a.view)[i]  *)
```

WhyML code for Vector

```
type t 'a = {
               dummy: 'a;
        mutable size: int63;
        mutable data: array 'a;
  ghost mutable view: seq 'a;
}
invariant { length view = size}
invariant { forall i. 0 ≤ i < size → view[i] = data[i] }
invariant { 0 ≤ size ≤ length data ≤ max_array_length }
invariant { forall i. size ≤ i < length data → data[i] = dummy }
```

WhyML code for Vector

```
type t 'a = {
               dummy: 'a;
        mutable size: int63;
        mutable data: array 'a;
  ghost mutable view: seq 'a;
}
invariant { length view = size}
invariant { forall i. 0 ≤ i < size → view[i] = data[i] }
invariant { 0 ≤ size ≤ length data ≤ max_array_length }
invariant { forall i. size ≤ i < length data → data[i] = dummy }

let resize (a: t 'a) (n: int63) : unit
  writes   { a.data, a.size, a.data.elts, a.view }
  ensures  { n = a.size }
  ensures  { forall i. 0 ≤ i < MinMax.min ((old a).size) n →
               Seq.([]) a.view i = Seq.([]) (old a).view i }
  raises   { Invalid_argument → not (0 ≤ n ≤ max_array_length) }
= if not (zero≤ n ≤max_array_length) then raise Invalid_argument;
  unsafe_resize a n
```

extracted code from Vector

```
type 'a t = {
  dummy: 'a;
  mutable size: int;
  mutable data: ('a array);
}

let resize (a: 'a t) (n: int) : unit =
  begin
    if not (0 <= n && n <= Sys.max_array_length)
     then raise (Invalid_argument );
    unsafe_resize a n
  end
```

driver to ocaml

```
module mach.int.Int63
  syntax type int63 "int"
  syntax val  ( + ) "%1 + %2"
  ...
end

