NSI en première (2019-2020)
Résumé Algorithmes 1

1 Introduction

Les écrits concernant la résolution d'équations de Al-Khwârizmî (780-850), savant de Bagdad, sont traduits en latin vers le XII^{e} siècle.

L'algorithme d'Euclide est l'un des plus ancien algorithme non trivial (livre VII des Éléments, datant d'environ 2300 ans).

On trouve aussi des algorithmes datant d'environ 2000 ans avant notre ère à Bagdad (voir Donald Knuth).

Algorithme d'Euclide
Calcul du pgcd de deux entiers m et n.
Étape 1 : on divise m par n ; soit r le reste ($0 \leq r < n$).
Étape 2 : si $r = 0$, le pgcd est n.
Étape 3 : sinon, on remplace m et n par n et r et on recommence à l'étape 1.

Programme en Python :

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

Des règles énoncées par Donald Knuth dans The Art of Computer Programming, ouvrage commencé dans les années 1960 :

— terminaison après un nombre fini d'étapes ;
— actions décrites de manière rigoureuse sans aucune ambiguïté ;
— des entrées, zéro ou plus, données avant ou pendant l'exécution.
— des sorties, quantités en relation avec les entrées.
— des instructions les plus basiques possibles.

2 Outils

2.1 Compteurs et accumulateurs

Un compteur permet de compter le nombre de passages dans une boucle ou le nombre d'actions d'un certain type exécutées dans une boucle.

Exemples de compteurs

```
def taille(n):
    cpt = 0
    while n > 0:
        cpt = cpt + 1
        n = n // 2
    return cpt
```

ou bien :

```
def nombre_de_1(n):
    cpt = 0
    while n > 0:
        if n % 2 == 1:
            cpt = cpt + 1
        n = n // 2
    return cpt
```

Exemple d'accumulateur
Calcul de la somme des termes d'une liste de nombres `liste` :

```
def somme(liste):
    acc = 0
    for x in liste:
        acc = acc + x
    return acc
```

2.2 Permutation de deux valeurs

Utilisation d'une troisième variable :

```
var1 = 15
var2 = 11
temp = var1
var1 = var2
var2 = temp
```

Une instruction simple en Python : `var1, var2 = var2, var1`.

2.3 Tests et boucles

Les tests
Tris types de structures : `if ...`, ou `if ... else ...`, ou `if ... elif ... else ...`.

Remarque : Si `x` est un booléen ou si `f(x)` est un booléen, on écrit `if x` ou `if f(x)`.
Il faut éviter d'écrire `if x == True` ou `if f(x) == True`.

Les boucles
Boucles `while` (tant que) et boucles `for` (pour).
Il faut bien comprendre l'utilisation de boucles imbriquées.
Exemple :

```
for i in range(5):
    j = 0
    while j <= i:
        print(i + j)
```

3 Validité et coût

Un algorithme est valide s'il produit un résultat en un temps fini et que ce résultat est correct (c'est-à-dire conforme à une spécification précise). Le coût en temps d'un algorithme nous indique s'il est utilisable ou pas avec un certain type de données.

3.1 Validité d'un algorithme itératif

Pour prouver la terminaison d'un algorithme itératif, nous utilisons la notion de variant qui est souvent une expression simple dont la suite des valeurs converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt.

Pour prouver la correction, nous utilisons la notion d'invariant de boucle. Il s'agit d'une propriété vérifiée avant l'entrée dans une boucle, à chaque passage dans cette boucle et à la sortie de cette boucle.

3.2 Coût

On parlerons du coût d'un algorithme ou de sa complexité.

Étudions quelques exemples simples qui sont à la base de ce type d'étude. Nous commençons par un algorithme rencontré précédemment.

Complexité linéaire
Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n + \beta$, avec α et β réels, $\alpha > 0$, nous disons que l'algorithme a un coût linéaire ou une complexité linéaire.

Complexité quadratique
Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n^2 + \beta n + \gamma$, avec α, β et γ réels, $\alpha > 0$, nous disons que l'algorithme a un coût quadratique ou une complexité quadratique.

Exemples avec des boucles "for" imbriquées ; dans le premier cas, le coût est linéaire. Dans les deux autres cas, le coût est quadratique.

Premier cas : `n` est la taille de la donnée, `k` est un nombre fixé.

```
for i in range(n):
    ...
    for j in range(k):
        ...
```

Deuxième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(n):
        ...
```

Troisième cas : `n` est la taille de la donnée.

```
for i in range(n):
    # q opérations
    for j in range(i):
        # r opérations
```

Le nombre total d'opérations est $\frac{r}{2}n^2 + (q - \frac{r}{2})n$ de la forme $\alpha n^2 + \beta n$ et le coût est quadratique.

4 Parcours séquentiel

4.1 Calcul d'une moyenne

Nous utilisons dans cette partie des variables de type list que nous appelons des listes. Nous pouvons aussi utiliser des p-uplets. Un parcours séquentiel signifie que la liste est parcourue élément par élément, suivant l'ordre des éléments.

Le calcul d'une moyenne s'effectue comme le calcul d'une somme déjà vu auparavant. On utilise un accumulateur.

Nous supposons la liste non vide.

