Modularité

Chapitre 07

NSI2

28 août 2023

Un problème,
de multiples réponses

Problème des anniversaires

Problème des anniversaires

- un groupe de personnes, chacune avec sa date anniversaire ;

Problème des anniversaires

- un groupe de personnes, chacune avec sa date anniversaire ;
- plus ce groupe est grand, plus il y a de chances que deux personnes soient nées le même jour de l'année.

Problème des anniversaires

- un groupe de personnes, chacune avec sa date anniversaire ;
- plus ce groupe est grand, plus il y a de chances que deux personnes soient nées le même jour de l'année.
- paradoxe des anniversaires :

Problème des anniversaires

- un groupe de personnes, chacune avec sa date anniversaire ;
- plus ce groupe est grand, plus il y a de chances que deux personnes soient nées le même jour de l'année.
- paradoxe des anniversaires : à partir de 23 personnes, il y a plus de 50% de chances que deux personnes soient nées le même jour de l'année.

Problème des anniversaires

- un groupe de personnes, chacune avec sa date anniversaire ;
- plus ce groupe est grand, plus il y a de chances que deux personnes soient nées le même jour de l'année.
- paradoxe des anniversaires : à partir de 23 personnes, il y a plus de 50% de chances que deux personnes soient nées le même jour de l'année.
- pas un « vrai » paradoxe.

Variante

Variante

- une liste d'entiers compris entre 0 et $2^{16} - 1$;
- à partir de 302 éléments, plus d'une chance sur 2 qu'il y ait au moins un doublon ;

Vous ne me croyez pas ?

Et bien regardez

```
 [13654, 26764, 60127, 56265, 45203, 54601, 23471, 64648, 49436, 32684, 4685, 61418,
8441, 10200, 29042, 55598, 35106, 59628, 16003, 52546, 61235, 61380, 58092, 15876,
41296, 5825, 11755, 46620, 33256, 21388, 34496, 50818, 24255, 21645, 59590, 46160,
29287, 28482, 7056, 62317, 7646, 48862, 580, 55506, 37346, 20788, 18739, 46029,
17621, 23795, 64827, 62778, 44784, 1732, 56030, 36325, 5513, 18255, 5423, 30071,
27916, 26456, 42655, 56515, 54266, 30311, 9712, 56000, 57606, 29080, 11732, 6675,
18147, 18031, 31923, 7587, 10177, 11595, 45194, 60765, 57430, 22114, 7692, 22005,
37297, 7817, 35883, 21041, 25233, 8245, 17171, 604, 15615, 49219, 5292, 61211, 32599,
27813, 59838, 15470, 35127, 19969, 15707, 60577, 28106, 54636, 18636, 10802, 45,
3962, 12676, 56137, 4457, 18793, 64445, 48181, 61137, 18182, 3579, 42258, 50192,
64379, 31680, 32806, 8665, 8581, 60429, 27189, 7004, 14490, 27959, 40120, 61965,
57446, 40767, 9506, 30011, 63676, 16650, 6053, 6459, 19781, 26735, 50643, 19042,
51938, 16298, 19033, 7838, 43301, 51725, 57656, 63232, 51368, 65031, 46605, 55392,
9509, 32286, 50079, 10218, 37000, 40932, 40890, 4415, 60392, 47891, 48141, 33494,
64130, 25837, 41840, 27717, 57910, 19235, 40414, 32108, 33204, 51667, 59269, 8221,
24221, 26572, 53731, 59583, 12556, 15280, 8670, 571, 20383, 25326, 13967, 13776,
30529, 2175, 7819, 8110, 64263, 16570, 35558, 55085, 12863, 21481, 53120, 54957,
30280, 2210, 16659, 52235, 27616, 42152, 33507, 29239, 51945, 32471, 47977, 8414,
10609, 27084, 2738, 40268, 8843, 59468, 27787, 56664, 61642, 25038, 39276, 9981,
21508, 17725, 64495, 7775, 63696, 2659, 17292, 27874, 55810, 35170, 19244, 13361,
40907, 13019, 21447, 16367, 34450, 54737, 54046, 65365, 28076, 49056, 61876, 4276,
8795, 26780, 24477, 43398, 35627, 18815, 24692, 8364, 39195, 29516, 33998, 9633,
32619, 21929, 3803, 58244, 49410, 45617, 9974, 49174, 8108, 19506, 4053, 12516, 502,
23668, 8665, 55033, 44229, 43647, 10663, 14877, 42960, 41370, 27958, 45473, 39233,
56912, 4757, 39967, 5703, 21297, 58081, 4677, 7777, 52981, 30204, 18837, 12346]
```

Bon, d'accord, ce n'est pas une preuve...

Algorithme de recherche de doublons

```
fonction doublon( contenu : liste ) ->
 ↪  booléen
    déjà_vu ← vide
    pour x dans contenu
        si x est dans déjà_vu
            renvoyer vrai
        sinon
            ajouter x à déjà_vu
    renvoyer faux
```

Pas si clair que ça...

Pas si clair que ça...

- Quelle structure de données pour implémenter `déjà_vu` ?

Pas si clair que ça…

- Quelle structure de données pour implémenter `déjà_vu` ?
- Quelles contraintes ?

Pas si clair que ça...

- Quelle structure de données pour implémenter `déjà_vu` ?
- Quelles contraintes ?Vitesse ?

Pas si clair que ça...

- Quelle structure de données pour implémenter `déjà_vu` ?
- Quelles contraintes ?Vitesse ?Mémoire ?

Pour la suite...