module mach.array.Array63
  syntax type array "(%1 array)"
  syntax val  ([])   "Array.get %1 %2"
  ...
end
```

Deductive program verification in a picture

the state of the verified OCaml modules with Vocal/Why3

module	spec	code	#VCs	
`UnionFind`	74	176	135	union-find
`PairingHeap`	41	245	52	persistent priority queues
`ZipperList`	66	180	87	zipper data structure for lists
`Arrays`	37	121	77	binary search and binary sort
`Queue`	54	185	119	mutable queues
`Vector`	149	309	142	resizable arrays
`HashSet`	21	34	12	sets using hash tables
`MergeSort`	12	401	630	in-place mergesort of lists
`Dfs`	-	58	5	depth-first graph marking
`Schorr-Waite`	-	184	172	in-place graph marking

all the VCs were proved automatically!

... and with Vocal/COQ/CFML

module	tool	loc	Coq
`Listmap`	Coq	50	170
`HashTable`	CFML	150	750
`UnionFind` *	CFML	60	800
`IntervalMap`	CFML	WiP	WiP

(∗) including (amortized) computational complexity

On the importance of efficient supporting data structure

the algorithmic quest

$n \setminus \mathcal{O}$	1	$log(n)$	n	$n\log(n)$	n^2	n^3	2^n	$n!$
5	1	3	4	15	25	125	32	120
10	1	4	10	33	10^2	10^3	10^3	3628800
10^2	1	7	10^2	664	10^4	10^6	10^{30}	$9,334 \times 10^{157}$
10^3	1	10	10^3	10^4	10^6	10^9	10^{301}	4.024×10^{5136}
10^4	1	13	10^4	10^5	10^8	10^{12}	10^{3000}	-
10^5	1	17	10^5	1.7×10^6	10^{10}	10^{15}	10^{30000}	-
10^6	1	20	10^6	2×10^7	10^{12}	10^{18}	$10^{3000000}$	-
10^7	1	23	10^7	2.3×10^8	10^{14}	-	-	-
10^8	1	27	10^8	2.7×10^9	10^{16}	-	-	-
10^9	1	30	10^9	3×10^{10}	10^{18}	-	-	-
10^{10}	1	33	10^{10}	3.3×10^{11}	10^{20}	-	-	-

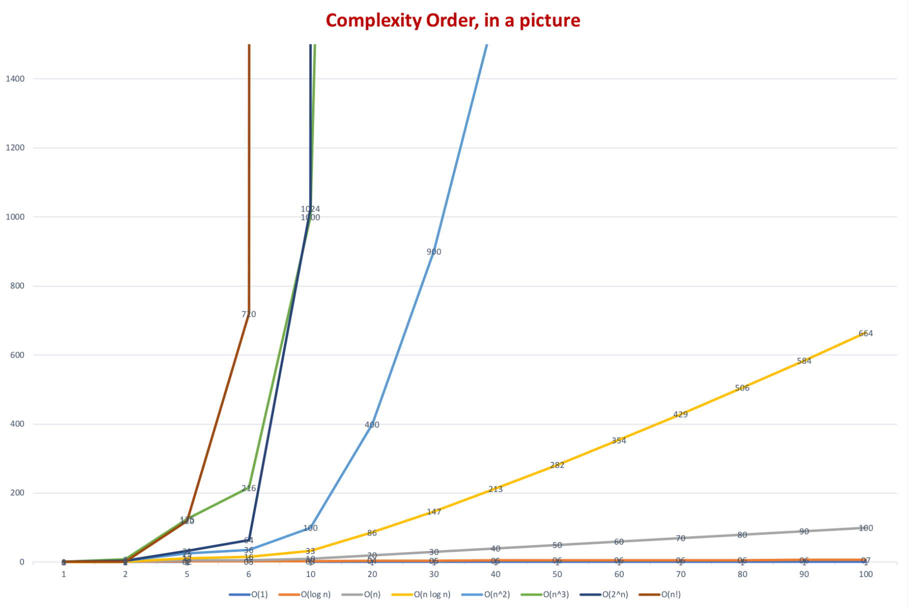
if we estimate 10^9 operations per second on current computers architectures then:

10 minutes $\approx 10^{12}$ **operations**
1 hour $\approx 10^{13}$ **operations**
1 day $\approx 10^{15}$ **operations**
1 year $\approx 10^{17}$ **operations**

the universe is approx. 14×10^9 years old

then *has been able to perform* $\approx 2 \times 10^{27}$ **operations**

in a picture


Complexity Order, in a picture
1400
1200
1000
800
600
400
200
0
1
2
5
6
10
20
30
40
50
60
70
80
90
100
720
1024
1000
900
400
216
125
120
100
64
36
32
25
16
33
86
147
213
282
354
429
506
584
664
10
20
30
40
50
60
70
80
90
100
O(1)
O(log n)
O(n)
O(n log n)
O(n^2)
O(n^3)
O(2^n)
O(n!)

an example

In a positively weighted graph with an order of 10^9 vertices, how much time it takes to determine the shortest path between two vertices?

using the Dijkstra algorithm? ... it depends on the data structure used to represent working data (e.g. the graph, the neighborhood relationship, the vertices to be processed)

naive implementations (*run from them!*) are **quadratic**.

If we accept that today's computers perform roughly 10^9 operations per second, this will take more than 30 years to find the shortest path

if one uses efficient data structures for the neighborhood relationship (**set, dictionaries, hash tables**, etc.) and **self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap** as a priority queue for the vertices to be processed, Dijkstra algorithm can perform on a $\mathcal{O}(|E| + |V| log(|V|))$ basis

... then we run under **10 seconds** to find the shortest path!

the case for minimum spanning trees

Prim's algorithm:

1. Pick some arbitrary start node s, initialize tree $T = \{s\}$
2. repeatedly add the shortest edge incident to T until the tree spans all the nodes

naive implementations are also quadratic.

