Université Paris-Saclay
M2 Droit de la création et du numérique

Approche de l'élaboration et du fonctionnement des logiciels

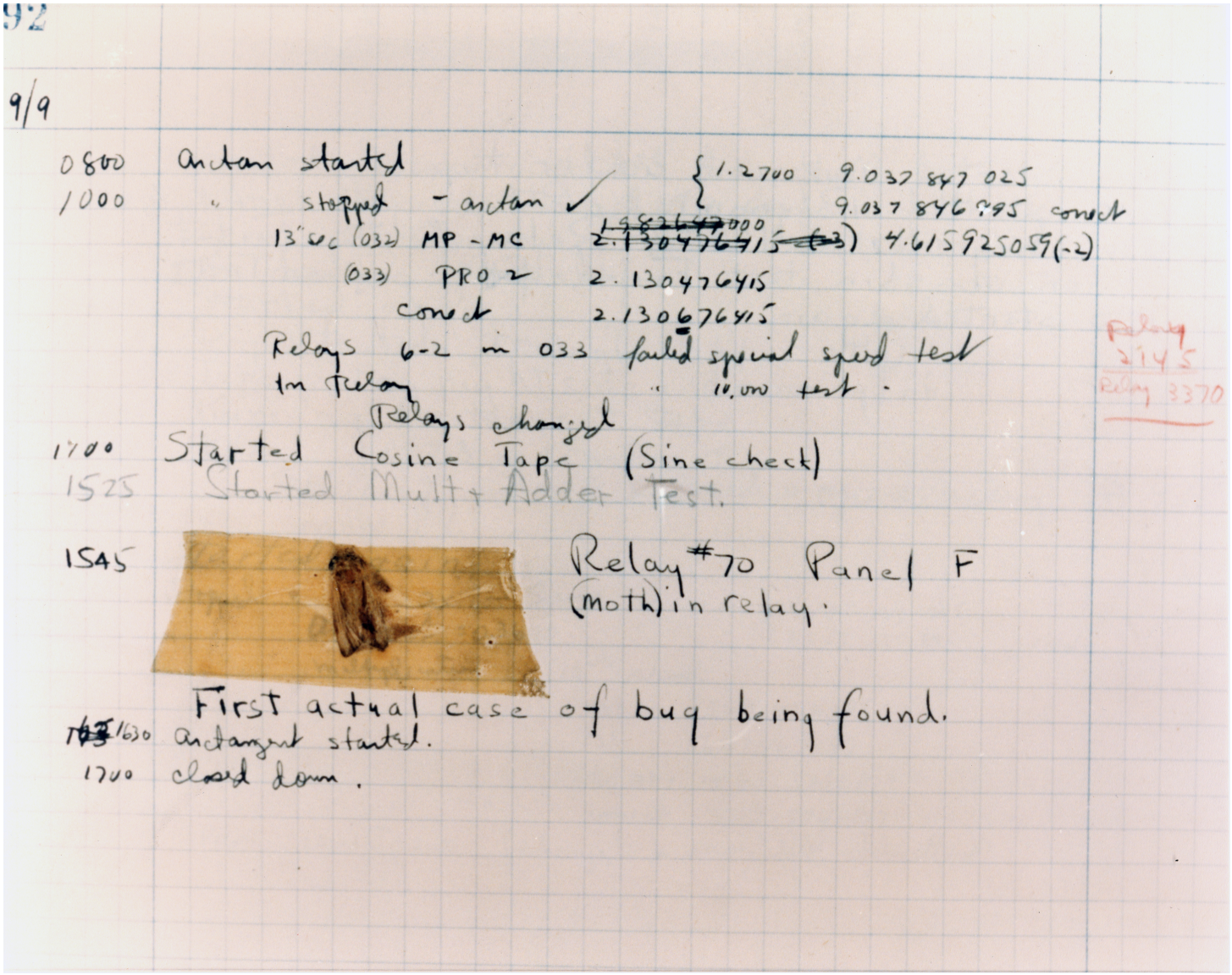
Alain Delaët

la chasse aux bugs

1. qu'est-ce qu'un bug ?
2. que peut-on faire ?

qu'est-ce qu'un bug ?

origine du terme

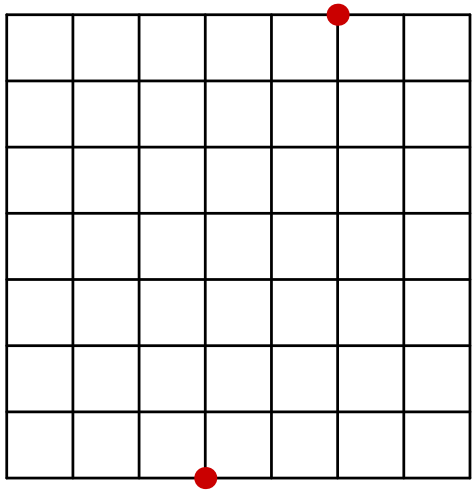


panne du Harvard Mark 2, journal de Grace Hopper (1946)

plus généralement, un **bug** est une erreur dans un programme informatique, entraînant un dysfonctionnement
quelques exemples réels :

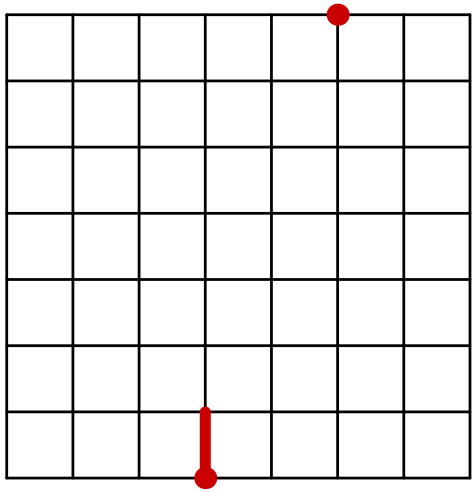
- panne du réseau téléphonique AT&T aux États-Unis
- un processeur qui ne calcule pas correctement et qui doit être rappelé
- une fusée qui explose peu après son décollage
- une sonde spatiale qui rate sa mise en orbite autour de Mars
- des patients irradiés

