Mon programme est-il correct ?

Jean-Christophe Filliâtre
CNRS

CINE, 4 février 2010

Citation

Software is hard.

Don Knuth

Pourquoi

- les raisons sont nombreuses et complexes
  - mauvaise interprétation des spécifications
  - programmation dans l'urgence
  - changements incompatibles
  - etc.

- que peut-on faire ?

Une première réponse

de meilleurs langages de programmation

- meilleure **syntaxe**
  (évite de considérer `DO 17 I = 1. 10` comme une affectation)
- plus de **typage**
  (évite de confondre des mètres et des yards)
- plus d'**avertissements** du compilateur
  (évite d'oublier certains cas)
- etc.

Une autre réponse

le **test** systématique et rigoureux est une autre réponse, complémentaire

mais le test est

- coûteux
- parfois très difficile à mettre en œuvre
- et surtout **incomplet** (à de très rares exceptions près)

Une approche mathématique

---

les méthodes formelles

Qu'est-ce qu'un programme ?

il y a plusieurs aspects en jeux

- ce que l'on calcule (**quoi**)
- la manière de le calculer (**comment**)
- la raison pour laquelle c'est correct (**pourquoi**)

Qu'est-ce qu'un programme ?

le programme, ce n'est que le « **comment** », et rien d'autre

le « **quoi** » et le « **pourquoi** » n'en font pas partie

ce sont des cahiers des charges, des commentaires, des pages web, des articles de recherche, etc.

Un exemple

- **comment** : 2 lignes de 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;}
```

- **quoi** : 15 000 décimales de π

- **pourquoi** : beaucoup de maths, dont

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

Méthode formelle

les méthodes formelles proposent une approche rigoureuse de la programmation, où on se donne

- une **spécification** écrite dans un langage mathématique
- une **preuve** que le programme vérifie cette spécification

Quel langage / quels outils

on peut faire une preuve mathématique « traditionnelle »
(C. A. R. Hoare, *Proof a of program : Find*, 1971)

en général, cependant, on recourt à la **logique assistée par ordinateur**

Quelle spécification

que souhaite-t-on prouver ?

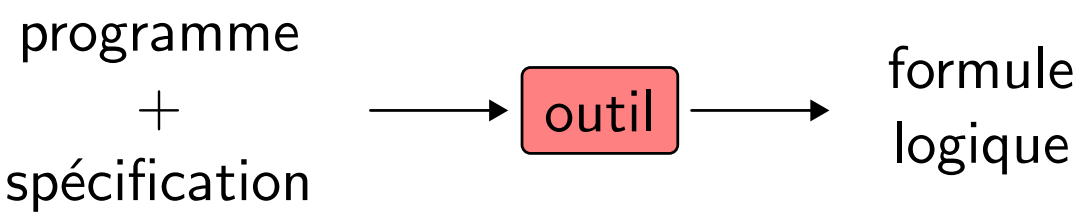
- **sûreté** : le programme ne « plante » pas
  - pas d'accès illégal à la mémoire
  - pas d'opération illégale (comme une division par zéro)
  - le programme termine

- **correction fonctionnelle**
  - le programme fait ce qu'il est censé faire

De nombreuses approches

on peut citer le model checking, l'interprétation abstraite, etc.
(cf INF563 par exemple)

cet exposé présente la **vérification déductive**



Exemple

```
int max(int x, int y) {
    if (x > y) return x;
    return y;
}
```

Exemple

```
//@ ensures \result >= x && \result >= y ;

int max(int x, int y) {
    if (x > y) return x ;
    return y ;
}
```

Exemple

```
//@ ensures \result >= x && \result >= y ;

int max(int x, int y) {
    if (x > y) return x ;
    return y ;
}
```

$$\forall x, \forall y, \quad x > y \Rightarrow x \geq x \wedge x \geq y$$
$$\wedge \quad x \leq y \Rightarrow y \geq x \wedge y \geq y$$

Un autre exemple

```
boolean search(int t[], int v) {
    for (int i = 0; i < t.length; i++)
        if (t[i] == v) return true;
    return false;
}
```

$$\forall t, \forall i, i < t.\texttt{length} \Rightarrow t \neq \texttt{null} \wedge 0 \leq i < t.\texttt{length}$$

Un autre exemple

