Preuves de programmes : Terminaison, correction, complexité. Rappels et compléments de PCSI

1 Terminaison, correction, complexité

En informatique, un programme se « prouve ». Précisément, on doit répondre à plusieurs questions :

1. Terminaison : le programme s'arrête-t-il ?
2. Correction : le programme est-il correct, c'est-à-dire fait-il bien ce que l'on veut qu'il fasse ?
3. Complexité : le programme prend il du **temps** pour terminer, et consomme-t-il beaucoup **d'espace** ?

1.1 Terminaison

Remarque — Démarche

Il s'agit de vérifier que les boucles while (et les appels récursifs, mais nous n'aborderons pas la question ici) se terminent bien, c'est-à-dire que l'on ne part pas en boucle infinie. Pour cela, on doit montrer que la condition de boucle dans la ligne `while condition de boucle` **devient fausse**.
Une méthode classique pour cela est de mettre en place un **variant de boucle**.
Un variant de boucle est une une suite strictement décroissante d'entiers naturels.
Si la condition de boucle restait vraie « éternellement », alors on ne sortirait jamais de la boucle, le variant de boucle serait une suite infinie d'entiers décroissante : absurde. Ainsi la boucle se termine.

Exemple

Considérons la fonction suivante :

```
# Code 1
def fonction1(n=100):
    i=0
    while i<n:
        i+=1
    return i
```

Montrons que si n est un entier, alors l'appel `fonction1(n)` termine. Considérons i_k la valeur contenue dans la variable i à la k eme itération de la boucle. Alors $(n - i_k)_{k\geqslant 0}$ est une suite d'entiers **strictement** décroissante et **positive**, ie un variant de boucle. Donc le programme termine (sinon par l'absurde on aurait une suite entière décroissante minorée par 0).

Exemple

Soit le programme suivant :

```
def division1(a,b):
    r=a
    q=0
    while r >=b:
        r=r-b
        q=q+1
    return q,r
```

Montrons que le programme termine lorsqu'il est éxécuté avec un entier $b > 0$. En effet, au cours de la boucle, r diminue à chaque fois d'au moins 1 à chaque itération (car $b \geqslant 1$). Par ailleurs, la suite des valeurs prises par r est minorée par b tant qu'on reste dans la boucle. Ainsi r est un variant de boucle.
(Remarque : Si b est négatif ou nul et a est positif ou nul, alors r n'est plus un variant de boucle et le programme ne termine pas.)

Exercice 1 Montrer que les programmes suivants terminent en précisant le variant de boucle.

```
# Code 1
def fonction1(n=100):
    i=0
    while i<n:
        i+=1
    return i
```

P. Connault – Reproduction et diffusion soumises à autorisation

```python
# Code 2
def fonction2(n=100):
    i=1
    while i<n:
        i=2*i
    return i
```

```python
# Code 2 bis
def fonction2bis(n=100):
    i=0
    x=1
    while x<=n:
        i+=1
        x=x*10
    return i
```

```python
# Code 3
def fonction3(x=1,debut=100):
    i=debut
    while i>=x:
        i-=1
    return i
```

```python
# Code 4
def fonction4(n=100):
    j=0
    while (j*j)<n:
        j+=1
    return j
```

```python
# code 5
def fonction5(l):
    i=0
    n=len(l)
    while i+1 <n and l[i]<l[i+1]:
        i+=1
    return i
```

```python
# Code 6
x=100
y=30
z=10
while x>y:
    x=x-z
```

```python
# Code 7
a=0
b=1000
c=10
while b-a>c:
    a+=1
    b-=1
```

```python
# Code 8
a=30
b=100
c=10
while b-a>c:
    a=a/2
    b=b/2
```

```python
# Code 9
P=0
for i in range(10):
    for j in range(10):
        P+=i*j
```

```
# Code 10
S=0
for i in range(101):
    j=0
    while j*j<i:
        j+=1
    S+=j
```

```
#Code 11
def fonction11(a=10.0,b=150.0,e=0.1):
    while b-a>e:
        c=(b-a)/2
        b=b-c
    return b-a
```

```
# Code 12
def fonction12(a=10,b=1, c=1):
    while a>b:
        a=a-c
        c=c//2
    return a
```

1.2 Correction

▲ Définition 1

On parle de **correction partielle** lorsque, **dans les cas où le programme s'arrête**, il donne le résultat attendu.
On parle de **correction totale** lorsque le programme s'arrête toujours et donne toujours le résultat attendu.
« terminaison + correction partielle = correction totale »

Exemple

Cette fonction

```
def est_puissance_de_deux(x):
    a=1
    while not(a==x):
        a=2*a
    return True
```

teste si x est une puissance de 2. Elle termine lorsque x est une telle puissance, et renvoie alors True, ce qui est exact. Elle ne termine pas sinon.

Méthode

Pour montrer qu'un programme est correct, il suffit parfois d'écrire les conditions dans la boucle. Garder en tête que la condition de boucle est **fausse au sortir de la boucle**, mais **était vraie juste avant.**
Dans la plupart des cas, on utilisera de plus un **invariant de boucle.**

Exemple

On reprend la `fonction1` de l'exemple plus haut, en supposant $n \in \mathbb{N}^*$. Soit i_f la valeur contenue dans i à la fin de l'éxécution de la fonction. On a donc $i_f \geqslant n$; mais on a aussi $i_f - 1 < n$, car à l'étape précédente la condition de boucle était vérifiée. Ainsi $n \leqslant i_f < n+1$ puis $n = i_f$ car i_f est entier. Ainsi on a $i_f = n$, c'est-à-dire que la fonction renvoie n.

▲ Définition 2

Un invariant de boucle est une propriété P, qui est vraie avant et après chaque exécution de la boucle.

Méthode

Comment prouver qu'une propriété est un invariant de boucle.
Il s'agit d'un raisonnement très proche de la récurrence finie **avec tous ses attendus formels**. • On pose la propriété • on vérifie son initialisation • puis sa conservation au cours de la boucle.
Les invariants de boucles sont essentiels pour **prouver** que le programme/algorithme a un certain comportement, autrement dit qu'il « fait bien ce que l'on veut qu'il fasse ».

P. Connault – Reproduction et diffusion soumises à autorisation

Exemple

Considérons la fonction `fonction2bis` ; montrons que la propriété « $x = 10^i$ » est un invariant de boucle.

1. **Avant la boucle :** Avant entrée dans la boucle on a $x = 1 = 10^0 = 10^i$.
2. **Conservation au cours de la boucle :** Notons x_p et i_p les valeurs respectives contenues dans x et dans i après la p eme iteration de la boucle. On a alors $x_{p+1} = 10x_p = 10 \times 10^{i_p} = 10^{1+i_p} = 10^{i_{p+1}}$. La propriété est donc conservée.