AAGADAGADADAAAGADAGAA



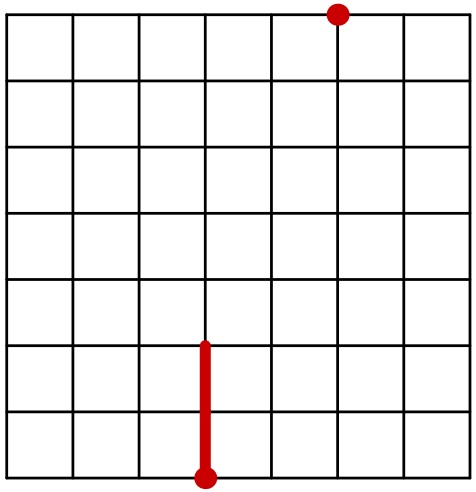
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



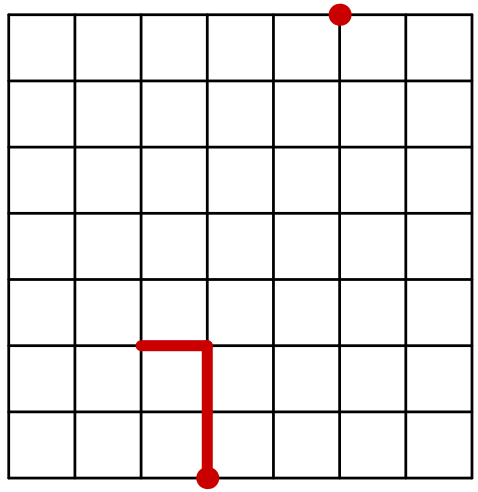
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



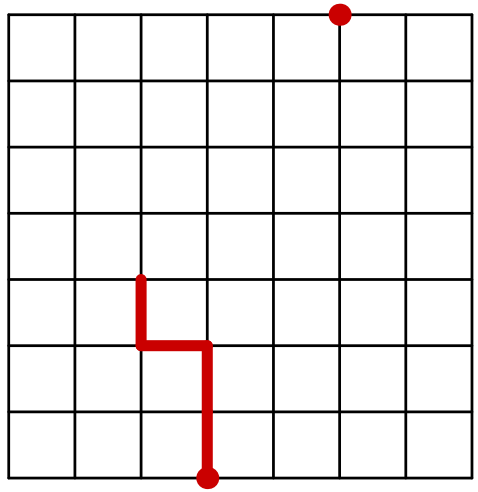
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



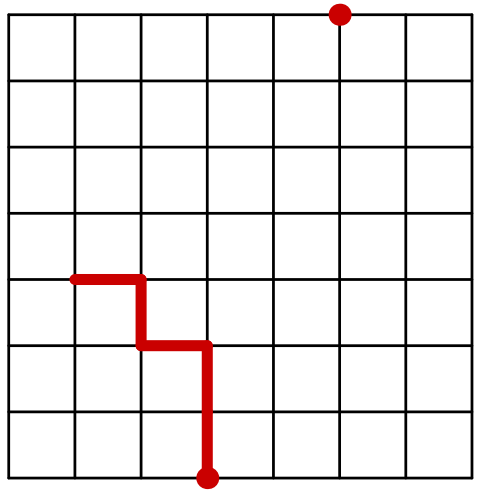
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



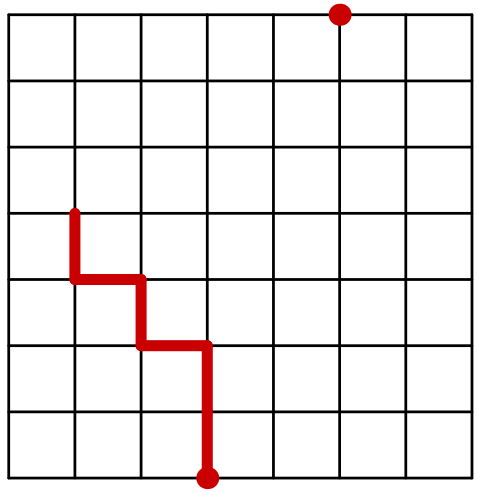
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



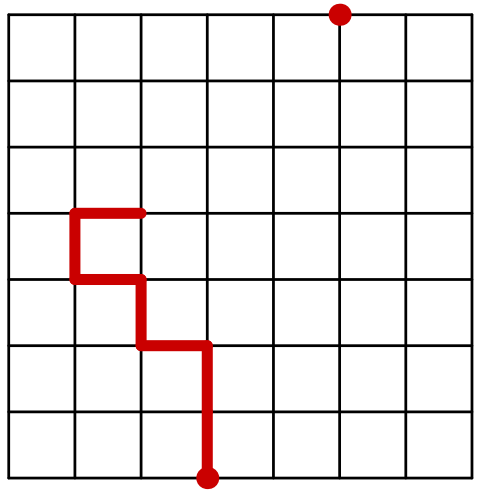
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