```
//@ requires t != null ;
boolean search(int t[], int v) {
    for (int i = 0 ; i < t.length ; i++)
        if (t[i] == v) return true ;
    return false ;
}
```

$$\forall t, t \neq \texttt{null} \Rightarrow \forall i, i < t.\texttt{length} \Rightarrow t \neq \texttt{null} \wedge 0 \leq i < t.\texttt{length}$$

Invariant de boucle

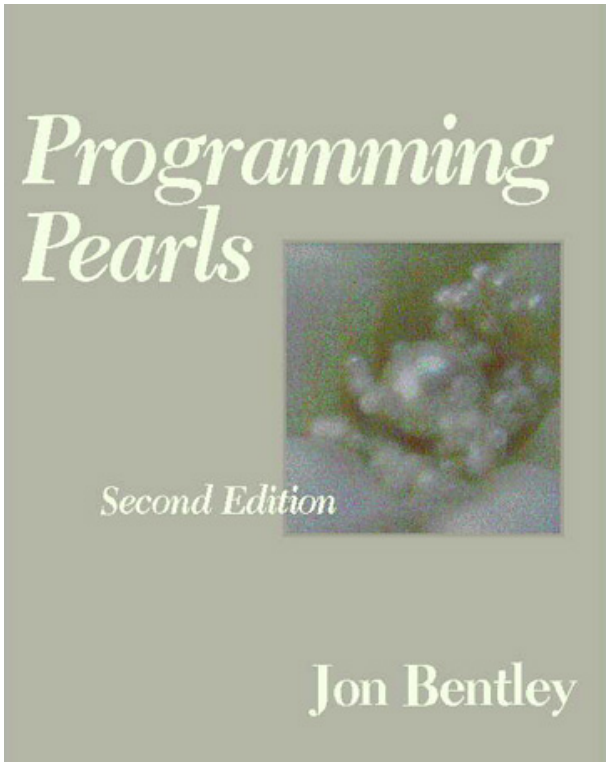
```
//@ requires t != null;
boolean search(int t[], int v) {
    //@ loop_invariant 0 <= i;
    for (int i = 0; i < t.length; i++)
        if (t[i] == v) return true;
    return false;
}
```

$$\forall t, t \neq \texttt{null} \Rightarrow 0 \leq 0$$

$$\forall t, t \neq \texttt{null} \Rightarrow \forall i, 0 \leq i \Rightarrow i < t.\texttt{length} \Rightarrow 0 \leq i + 1$$

$$\forall t, t \neq \texttt{null} \Rightarrow \forall i,\ 0 \leq i \Rightarrow i < t.\texttt{length} \Rightarrow t \neq \texttt{null} \wedge 0 \leq i < t.\texttt{length}$$

Un exemple célèbre



Writing correct programs

the challenge of binary search
(recherche dans un tableau trié)

première publication en 1946
première publication **sans bug** en 1962

démo

Binary Search

prouvons la sûreté d'exécution

```
//@ requires t != null ;
static int binary_search(int t[], int v) {
    int l = 0, u = t.length - 1 ;
    //@ loop_invariant 0 <= l && u <= t.length-1 ;
    while (l <= u ) {
        int m = (l + u) / 2 ;
        if (t[m] < v) l = m ;
        else if (t[m] > v) u = m ;
        else return m ;
    }
    return -1 ;
}
```

Binary Search

prouvons la terminaison

```
//@ requires t != null;
static int binary_search(int t[], int v) {
    int l = 0, u = t.length - 1;
    /*@ loop_invariant 0 <= l && u <= t.length-1;
      @ loop_variant   u - l;
      @*/
    while (l <= u ) {
        int m = (l + u) / 2;
        if (t[m] < v) l = m;
        else if (t[m] > v) u = m;
        else return m;
    }
    return -1;
}
```

Binary Search

on ne parvient pas à prouver la terminaison, et pour cause !

avec par exemple `l = 0`, `u = 1`, et `t[0] < v`, le programme boucle

on peut rectifier ainsi