largely more efficient implementations make use of a **priority queue** to store neighbors of the current tree.

If implemented by a **binary heap**, the complexity is $\mathcal{O}(|E| log(|V|))$

with a **Fibonacci heap**, we have $\mathcal{O}(|E| + |V| log(|V|))$ (nicer for dense graphs)

the case for minimum spanning trees

Kurskal's algorithm:

1. sort edges by cost and examine them from cheapest to most expansive
2. put each edge into the current forrest if it does not form a cycle with the edges chosen so far

again, naive implementations are quadratic.

the initial sort costs $\mathcal{O}(|E| log(|E|))$, the challenge is to detect cycles efficiently (i.e. determine connected components).

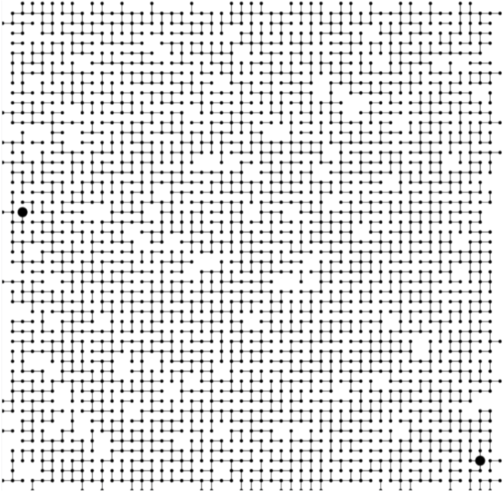
The use of **union-find data structure** allows for such efficiency (below $\mathcal{O}(|E| log(|E|))$!!) and then we can then obtain an overall (worst case) complexity of $\mathcal{O}(|E| log(|E|))$!

(epilogue: Borůvka Alg. in $\mathcal{O}(|E| log(|V|)$ (**and** easily parallelizable) and Chazelle Alg. in $\mathcal{O}(|E|\alpha(|E|, |V|))$, that is in practice... **linear** !!)

Union-Find

Are we connected?

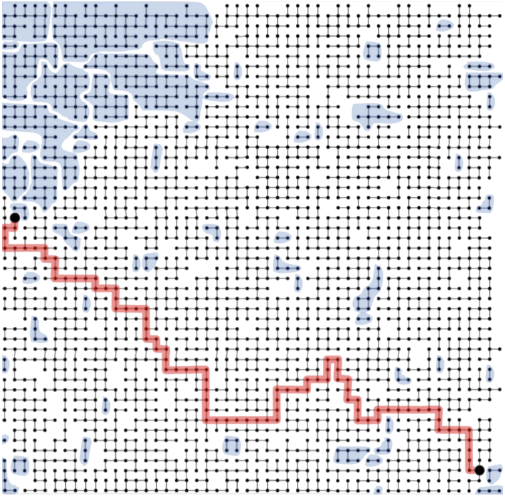
Can these two points communicate?



(source: R. Sedgewick and K. Wayne)

Are we connected?

YES! - (63 connected components)



(source: R. Sedgewick and K. Wayne)

Use Union-Find!

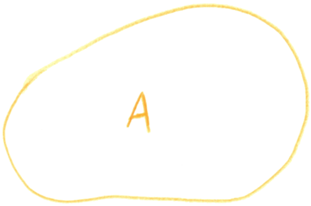
If we set up the Union-Find data structure when setting up the network configuration itself, **the answer of the question is in practice … constant!**

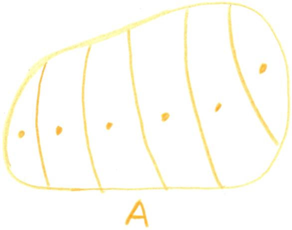
what is union-find?

union-find is a specialized data-structure to keep track of disjoint subsets of a set, say *A*, that form a partition of *A*

this implies the existence of a equivalence relation over *A* that defines the partition

as usual in equivalente classes, each partition is denoted by a **representative** element





operations over set partitions

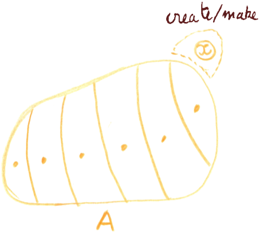
we expect three main operations (thus, efficiently implemented) over set partitions

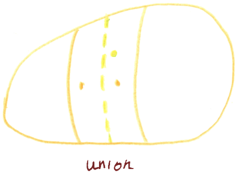
create s: creates a “singleton” partition $\{s\}$ and adds it to the partitions set.

find n: returns the representative element of the partition that contains n

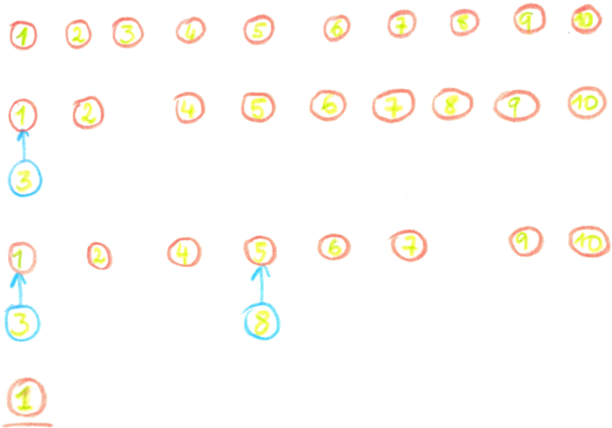
union a b: merge the partitions containing a and b (identity if a and b belongs to the same partition)

two elements are in the same partition (***in the same class***) if they have the same representative element





example of use



create 1 .. 10

union 1 3

union 5 8

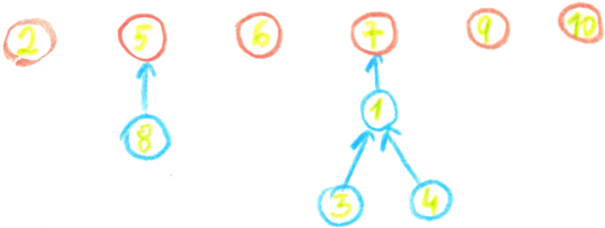
find 1

example of use

actual configuration

find 8

union 3 4



union 7 3

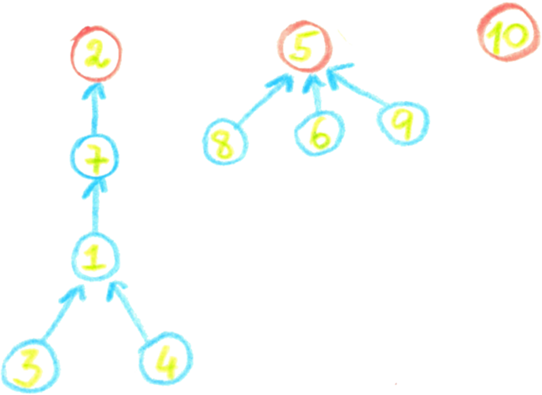
find 4

example of use

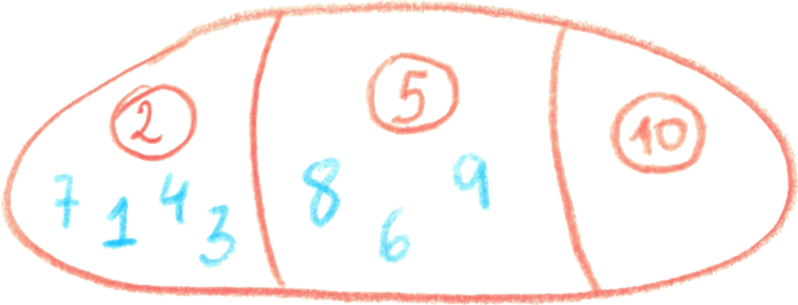
union 2 7

example of use

union 8 6
union 6 9



final configuration



implementation details

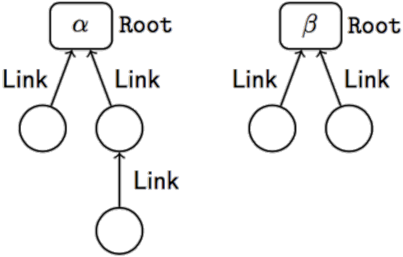
since a node has at most one parent, one can use a (resizable) array to encode the adjacency relation of the forest (seen as a graph)

1	2	3	4	5	6	7	8	9	10
7	2	1	1	5	5	2	5	5	10

this is a standard choice that implies the definition of a bijection of the set A to $\{1..n\}$ (with $|A| = n$)

or, we could opt for the more direct encoding as a collection of pointers to the following structure

