Cours d'algorithmique NSI 1ere

Les principaux algorithmes rencontrés en classe de première NSI.

- Recherche séquentielle (linéaire) et recherche dichotomique;
- Tris par insertion et par sélection;
- les k plus proches voisins.
- Algorithmes gloutons

Recherche dans un tableau

Introduction

Pour *Thomas Cormen*, très grand professeur d'informatique, un algorithme est **une procédure de calcul, bien définie, qui prend en entrée une valeur ou un ensemble de valeurs, et qui donne en sortie une valeur ou un ensemble de valeurs**. C'est un outil permettant de résoudre un problème de calcul bien spécifié.
Un algorithme doit se terminer dans un temps raisonnable et doit résoudre correctement le problème donné. Ces deux propriétés sont respectivement nommées:

- complexité
- correction

Cette première leçon d'algorithmique sera orientée vers la recherche (*séquentielle*) d'un élément dans un tableau.

Parcours séquentiel d'un tableau

Recherche d'une occurence

Soit le problème suivant: étant donné un tableau `t` de n éléments, écrire un algorithme qui permet de dire si un élément x appartient à ce tableau.
Le problème: x appartient-il au tableau ?

Les entrées: un tableau `t` et un élément `x`;
La sortie: une réponse vraie ou fausse (`True` ou `False`)

Une solution:

```
fonction appartient(x, t):

Entrées:
- t: tableau
- x: élément de même type que les éléments du tableau t
Sortie: booléen (réponse)

Variable: n, entier naturel

n = taille(t)
Pour i allant de 0 à n-1
    Si t[i] = x
        retourner Vrai
Retourner Faux
```

```python
#Implémentation en python
def appartient_v1(x, t):
    """
    retourne un booléen correspondant à la présence x dans t (ou non);
    x: élément de même type que les éléments de t
    """
    n = len(t)
    for i in range(n):
        if t[i] == x:
            return True
    return False
```

```
appartient_v1(-5, [5, 10, 15, 20])
```

```
False
```

Travail: réaliser l'implémentation en python en utilisant une boucle `while` plutôt qu'une boucle `for`.

```
def appartient_v1b(x, t):
    """
    retourne un booléen correspondant à la présence x dans t (ou non);
    t: tableau d'entiers
    x: élément de même type que les éléments de t
    """
    n = len(t)
    i = 0
    while i < n and t[i] != x:
        i = i + 1
    if i < n:
        return True
    return False
```

```
appartient_v1b(-1, [5, 10, 15, 20])
```

```
False
```

Remarque: on aurait pu boucler sur éléments plutôt que les indices; cela nous dispense de la variable n.

```
def appartient_v2(x, t):
    """
    retourne un booléen correspondant à la présence x dans t (ou non);
    x: élément de même type que les éléments de t
    """
    for t_i in t:
        if t_i == x:
            return True
    return False
```

Recherche d'un extrémum

Deuxième problème: étant donné un tableau `t` de n éléments, écrire un algorithme qui permet de trouver le minimum parmi les éléments de ce tableau.
Le problème: quellle est la plus petite valeur du tableau ?

```
fonction minimum(t):
L'entrée: un tableau t;
La sortie: le plus petit élément de t
Variables:
    n, entier naturel
    mini, même type que les éléments de t
Précondition: t non vide

n = taille(t)
mini = t[0]
Pour i allant de 1 à n-1:
    Si t[i] <= mini
        mini = t[i]
retourner mini
```

```
def minimum(t):
    """
    Retourne le plus petit élément de t
    """
    assert t != [], "Erreur: tableau vide"
    n = len(t)
    mini = t[0]
    for i in range(1,n):
        if t[i] < mini:
            mini = t[i]
    return mini
```

```
minimum([5, 10, -15, 20])
```

```
-15
```

```
minimum([0, 12])
```

```
0
```

Travail: écrire et tester l'implémentation d'une fonction retournant l'élément **le plus grand** de `t`

Calcul d'une moyenne

Troisième problème: étant donné un tableau `t` de n éléments, écrire un algorithme qui permet de trouver la moyenne des éléments de ce tableau.
Le problème: quellle est la valeur moyenne des éléments du tableau ?

```
fonction moyenne(t):
L'entrée: un tableau t de flottants;
La sortie: la moyenne de tous les éléments de t
Variables:
    n, entier naturel
    somme, flottant
Précondition: t non vide

n = taille(t)
somme = 0
Pour i allant de 0 à n-1:
    somme = somme + t[i]
retourner somme / n
```

Travail: implémenter cet algorithme en python

```python
def moyenne(t):
    """
    Renvoie la moyenne des éléments d'un tableau t non vide;
    """
    assert len(t) != 0, "Erreur: tableau vide"
    n = len(t)
    somme = 0
    for i in range(n):
        somme = somme + t[i]
    return somme / n
```

```python
moyenne([5.5, 10.0, 15.0, 19.5])
```

```
12.5
```

Coût des algorithmes

La boucle est parcourue:

- n fois dans le pire des cas (*si x n'appartient pas au tableau*) dans le problème 1
- $n-1$ fois dans le problème 2;
- n fois dans le problème 3;

Le nombre d'opérations effectuées dépend de la taille de l'entrée n, plus exactement est proportionnel à n (ou $n-1$).
On dit que la complexité de ces algorithmes est **linéaire**.