```
...
if (t[m] < v) l = m + 1 ;       // et non pas l = m
else if (t[m] > v) u = m - 1 ; // et non pas u = m
...
```

Binary Search

prouvons maintenant la **correction fonctionnelle**

un sens : si le résultat est positif ou nul, il indique une occurrence de `v`

```
/*@ requires t != null;
  @ ensures  \result >= 0 ==> t[\result] == v;
  @*/
static int binary_search(int t[], int v) {
    ...
}
```

la preuve est immédiate, sans annotation supplémentaire

Binary Search

l'autre sens : si le résultat vaut -1 alors `v` n'apparaît pas dans `t`

$\Rightarrow$ il faut maintenant indiquer que le tableau est trié

```
/*@ requires
  @   t != null &&
  @   \forall integer k1 k2 ;
  @       0 <= k1 <= k2 <= t.length-1 ==> t[k1] <= t[k2] ;
  @ ensures
  @   (\result >= 0 && t[\result] == v) ||
  @   (\result == -1 && \forall integer k ;
  @                       0 <= k < t.length ==> t[k] != v) ;
  @*/
static int binary_search(int t[], int v) {
    ...
}
```

Binary Search

cela ne suffit pas : il faut aussi renforcer l'invariant de boucle

```
static int binary_search(int t[], int v) {
    int l = 0, u = t.length - 1;
    /*@ loop_invariant
      @   0 <= l && u <= t.length-1 &&
      @   \forall integer k;
      @      0 <= k < t.length ==> t[k] == v ==> l <= k <= u;
      @ ...
      @*/
    ...
}
```

Binary Search

montrons enfin qu'il n'y a pas de débordement arithmétique
$\Rightarrow$ on trouve un autre **bug** ! en effet, le calcul de `l + u` dans

```
int m = (l + u) / 2;
```

peut provoquer un débordement de capacité (sur une machine 32 bits, avec plus d'un milliard d'éléments dans `t`)

on peut corriger ainsi

```
int m = l + (u - l) / 2;
```

Binary Search

ce bug a été trouvé en 2004 dans la bibliothèque Java (`java.util.Arrays.binarySearch`), au bout de 9 ans seulement

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

Un autre exemple

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

Un peu de clarté

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int main(int q) {
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

Le même code en Java

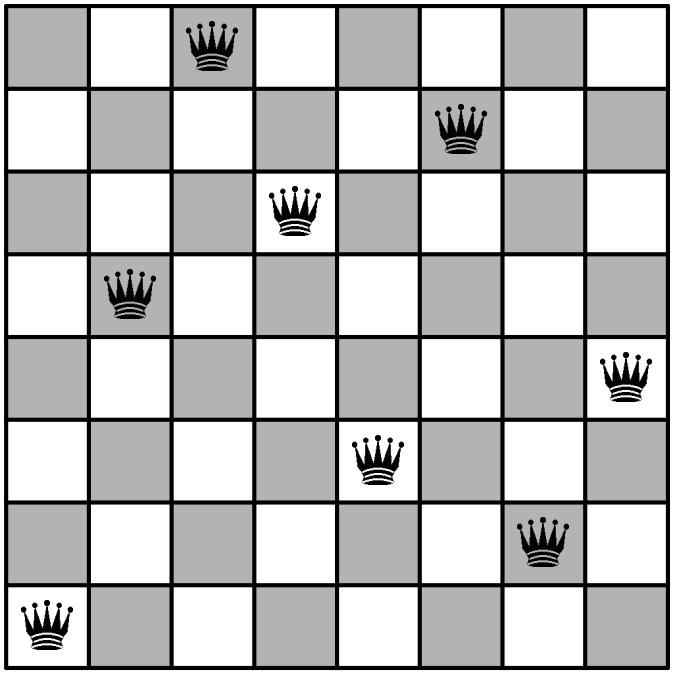
```
static int t(int a, int b, int c) {
    int d, e=a&~b&~c, f=1;
    if (a != 0)
        for (f=0; (d=e&-e) != 0; e-=d)
            f += t(a-d, (b+d)*2, (c+d)/2);
    return f;
}