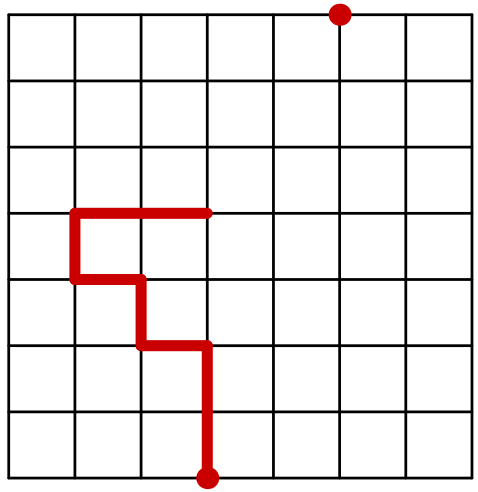
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



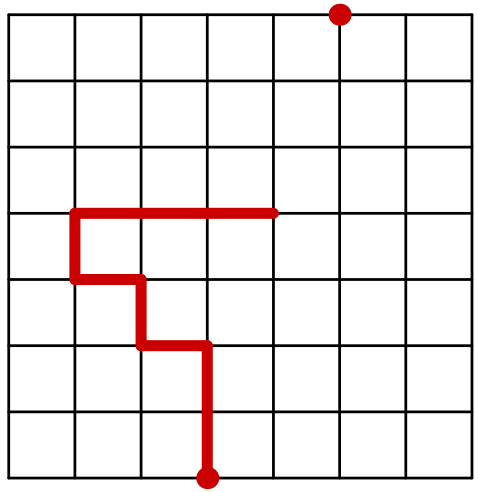
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



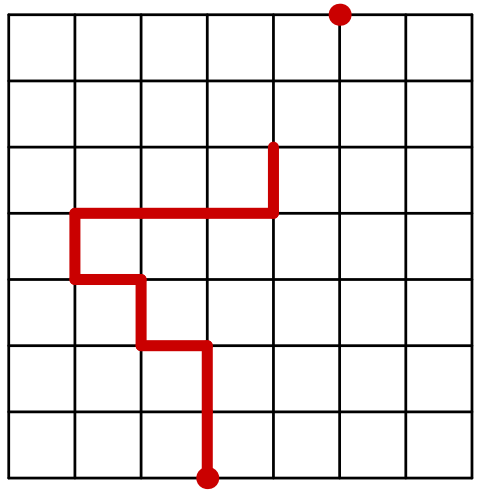
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



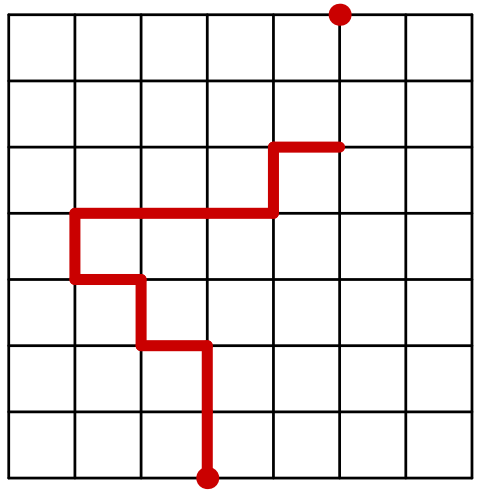
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



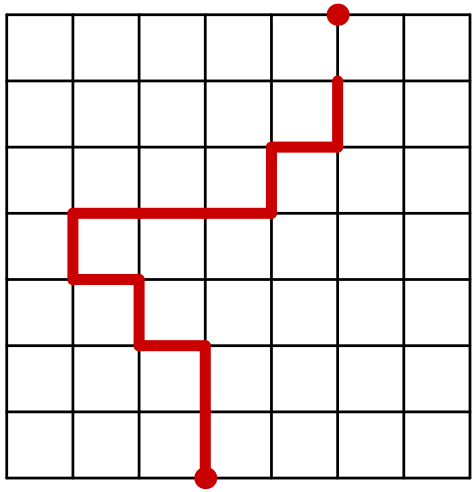
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



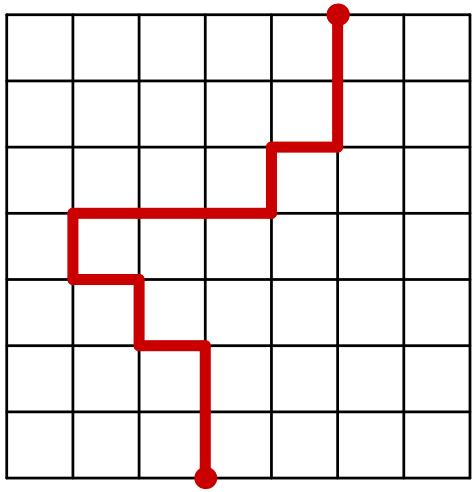
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



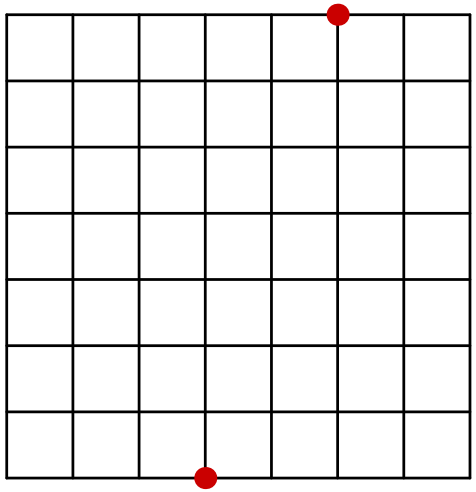
(sur une idée de Gérard Berry)

AAGADAGADADAAAGADAGAA



(sur une idée de Gérard Berry)