La recherche dichotomique

Principe

On souhaite rechercher une valeur x dans un tableau t trié (*par ordre croissant pour se fixer les idées*). L'idée principale est de comparer x avec la valeur située au milieu du tableau:

- si elle est plus petite, il suffit de restreindre la recherche dans la partie gauche du tableau;
- sinon, on la restreint à la partie droite de t.
- ou alors on a trouvé x

En répétant ce procédé on divise la zone de recherche par deux à chaque fois. Il s'agit d'une application du principe *diviser pour régner*.

Implémentation en python

```python
def est_trie(t):
    """
    Vérifie si t est trié et renvoie un booléen comme réponse
    """
    return all(t[i] <= t[i + 1] for i in range(len(t) - 1))


def recherche_dichotomique(x, t):
    """
    Renvoie la position de x dans le tableau t trié; None si non trouvé
    Précondition: t trié
    """
    assert est_trie(t), "Erreur: tableau non trié"
    g = 0#gauche
    d = len(t) - 1#droit
    while g <= d:
        m = (g + d) // 2
        if x > t[m]:
            g = m + 1
        elif x < t[m]:
            d = m - 1
        else:
            return m
    return None
```

Initialement, `g = 0` et `d = len(t) - 1`. On calcule l'index central m du tableau (*ligne 8*) en effectuant une **division entière**.
Ensuite on compare x à `t[m]`. Suivant le résultat de la comparaison, on ajuste la borne gauche g ou droite d de la zone de recherche.

Si `x == t[m]` (*ligne 13*), on a trouvé x et on renvoie son index. Il se peut que l'on sorte de la boucle `while` (si `g > d`, c'est dire que la zone de recherche est vide), dans ce cas, on signale l'absence de x en retournant `None` (*ligne 15*).

Terminaison de l'algorithme

Lorsqu'un programme contient une boucle `while` il est suceptible de **diverger** (*on dit aussi qu'il peut entrer dans une boucle infinie*). Pour montrer l'arrêt de la boucle, on utilise une technique de raisonnement appelé **technique du variant de boucle**. Il s'agit de trouver parmi les éléments du programme une quantité:

- entière;
- positive;
- qui décroit strictement à chaque tour de boucle.

g	d	m	t[m]	d-g	trouvé ?
Initialisation					
Fin itération 1					
Fin itération 2					
Fin itération 3					

Soit `t = [1,7,8,9,12,15,15,22,30,31]`. On recherche $x = 15$ dans `t`.
Compléter le tableau ci-dessus. Conclure

Tri par insertion

Principe

Voir le principe à l'adresse: Algorithmes de tri

L'idée principal est de **parcourir le tableau de la gauche vers la droite, en maintenant la partie déjà triée sur sa gauche**. Concrètement, on va décaler d'une case vers la droite tous les éléments déjà triés, qui sont plus grands que l'élément à classer, puis déposer ce dernier dans la case libérée.

Implémentation en python

```python
def tri_insertion(t):
    """
    Trie le tableau t par ordre croissant
    """
    for i in range(1,len(t)):
        elt_a_classer = t[i]
        j = i
        #décalage des éléments du tableau pour trouver la place de t[i]
        while j>0 and t[j - 1] > elt_a_classer:
            t[j] = t[j - 1]
            j = j - 1
        #on insère l'élément à sa place
        t[j] = elt_a_classer
```

Validité de l'algorithme

Au début de chaque itération d'index `i`, le sous tableau `t[0..i-1]` est trié (attention, on fait référence aux éléments d'index 0 à `i-1` **inclus**, ce n'est pas une notation python). Cette propriété est appelée **invariant de boucle**. L'invariant est une propriété qui doit:

être vérifié avant la première itération (initialisation)

Lorsqu'on est en ligne 5 la première fois, `i = 1` et cette propriété se traduit par: la tableau `t[0]` est trié. Ce qui est vrai évidemment car on n'a qu'un seul élément.

être conservé (conservation)

Si l'invariant est vrai avant une itération, il l'est aussi avant la prochaine.