Exemple

Montrons que la propriété « $a = bq + r$ » est un invariant de boucle dans le programme `division1` précédent. Appelons q_n et r_n les valeurs contenues dans q et r lors de la n eme itération.

1. **Avant la boucle :** on a $a = b * 0 + a$, or avant la boucle $q_0 = 0, r_0 = a$. Donc $a = bq_0 + r_0$, c'est-à-dire $a = bq + r$.
2. **Conservation au cours de la boucle :** Supposons qu'après la n eme itération de la boucle, la variable q contient la valeur q_n, que r contient la valeur r_n, et que l'on a $a = bq_n + r_n$ lors de la n eme itération de la boucle ; alors à l'itération $n+1$, on a $r_{n+1} = r_n - b$ et $q_{n+1} = q_n + 1$. Ainsi $bq_{n+1} + r_{n+1} = b(q_n + 1) + (r_n - b) = bq_n + r_n = a$. La propriété est conservée.

Ainsi « $a = bq + r$ » est un invariant de boucle du programme qui précède.

Exercice 2 Montrer les invariants de boucle demandés :

- Code 2 : « Après la k eme itération, au sortir de la boucle, on a $i = 2^k$ »
- Code 3 : « A la k eme itération, i contient (debut $-k$) »
- Code 5 : « Après la k eme itération, la suite finie $(l[0], l[1], \cdots, l[i])$ est strictement croissante »
- Code 8 : A l'issue de n passages dans la boucle, on appelle a_n et b_n les valeurs contenues dans a et b. Invariant de boucle à prouver : « $a_n = \dfrac{30}{2^n}$ et $b_n = \dfrac{100}{2^n}$ ».

Méthode

Comment utiliser un invariant de boucle pour prouver la correction d'un programme/algorithme.
La propriété invariante de boucle est vraie à la sortie de la boucle **et** la condition de boucle est fausse **et** elle était vraie avant de sortir de la boucle.
Ces trois propriétés conjointes permettent souvent de prouver que le programme retourne le bon résultat.

Exemple

Considérons la fonction `fonction2bis` ; on note i_f et x_f les valeurs contenues dans i et x au sortir de la boucle. On a alors les trois propriétés suivantes : $10^{i_f} = x_f$ (invariant de boucle) ; $x_f > n$ (condition de boucle fausse) ; et $\dfrac{x_f}{10} \leqslant n$ (condition de boucle vraie à l itération précédnte).
Ainsi $n < 10^{i_f} \leqslant 10n$. Passons au logarithme en base 10 : $\log_{10}(n) < i_f \leqslant \log_{10}(n) + 1$ donc $i_f = \lfloor \log_{10}(n) \rfloor + 1$.
Autre conclusion si vous ne voyez pas à quoi correspond ce résultat : i_f est le plus petit entier i tel que $10^i > n$. C'est simplement le nombre de chiffres dans l'écriture décimale de n.

Exemple

Montrons que si $b > 0$, alors `division1(a,b)` retourne q,r respectivement le quotient et le reste la division euclidienne de a par b.
AUtrement dit on veut prouver : $a = bq + r$, $0 \leqslant r < b$, et $q, r \in \mathbb{Z}$.
En effet division1(a,b) termine, et retourne bien deux entiers q et r (par stabilité de $\mathbb{Z}$ par les opérations +, - et *)
De plus au sortir de la boucle on sait que :

- $a = bq + r$, car cette propriété est un invariant de boucle qui est donc vrai lorsqu'on sort juste de la boucle
- $r < b$ car la condition de boucle était $r \geqslant b$ et qu'elle n'est **plus** vérifiée car on est sorti de la boucle
- Enfin, notons R la valeur contenue dans la variable r à la dernière étape de la boucle ; on avait « $R \geqslant b$ » et on a ôté b à r, donc à la sortie de la boucle on a $r = R - b \geqslant 0$.

Ainsi $0 \leqslant r < b$; $a = bq + r$; $q \in \mathbb{Z}$; $r \in [\![0, b-1]\!]$. Par définition de la division euclidienne, q et r sont donc respectivement le quotient et le reste la division euclidienne de a par b.

Exercice 3 En utilisant les invariants de boucles prouvés, dire ce que « font » les codes 2, 5 et 8.

Remarque

Comment trouver les bons invariants de boucle ? **En général, il faut voir quelle est la propriété du programme qu'on veut montrer**. Ici, c'était « $a = bq + r$ et $r \in [\![0; b-1]\!]$ ». L'invariant de boucle est souvent une propriété proche, mais pas tout à fait la même.
Rappel : en mathématique, pour démontrer une assertion on se concentre sur sa conclusion. De façon proche, en informatique pour savoir quel invariant de boucle démontrer, on se concentre sur les propriétés attendues pour les objets retournés en sortie.

P. Connault – Reproduction et diffusion soumises à autorisation

1.3 Complexité

Définition 3

Un programme peut effectuer un grand nombre d'opérations, et un ordinateur stocker un grand nombre de données. Même si chacune des actions prend très peu de temps, et chaque stockage peu d'espace, **un grand nombre d'opérations et/ou un grand nombre de données peuvent prendre beaucoup de temps ou d'espace.** Ce coût est appelé la **complexité** : c'est le nombre d'opérations que fait un programme.
On distingue complexité en mémoire (taille et gestion de la mémoire) et complexité en temps (combien de temps prend le programme pour donner le résultat). La complexité en temps peut être envisagée dans le pire des cas, le meilleur des cas et en moyenne (hors-programme).
Nous n'abordons que la complexité en temps.

Exemple

Les deux programmes suivants renvoient le même résultat :

```
def somme1(n):
    return n*(n+1)//2
```

```
def somme2(n):
    s=0
    for k in range(n+1):
        s+=k
    return s
```

Le premier a un **cout constant** en fonction de n, tandis que le second effectue n passages de boucle : il a un coût proportionnel à n, ou encore **linéaire** en n.

Remarque

A votre niveau, on part souvent du principe que chaque opération $+, \times, -, \ldots$ a un cout constant. Le facteur déterminant est donc le nombre de passages dans les boucles présentes. Attention bien sûr à la présence d'appels à d'autres programmes, qui peuvent eux memes contenir des boucles.