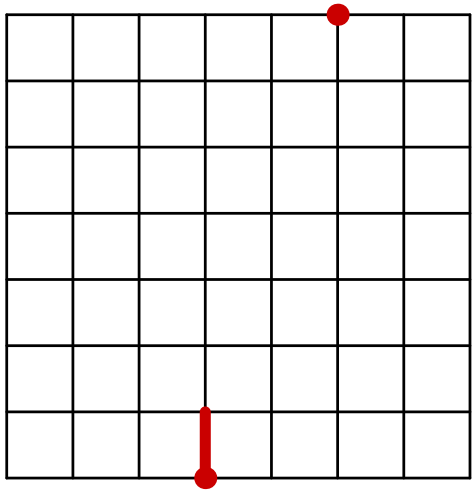
AAGADAGADADAAA**D**ADAGAA



(sur une idée de Gérard Berry)

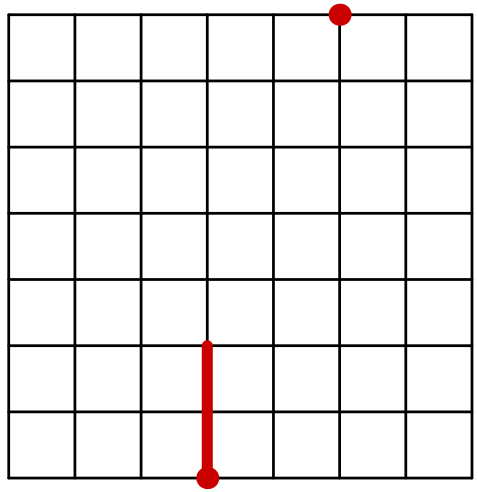
une typo malheureuse

AAGADAGADADAAA**D**ADAGAA



(sur une idée de Gérard Berry)

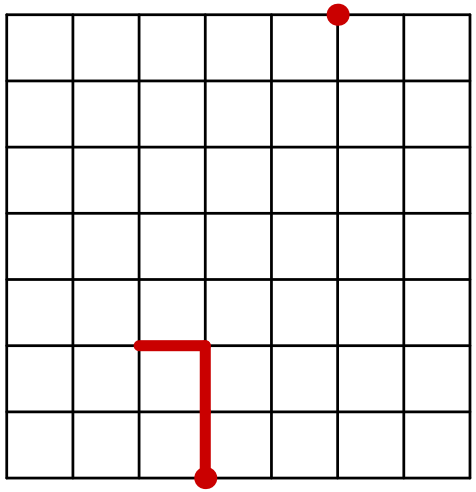
AAGADAGADADAAA**D**ADAGAA



(sur une idée de Gérard Berry)

une typo malheureuse

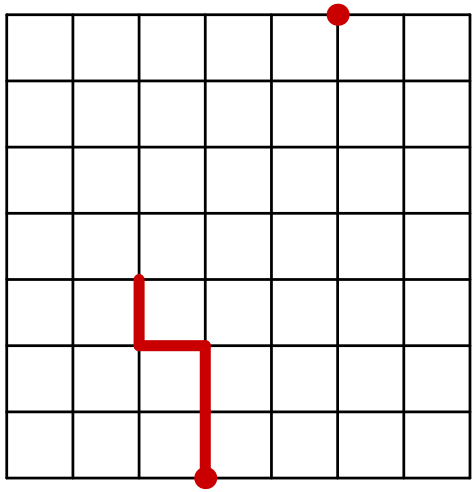
AAGADAGADADAAA**D**ADAGAA



(sur une idée de Gérard Berry)

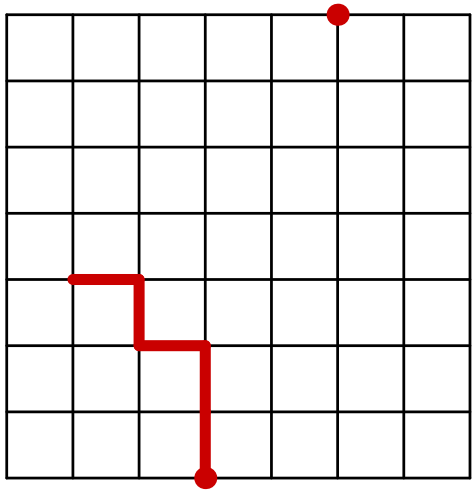
une typo malheureuse

AAGADAGADADAAA**D**ADAGAA



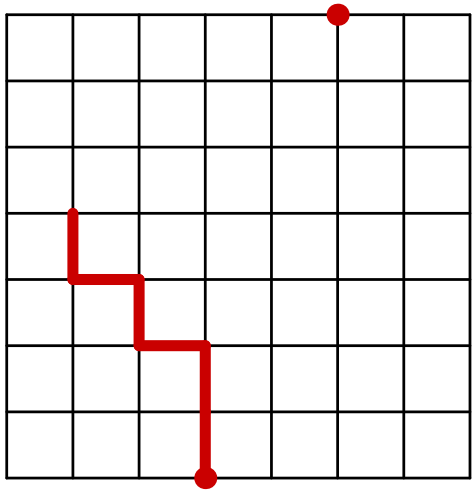
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



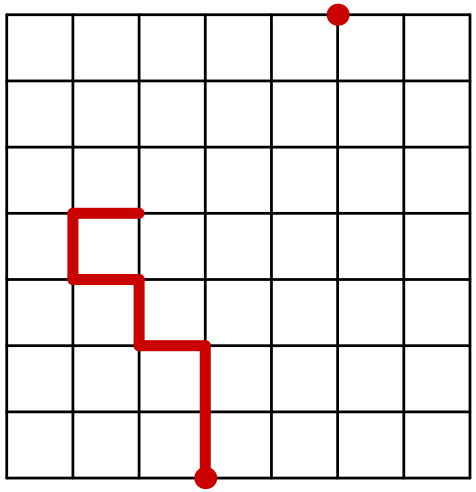
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