public static void main(String arg[]) {
    int q = Integer.parseInt(arg[0]);
    System.out.println(t(~(~0<<q), 0, 0));
}
```

c'est donc une fonction f de $\mathbb{N}$ dans $\mathbb{N}$

Les n reines

c'est le nombre de solutions du problème des n reines

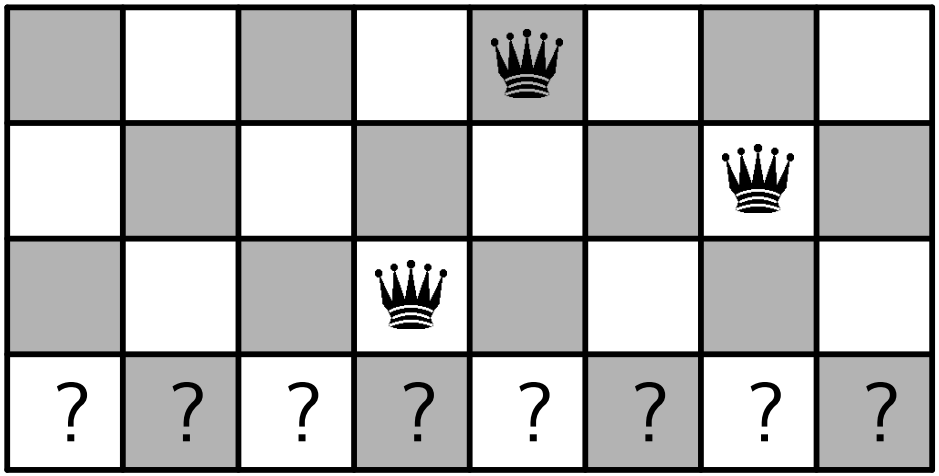
exemple : $f(8) = 92$



Principe

on procède par force brute

pour chaque rangée, on essaye successivement toutes les positions encore possibles

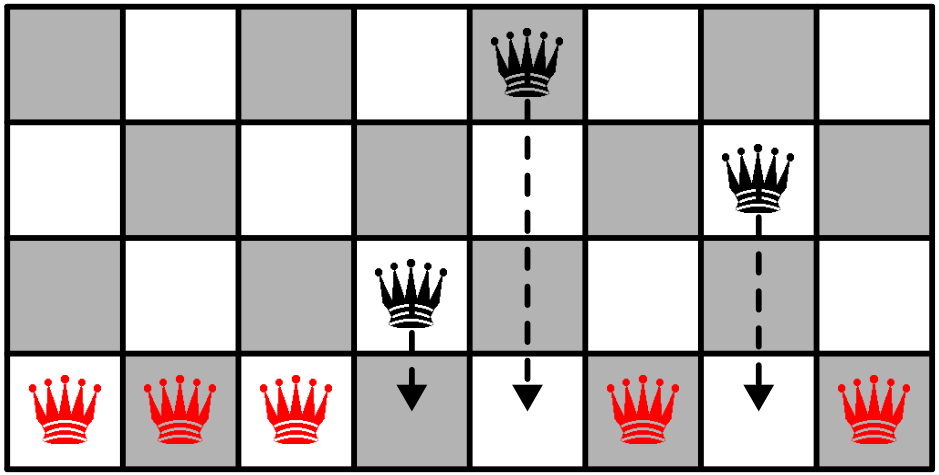


pour chacune, on passe à la rangée suivante, récursivement

Principe