Exercice 4 Concernant les fonctions suivantes, donner leur coût (temporel) en fonction de n.

```
def fonction1(n):
    x=0
    for i in range(n):
        x+=1
    return x
```

```
def fonction2(n):
    x=0
    for i in range(n):
        for j in range(n):
            x+=1
    return x
```

```
def fonction3(n):
    x=0
    for i in range(n):
        for j in range(i):
            x+=1
    return x
```

```
def fonction4(n):
    i=0
    while i**2<=n:
        i+=1
    return i
```

```
def fonction5(n):
    x=0
    for i in range(n):
       j = 0
       while j*j < i:
           x += 1
           j += 1
    return x
```

P. Connault – Reproduction et diffusion soumises à autorisation

```python
def fonction6(n):
    x=0
    i=2**n
    while i>1:
        i=i//2
        x+=1
    return x
```

```python
def fonction6bis(n):
    x=0
    i=n   ###on pose i=n ! difference
        avec la fonction precedente
    while i>1:
        i=i//2
        x+=1
    return x
```

```python
def fonction7(n):
    x=0
    i=n
    while i>1:
        j=2**n
        while j>1:
            j=j//2
            x+=1
        i=i//2
    return x
```

1.4 Notations de Landau

Dans les exemples précédents, on ne se focalise que sur l'ordre de grandeur des opérations, et on emploira les notations $\mathrm{O}()$ pour montrer que les temps constants permettant d'effectuer les opérations élémentaires ne nous intéressent pas. Ainsi la complexité de `somme1` est $\mathrm{O}(1)$, la complexité de `somme2` est $\mathrm{O}(n)$.

Les coûts les plus classiques sont les suivants pour un ordinateur personnel actuel, en ordre de grandeur :

c			valeurs	de	n		exemple
o		10^2	10^3	10^4	10^5	10^6	d'algorithme
m	$\ln(n)$	10^{-9}s	10^{-9}s	10^{-9}s	10^{-9}s	10^{-9}s	recherche par dichotomie ds liste
p	n	10^{-7}s	10^{-6}s	10^{-5}s	10^{-4}s	10^{-3}s	maximum d'une liste
l	$n\ln(n)$	10^{-6}s	$\cdots$	$\cdots$	$\cdots$	10^{-2}s	tri rapide d'une liste
e	n^2	10^{-5}s	10^{-3}s	$\cdots$	10s	20 minutes	tri par sélection d'une liste
x	n^3	10^{-3}s	1s	17s	10 jours	30 **ans**	multiplication de matrices – naïf
i té	2^n	4 **ans**	$15\times$ age de l'univers (depuis le 'Big Bang')				énumération des parties de $[\![1,n]\!]$ probleme du sac à dos

Noms à connaître :

complexité	Nom
$O(1)$	coût constant
$O(\ln(n))$	coût logarithmique
$O(n)$	coût linéaire
$O(n\ln(n))$	coût semi-linéaire
$O(n^2)$	coût quadratique
$O(n^k)$	coût polynomial
$O(2^n), O(a^n)$ avec $a>1$	coût exponentiel

Exercice 5 Donner les couts des algorithmes de l'exercice 1 en utilisant la notation de Landau.

Remarque

On comprend mieux pourquoi les constantes détaillées n'interviennent pas en première analyse : les ordres de grandeur dépendent avant tout du $\mathrm{O}(\text{nombre d'opérations})$. Une **distance particulièrement immense** [a] sépare les algorithmes de complexité exponentielle ($\mathrm{O}(k^n), k>1$) de ceux de complexité polynomiale $\mathrm{O}(n^\alpha)$. En pratique, on se permet au plus une complexité quadratique $\mathrm{O}(n^2)$.

a. Lire Why Philosophers Should Care About Computational Complexity de Scott Aaronson, par exemple.

Lorsque les paramètres sont plus complexes (listes, vecteurs par exemple) la durée d'éxécution peut être sujet à variabilité. On parle de complexité dans le pire des cas, meilleur des cas (et en moyenne).

P. Connault – Reproduction et diffusion soumises à autorisation

Exemple

Considérons les deux fonctions suivantes dont le comportement est identique

```
def appartient1(a,l):
    for x in l:
     if x==a:
      return True
    return False

def appartient2(a,l):
    n,i=len(l),0
    while i<n and not(l[i]==a):
        i=i+1
    return (i==n)
```

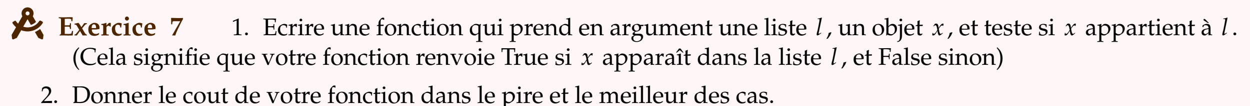
La complexité dans le meilleur des cas est O(1), et celle dans le pire des cas est O(len(l)).

Exercice 6 Ecrire une fonction vérifiant le cahier des charges demandé, et donner sa complexité dans le pire et le meilleur des cas :

1. Ecrire une fonction qui prend en argument une liste l et renvoie la somme des éléments de l.
2. Ecrire une fonction qui prend deux listes u et v et renvoie leur produit scalaire $u.v$.
3. Ecrire une fonction qui prend en argument une liste l et renvoie la liste l' obtenue en supprimant les doublons (consécutifs) d'une liste.
4. Ecrire une fonction qui prend en argument une liste l et deux nombres m et M, et décide si la liste l est majorée par M et minorée par m.
5. Ecrire une fonction qui prend en argument une chaine de caractères et décide si celle-ci est un palindrome.
6. Ecrire une fonction qui prend en argument une liste l et renvoie la même liste l sans repetitions consecutives. Par exemple [1,1,1,2,2,1,1,1,1,3,3,3,2,2] devient [1, 2, 1, 3, 2].
7. Ecrire une fonction qui prend en argument une matrice M (représentée par une liste de listes) et décide si celle-ci est une matrice symétrique.

2 Application à quelques fonctions vues en PCSI

2.1 Recherche par dichotomie



Exercice 7

1. Ecrire une fonction qui prend en argument une liste l, un objet x, et teste si x appartient à l. (Cela signifie que votre fonction renvoie True si x apparaît dans la liste l, et False sinon)
2. Donner le cout de votre fonction dans le pire et le meilleur des cas.

Exercice 8 **Recherche par dichotomie dans une liste triée (programme de PCSI)** On reprend l'exercice précédent, mais on suppose de plus que l est **triée** par ordre **croissant**.

1. Ecrire une fonction qui prend en argument une liste l, un objet x, et décide si x appartient à l. (Cela signifie que votre fonction renvoie True si x apparaît dans la liste l, et False sinon).
   Vous mettrez en place l'algorithme suivant :
   (a) poser $i = 0$, $j =$indice de fin de liste, $k = \frac{i+j}{2}$ indice milieu
   (b) Regarder l'élément $a = l[k]$ au milieu de la liste.
   (c) Si $x = a$, on a fini.
   (d) Sinon, selon que $x < a$ ou que $x > a$, remplacer i ou j par k, et prendre $k = \frac{i+j}{2}$ et recommencer le même procédé.
   (e) Boucler ainsi tant que $i < j$.