Ici pour une itération d'index `i` donné, on va décaler les `t[i-1]`, `t[i-2]`, .. etc, vers la droite pour trouver la place de `t[i]`. On place alors `t[i]`. Ces éléments sont triés et comme on incrémente `i` au début de la boucle `for` de la prochaine itération, l'invariant est conservé.

Par ailleurs, la combinaison de l'invariant avec la **terminaison** de la boucle permet de conclure à la validité, on dit aussi **correction totale**, de l'algorithme.

Dans notre cas, à la fin de la boucle, `i = len(t)`. Si on substitue cette valeur dans l'expression de l'invariant, on a : à l'itération `i = len(t)`, le sous tableau `t[0..len(t)-1]` est trié, soit:

le tableau t est trié, l'algorithme est correct.

Efficacité: complexité de l'algorithme

Dans le pire des cas, pour un tableau de taille `n`, il faudra effectuer:

$$1+2+\cdots+(n-2)+(n-1)=\frac{n}{2}\times(n-1)$$

comparaisons et échanges. Le coût (on dit aussi *complexité*) est **quadratique**. On écrit souvent que la complexité est en $\Theta(n^2)$.

> **Note**
>
> Voir le document "Complexité" en complément

Tri par sélection

Principe

On commence par rechercher le plus petit élement du tableau puis on l'échange avec le premier élement. Ensuite, on cherche le deuxième plus petit élement et on l'échange avec le deuxième élément du tableau et ainsi de suite jusqu'à ce que le tableau soit entièrement trié.

Voir une animation à cette adresse

Algorithme et exemple d'implémentation en python

On peut formaliser l'algorithme du tri par sélection avec la procédure décrite ci-dessous.

> **Note**
>
> Contrairement à une fonction, une procédure ne renvoit rien et ne modifie pas ses arguments

```
procedure tri_selection(t)
t: tableau de n éléments (t[0..n-1)
Pour i allant de 0 à n-2:
    idxmin = i
    Pour j allant de i+1 à n-1:
        Si t[j] < t[idxmini]:
            idxmini = j
    Echanger t[i] et t[idxmini]
```

Travail:

- Appliquer cet algorithme à la main sur le tableau `t = [3, 4, 1, 7, 2]`.
- donner une implémentation possible en python de cet algorithme et tester.

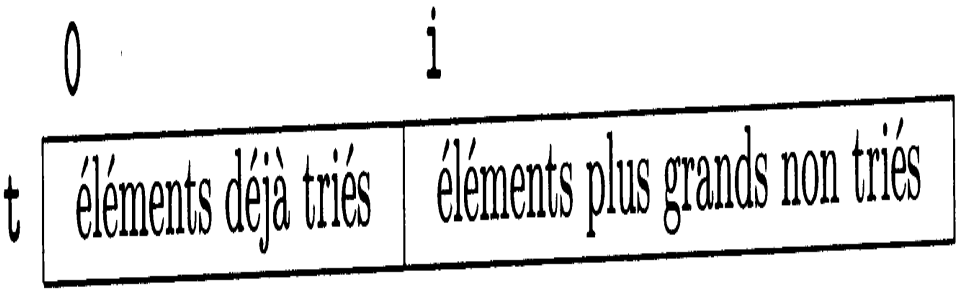
```python
def echange(t, i, j):
    """
    Permute les éléments situés aux index i et j du tableau t
    t: tableau non vide
    i, j: entiers dans l'intervalle [0, len(t)-1]
    """
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def tri_selection(t):
    """
    trie par ordre croissant les éléments de t
    """
    n = len(t)
    #Compléter le code
```

Pour tester votre code, exécuter la séqunce suivante:

```python
t = [5, 6, 1, 1, 15, 0, 4]
tri_selection(t)
assert t == [0, 1, 1, 4, 5, 6, 15]
```

Validité de l'algorithme

La situation au $i^{ème}$ tour de boucle peut être représentée de la manière suivante:



Tous les éléments d'indice compris entre 0 et $i-1$ inclus sont triés **et** ils sont tous inférieurs ou égaux aux éléments de la partie non triée, entre i et $n-1$.
Il s'agit d'un **invariant** pour l'algorithme `tri_selection`.
La terminaison est assurée car l'algorithme fait intervenir deux boucles bornées (boucle `for`).

Complexité

Le contenu de la boucle interne prend un temps d'exécution constant. Evaluons le nombre de fois qu'elle est exécutée.

Pour $i=0$, elle est exécutée $(n-1)-(0+1)+1=n-1$ fois. Pour $i=1$, elle est exécutée $(n-1)-(1+1)+1=n-2$ fois. Si on généralise, le nombre d'exécutions de la boucle interne est:

$$(n-1)+(n-2)+\cdots+2+1$$

Cette somme correspond à la somme des termes consécutifs d'une suite arithmétique, dont la valeur pour $n>1$ est donnée par:

$$\frac{n}{2}\times(n-1)=\frac{1}{2}n^2-\frac{1}{2}n$$

Pour une taille n très grande de l'entrée, le terme en n^2 devient prépondérant. En d'autres termes, le nombre d'opérations effectuées, donc le temps d'exécution, est proportionnel à n^2.
La complexité du tri par sélection est quadratique.