```
type 'a content =
  | Link of 'a content ref
  | Root of int * 'a
```



(worst case) complexity of this union-find

cost model: number of (pointer) allocation, update and dereferencing, for a set of size N

create	union	find
1	N	N

actually, this not the cost of individual operation seen alone that matters here

typical use of Union-Find is as an auxiliary data-structure, ***as a service***

what matters is the overall behavior of a sequence of, say M, Union-Find operations (as witnessed by its use in the Kruskal Algorithm)

$\Longrightarrow$ **amortized complexity**

$$\mathcal{O}(M \times N)$$

(N of these M operations are `create` operations, the remaining operations have cost N - e.g. the cost of a sequence of *Unions* is quadratic)

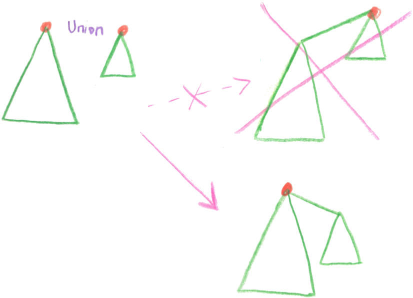
can we do better?

YES!

first: where is the bottleneck? in the size of underlying trees

can we do clever union when we know the size of each class? yes.

this is **union by rank**.



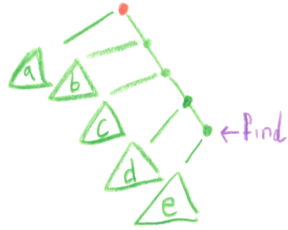
if we encode in each class representative the maximal depth of its underlying tree (... *its rank*) we can cleverly choose which class goes under the other

the representative element of the class with higher rank remains the representative of the union.

can we do even better?

YES!

each find operation go through the tree to the root



we can take this opportunity to branch all visited nodes directly to the root.

this is **find with path compression**

complexity analysis of union-find with rank and path compression

technically more challenging to characterize precisely

but surprisingly fast in terms of performance

M union-find operations over N elements (so N create operations)

$$\mathcal{O}(N + M \times \alpha(N))$$

where α is the **inverse Ackermann function**, which is a function that grows astoundingly slow

e.g. α(number of atoms in the whole universe) ≤ 5

in practice, we can consider that each union-find operation takes constant time!

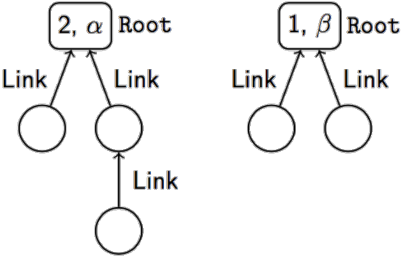
complexity epilogue - other data-structure with this amortized complexity: **splay tree** (balanced BST in which elements under use are pushed to the root)

correct-by-construction implementation using Vocal/Why3

union-find data-structure