```
static int t(int a, int b, int c) { ...
```

exemple : a=229, b=104, c=9

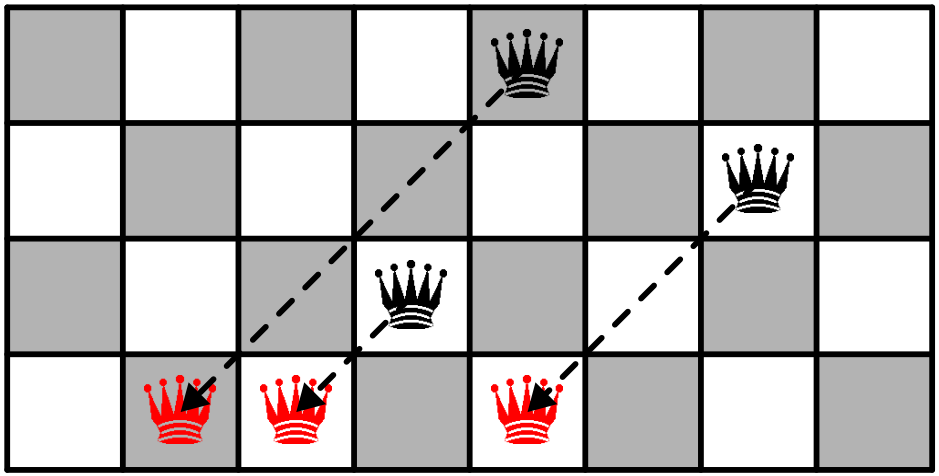


a = colonnes à remplir = 229 = 11100101_2

Principe

```
static int t(int a, int b, int c) { ...
```

exemple : a=229, b=104, c=9

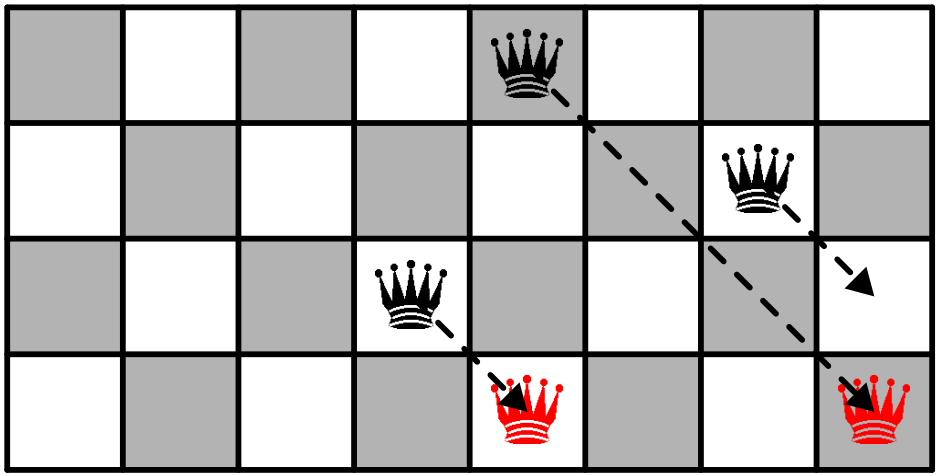


b = positions interdites (1/2) = 104 = 01101000_2

Principe

```
static int t(int a, int b, int c) { ...
```

exemple : a=229, b=104, c=9

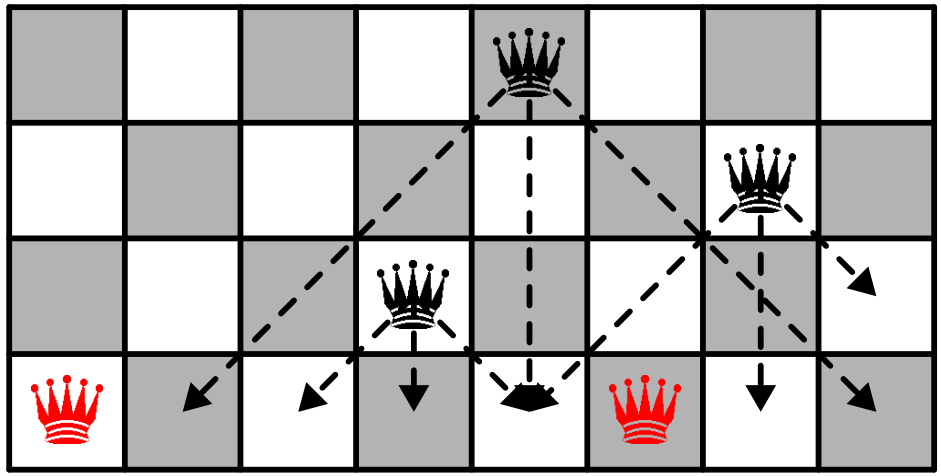


c = positions interdites (2/2) = 9 = 00001001_2

Principe

