Algorithmique & Programmation

Cas d'étude: le jeu du Loto

yann.secq@univ-lille.fr

ABIDI Sofiene, ALMEIDA COCO Amadeu, BONEVA Iovka, CASTILLON Antoine, DELECROIX Fabien, LEPRETRE Éric, Timothé ROUZÉ, SANTANA MAIA Deise, SECQ Yann

Saison 2023-2024 / R1.01+R1.02 : Initiation au développement

John Mc Carthy (1957-2011)

It's difficult to be rigorous about whether a machine really 'knows', 'thinks', etc., because we're hard put to define these things. We understand human mental processes only slightly better than a fish understands swimming.

The Little Thoughts of Thinking Machines

Le jeu du LOTO

- Préliminaire: n'y jouez jamais ! ;-)
- Concevoir un algorithme qui effectue le tirage du loto national
- Extraire six nombres distincts parmi les 49 proposés (de 1 à 49).
- Primitive permettant d'obtenir une valeur aléatoire entre `[0..max] : int random(int max)`
- Etude de trois approches différentes

Approche « zéro »

- Répétition de l'appel de la fonction `random` six fois (autant que de nombres demandés)
- Quel soucis avec cette approche directe ?

Quel est la probabilité que l'on tire la même valeur ?

Apparition de doublons

- Lors du premier nombre pas de problème (normal, on n'a rien tiré).
- Pour le 2ième, 1 chance sur 49
- Pour le 3ième, 2 chances sur 49
- … pour le 6ième, 5 chances sur 49
- Soit: (1+2+3+4+5)/49 = 0.3 !

Corrections possible

- la création d’une structure de données pour mémoriser les numéros tirés et,
- un dispositif qui assure, soit qu’un numéro n’est tiré que parmi les numéros libres ou qui tire à nouveau un numéro déjà sorti.

Quelle structure de données ?

- **Mémoriser les numéros sortis**
  - Incontournable … sans mémoire, point de salut!
- **Affichager les numéros sortis**
  - Pendant le calcul ou après (mieux) … éventuellement en ordre croissant ?
- **Vérifier qu’il n’y a pas de doublon**
  - il est nécessaire de pouvoir tester si un nombre donné fait déjà parti du tirage

Structures possibles

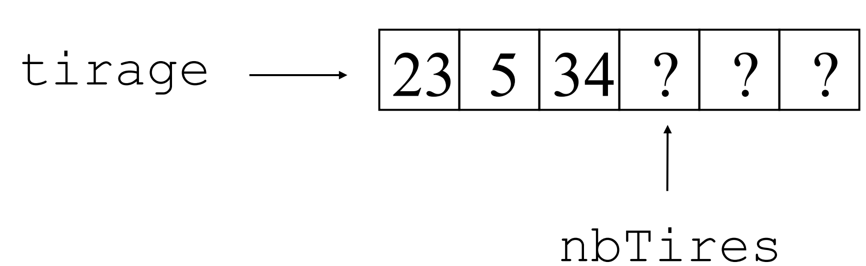
- Plusieurs réponses plus ou moins naturelles, plus ou moins efficaces et plus ou moins complexes ...
- Trois structures de données envisageables
  - un tableau de 6 nombres,
  - un tableau de 49 booléens,
  - un tableau de 49 nombres.

Structures possibles

- Plusieurs réponses plus ou moins naturelles, plus ou moins efficaces et plus ou moins complexes ...
- Trois structures de données possibles:
  - **un tableau de 6 nombres**,
  - un tableau de 49 booléens,
  - un tableau de 49 nombres.

Tableau de 6 nombres

- Un tableau d'entiers (`tirage`) pour les nombres déjà tirés
- Un indice (`nbTires`) permet de savoir combien de nombre ont déjà été tirés
- Retour sur les trois fonctionnalités
  - déterminer si un nombre est déjà sorti
  - afficher la liste des nombres tirés
  - tirer six nombres distincts



Nombre déjà sorti ?

- Un nombre est déjà sorti si il figure dans la partie utile du tableau (i.e. cases dont l'indice est inférieur strictement `nbTires`)

```
boolean dejaTire = false;
int ieme = 0;

while (ieme < nbTires && ! dejaTire) {
  if (tirage[ieme] == x) {
    dejaTire = true;
  }
  ieme = ieme + 1;
}
```

Affichage des nombres

- Afficher les nombres

A n'utiliser qu'ici !

```
for (int ieme=0; ieme<length(tirage); ieme++) {
  print(tirage[ieme]+" ");
}
```

- Les nombres ne seront pas forcément affichés en ordre croissant
- Cette opération peut être incluse dans l'algorithme de tirage (mais à éviter)

