Exercices 1

Philipp Ahrendt

December 30, 2025

Contents

1 Exercice 1: Années bissextiles

Les années bissextiles (les années avec 29 jours en février au lieu de 28) sont les années qui respectent l'une des conditions suivantes:

1. L'année est divisible par 4 sans être divisible par 100.
2. L'année est divisible par 400.

Écrire une fonction `bissextile` qui prend en entrée un entier (représentant l'année) et retourne `True` si l'année est bissextile, et `False` sinon.

Réponse:

```
def bissextile(n):
    return (n % 4 == 0 and n % 100 != 0) or n % 400 == 0
```

2 Exercice 2: Algorithme d'Euclide

Le plus grand diviseur commun de deux entiers non nuls n,m, noté pgcd(n,m), est le plus grand entier k tel que n/k et m/k soient tous les deux entiers. Les deux propriétés suivantes sont vérifiées par le pgcd:

1. Si m divise n, alors pgcd(n,m)=m.
2. Sinon, pgcd(n,m)=pgcd(m,n % m), où n % m est le reste de la division euclidienne.

2.1 Question a:

Utiliser les deux propriétés ci-dessus pour écrire une fonction `pgcd`, qui prend en entrée deux entiers, et donne en sortie leur pgcd.

Réponse:

```
def pgcd(n,m):
    a,b = n,m
    while a % b != 0:
        a,b = b,a%b
    return b
```

2.2 Question b:

Pouvez-vous expliquer pourquoi cette fonction finit toujours par rendre un résultat (i.e pourquoi elle ne peut jamais tourner à l'infini)? Pouvez-vous expliquer pourquoi elle calcule bien le pgcd?

Réponse: L'algorithme termine au plus tard quand `b = 1` (`b` ne peut pas devenir nul ou négatif: à chaque itération c'est le reste positif d'un nombre positif, et la condition d'arrêt de la boucle empêche `b = 0`). On a toujours `a%b < b`, donc `b` décroît strictement à chaque itération. Cela montre que `b` finit toujours par atteindre 1, si l'algorithme ne termine pas avant cela.

Pour voir que l'algorithme calcule le pgcd, il suffit de regarder les conditions 1 et 2. On peut voir que les itérations de la boucle laissent "invariante" la condition 2: si, avant une itération, on a pgcd(a,b)=pgcd(n,m), alors on a pgcd(b,a%b)=pgcd(a,b)=pgcd(n,m). C'est vrai au départ, quand a,b=n,m, et donc cela reste vrai tant que l'algorithme tourne. La condition 2. est ce qu'on appelle un "invariant de boucle" pour la boucle while. Quand l'algorithme termine, on peut regarder la condition 1, qui dit que b=pgcd(a,b)=pgcd(n,m).

3 Exercice 3: Recherche de fonctions inverses

Si f(x) est une fonction strictement croissante en x, elle admet une fonction inverse g(y) telle que g(f(x))=x et f(g(y))=y. Dans cet exercice, le but est de trouver algorithmiquement des approximations de la fonction inverse d'une fonction f croissante, par différentes méthodes.

3.1 Question a: Inversion sur les entiers

Une première approche consiste à chercher les valeurs de la fonction inverse sur les entiers uniquement (positifs et négatifs). Pour une fonction f strictement croissante, on peut définir son inverse g sur les entiers par:

- g(n)=k, où k est le plus grand entier tel que f(k) $\leq$ n

Écrire une fonction `inv_int` qui prend en entrée une fonction f (que l'on suppose strictement croissante), et un entier n, et donne en résultat la valeur g(n).

Pour tester cette fonction, vous pouvez utiliser par exemple f(x) = x^3, et tester contre la racine cubique.

```
import math

def f(n):
    return n**3

def g_int(n):
    # La fonction que doit trouver l'algorithme
    return math.floor(n**(1/3))


# Tester sur cette fonction aussi. L'algoroithme marche-t-il encore?
def f_bis(n):
    return (n+5)**3

def g_bis_int(n):
    return math.floor(n**(1/3)-5)
```