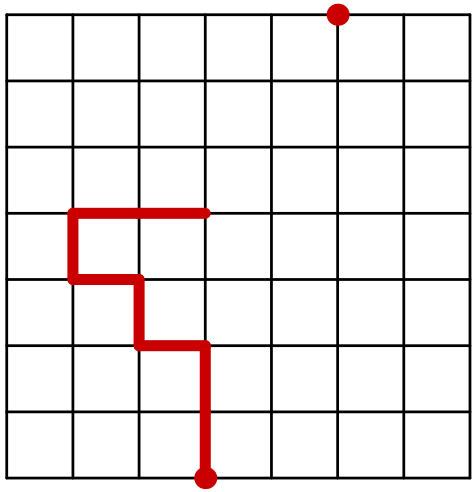
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



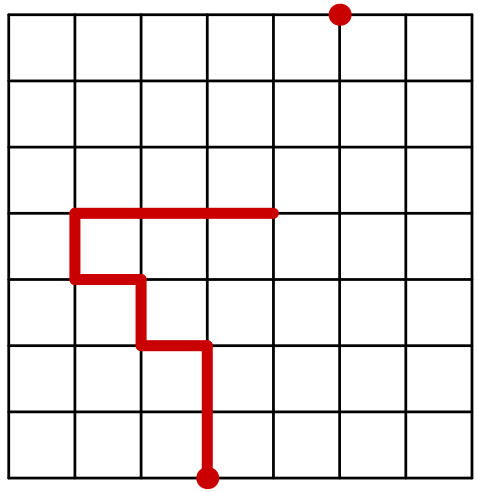
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



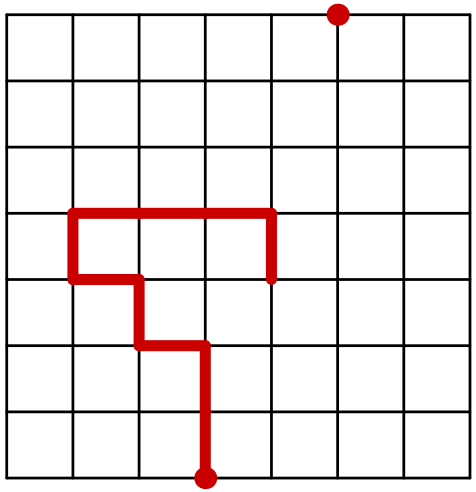
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



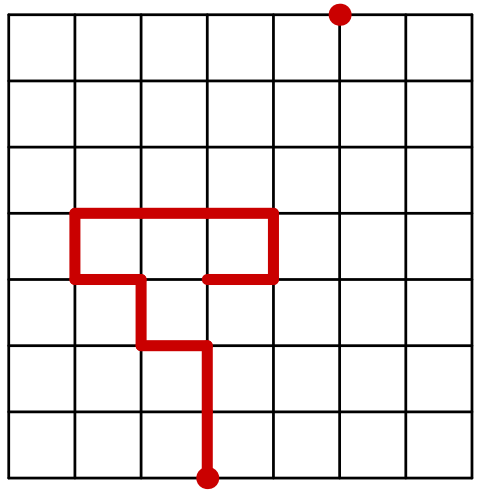
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



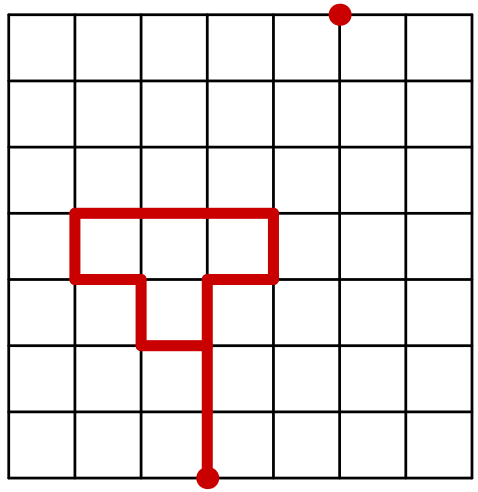
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



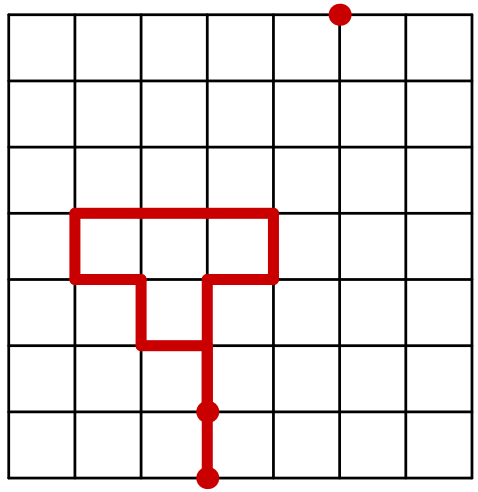
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



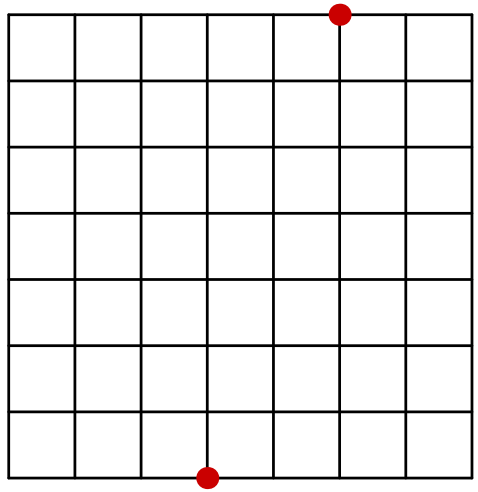
(sur une idée de Gérard Berry)