```
def moyenne(liste):
    n = len(liste)
    s = 0
    for u in liste:
        s = s + u
    return s / n
```

Le coût est linéaire en n la taille de la liste, puisque nous avons n additions et n affectations effectuées dans la boucle. Nous supposons, et c'est le cas, que l'instruction `n = len(liste)` a un coût constant.

4.2 Recherche d'une occurrence

Il s'agit de rechercher de manière séquentielle la présence d'une valeur dans un tableau. Cette méthode est aussi appelée méthode par balayage ou recherche linéaire (en anglais, linear search). Un tableau peut être ici une liste, un p-uplet ou une chaîne de caractères. On cherche donc une valeur précise dans une liste ou un p-uplet ou un caractère précis dans une chaîne. Pour cela on compare la valeur cherchée successivement à toutes les valeurs du tableau.

L'algorithme s'arrête dès que l'élément est trouvé ou si la fin du tableau est atteinte.

Algorithme de recherche d'un élément `x` dans un tableau `t` de longueur `n` :

```
i = 0
tant que i < n et x différent de t[i]
    i = i + 1
fin tant que
si i < n
    renvoyer i
```

Programme en Python :

```
def recherche(x, t):
    n = len(t)
    i = 0
    while i < n and x != t[i]:
        i = i + 1
    if i < n:
        return i
```

Pour évaluer le coût, nous comptons le nombre de comparaisons qui sont effectuées.

Si l'élément recherché n'est pas dans le tableau ou si c'est celui qui est examiné en dernier, il est nécessaire de parcourir tout le tableau, donc d'effectuer n itérations si la taille du tableau est n, pour examiner chaque élément du tableau avec n comparaisons. Le coût est donc linéaire en n.

En Python, on peut aussi utiliser une boucle for avec une sortie de boucle éventuellement prématurée.

```
def recherche(x, t):
    for i in range(len(t)):
        if t[i] == x:
            return i
```

4.3 Recherche d'un extremum

Nous disposons d'un ensemble de nombres dans lequel nous cherchons un extremum. Ce peut être un maximum, un minimum, ou les deux.

Algorithme de recherche du maximum : si la liste est non vide, on suppose que le maximum est le premier élément, puis on parcourt la liste et chaque fois qu'on rencontre un élément plus grand que le maximum provisoire, on dit que c'est le nouveau maximum provisoire.

Il s'agit encore d'un parcours séquentiel dans une liste de nombres.

```
def maximum(liste):
    maxi = liste[0]
    for x in liste:
        if x > maxi:
            maxi = x
    return maxi
```

Le même algorithme est utilisé pour chercher une valeur approchée du maximum d'une fonction f sur un intervalle $[a; b]$.

```
def maximum(f, a, b, n):
    maxi = f(a)
    dx = (b - a) / n
    x = a
    for k in range(n):
        x = x + dx
        y = f(x)
        if y > maxi:
            maxi = y
    return maxi
```

La recherche d'un minimum est similaire. Il suffit de remplacer le signe $>$ par le signe $<$ dans la comparaison.

La recherche d'un maximum ou d'un minimum est basée sur le parcours séquentiel d'une liste avec un nombre borné d'opérations pour chaque élément. Le coût est donc linéaire.

On peut rechercher conjointement le minimum et le maximum.

Exemple de recherche dans une liste :

```
def minmax(liste):
    mini = liste[0]
    maxi = liste[0]
    for x in liste:
        if x > maxi:
            maxi = x
        if x < mini:
            mini = x
    return mini, maxi
```

5 Recherche dichotomique

La recherche par dichotomie s'effectue dans un tableau trié.

On utilise ici le type `list` pour pouvoir préalablement trier le tableau. Pour cela, nous disposons avec Python de la fonction `sorted` qui prend en argument une liste et renvoie la liste triée. La liste initiale n'est pas modifiée. Nous disposons aussi de la méthode `sort` qui trie la liste à laquelle elle s'applique.