```
static int t(int a, int b, int c) {
  int e = a & ~b & ~c;
  ...
```

exemple : a=229, b=104, c=9



a & ~b & ~c $= 229$ & ~104 & ~9 $= 132$

$= 11100101_2$ & 10010111_2 & $11110110_2 = 10000100_2$

= positions restant à essayer

Principe

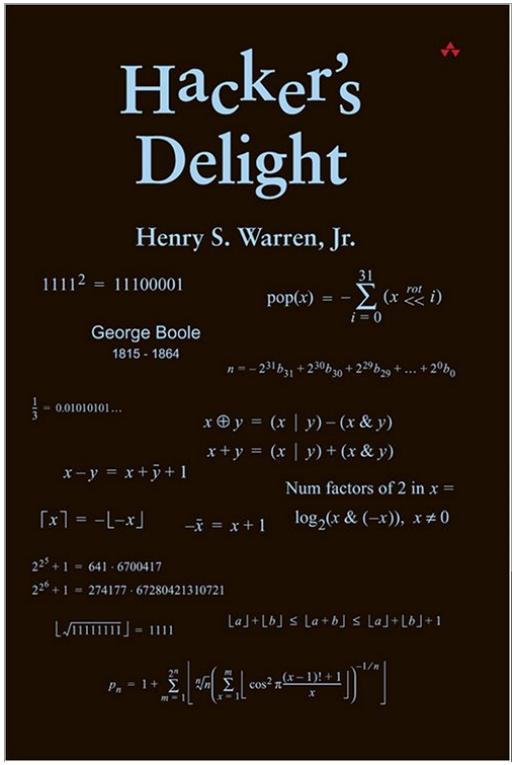
ce programme exploite astucieusement la représentation en base 2

`e & -e` calcule le plus petit bit à 1 de `e`,

donc la boucle

```
for ( ; (d=e&-e) != 0 ; e-=d)
   ...
```

parcourt tous les bits à 1 de `e`



Vision ensembliste

on peut voir les entiers comme des ensembles

entiers	ensembles
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, quand $a \cap b = \emptyset$
`a-b`	$a \setminus b$, quand $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min(a)$, quand $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i + 1 \mid i \in a\}$
`a/2`	$\{i - 1 \mid i \in a \wedge i \neq 0\}$

Vision ensembliste

int $t(a,\ b,\ c)$
    if $a = \emptyset$ return 1
    $f \leftarrow 0$
    for each d in $(a \backslash b) \backslash c$
        $f \leftarrow f + t(a \backslash \{d\},\ \{x + 1 \mid x \in b \cup \{d\}\},\ \{x - 1 \mid x \in c \cup \{d\}\})$
    return f

int $queens(n)$
    return $t(\{0, 1, \ldots, n - 1\},\ \emptyset,\ \emptyset)$

Questions

on peut montrer facilement que ce programme est sûr et termine

mais peut-on donner une spécification fonctionnelle à ce programme et montrer qu'il est correct ?

Spécification

une difficulté : le programme ne fait que calculer le **nombre** de solutions

une solution : on va **stocker les solutions** qui sont implicitement parcourues par le programme, pour pouvoir en parler

ce n'est pas la seule solution, mais elle est naturelle pour le programmeur

Code fantôme

on ne souhaite pas modifier le programme, mais juste l'observer

pour cela, on va utiliser du **code fantôme** :

- peut lire les données du programme
- ne peut pas les modifier
- peut modifier des données fantômes

Code fantôme

on introduit des variables fantômes

- le nombre de reines

```
//@ ghost static int N;
```

- les solutions trouvées

```
//@ ghost static int sol[][];
```

- le prochain emplacement libre dans `sol`