   Au besoin, prendre le programme du corrigé. Retenez–le par coeur, c'est du cours.
2. Prouver que le programme termine,
3. est correct,
4. et donner sa complexité.

P. Connault – Reproduction et diffusion soumises à autorisation

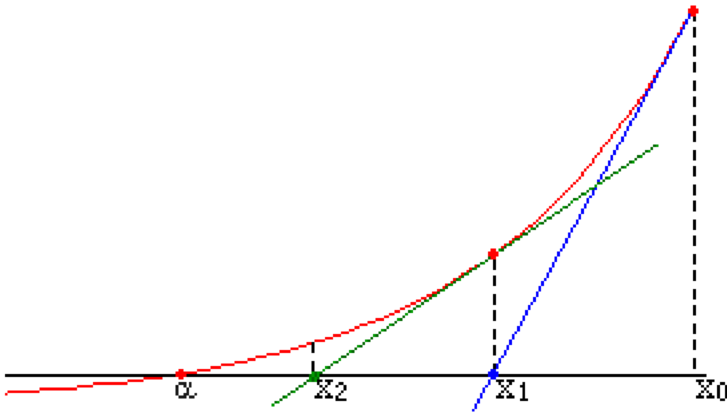


FIGURE 1 – Methode de Newton. Source : Wikipedia

2.2 Résolution d'équations du type $f(x) = 0$

Exercice 9 Si f est une fonction strictement monotone sur $[a, b]$, on peut chercher à résoudre l'équation $f(x) = 0$ par dichotomie.

1. Rappeler l'algorithme de dichotomie.
2. Ecrire les spécifications de la fonction recherchée, et sa ligne d'en–tête et dernière ligne.
3. Ecrire la fonction.
4. Prouver qu'elle termine.
5. Donner sa complexité.
6. Evaluer son coût en fonction de a, b, ϵ.

Exercice 10 Compléter le code ci-dessous :

```
def newton1(f,x, fprime, esp=10**(-8)):
    x0=x
    x1=x-f(x)/fprime(x)
    while abs(x0-x1)>eps and n_iter>0: ## on arrete qd ecart |xn - x(n+1)| 'petit'
        n_iter=n_iter-1
        x0, x1=x1, ???
    return x1
```

On décide de modifier le code ainsi :

```
def newton2(f,x, fprime, esp=10**(-8), n_iter=100):
    x0=x
    x1=x-f(x)/fprime(x)
    while abs(x0-x1)>eps and n_iter>0: ## on arrete qd ecart |xn - x(n+1)| 'petit'
        n_iter=n_iter-1
        x0, x1=x1, ...
    return x1
```

Expliquez.

2.3 Recherche de mot dans un texte

Exercice 11 On dit que la liste $[b_0, b_1, \ldots, b_m]$ est **incluse** dans $[a_0, a_1, \ldots, a_n]$ si $\exists k \in [\![1, n]\!]\ \forall j \in [\![1, m]\!]\ a_{j+k} = b_j$.
Par exemple, les listes `[1,2,3]`, `[1,3, 4]`, `[1,1]` sont incluses dans `[1,2,3,1,1,3,4]`; mais `[1,2,4]`, `[1,1,1]`, `[3,2,1]` ne le sont pas.
Ecrire un programme qui teste si une liste est incluse dans une autre. Terminaison, correction, complexité ?

3 Résumé

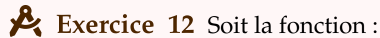
Un programme se prouve : « Terminaison, Correction, Complexité. »

1. Terminaison : absurde + suite convergente/suite strictement décroissante d'entiers (=variant de boucle).

P. Connault – Reproduction et diffusion soumises à autorisation

2. Correction : trouver, prouver, et utiliser invariant de boucle
3. Complexité : nombre d'opérations élémentaires en $O()$.

4 Exercices



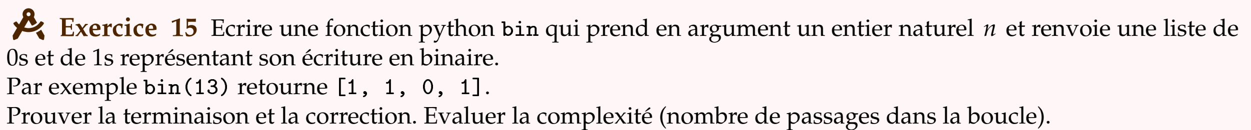
Exercice 12 Soit la fonction :

```python
def fonction_mystere(x,n):
    p = 1 # p comme produit
    a = x
    e = n # e comme exposant
    while e != 0:
        if e % 2 == 1:
            p = p*a
            e = e - 1
        else:
            e = e // 2
            a = a * a
    return(p)
```

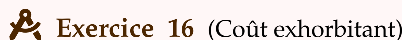
1. Prouver que si x est un flottant et n un entier naturel, alors `fonction_mystere` termine.
2. Prouver l'invariant de boucle « $x^n = pa^e$ »
3. En déduire par une phrase en français ce que calcule `fonction_mystere`
4. Donner le coût temporel de la fonction, et comparer avec son analogue « naïf »

Exercice 13 Rappeler l'algorithme du crible d'Eratosthène. Montrer que cet algorithme termine et est correct.

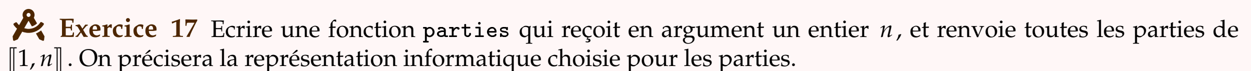
Exercice 14 Prouver la correction du programme `somme2`. La terminaison pose-t-elle problème ?



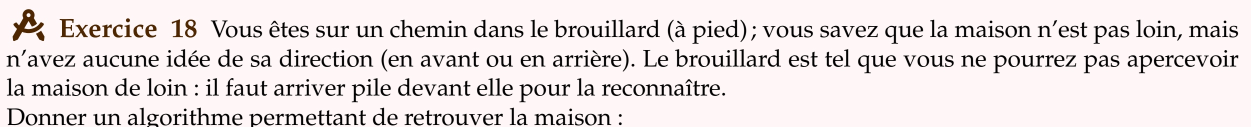
Exercice 15 Ecrire une fonction python `bin` qui prend en argument un entier naturel n et renvoie une liste de 0s et de 1s représentant son écriture en binaire.
Par exemple `bin(13)` retourne `[1, 1, 0, 1]`.
Prouver la terminaison et la correction. Evaluer la complexité (nombre de passages dans la boucle).



Exercice 16 (Coût exhorbitant)
Ecrire un programme sui prend en argument un entier strictement positif n et retourne la liste de toutes les listes de taille n contenant des 0 et des 1. Par exemple si $n = 3$, on attend la réponse : `[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]`
Evaluer le coût temporel et le coût spatial (=le nombre d'entiers à stocker en fonction de n) de votre programme.