```
>>> liste = [4, 1, 3, 2]
>>> liste2 = sorted(liste)
>>> liste2
[1, 2, 3, 4] # liste2 est triée
>>> liste
[4, 1, 3, 2] # liste n'est pas modifiée
>>> liste.sort()
>>> liste
[1, 2, 3, 4] # liste est triée
```

Principe de dichotomie (binary search en anglais) : à chaque étape, on coupe le tableau en deux et on effectue un test pour savoir dans quelle partie se trouve l'élément cherché. C'est le principe *diviser pour régner* (en anglais *divide-and-conquer*). Voici un exemple de programme :

```
def dichotomie(x, liste):
    g = 0
    d = len(liste)
    while g < d - 1:
        k = (g + d) // 2
        if x < liste[k]:
            d = k
        else:
            g = k
    if x == liste[g]:
        return g
    else:
        return False
```

Preuve de la terminaison avec le variant de boucle $d - g$.

Si la taille du tableau est inférieure à 2^n, après k itérations, $d - g \leq \dfrac{2^n}{2^k}$ soit $d - g \leq 2^{n-k}$. La suite des valeurs $d - g$ est strictement décroissante et après n étapes, $d - g \leq 1$, donc la boucle s'arrête.

Par exemple, il faut sept étapes pour une taille de tableau égale à 100 et 10 étapes pour une taille égale à 1000.

Ceci prouve aussi que le coût de la recherche dichotomique est de l'ordre du nombre de chiffres dans l'écriture binaire de n, donc nettement inférieur à celui d'une recherche linéaire.

Preuve de la correction.

Avant l'entrée dans la boucle, `g` a pour valeur 0 et `d` a pour valeur `len(liste)`. Si $x <$ `liste[0]`, ou si $x >$ `liste[d-1]`, alors x n'est pas dans la liste et la recherche n'aboutit pas : la fonction renvoie `False`.

Nous pourrions ajouter une assertion dans le programme, avant de commencer la recherche, afin ne pas effectuer la boucle pour rien.

```
def dichotomie(x, liste):
    g = 0
    d = len(liste)
    assert liste[g] <= x <= liste[d-1]
    ...
```

Si l'assertion est vérifiée, nous avons alors `liste[g]` $\leq x \leq$ `liste[d-1]` avant l'entrée dans la boucle.

Nous considérons alors que nous ajoutons un élément en fin de liste, strictement supérieur au dernier élément, par exemple égal à `liste[d-1]+1`. Et nous allons montrer que la propriété `liste[g]` $\leq x <$ `liste[d]` est un invariant de la boucle.

La propriété est vraie avant l'entrée dans la boucle d'après ce qui précède.

Supposons qu'elle soit vraie avant un passage dans la boucle : `liste[g]` $\leq x <$ `liste[d]`.

D'après le choix de `k`, `liste[g]` $\leq$ `liste[k]` $\leq$ `liste[d]` puisque la liste est triée.

Donc, si $x <$ `liste[k]`, on obtient `liste[g]` $\leq x <$ `liste[k]`. Dans ce cas la nouvelle valeur de `d` est `k`, et donc `liste[g]` $\leq x <$ `liste[d]` après le passage dans la boucle.

Sinon, `liste[k]` $\leq x$, et on obtient `liste[k]` $\leq x <$ `liste[d]`. Dans ce cas la nouvelle valeur de `g` est `k`, et donc `liste[g]` $\leq x <$ `liste[d]` après le passage dans la boucle.

Il nous reste à examiner l'état des variables à la fin de la boucle.

Lorsque le dernier passage dans la boucle est effectué, $d - g \geq 2$. Donc $g < k < d$. Si d prend la valeur k ou si g prend la valeur k, alors $d - g > 0$. Mais s'il n'y a plus de passage dans la boucle, cela signifie que $d - g \leq 1$. La seule possibilité est donc $d - g = 1$.

Or `liste[g]` $\leq x <$ `liste[d]`, donc `liste[g]` $\leq x <$ `liste[g+1]`. En conclusion, nous avons deux possibilités : soit $x =$ `liste[g]`, soit x n'est pas dans la liste.

Le principe est le même pour trouver une solution approchée d'une équation du type $f(x) = 0$ sur un intervalle $[a; b]$ à epsilon près.

```
def dichotomie(f, a, b, epsilon):
    while b - a > epsilon:
        m = (a + b) / 2
        if f(a) * f(m) <= 0:
            b = m
        else:
            a = m
    return (a + b) /2
```

À chaque étape, l'amplitude de l'intervalle contenant la solution est divisée par deux. Ceci signifie que la précision gagne un chiffre dans l'écriture binaire. (10 chiffres dans l'écriture binaire représente environ 3 chiffres dans l'écriture décimale).

Note

De nombreuses bibliothèques sont disponibles en Python pour les calculs mathématiques. En particulier, le module `optimize` de la bibliothèque scientifique **Scipy** contient les fonctions `bisect` et `newton` dans lesquelles sont programmées respectivement la méthode de dichotomie et la méthode de Newton.

Voici un code pour résoudre de manière approchée l'équation $x^2 = 2$ avec la fonction `bisect` et avec la fonction `newton` :

```
import scipy.optimize

def f(x):
    return x**2-2

x = scipy.optimize.bisect(f, 1, 2) # intervalle initial [1;2]
x = scipy.optimize.newton(f, 1) # valeur initiale 1
```