AAGADAGADADAAA**D**ADAGAA



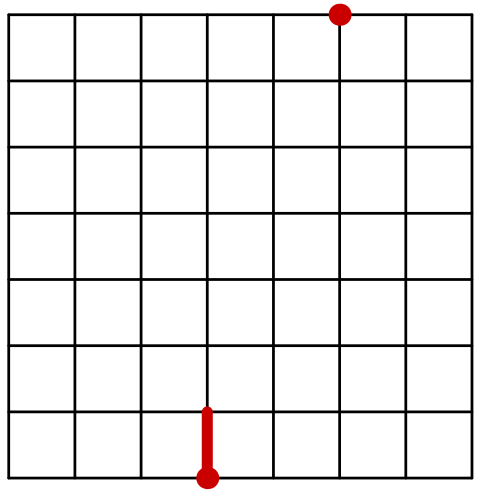
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



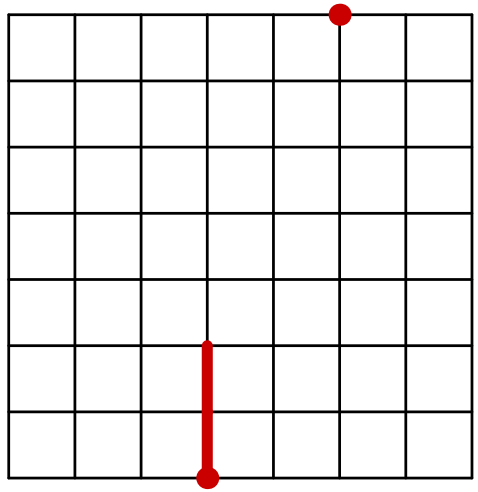
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



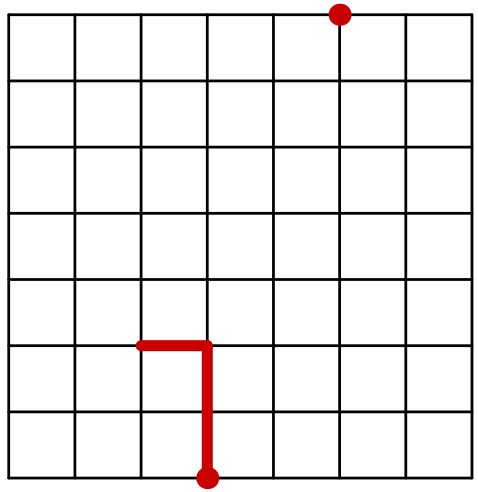
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



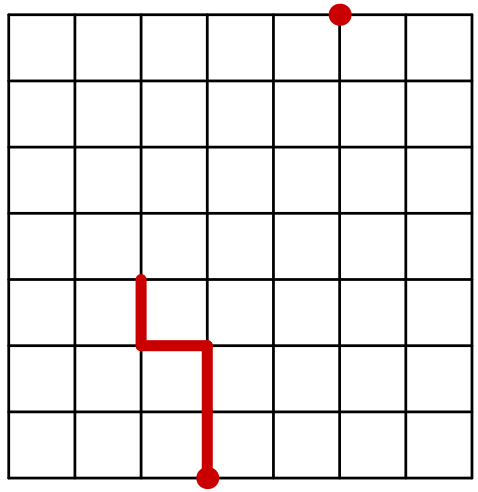
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



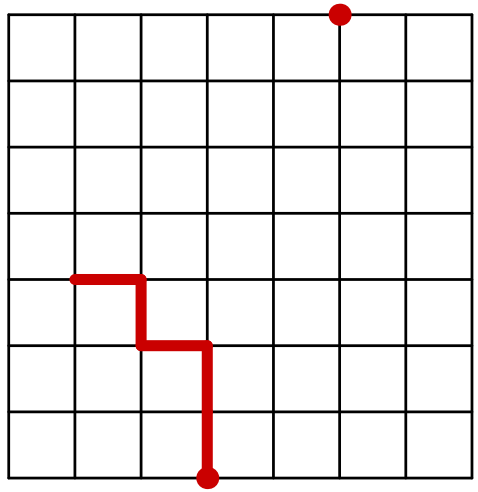
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



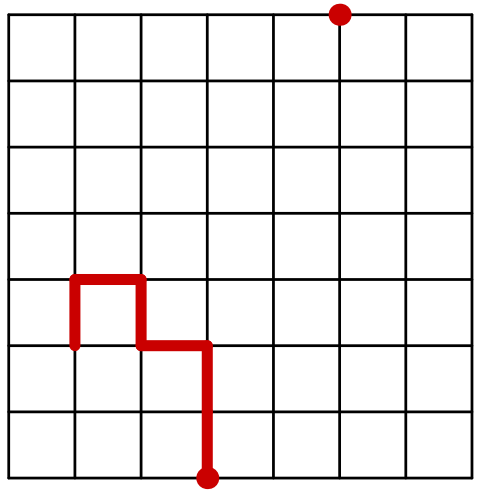
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



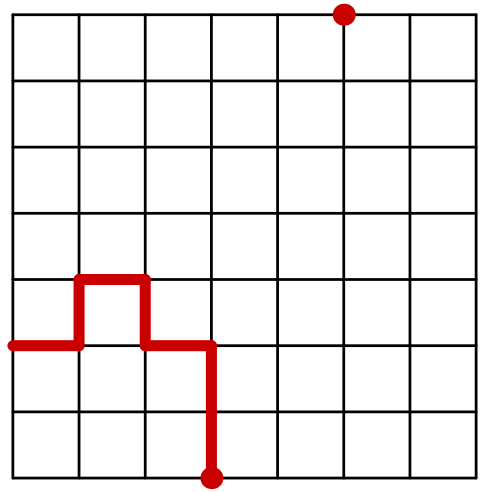
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