```
//@ ghost static int s = 0;
```

- la solution en cours de construction

```
//@ ghost static int col[];
```

- la rangée courante dans `col`

```
//@ ghost static int k = 0;
```

Stocker les solutions

on modifie ensuite le programme pour qu'il stocke les solutions

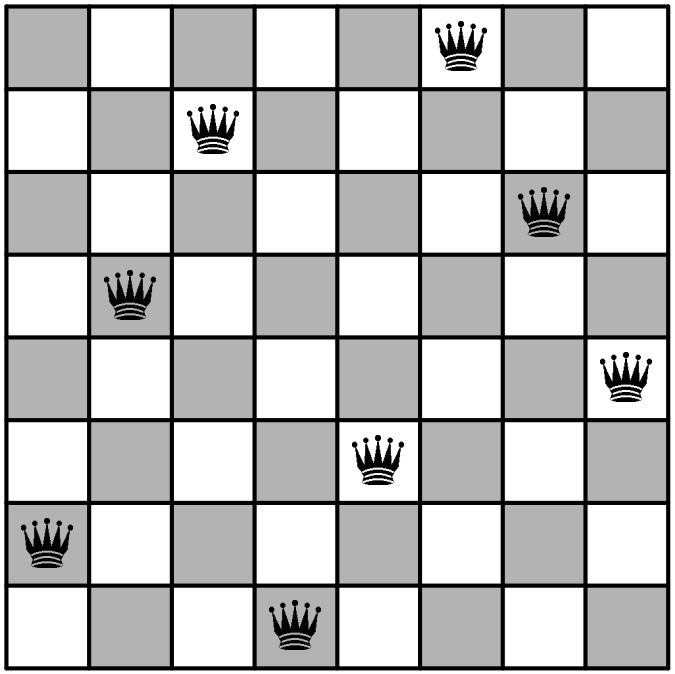
```
int t(int a, int b, int c){
  int d, e=a&~b&~c, f=1;
  if (a != 0)
    for (f=0; (d=e&-e) != 0; e-=d) {
      // ghost col[k] = log2(d);
      // ghost k++;
      f += t3(a-d, (b+d)*2, (c+d)/2);
      // ghost k--;
    }
  // ghost else {
  //   for (int i = 0; i < N; i++) sol[s][i] = col[i]; s++;
  // }
  return f;
}
```

Ordre

on peut ordonner les solutions par ordre lexicographique

coïncide avec l'ordre de découverte

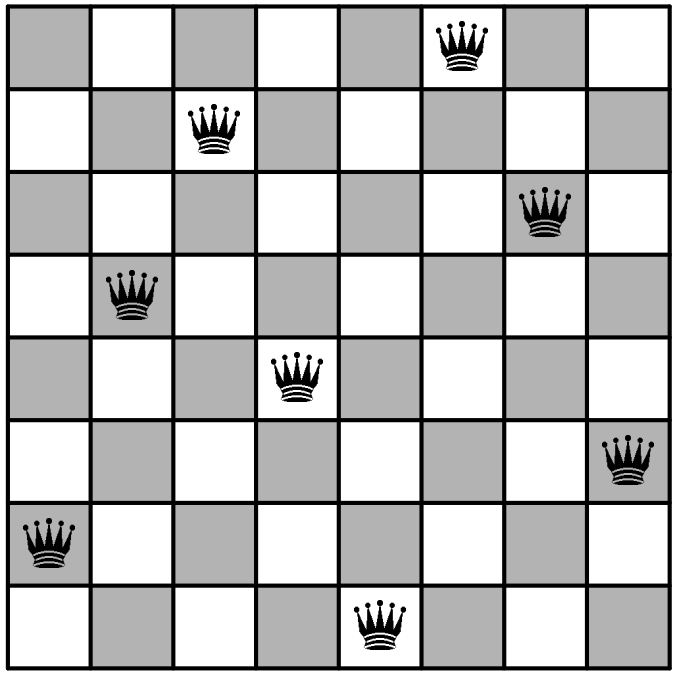
6-ième solution



1 ; 4 ; 6 ; 0 ; 2 ; 7 ; 5 ; 3

$<$

7-ième solution



1 ; 4 ; 6 ; 3 ; 0 ; 7 ; 5 ; 2

Code fantôme

on peut alors exprimer qu'on trouve

- uniquement des solutions
- toutes les solutions
- chaque solution une seule fois