```
type 'a content =
  | Link of 'a content ref
  | Root of int * 'a

type 'a elem = 'a content ref
```



union-find in Why3 : first attempt

```
type content 'a =
  | Link (ref (content 'a))
  | Root int 'a
```

```
Error:
This field has non-pure type, it cannot be used in a recursive
type definition
```

the Why3 type and effect system:

mutability of **bounded depth** ⟹ **all** aliases must be known **statically**

solution : built-in memory model

embed a custom memory model into the why3's logic :

- a type for memory pointers
- operations for pointer allocation, read and write
- an association table from pointers to their values

examples : Frama-C, Dafny, VeriFast, VCC, CFML

here we use the ***component-as-array*** memory model design technique [Burstall, 1972]

memory model for union-find (1/2)

```
type loc 'a

type content 'a =
  | Link (loc 'a)
  | Root int 'a

type memory 'a = {
  ghost mutable refs: loc 'a → option (content 'a);
}
```

here `None`/`Some` mean non-allocated/allocated

memory model for union-find (2/2)

```
val alloc (ghost mem: memory 'a) (v: content 'a) : loc 'a
  writes   { mem }
  ensures  { (old mem).refs result = None }
  ensures  { mem.refs = (old mem.refs)[result ← Some v] }

val set_ref (ghost mem: memory 'a) (l: loc 'a) (v: content 'a)
  requires { mem.refs l ≠ None }
  writes   { mem }
  ensures  { mem.refs = (old mem.refs)[l ← Some v] }
...
```

implementation: union-find data structure

```
type uf  'a = {
          memo: memory 'a;
  mutable dom : set (loc 'a);    (* which "pointers" are involved *)
  mutable rep : loc 'a → loc 'a;(* representative element *)
  mutable img : loc 'a → 'a;    (* representative element value *)
  mutable dst : loc 'a → int;   (* distance *)
  mutable maxd: int;
  }
  invariant { forall x. mem x dom → img x = img (rep x) }
  invariant { forall x. mem x dom → rep (rep x) = rep x }
  invariant { forall x. mem x dom → mem (rep x) dom }
  invariant { forall x y. mem x dom → mem y dom →
                rep x = rep y → img x = img y }
  invariant { forall x y. mem x dom → rep x = y → mem y dom }
  invariant { forall x. mem x dom ↔ allocated memo x }
  ...
```

implementation: union-find data structure