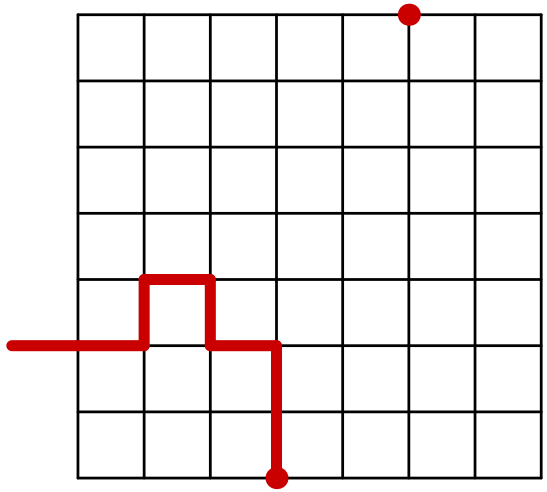
(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



(sur une idée de Gérard Berry)

AAGADAGA**G**ADAAAGADAGAA



(sur une idée de Gérard Berry)

une seule erreur dans un programme, même d'un unique caractère, peut conduire

- à un programme qui ne **s'exécute plus** du tout
  - problème surtout pour le programmeur
- à un programme qui finit par "**planter**"
  - gênant, possiblement catastrophique
- à un programme qui donne le **mauvais résultat**
  - gênant, possiblement insoupçonné, possiblement catastrophique

le système d'exploitation Linux, c'est **28 millions** de lignes de code
en comparaison, le module de commande Apollo qui a emmené les hommes sur la lune, c'est seulement 2 millions de pièces

que peut-on faire ?

en premier lieu, on peut adopter de **bonnes pratiques**, comme

- utiliser des **noms pertinents** pour les fonctions, paramètres, variables, etc.
- utiliser des **commentaires** dans le code source, pour expliquer tout ce qui n'est pas évident
- **relire** son code, le faire relire

```
...
d = date("1957.100481")
...
```

```
...
d = date("1957.100481") # lancement Spoutnik 1
# le 4 octobre 1957 à 19:28
...
```

```
...
lancement_Spoutnik_1 = date("1957.100481")
# 4 octobre 1957 à 19:28
...
```

parfois, le commentaire explique ce que fait le programme, ou ce que fait une fonction

```
    # nombre de jours dans le mois m dans l'année a
    def nbjoursmois(a, m):
    ...
```

on appelle cela une **spécification**

en Python, on peut associer une chaîne de **documentation** à toute fonction, et la consulter avec la commande `help`
exemple :

```
help(randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
Return random integer in range [a, b], including both end
points.
```

on peut le faire sur nos propres fonctions

```
    def nbjoursmois(a, m):
    """nombre de jours dans le mois m de l'année a"""
    ...
```

parfois, les programmes font des suppositions sur leurs entrées, ou les fonctions font des suppositions sur leurs paramètres
on appelle cela une **précondition**
exemple :

```
def nbjoursmois(a, m):
"""Nombre de jours dans le mois m de l'année a.
Le mois m est compris entre 1 et 12."""
...
```

précondition non satisfaite

que faire si la précondition n'est pas satisfaite ?

- l'exécution se poursuit
  - et possiblement le résultat est incorrect
  - et possiblement le programme plante plus loin

```
IndexError: list index out of range
```

- le programme signale une erreur et s'interrompt

```
le mois doit être entre 1 et 12
```

- le programme signale une erreur et poursuit son exécution

```
def nbjoursmois(a, m):
"""Nombre de jours dans le mois m de l'année a.
Le mois m est compris entre 1 et 12."""
if m < 1 or m > 12:
exit("le mois doit être entre 1 et 12")
...
```

il existe même une construction Python pour cela

```
def nbjoursmois(a, m):
"""Nombre de jours dans le mois m de l'année a.
Le mois m est compris entre 1 et 12."""
assert 1 <= m <= 12
...
```

- si la condition est vérifiée, le programme continue
- sinon, il s'interrompt en signalant le problème

```
File "jours.py", line 30, in nbjoursmois
assert 1 <= m <= 12
AssertionError
```

le test

pour s'assurer du bon fonctionnement d'un logiciel, on peut le **tester**
on procède à des exécutions variées, en comparant les résultats avec des résultats attendus

on l'a fait pendant la dernière séance

```
    print(nbjoursmois(2016, 2)) # 29
    print(nbjoursmois(2015, 2)) # 28
```

```
assert nbjoursmois(2016, 2) == 29
assert nbjoursmois(2015, 2) == 28

assert nbjours(1, 1, 2000, 1, 1, 2001) == 366
```

idéalement, on cherche à tester les différents composants d'un logiciel un par un, indépendamment les uns des autres
on appelle cela le **test unitaire**
mais c'est rapidement très difficile, voire impossible

les tests visent à couvrir toutes les parties d'un logiciel ; on appelle cela la **couverture de code**

pour y parvenir, on peut construire les tests en inspectant le code source

exemple :

```
    def nbjoursmois(a, m):
    if m == 2 and bissextile(a):
    return 29
    else:
    return joursmois[m]
```