Réponse:

```
def inv_int(f,n):
```

```
k = 0

# Manière simple de chercher: on teste jusqu'où f(k) < 1
while f(k) <= n:
    k += 1

# La même chose si k est trop grand
# Cette boucle étant placée après la première, elle s'occupe aussi
# de ajustements si besoin; on cherche le k juste avant de sortir de la première bo
# que l'on retrouve avec une itération de la deuxième. Ceci n'est pas symétrique;
# en allant vers les négatifs, le k cherché est bien celui qui sort de cette boucle
while f(k) > n:
    k -= 1

return k
```

3.2 Question b: Recherche dichotomique

Une manière de calculer efficacement l'inversion entière d'une fonction est par **recherche dichotomique**. Dans un premier temps, on va supposer que la fonction f vérifie $n \leq f(n)$. Le but est toujours de trouver g(n):

- g(n)=k, où k est le plus grand entier tel que $f(k) \leq n$.

L'algorithme de recherche dichotomique consiste à diminuer progressivement l'intervalle dans lequel on cherche g(n). A chaque étape, on a un intervalle $[k_{min}, k_{max}]$, et on prend k dans cet intervalle. Si k est plus grand que g(n), on remplace k_{max} par k, et on cherche relance la recherche dans $[k_{min}, k]$, et si k est plus petit, on remplace k_{min} et relance. On répète cela jusqu'à ce qu'on trouve g(n). Voici une manière d'implémenter cela:

1. Initialiser:
   - $k_{min} \leftarrow 0$
   - $k_{max} \leftarrow n$
2. Tant que g(n) n'est pas trouvée:
   - $k \leftarrow$ partie entière de $(k_{min}+k_{max})/2$
   - Si $f(k) \leq n$:
     - Si $f(k+1) > n$, **fin**: on sort g(n)=k

- Sinon, $k_{min} \leftarrow k$

- Si $f(k) > n$, $k_{max} \leftarrow k$

Écrire une fonction `inv_int_dicho` qui prend une fonction f et un entier n, et trouve l'inverse g(n) par recherche dichotomique.

Réponse:

```
# Première version; avec un intervalle de départ [0,n] fixe
def inv_int_dicho(f,n):
    k_min,k_max = 0,n
    k = (k_min+k_max)//2

    while (f(k) <= n and f(k+1) <= n) or f(k) > n:
        if f(k) <= n:
            k_min = k
        else:
            k_max = k

        k = (k_min+k_max)//2

    return k
```

Comment peut-on adapter cette fonction pour qu'elle marche sur n'importe quelle fonction croissante?

Réponse:

```
def inv_int_dicho_bis(f,n):

    # On commence par trouver un intervalle qui marche
    # Pour faire simple, on multiplie juste par 2 d'un cote ou l'autre
    # jusqu'à ce que f(k_min) <= n <= f(k_max)
    if f(0) < n:
        k_min = 0
        k_max = 1

        while f(k_max) < n:
            k_max *= 2

    else:
```

```
        k_max = 0
        k_min = -1

        while f(k_min) > n:
            k_min *= 2

    # Le reste est juste le même algorithme
    k = (k_min+k_max)//2

    while (f(k) <= n and f(k+1) <= n) or f(k) > n:
        if f(k) <= n:
            k_min = k
        else:
            k_max = k

        k = (k_min+k_max)//2

    return k
```

3.3 Question c: Recherche dichotomique sur les nombres réels

On peut utiliser le même principe pour approcher l'inverse sur les nombres réels. Une fonction inverse approchée d'une fonction f(x) est une fonction g(y) telle que |g(f(x))-x| $\leq \epsilon$, pour une tolérance d'erreur $\epsilon > 0$ choisie au préalable.

Adapter l'algorithme précédent pour écrire une fonction `inv_dicho`, qui prend en entrée une fonction f, un nombre x, et un nombre positif ϵ, et trouve g(x) avec une erreur maximale de ϵ.

```
def inv_dicho(f,x,eps):
    if f(0) < x:
        r_min = 0
        r_max = 1

        while f(r_max) < x:
            r_max *= 2

    else:
        r_max = 0
```

```
    r_min = -1

    while f(r_min) > x:
        r_min *= 2

r = (r_min+r_max)/2

while f(r) < x - eps or f(r) > x + eps:
    if f(r) < x - eps:
        r_min = r
    else:
        r_max = r

    r = (r_min+r_max)/2

return r
```