```
type uf  'a = {
          memo: memory 'a;
  mutable dom : set (loc 'a);    (* which "pointers" are involved *
  mutable rep : loc 'a → loc 'a;(* representative element *)
  mutable img : loc 'a → 'a;    (* representative element value *)
  mutable dst : loc 'a → int;   (* distance *)
  mutable maxd: int;
  } ...
 invariant { forall x. match memo.refs x with
               | Some (Link y)   → x ≠ y ∧ allocated memo y ∧
                                   rep x = rep y ∧ dst x < dst y
               | Some (Root r v) → img x = v ∧ rep x = x
               | None → true end }
  invariant { 0 ≤ maxd }
  invariant { forall x. mem x dom → dst x ≤ maxd }
  invariant { forall x. mem x dom → match memo.refs (rep x) with
               | Some (Root r _) → true
               | _               → false end }
```

implementation: union-find data operations

```
(* with path compression *)
let rec find (ghost uf: uf 'a) (x: loc 'a) : loc 'a
  requires { mem x uf.dom }
  writes   { uf.memo }
  variant  { uf.maxd - uf.dst x }
  ensures  { result = uf.rep x }
  ensures  { uf.dst result ≥ uf.dst x }
= match get_ref uf.memo x with
| Root _ _ → x
| Link y → let rx = find uf y in
            set_ref uf.memo x (Link rx);
            rx end
```

implementation: union-find data operations

```
let link (ghost uf: uf 'a)  (x y: loc 'a) : ghost loc 'a
 requires { mem x uf.dom }
 requires { x = uf.rep x }
 requires { mem y uf.dom }
 requires { y = uf.rep y }
 ensures  { (result = old (rep uf x)) || (result = old (rep uf y))
 ensures  { forall z. mem z uf.dom →
     rep uf z = if old (equiv uf z x || equiv uf z y) then result
                else old (rep uf z) }
 ensures  { forall z. mem z uf.dom →
     img uf z = if old (equiv uf z x || equiv uf z y) then img uf re
                else old (img uf z) }
```

implementation: union-find data operations

```
let union (ghost uf: uf 'a) (x y: loc 'a) : ghost loc 'a
  requires { mem x uf.dom }
  requires { mem y uf.dom }
  ensures  { result = old (rep uf x) || result = old (rep uf y) }
  ensures  { forall z. mem z uf.dom →
      rep uf z = if old (equiv uf z x || equiv uf z y) then result
                 else old (rep uf z) }
  ensures  { forall z. mem z uf.dom →
      img uf z = if old (equiv uf z x || equiv uf z y) then img uf r
                 else old (img uf z) }
= let a = find uf x in
  let b = find uf y in
  link uf a b
```

OCaml driver for union-find

```
(* custom driver for UnionFind_impl, to map the custom memory
    model  to OCaml references. *)

module UnionFind_impl.Mem
  syntax type loc      "(%1 content) ref"
  syntax function Link "Link %1"
  syntax function Root "Root (%1, %2)"
  syntax val  (==)     "%1 == %2"
  syntax val  (!=)     "%1 != %2"
  syntax val  alloc    "ref %1"
  syntax val  get_ref  "!%1"
  syntax val  set_ref  "%1 := %2"
end

module UnionFind_impl.Impl
 prelude "type 'a content = Link of 'a content ref
                           | Root of int * 'a"
end
```

conclusion

it works like a charm

how to generate a perfect maze of size $N \times N$?

use **union-find**!

DEMO

future work

obviously:

- more correct-by-construction data-structures and algorithms
- integration into critical software / client code

in the Vocal/CFML/COQ, the proof process is not automatic but it is possible to prove very subtle or complex properties, for instance about computational complexity

our plan to achieve this level of proof power can be divided in two points:

1. separation logic support in why3 (as a library), for a better systematic memory region / frame reasonning
2. integrate time credits techniques for checking computational complexity

Concluding Remark

Un peu de programmation éloigne de la logique mathématique;
beaucoup de programmation y ramène.

Xavier Leroy.