à la lecture de ce code, on voit qu'il est intéressant de tester en particulier

- le cas du mois de février d'une année bissextile
- le cas du mois de février d'une année non bissextile
- le cas d'un mois autre que février

dans certains cas, on ne dispose pas du code source
cela ne nous empêche pas pour autant de tester le programme ou la fonction ; on parle de **test en boîte noire**

on peut ainsi tester la fonction `pow` de Python

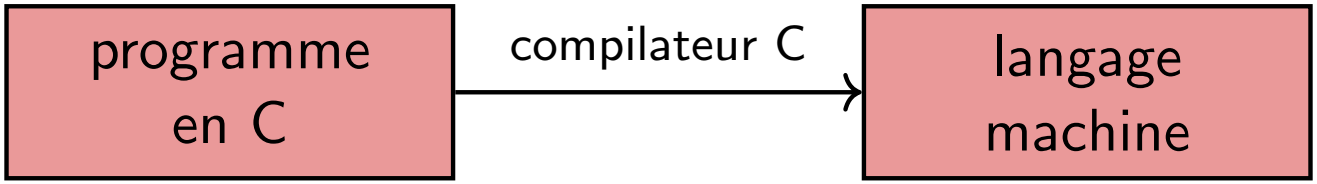
```
>>> pow(2, 0)
1
>>> pow(2, 10)
1024
>>> pow(-2, 3)
-8
>>> pow(2, -2)
0.25
...
```

Le test montre la présence, mais pas l'absence d'erreurs.
(Edsger Dijkstra, informaticien néerlandais, 1969)

il y a en général **une infinité** d'entrées possibles pour un programme
il est dès lors impossible de procéder à un test exhaustif

au-delà du test

les **langages de programmation** peuvent aider à la qualité du logiciel, en procédant à des vérifications **avant** de chercher à exécuter le programme
c'est typiquement le compilateur qui va effectuer ces vérifications



le programme C

```
int f() { return x; }
```

est rejeté par le compilateur, car la variable `x` n'existe pas

```
gcc -c test.c
test.c: In function 'f':
test.c:2:18: error: 'x' undeclared
2 | int f() { return x; }
  |                  ^
```

ici, le langage machine n'est **pas produit**

la même erreur en Python est signalée seulement **pendant l'exécution**, au moment où on cherche à accéder à la variable qui n'existe pas

```
$ python test.py
...
NameError: name 'x' is not defined
```

la différence est **majeure** :

- avec le programme C, c'est un problème pour le **programmeur**
- avec le programme Python, c'est un problème pour l'**utilisateur**

beaucoup de langages permettent au programmeur d'indiquer la nature des variables

- ≪ ceci est un entier ≫
- ≪ ceci est une chaîne de caractères ≫
- ≪ ceci est un tableau ≫
- etc.

on appelle cela des **types**

dans le langage Java, on peut écrire

```
String f(String s, int n) {
...
```

pour indiquer que la fonction `f`

- reçoit un paramètre `s` qui est une chaîne de caractères
- reçoit un paramètre `n` qui est un entier
- renvoie un résultat qui est une chaîne de caractères

et si on tente une opération illégale

```
String f(String s, int n) {
    return s / n;
  }
```

on obtient une erreur du compilateur

```
C.java:4: error: bad operand types for binary operator '/'
return s / n;
^
first type:  String
second type: int
1 error
```

c'est une erreur pour le programmeur

là encore, il y a une différence avec Python, qui ne signalera un tel problème que **pendant l'exécution** du programme

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

de manière générale, on distingue

- les analyses **statiques**, réalisées **avant** l'exécution
- les analyses **dynamiques**, réalisées **pendant** l'exécution

les vérifications que le compilateur peut effectuer automatiquement sont **limitées**

il est **impossible** pour le compilateur de vérifier que le programme fait ce qu'il est censé faire

plus généralement, il est impossible de vérifier toute propriété intéressante relative à un programme, comme

- il ne contient pas de bug
- il ne fait pas de division par zéro
- il n'accède pas en dehors des limites d'un tableau
- il ferme bien tous les fichiers qu'il a ouvert
- il se termine
- etc.

en revanche, il est possible de faire une **preuve mathématique** que le programme fait bien ce qu'il est censé faire
ceci demande

- qu'on écrive la spécification dans un langage mathématique
- qu'on détaille les étapes de la preuve

on parle de **méthodes formelles**

on ajoute dans un programme Python des annotations écrites dans un langage mathématique

```
a = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
#@ assert len(a) == 10

a[0] = 0
a[1] = 1
for i in range(2, 10):
#@ invariant forall j. 0 <= j < i -> 0 <= a[j]
a[i] = a[i-2] + a[i-1]

#@ assert a[9] >= 0
```

elles sont comprises par un outil de **preuve de programme**

- le code de la ligne 14 du métro parisien a été en partie vérifié avec des méthodes formelles
- le code d'un avion Airbus est garanti contre tout "plantage" par des méthodes formelles (mais il n'est en revanche pas prouvé qu'il fait ce qu'il est censé faire)

il y a une différence fondamentale entre

- le **test**, qui a été effectué sur **certaines entrées** seulement
- la **preuve**, qui garantit que le programme fonctionne correctement pour **toute entrée possible**

le test est omniprésent dans l'industrie du logiciel

- des méthodologies, des pratiques, des outils, etc.
- une part importante du temps de développement

la preuve, en revanche, n'est que très rarement faite, car elle nécessite une grande expertise et un effort incroyable

- un ordre de grandeur : la ligne 14, c'est 15 ingénieurs pendant 3 ans pour vérifier 90 000 lignes de code

TP 4 : identifier les mauvais tris

1. télécharger le fichier `tris.py` sur ecampus et le téléverser sur JupyterLab
2. créer un nouveau fichier `tp4.py` et y ajouter la ligne suivante

```
from tris import *
```

3. ceci importe des fonctions `tri1`, `tri2`, ..., `tri10`
4. chaque fonction a un paramètre (un tableau Python) et est censée en modifier le contenu pour le trier par ordre croissant
5. une seule fonction est correcte et il s'agit de l'identifier

```
a = [3, 2, 1]
tri4(a)
print(a)
```

```
[1, 2, 3]
```

- mercredi 14 février
  - développer de gros logiciels
  -  si possible, apporter un jeu de 52 cartes