Exercice 17 Ecrire une fonction `parties` qui reçoit en argument un entier n, et renvoie toutes les parties de $[\![1, n]\!]$. On précisera la représentation informatique choisie pour les parties.



Exercice 18 Vous êtes sur un chemin dans le brouillard (à pied) ; vous savez que la maison n'est pas loin, mais n'avez aucune idée de sa direction (en avant ou en arrière). Le brouillard est tel que vous ne pourrez pas apercevoir la maison de loin : il faut arriver pile devant elle pour la reconnaître.
Donner un algorithme permettant de retrouver la maison :

1. En temps quadratique dans le pire des cas,
2. En temps linéaire, est-ce possible ?

P. Connault – Reproduction et diffusion soumises à autorisation

5 Solutions

Solution

Solution 1.1
1. $(n-i)$
2. $(n-i)$

2bis (x_k) suite d'entiers strictement croissante et majorée par n

3. $(i_k - x)$ suite d'entiers strictement décroissante et minorée par 0
4. (j_k) suite d'entiers strictement croissante et majorée par $\sqrt{n}$; donc $(\sqrt{n} - j)$ variant de boucle
5. (i_k) suite d'entiers strictement croissante et majorée par n
6. (x_k) suite d'entiers strictement décroissante et minorée par y ; variant de boucle : $(y-x)$
7. $(b_n - a_n)$ suite d'entiers relatifs strictement décroissante et minorée par c ; variant de boucle : $b-a-c$
8. $(b_n - a_n)$ suite de flottants (réels), positifs, géométrique de raison $1/2$ donc tendant vers 0 et minorée par un nombre strictement positif c
9. Il n'y a que des boucles for (et i et j ne sont pas modifiés sinon) : le programme termine évidemment
10. Deux boucles imbriquées; la for termine toujours, et la while termine car à chaque éxécution on aura la suite (j_k) suite d'entiers strictement croissante et majorée par $\sqrt{i}$
11. Notons b_n la valeur contenue dans b suite à la n eme itération de boucle. On a $b_{n+1} = \dfrac{a+b_n}{2}$ de sorte que la suite b_n tend vers a. Ainsi $a_n - b$ n'est pas majoré par un $e > 0$
12. **Attention** ce programme ne termine pas pour les valeurs par défaut entrées. En effet Les valeurs prises pas c sont $c_0 = 1, \frac{1}{2}c_0, \frac{1}{4}c_0, \frac{1}{8}c_0$ etc et donc on a à l'étape n la valeur de a qui est $a_n = a_0 - \sum_{k=0}^{n} frc2^k$. On a $a_n \to a_0 - 2c_0$ lorsque n tend vers l'infini, ce qui ne garantit **pas** que a_n est inférieur à b pour une certaine valeur de n.

Solution

Solution 1.2
- Code 2 : Soit i_f la valeur de i une fois sorti de la boucle. On a donc $i_f \geqslant n$ et (passage précédent) $\frac{i_f}{2} < n$. De plus après la k eme itération, au sortir de la boucle, on a $i = 2^k$. On a donc $i = 2^k$ et $2^k \geqslant n > 2^{k-1}$. Ainsi i contient la plus petite puissance de 2 qui dépasse n. Comme i ets la valeur retournée par la fonction, on a donc prouvé que `fonction2(n)` retourne la première puissance de 2 qui dépasse n.
- Code 5 : Soit i la valeur lors de la sortie de boucle; « après la i eme itération, la suite finie $(l[0], l[1], \cdots, l[i])$ est strictement croissante » ; de plus comme la condition de boucle est fausse pour i on a $l[i] \geqslant l[i+1]$.
  Ainsi la fonction retourne l'indice i tel que la liste $[l[0], l[1], \cdots, l[i])$ est stmt croissante et qui est maximal pour cette propriété : c'est donc la plus longue séquence croissante partant du premier terme de la liste.
- Code 8 : A l'issue des différents passages dans la boucle, on a $a = \dfrac{30}{2^n}$ et $b = \dfrac{100}{2^n}$ où n ets le nombre de pasages de boucle. On a de plus $\dfrac{100}{2^n} - \dfrac{30}{2^n} \leqslant 10 < \dfrac{100}{2^{n-1}} - \dfrac{30}{2^{n-1}}$ de sorte que $7 \leqslant 2^n < 14$ et $n = 3$.
  On trouve donc que les contenus des variables sont $a = \dfrac{30}{8}$ et $b = \dfrac{100}{8}$

Solution

Solution 1.3
`fonction1` On fait n opérations,, soit un coût de n

`fonction2` On fait n fois n opérations, soit un coût de n^2

`fonction3` La boucle `for j in range(i)` effectue i opérations, et elle est appelée pour i allant de 0 à $n-1$ c'est-à-dire :

$$0 + 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} \text{ opérations.}$$

`fonction4` La condition de la boucle while devient fausse pour $i^2 > n$, elle est donc exécutée pour i valant successivement $0, 1, \cdots, \lfloor\sqrt{n}\rfloor$: cela fait $\lfloor\sqrt{n}\rfloor + 1$ opérations

`fonction5` On reprend l'analyse du programme qui précède, mais pour chaque valeur de i dans la boucle for. Cela fait $\sum_{i=0}^{n}(\lfloor\sqrt{i}\rfloor + 1)$ opérations.

`fonction6` A chaque itérations i est divisé par 2 ; au cours de la boucle il prendra donc comme valeurs la suite des puissances de 2 (en décroissant), et terminera par 1 puis 0 (rappel : `1//2` rend 0, c'est le quotient de la division euclidienne de 1 par 2). Le nombre d'opérations est donc n.

`fonction6bis` Cette fonction divise n par 2 tant qu'on obtient un résultat > 1. Si on s'arrêt au bout de N itérations de boucle, cela signifie que $\dfrac{n}{2^N} \leqslant 1 < \dfrac{n}{2^{N-1}}$ c'est-à-dire : $2^{N-1} < n \leqslant 2^N$. On en déduit que $N = \lceil\log_2(n)\rceil$

`fonction7` On a $n\lceil\log_2(n)\rceil$ opérations.

Solution

Solution 1.4 A savoir $\lfloor x\rfloor \sim \lceil x\rceil \sim x$. En effet on a $\lfloor x\rfloor \leqslant x < \lfloor x\rfloor + 1$, donc $x-1 < \lfloor x\rfloor \leqslant x$ puis $\frac{x-1}{x} < \frac{\lfloor x\rfloor}{x} \leqslant \frac{x}{x}$ et par gendarmes, $\frac{\lfloor x\rfloor}{x} \to 1$ c'est-à-dire $\lfloor x\rfloor \sim x$.