Pour la suite...

- On appellera `s` la structure de donnée qui représente `déjà_vu`.

Pour la suite...

- On appellera `s` la structure de donnée qui représente `déjà_vu`.
- On notera *n* la taille de la liste à examiner et `content` son nom.

Pour la suite...

- On appellera `s` la structure de donnée qui représente `déjà_vu`.
- On notera *n* la taille de la liste à examiner et `content` son nom.
- On définit une OPEL comme un accès à `content` ou à `s`.

Pour la suite...

- On appellera `s` la structure de donnée qui représente `déjà_vu`.
- On notera n la taille de la liste à examiner et `content` son nom.
- On définit une OPEL comme un accès à `content` ou à `s`.
- On appellera complexité l'ordre de grandeur du nombre d'OPEL en fonction de n.

Au plus simple :

Au plus simple : avec une liste

Code Python

```
def has_duplicates_1(content: list) ->
 ↪  bool:
    s = []
    for x in content:
        if x in s:
            return True
        else:
            s.append(x)
    return False
```

Analyse

Analyse

- Simple !

Analyse

- Simple !
- Complexité dans le pire des cas :

Analyse

- Simple !
- Complexité dans le pire des cas : n^2.

Avec une liste de booléens

Principe

- 2^16 valeurs possibles pour chaque élément ;

Principe

- 2^16 valeurs possibles pour chaque élément ;
- créer une liste `s` de 2^16 booléens ;

Principe

- 2^16 valeurs possibles pour chaque élément ;
- créer une liste `s` de 2^16 booléens ;
- pour chaque `x` de `content`, mettre `s[x]` à `True` et, s'il y est déjà, s'arrêter car on a un doublon.

Code Python

```python
def has_duplicate2(content: list) -> bool:
    s = [False] * 65536
    for x in content:
        if s[x] == True:
            return True
        else:
            s[x] = True
    return False
```

Analyse

- Toujours simple ;

Analyse

- Toujours simple ;
- Complexité : n ;

Analyse

- Toujours simple ;
- Complexité : n ;
- Malheureusement, en PYTHON chaque booléen est stocké sur 64 bits.

Analyse

- Toujours simple ;
- Complexité : n ;
- Malheureusement, en PYTHON chaque booléen est stocké sur 64 bits.
  Cela fait un sacré gaspillage de mémoire (plus de 500 ko) !

En jouant sur les bits d'un grand nombre

Principe

Principe

- utiliser une variable `s` de type `int` pour « stocker » une valeur sur 65 536 bits ;

Principe

- utiliser une variable `s` de type `int` pour « stocker » une valeur sur 65 536 bits ;
- méthode identique à la précédente : lors du parcours de `content`, si on trouve l'élément `x`, on regarde le x^e bit de `s`…

Code Python

```python
def has_duplicate3(content: list) -> bool:
    s = 0
    for x in content:
        v = 1 << x
        if s & v:
            return True
        else:
            s = s | v
    return False
```

Analyse

Analyse

- plus technique ;

Analyse

- plus technique ;
- complexité : n ;

Analyse

- plus technique ;
- complexité : n ;
- en théorie, ça marche bien…

Analyse

- plus technique ;
- complexité : n ;
- en théorie, ça marche bien…et en pratique ?

Solution hybride : la liste de paquets

Principe

Principe

On rappelle que `content` contient 302 nombres.

Principe

On rappelle que `content` contient 302 nombres.

- `s` est une liste de taille 302...

Principe

On rappelle que `content` contient 302 nombres.

- `s` est une liste de taille 302...
- mais pas une liste d' `int`, plutôt une liste de listes ;

Principe

On rappelle que `content` contient 302 nombres.

- `s` est une liste de taille 302...
- mais pas une liste d' `int`, plutôt une liste de listes ;
- lors du parcours de `content`, `x` est placé dans `s[x % 302]`.

Code Python

```python
def has_duplicate4(content: list) -> bool:
    s = [[] for _ in range(302)]
    for x in content:
        if x in s[x % 302]:
            return True
        else:
            s[x % 302].append(x)
    return False
```

Analyse

Analyse

- `s` a une taille raisonnable ;

Analyse

- `s` a une taille raisonnable ;
- peu de calculs ;
- le test `if x in s[x % 302]` porte sur un paquet qui sera probablement de taille très réduite.

Pourquoi tout cela ?

Définition

Définition

Implémentation

choix concret de programmation pour répondre à un problème qui, lui, peut s'avérer *abstrait* :

Définition

Implémentation

choix concret de programmation pour répondre à un problème qui, lui, peut s'avérer *abstrait* :

- choix des structures de données pour représenter les variables ;

Définition

Implémentation

choix concret de programmation pour répondre à un problème qui, lui, peut s'avérer *abstrait* :

- choix des structures de données pour représenter les variables ;
- parfois, choix d'un algorithme particulier, ou d'un paradigme de programmation.

Concrètement

Concrètement

On vient de rencontrer 4 implémentations différentes destinées à résoudre le problème de départ.

Modularité !

Points communs des implémentations

```
def has_duplicate(content: list) -> bool:
    s = create() # <-- MODULAIRE
    for x in content:
        if contains(s,x): # <-- MODULAIRE
            return True
        else:
            add(s,x) # <-- MODULAIRE
    return False
```

Principe de modularité

Principe de modularité

- découper un programme en composants (fonctions ou objets);

Principe de modularité

- découper un programme en composants (fonctions ou objets);
- chaque composant a ses propres *spécifications* et fonctionne le plus indépendamment possible.