Tirage de 6 nombres distincts

- Tirer à nouveau un nombre ne garantit pas que le nouveau nombre est original ! même si il s'agit du 2ème , voire du nième retirage !
- Nécessité d'une boucle supplémentaire pour le retirage d'un nombre ...

```
for (int nbTires=1; nbTires<=6; nbTires++) {
  do {
    x = random(48)+1;
    //calcul de déjàTire (vérification d'existence
     ou non dans les numéros déjà sortis)
  } while (dejaTire);
  tirage[nbTires-1] = x;
}
```

Tirage de 6 nombres distincts

- et en détaillant la ligne intermédiaire (le calcul de `dejaTire`)

```
for (int nbTires=1; nbTires<=6; nbTires++) {
  do {
    x = random(48)+1;
    dejaTire = false;
    ieme = 0;
    while (ieme < nbTires && ! dejaTire) {
      if (tirage[ieme] == x) {
        dejaTire = true;
      }
      ieme = ieme+1;
    }
  } while (dejaTire);
  tirage[nbTires-1] = x;
}
```

Analyse de cette solution

- Pas si évidente que cela à comprendre
- Quatre niveaux d'imbrication de structures de contrôle
- Lisibilité améliorable en utilisant des fonctions

```
boolean estDejaTire(int[] tires, int numero) {}

for (int nbTires=1; nbTires<=6; nbTires++) {
  do {
    x = random(48)+1;
  } while (estDejaTire(tirage, x));
  tirage[nbTires-1] = x;
}
```

Structures possibles

- Plusieurs réponses plus ou moins naturelles, plus ou moins efficaces et plus ou moins complexes ...
- Trois structures de données possibles:
  - un tableau de 6 nombres,
  - **un tableau de 49 booléens**,
  - un tableau de 49 nombres.

Tableau de 49 booléens

- Utilisation d'un tableau de 49 booléens (1 par numéro)
- Tableau dont chaque case est initialisée à `FAUX`
- La case correspondant à un numéro est mise à `VRAI` lorsque ce numéro a été tiré
- Vérification du tirage d'un numéro par un accès direct, c-à-d. sans avoir à parcourir l'intégralité du tableau !

```
boolean[] tires = new boolean[49];

// Indices de 0 à 48, si i est sorti (t[i-1]==true)
```

Tableau de 49 booléens

- Le tableau `tirage` n'est plus nécessaire
- On retrouve l'ensemble des six nombres sortis en parcourant le tableau `tires`
- **Attention à l'affichage**, il faut faire `x+1` (car `0<=x<=48`)

```
for (int idx=0; idx<length(tires); idx++) {
  tires[idx] = false;
}
for (int nbTires=1; nbTires<=6; nbTires++) {
  do {
    x = random(48);
  } while (tires[x]);
  tires[x] = true;
}
```

Analyse de cette solution

- Simplifie radicalement la recherche de doublon …
- **De l’importance du choix de la structure de données !**
- 3 structures de contrôles, mais juste un niveau d’imbrication
- Programme bien plus simple à comprendre et plus efficace
- Il n’existe aucune garantie sur le temps d’exécution global
- Si on pinaille : le programme pourrait ne pas s’arrêter avec un mauvais générateur aléatoire …

Structures possibles

- Plusieurs réponses plus ou moins naturelles, plus ou moins efficaces et plus ou moins complexes ...
- Trois structures de données possibles:
  - un tableau de 6 nombres,
  - un tableau de 49 booléens,
  - **un tableau de 49 nombres**.

Tableau de 49 nombres

- `int[] balles = new int[49];`
- Tableau initialisé avec les valeurs 1 à 49
- Déplacer les nombres tirés à la fin du tableau :)
- Pseudo-algorithme global

```
// initialiser balles
for (int ieme=0; ieme<6; ieme++) {
  // tirer x entre 0 et 48-ieme (bornes comprises)
  // échanger la case x et la case d'indice 48-ieme
}
// afficher les 6 dernières cases
```

Tableau de 49 nombres

```
for (int idx=0; idx<length(balles); idx++) {
  balles[idx] = idx+1;
}
for (int ieme=0; ieme<6; ieme++) {
  // tirage de x entre 0 et 48-ieme
  x = random(48-ieme);
  // échange des cases x et 48-ieme
  int temporaire  = balles[x];
  balles[x]       = balles[48-ieme];
  balles[48-ieme] = temporaire;
}
for (int idx=0; idx<6; idx++) {
  print(balles[length(balles)-idx-1]+ " ");
}
```

Analyse de cette solution

- La question du doublon ne se pose même plus :)
- 3 structures de contrôles, mais en série, sans imbrication
- Programme le plus simple et le plus efficace
- **De l’importance du choix de la structure de données !**

Synthèse

- La conception … c'est compliqué ! (mais passionnant ;))
- Toujours envisager les différents cas possibles
- **Comprendre l'impact du choix d'une structure de données dans la résolution**
- Ne pas s'arrêter à la 1ère solution trouvée
- Réfléchir, essayer, ruminer, réessayer !
- Plus on s'imprègne d'un problème, plus l'on identifie les étapes critiques, au mieux on améliore leur résolution

Made with Lisp