`fonction1` On fait n opérations,, soit un coût linéaire $O(n)$

`fonction2` On fait n fois n opérations, soit un coût quadratique $O(n^2)$

`fonction3` La boucle `for j in range(i)` effectue i opérations, et elle est appelée pour i allant de 0 à $n-1$ c'est-à-dire :

$0+1+2+\cdots+(n-1) = \frac{(n-1)n}{2} = O(n^2)$ opérations. Le coût est donc quadratique.

`fonction4` La condition de la boucle while devient fausse pour $i^2 > n$, elle est donc éxécutée pour i valant successivement $0, 1, \cdots, \lfloor\sqrt{n}\rfloor$: cela fait $\lfloor\sqrt{n}\rfloor + 1 = O(\sqrt{n})$ opérations

`fonction5` On reprend l'analyse du programme qui précède, mais pour chaque valeur de i dans la boucle for.
Cela fait $\sum_{i=0}^{n}(\lfloor\sqrt{i}\rfloor + 1) = \sum_{i=0}^{n}\lfloor\sqrt{i}\rfloor + \sum_{i=0}^{n} 1 = \sum_{i=0}^{n}\lfloor\sqrt{i}\rfloor + O(n)$ opérations. Mais on a $\sum_{i=0}^{n}\lfloor\sqrt{i}\rfloor \leqslant \sum_{i=0}^{n}\sqrt{n} = O(n\sqrt{n})$. Ainsi on a $O(n\sqrt{n}) + O(n) = O(n\sqrt{n})$ opérations.
Façon plus informaticenne de rédiger : le coût est $\sum_{i=0}^{n} O(\sqrt{i}) = nO(\sqrt{n}) = O(n\sqrt{n})$

`fonction6` A chaque itérations i est divisé par 2 ; au cours de la boucle il prendra donc comme valeurs la suite des puissances de 2 (en décroissant), et terminera par 1 puis 0 (rappel : `1//2` rend 0, c'est le quotient de la division euclidienne de 1 par 2). Le nombre d'opérations est donc $n = O(n)$, cout linéaire.

`fonction6bis` Cette fonction divise n par 2 tant qu'on obtient un résultat > 1. Si on s'arrêt au bout de N itérations de boucle, cela signifie que $\frac{n}{2^N} \leqslant 1 < \frac{n}{2^{N-1}}$ c'est-à-dire : $2^{N-1} < n \leqslant 2^N$. On en déduit que $N = \lceil\log_2(n)\rceil = O(\log_2(n)) = O(ln(n))$, cout logarithmique.

`fonction7` On a $n\lceil\log_2(n)\rceil = O(n\ln n)$ opérations ; cout semi-linéaire.

Solution

Solution 1.4

```python

def somme(l):
    S=0
    for x in l:
        S+=x
    return S

def produit_scalaire(u,v):
    n=len(u)
    S=0
    for i in range(n):
        S+=u[i]*v[i]
    return S


def est_bornee_par(l,m,M):
    for x in l:
        if x<m or x>M:
            return False
    return True



def est_palindrome(ch):
    n=len(ch)
    i=0
    j=n-1
    while i<=j and ch[i]==ch[j] :
        i+=1
        j-=1
    return i>j


def supprime_repetitions_consecutives(l):
    i=0
    while i+1< len(l):
        if l[i]==l[i+1]:
            del l[i]
        else:
            i+=1
    return l


def est_symetrique(M):
```

P. Connault – Reproduction et diffusion soumises à autorisation

```
45      n=len(M)
46      M_est_sym=True
47      i,j=0,0
48      while j<n and M_est_sym:
49          M_est_sym=(M[i][j]==M[j][i])
50          i+=1
51          if i >=n:
52              i=0
53              j+=1
54      return M_est_sym
```

Solution

Solution 2.1

```
1
2   def recherche(x,l):
3       n=len(l)
4       for i in range(n):
5           if l[i]==x:
6               return True
7       return False
```

Le cout est $O(n)$ avec n la longueur de la liste ℓ.

Solution

Solution 2.1 1.

```
1
2   def recherche_dicho(x, l):
3       i=0
4       j =len(l)
5       while i < j:
6           k = (i + j) // 2
7           if l[k] == x:
8               return True
9           elif l[k] > x:
10              j = k
11          else:
12              i = k + 1
13      return False
```

Terminaison : au cours de la boucle on a $j - i//2$ qui est une suite d'entiers strictement décroissante (car $j > i$) et minorée par 0. **Correction** Invariant de boucle « $x \in l[i:j]$ ou $x \notin l$ ». En effet, tautologiqument vrai avant la boucle. AU cours de la bocule, conservé car l est triée par ordre croissant.

Ainsi ou bien x apparait dans l, et on a le résultat ; ou bien non et on se retrouve avec $i \geqslant j$ en sortie de boucle. L'invariant dit alors « $x \in []$ ou $x \notin l$ ».

Complexité Meilleur des cas : O(1) (on trouve tout de suite que $a = x$).

Pire des cas : on effectue un comparaison dans l, une dans « $l/2$ », une dans « $l/4$ », etc jusqu'à ne plus pouvoir diviser l en deux morceaux, ie jusqu'à se retouver avec une liste « $l/2^m$ » avec $2^m \leqslant \mathrm{lgr}(l) < 2^{m+1}$. On a donc $m = \log_2(\mathrm{lgr}(l)) = \mathrm{O}(\ln(\mathrm{lgr}(l)))$.

Le cout de la recherche dichotomique est logarithmique en la longueur de la liste

Solution

Solution 2.2

```
1
2   def dichotomie(f,a,b,eps=10**(-12)):
3       assert f(a)*f(b)<0
4       u,v=a,b
5       while abs(v-u)>eps:
6           c=(u+v)/2
7           if f(u)*f(c)<=0:
8               v=c
9           else:
10              u=c
11      return((u+v)/2)
```

Terminaison. Appelons u_n (resp v_n) la valeur de a (resp b) lors du n ieme passage dans la boucle while. Supposons

P. Connault – Reproduction et diffusion soumises à autorisation

la boucle infinie. Alors la suite $|v_n - u_n| \underset{\text{géometrique}}{=} \frac{|b-a|}{2^n} \to 0$ donc $0 \geqslant$ eps, ce qui ne peut être vérifié car eps>0 vu la première ligne.

Correction. On a l'invariant de boucle « $f(u_n)f(v_n) \leqslant 0$ ». Appelons $c^* \in \mathbb{R}$ la vraie valeur telle que $f(c^*) = 0$. On a alors que les nombres c^* et $\frac{u_n+v_n}{2}$ sont tous les deux compris entre u_n et v_n et $|v_n - u_n| \leqslant$ eps. Ainsi $|c^* - \frac{u_n+v_n}{2}| \leqslant$eps.