Application directe

En supposant que le tri par sélection prenne un temps directement proportionnel à n^2 et qu'un tri de 16000 valeurs nécessite 6.8 s. Calculer le temps nécessaire pour le tri d'un million de valeurs avec cet algorithme.

Exercice: temps d'exécution

Pour mesurer le temps d'exécution d'un programme, on importe la fonction `time` du module `time`. Cette fonction renvoie le temps en secondes écoulé depuis le 1^{er} janvier 1970.
Le code qui suit permet par exemple d'afficher le temps pris par l'exécution du tri d'un tableau.

```
from time import time
top = time()
tri_selection(t)
print(time() - top)
```

On souhaite comparer les temps d'exécution des tri sélection et insertion sur deux types de tableau: un tableau de nombre au hasard et un tableau de nombres déjà triés. On reprend le code des fonctions de tri du cours.

1. Construire un tableau de 3000 entiers pris au hasard entre 1 et 10000, bornes comprises. Mesurer le temps d'exécution du programme de tri sélection et de tri insertion pour trier ce tableau. *Attention: il faut reconstruire le tableau entre les deux tris*. Quel commentaire peut-on faire concernant les deux résultats ?

2. Construire un tableau de 3000 entiers de 0 à 2999, bornes comprises. Mesurer le temps d'exécution du programme de tri sélection et de tri insertion pour trier ce tableau. Quel commentaire peut-on faire concernant les deux résultats ?
3. Mesurer sur un tableau de 100000 entiers, choisis de manière aléatoire entre 1 et 100000, le temps d'exécution de la méthode `sort()` de python. Syntaxe: `t.sort()`. Commentez.

Complexité

Introduction: complexité ou coût d'un algorithme

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Un critère de choix possible est le *temps d'exécution* le plus faible.
Le temps d'exécution caractérise le coût -appelé complexité- d'un algorithme.
En réalité on ne va pas mesurer réellement le temps d'exécution mais plutôt utiliser un *modèle* simplifié, indépendant de la machine utilisée et dont l'unité de mesure du coût est une *opération* (affectation, comparaison, etc..)

Complexité linéaire

Pour exprimer le coût d'un algorithme, on peut évaluer le nombre d'opérations effectué.

Exemple: le problème est d'afficher les diviseurs d'un entier n. On propose l'algorithme naïf ci-dessous, exprimé en python.

```
def diviseurs(n):
    for i in range(1, n+1):
        if n % i == 0:
            print(i)
```

On a: n incrémentations et n tests à la ligne 2, n calculs de restes de divisions et n tests à la ligne 3 et au plus n affichages soit $5n$ opérations au maximum (on dit aussi dans le *pire de cas* ou *worst case* en anglais).
En général, le coût d'un algorithme varie en fonction d'un paramètre appelé **taille de l'entrée**, ici le nombre n. L'algorithme naïf présenté a un coût évalué à $5n$ opérations.
Il n'est pas utile en première NSI d'aller vers un niveau de détail tel que celui qui vient d'être présenté. On dira simplement que:

le nombre d'opérations de cet algorithme est proportionnel à n ou encore qu'il a une complexité linéaire.

Autres cas courants

Si la complexité ne **dépend pas** de la taille de l'entrée, l'algorithme a une **complexité constante**.

Existe-t-il des algorithmes plus performants que ceux ayant une complexité linéaire ?
Oui, les algorithmes ayant une **complexité logarithmique**. La fonction logarithme est une fonction qui croît très lentement.

Existe-t-il des algorithmes moins performants que ceux ayant une complexité linéaire ?
Oui, les algorithmes ayant une **complexité quadratique, cubique, etc** c'est-à-dire des algorithmes dont le nombre d'opérations est une fonction du carré de la taille de l'entrée, du cube, etc.

Ordre de grandeur du temps d'exécution sur un ordinateur personnel de quelques algorithmes dont la complexité a été évoquée ci-dessus

Nom courant	Temps pour une taille n = 1 million
logarithmique	10 ns
linéaire	1 ms
quadratique	1/4 h
cubique	30 ans

Quelques exemples

Donner la complexité des exemples ci-dessous.

```
def table1(n):
    for i in range(11):
        print(i * n)

def table2(n):
    for i in range(n):
        print( i * i)

def table3(n):
    for i in range(n):
        for j in range(n):
            print(i * j, end=" ")
        print()
```

By Bruno DARID