```
/*@ ensures
  @   sorted(sol, 0, \result) &&
  @   \forall int t[] ;
  @      solution(t) <==>
  @      (\exists integer i ;
  @         0 <= i < \result && eq_sol(t, sol[i])) ;
  @*/
int queens(int n) {
  return t(~(~0<<n),0,0) ;
}
```

Au final

```
/*@ requires
  @   0 <= k && k + card(iset(a)) == N() && 0 <= s &&
  @   pre_a : : (\forall int i ; in_(i,iset(a)) <==>
  @            (0<=i<N() && \forall int j ; 0<=j<k ==> i != col[j])) &&
  @   pre_b : : (\forall int i ; i>=0 ==> (in_(i,iset(b)) <==>
  @            (\exists int j ; 0<=j<k && col[j] == i+j-k))) &&
  @   pre_c : : (\forall int i ; i>=0 ==> (in_(i,iset(c)) <==>
  @            (\exists int j ; 0<=j<k && col[j] == i+k-j))) &&
  @   partial_solution(k, col) ;
  @ assigns
  @   col[k..], s, k, sol[s..][..] ;
  @ ensures
  @   \result == s - \old(s) && \result >= 0 && k == \old(k) &&
  @   sorted(sol, \old(s), s) &&
  @   \forall int t[] ; ((solution(t) && eq_prefix(col,t,k)) <==>
  @              (\exists int i ; \old(s)<=i<s && eq_sol(t, sol[i]))) ;
  @*/
int t(int a, int b, int c){
  int d, e=a&~b&~c, f=1 ;
  //@ label L
  if (a != 0)
    /*@ invariant
      @   included(iset(e),\at(iset(e),L)) &&
      @   f == s - \at(s,L) && f >= 0 && k == \old(k) &&
      @   partial_solution(k, col) &&
      @   sorted(sol, \at(s,L), s) &&
      @   \forall int *t ;
      @     (solution(t) &&
      @      \exists int di ; in_(di, diff(\at(iset(e),L), iset(e))) &&
      @          eq_prefix(col,t,k) && t[k]==di) <==>
      @     (\exists int i ; \at(s,L)<=i<s && eq_sol(t, sol[i])) ;
      @ loop_assigns
      @   col[k..], s, k, sol[s..][..] ; */
    for (f=0 ; (d=e&-e) != 0 ; e-=d) {
      // ghost col[k] = min_elt(d) ;
      // ghost k++ ;
      f += t(a-d, (b+d)*2, (c+d)/2) ;
      // ghost k-- ;
    }
  // ghost else {
  //   for (int i = 0 ; i < N ; i++) sol[s][i] = col[i] ; s++ ;
  // }
  return f ;
}
```

```
/*@ requires
  @   n == N() && s == 0 && k == 0 ;
  @ ensures
  @   \result == s &&
  @   sorted(sol, 0, s) &&
  @   \forall int t[] ;
  @      solution(t) <==>
  @      (\exists int i ; 0<=i<\result && eq_sol(t,sol[i])) ;
  @*/
int queens(int n) {
  return t(~(~0<<n),0,0) ;
}
```

La preuve

- la fonction principale : **15** buts à prouver
  - **tous** prouvés automatiquement
- fonction récursive `t` : **51** buts à prouver
  - **42** prouvés automatiquement
  - **9** prouvés avec un assistant de preuve (Coq ; cf INF565)

conclusion

Un peu d'anthropomorphisme

on manque de respect pour les ordinateurs

on leur dit ce qu'il faut faire, sans expliquer ce que l'on cherche à obtenir, et encore moins pourquoi il est justifié de procéder ainsi

avec un peu plus de bonne volonté de notre part, on peut obtenir beaucoup

Conclusion

la vérification déductive

- permet de garantir la **sûreté** d'exécution
- mais aussi le respect de **spécifications** arbitrairement complexes

- est assistée par des **outils de démonstration** automatique et/ou interactive

c'est encore un processus coûteux, mais qui se simplifie et se démocratise chaque jour un peu plus
(exemple : SPEC# dans Visual Studio depuis 2008)