Complexité. Soit n le nombre d'itérations pour sortir de la boucle. On a $\frac{b-a}{2^{n+1}} \leqslant \epsilon < \frac{b-a}{2^n}$ donc $n = \lceil \log_2(\frac{b-a}{\epsilon}) \rceil \sim -\ln(\epsilon)$. Si $\epsilon = \frac{1}{N}$, alors la complexité est $O(\ln(N))$. Elle est linéaire en 1/(l'ordre de grandeur) de ϵ.

Solution

Solution 11 Par exemple :

```python
def est_inclus(grand, petit):
    n=len(grand)
    m=len(petit)
    inclusion=(petit==[])
    for k in range(n):
        inclusion= True
        for l in range(m):
            if (k+l>=n) or (grand[k+l]!=petit[l]):
                inclusion=False
                break
        if inclusion:
            break
    return inclusion
```

1. Terminaison : évidente car boucle for
2. Correction : l'invariant de boucle est « petit n'est pas incluse dans grand[0, k-1], et [petit[0], ...,petit[j]]=[grand[k], ..., grand[k+j]] »
3. complexité : dans le pire des cas, on effectue environ nm comparaisons (précisément, $(n-m)m$). Dans le meilleur, on en effectue m. La complexité de `est_inclus` est $O(mn)$ (pire des cas) et $O(m)$ (meilleur des cas).

Solution

Solution 4

1. **Terminaison** La variable e contient un entier naturel, et les opérations effectuées sur e conservent cette propriété. Ainsi si (par l'absurde) la boucle est infinie, alors les valeurs prises par e sont strictement positives, mais dans ce cas les opérations effectuées sur e produisent une suite d'entiers strictement décroissante : absurde. Donc la fonction termine.
Invariant de boucle
2. Initialisation : on a $pa^e = 1x^n = x^n$.
3. Hérédité : Notons $p_m; a_m; e_m$ les contenus des variables p, a, e à la m eme itération dans le boucle. Si à l'itération m on a $x^n = p_m a_m^{e_m}$, alors à l'itération $m+1$ on a
OU BIEN e_m est impair, et donc $p_{m+1} = p_m \times a_m$, $e_{m+1} = e_m - 1$, $a_{m+1} = a_m$.
Ainsi $p_{m+1}a_{m+1}^{e_{m+1}} = p_m \times a_m (a_m)^{e_m-1} = p_m a_m^{e_m} = x^n$.
OU BIEN e_m est pair, et donc $p_{m+1} = p_m$, $e_{m+1} = \frac{e_m}{2}$, $a_{m+1} = a_m^2$.
Ainsi $p_{m+1}a_{m+1}^{e_{m+1}} = p_m(a_m^2)^{\frac{e_m}{2}} = p_m \times a_m^{e_m} = x^n$.
On a donc prouvé l'invariant de boucle.
Montrons que le programme retourne x^n. En effet à l'issue de la boucle on a à la fois $e = 0$ et $pa^e = x^n$. Ainsi on trouve $p = x^n$, ce que la fonction retourne.
Complexité
4. Remarquons que l'on a toujours $e_{m+2} \leqslant \frac{e_m}{2}$. En effet sur deux itérations successives, la valeur de e passe toujours par une valeur paire et sera divisée par 2. Ainsi au bout de $2k$ itérations, partant de $e_0 = n$, on a $e_{n-2k} \leqslant \frac{n}{2^k}$, et aussi $e_{n-2k-1} \leqslant \frac{n}{2^k}$ Ainsi dès que $\frac{n}{2^k} < 1$, le programme termine ; en particulier, soit k l'entier tel que $\frac{n}{2^k} < 1 \leqslant \frac{n}{2^{k-1}}$, c'est-à-dire tel que $k-1 \leqslant \log_2(n) < k$ ou encore : $k = \lfloor \log_2(n) \rfloor + 1$. Alors au bout $2k$ itérations au plus, la boucle termine.
On a donc une complexité dans le pire des cas qui est $2(\lfloor \log_2(n) \rfloor + 1) \leqslant \frac{2\ln n}{\ln 2} + 1 = O(\ln n)$.
L''exponentiation naive, en n'employant que des multiplications, est bien sûr de cout linéaire en n. Ainsi la

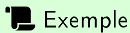
P. Connault – Reproduction et diffusion soumises à autorisation

fonction mystere, dont l'algorithme est appelé **exponentiation rapide**, a un cout bien moindre puisque logarithmique en n.

Solution

Solution 4 C'est un **algorithme donnant la liste des tous les nombres premiers strictement inférieurs** à un entier n donné. On suppose que l'on dispose de deux opérations : marquer un nombre comme premier, et supprimer un élément.

1. Soit L la liste de tous les entiers de 2 à $n-1$.
2. **Tant qu'il** y a dans L des éléments non marqués premiers :
   Soit p le premier nombre de L non marqué premier. Supprimer dans L tous les multiples stricts de p. Marquer p comme premier.
3. Les nombres premiers strictement inférieurs à n sont les nombres marqués premiers.

 Exemple

Trouvons tous les nombres premiers inférieurs ou égal à 15.
On obtient les liste suivantes :

1. 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2. **2** 3 5 7 9 11 13 15
3. **2 3** 5 7 11 13
4. **2 3 5** 7 11 13
5. ...
6. **2 3 5 7 11 13**

1. Terminaison : Le nombre de nombre marqués diminue strictement à chaque itération, et on ne rajoute jamais d'élément. Ainsi la condition de boucle finit par devenir fausse.
2. Correction : l'invariant de boucle est « Tous les entiers marqués sont premiers et tous les effacés sont non premiers » Preuve :

   - Initialisation :on commence par marquer 2, qui est le plus petit nombre premier
   - Hérédité : soit b le plus petit nombre non marqué strictement supérieur à a. Alors b n'est pas composé, sinon il aurait un diviseur premier strict ; cela aurait amené à l'effacement de b au cours d'un des passages qui précèdent. Ainsi b est premier. Les nombres inférieurs ou égaux à b, marqués, sont premiers. De plus, les nombres qui ont été effacés sont tous non premiers. Ainsi les nombres présents marqués sont tous premiers, les nombres effacés tous non premiers.

   Ensuite, au sortir de la boucle, tous les nombres inférieurs à n ou bien sont marqués et premiers, ou bien effacés et non premiers. La liste contient donc exactement les nombres premiers entre 2 et n.

Solution

Solution 4 Le programme termine car il ne fait appel à aucun programme qui pourrait ne pas terminer, et ne contient pas de boucle while.
La propriété :« Avant de reboucler, s $= \frac{k(k+1)}{2}$ » est un invariant de boucle.

Solution

Solution 4 **Solution 1**

```

def plugrandepuissance(n):
    k=0
    while 2**k <=n:
        k+=1
    return k-1

def bin_lent(N):
    n=N
    m=plugrandepuissance(n)
    res=[1]+[0]*(m)
    n=n-2**m
    while n!=0:
        k=plugrandepuissance(n)
        res[m-k]=1
        n=n-2**k
    return res
```

Les deux programmes terminent car les suites (2^n) et $(n-2^k)$ ne sont pas bornées.

P. Connault – Reproduction et diffusion soumises à autorisation

Le programme `plugrandepuissance` calcule $\max\{j \in \mathbb{N} \mid 2^j \leqslant n\}$. On le prouve car un invariant maintenu au cours de la boucle dès la deuxième itération est « $2^{k-1} \leqslant n$ », et au sortir de la boucle on a $2^k > n$ Ainsi on a $2^{k-1} \leqslant n < 2^k$ et on retourne $k-1$.
Le programme `bin_lent` vérifie l'invariant de boucle « $\sum_{j=k}^{m} r_j 2^j \leqslant N < \sum_{j=k}^{m} r_j 2^j + 2^k$ et $n = N - \sum_{j=k}^{m} r_j 2^j$ » c'est en effet initialisé par définition de m qui vérifie $2^m \leqslant N < 2^m + 2^m$; et héréditaire pour la mêm raison : k vérifie en effet $2^k \leqslant n < 2^k + 2^k$.
On a bien à la sortie de boucle $\sum_{j=k}^{m} r_j 2^j \leqslant N < \sum_{j=k}^{m} r_j 2^j + 2^k$ et $n = N - \sum_{j=k}^{m} r_j 2^j = 0$ donc $N = \sum_{j=k}^{m} r_j 2^j$, qui est la décomposition en base 2 de N.

La complexité en temps du premier programme est exactement k, et de $2^k \leqslant n < 2^{k+1}$ on tire $k = \lfloor \frac{\ln n}{\ln 2} \rfloor$.
La complexité en temps du second programme est plus compliqué à évaluer.
Dans le meilleur des cas, on a N ets une puissance de 2 et on effectue un seul appel à `plugrandepuissance`. Complexité $O(\ln(N))$.
Dans ce qu'on peut deviner être le pire des cas, $N = \sum_{k=0}^{m} 2^k = 2^{m+1} - 1$: En fait $N = \underbrace{111\ldots1}_{m \text{ fois}}$ en base 2. On fera alors m appels à `plugrandepuissance`, et les coûts successifs seront $m-1, m-2, \ldots, , 3, 2, 1$: c'est-à-dire $\mathrm{O}(m^2)$ opérations. Le coût est alors en $\mathrm{O}(\ln(n)^2)$.
Solution 2
On implémente

```
plus_gde_puissance_bis(n):
    return floor(ln(n)/ln(2))
```

qui est de coût constant. Le coût du programme précédent passe alors à $O(\ln(n))$.
Solution 3

```
def bin(N):
    """Convertit un nombre en binaire"""
    n=N
    q = N
    res = []
    while q != 0:
        q = n // 2
        r = n % 2
        res = [r] + res
        n = q
    return res
```

On appelle q_k la valeur de q à la k eme itération de la boucle.
Si le programme ne termine pas alors $(q_k)_{k \geqslant 2}$ est une suite d'entiers strictement positifs, et strictement décroissante. (en effet $q_{k+1} \leqslant \frac{1}{2} q_k < q_k$). Absurde, donc le programme termine.
Ecrivons $n = \sum_{j=0}^{m} a_j 2^j$ l'écriture en binaire de n (avec $a_j \in \{0, 1\}$ pour tous j).
On montre l'invariant de boucle suivant :« A la fin de la k eme itération, $q_k = \sum_{j=k}^{m} a_j 2^{j-k}$ et $res = [a_{k-1}, \ldots, a_2, a_1, a_0]$ »
En effet si la propriété est vraie à la fin de k eme iutéaration, alors on a à la fin de l'itération suivante (la $k+1$ eme) : $\sum_{j=k}^{m} a_j 2^{j-k} = 2\left(\sum_{j=k+1}^{m} a_j 2^{j-k}\right) + a_k$ de sorte que $q_{k+1} = \sum_{j=k+1}^{m} a_j 2^{j-k}$ et $r = a_k$, puis $res = [a_k, a_{k-1}, \ldots, a_2, a_1, a_0]$
A la sortie de la boucle on a $q = 0$ ce qui signifie que la somme $\sum_{j=k}^{m} a_j 2^{j-k}$ est alors vide : on a donc $k = m+1$, et on avait $k = m$ lors de la dernière itération. A ce stade,le contenue de res était $[a_m, a_{m-1}, \ldots, a_2, a_1, a_0]$ qui est exactement l'écriture binaire de n.
A chaque passage, on écrit une décimale de la décomposition : or le nombre total m de décimales vérifie $2^m \leqslant n < 2^{m+1}$ *i.e.* $m = \lfloor \frac{\ln n}{\ln 2} \rfloor$. C'est le nombre de passages dans la boucle, ainsi on trouve un coût en $\mathrm{O}(\ln(n))$.
C'est un coût **logarithmique en** n. Cependant, c'est un coût **linéaire en le nombre de décimales de** n**.**

Solution

Solution 4

```


def parties_zero_un(n):
    liste=[]
    for i in range(2**n):
        longueur=len(bin(i))  "conversion en binaire"
        liste.append((n-longueur)*[0]+bin(i)) "on rajoute des 0s pour mettre la liste a
            la bonne taille"
    return liste
```

Cout temporel $= \sum_{i=1}^{2^n-1} O(\ln(i)) = \mathrm{O}(\ \ln[(2^n-1)!]\ )$. On peut aussi remarquer que le cout est d'une prt $> \sum_{i=0}^{2^n-1} O(1) = \mathrm{O}(2^n)$ (c'est donc au moins exponentiel), et d'autre part $< \ln(2^n)2^n = O(n2^n)$.

P. Connault – Reproduction et diffusion soumises à autorisation

Solution

Solution 4 Si on choisit de représenter une partie par une liste de 0s et de 1s, alors on a **déjà répondu lors de l'exercice qui précède**.
Si on choisit de représenter une partie par la liste croissante de ses éléments, on a besoin d'un traducteur :

```


def parties_zero_un(n):
    liste=[]
    for i in range(2**n):
        longueur=len(bin(i))
        liste.append((n-longueur)*[0]+bin(i))
    return liste

def traduction(liste): #transforme la liste reprsentant une indicatrice en liste de parties de [1,n]
    liste_traduite=[]
    for i in range(len(liste)):
        if liste[i]==1:
            liste_traduite.append(i+1)
    return liste_traduite

def parties(n):
    return [traduction(l) for l in parties_zero_un(n)]
```