université
PARIS-SACLAY

Analyse d'algorithmes

DIU - EIL

Christine Froidevaux

chris@lri.fr

Juin 2019

UNIVERSITÉ PARIS SUD

Algorithme

- 3e siècle av-JC : algorithme d'***Euclide*** (Grèce)

- 9e siècle ap-JC : ***Al-Khoarizmi*** (mathématicien, géographe, astrologue et astronome perse), écrit un ouvrage d'arithmétique utilisant les règles de calcul sur la représentation décimale des nombres

  ➔***Algorithme***

- 20e siècle : ***Formalisation logique*** ; thèse de Church, machine de Turing

  ➔début de l'informatique théorique

Algorithme (1/2)

- Un ***algorithme*** est un procédé de calcul automatique composé d'un ensemble fini d'étapes (chaque étape étant formée d'un nombre fini d'étapes élémentaires) qui permet de résoudre un problème ou de calculer une fonction. Chaque étape élémentaire est :
  - définie de façon rigoureuse et ***non ambiguë***
  - ***effective***, c-à-d, pouvant être réalisée par une machine

Analyse d'algorithmes

1. **Complexité**
2. Exemple d'analyse : recherche séquentielle
3. Principes de conception d'un algorithme (illustré sur le tri par insertion)
4. Ordre de grandeur et échelle de fonctions
5. Tris

Algorithmique

- Etude d'algorithmes : plusieurs algos sont possibles pour résoudre le même problème

- **Comparer des algorithmes** : « sur toute machine, quel que soit le langage de programmation, l'algorithme A1 est « meilleur » que l'algorithme A2 pour les données de grande taille »

➔ Divers critères de comparaison, mais on va surtout évaluer les ressources nécessaires pour réaliser l'algorithme (**place mémoire** et **temps d'exécution**)

Introduction à la complexité en temps

2 algorithmes peuvent résoudre le même problème

```
Fonction1(n) :
    Pour tout i de 1 à n :
        si i divise n :
            afficher i
```

```
Fonction2(n) :
    Pour tout i de 1 à Ent(√ n) :
        si i divise n :
            afficher i
            si n/i différent de i :
                afficher n/i
```

Introduction à la complexité en temps

… mais ils n'utilisent pas forcément les mêmes ressources

```
Fonction1(n) :
    Pour tout i de 1 à n :
        si i divise n :
            afficher i
```

Algorithme plus simple à comprendre

Au plus 2n tests et affichages

```
Fonction2(n) :
    Pour tout i de 1 à √ n :
        si i divise n :
            afficher i
            si n/i différent de i :
                afficher n/i
```

Effectue **au plus**
4 √ n tests et affichages

Complexité (= coût d'exécution)

- **Temps** : regarder le temps CPU ?
- Pour évaluer le coût d'un algorithme, l'un des modèles consiste à compter le ***nombre des opérations effectuées*** : opérations arithmétiques, comparaisons, affectations (échanges) entre variables, affichages
- On peut aussi se concentrer sur certains opérations seulement : ***opérations fondamentales***
- On peut aussi s'intéresser à **l'espace** : place mémoire utilisée pendant l'exécution de l'algorithme
  - Allocation dynamique de la mémoire
  - Fonctions récursives : pile des variables locales…

Complexité en temps

- **Notion d'opération fondamentale** (op fond)

On détermine la complexité intrinsèque d'un algorithme en comptant le nombre d'***opérations fondamentales***, telles que le temps d'exécution de l'algorithme est proportionnel au nombre de ces opérations fondamentales.

Exemples :

1) **Recherche d'un élément dans une liste**

Op fond : comparaison entre cet élément et les entiers de la liste

2) **Tri** d'une liste rangée dans un tableau

Op fond : comparaison entre éléments (clés)
déplacements d'éléments (structures)

3) **Produit de matrices carrées de réels**

Op fond : addition et multiplication entre réels

Calcul du nbre d'op fond d'une instruction I (1/4)

1. **Séquence** : I = {I1, I2} ;

   op(I)= op(I1) +op(I2)

2. **Branchement conditionnel** : I = <u>if</u> C {I1} <u>else</u> {I2} ;

   $op(I) \leq op(C) + Max(op(I1), op(I2))$

3. **Boucle** : $I = B_{m \leq i \leq n}$ (J, i), où J est le corps de la boucle ;

   op(I) =

Calcul de op(I), nbre d'op fond d'une instruction I (1/4)

1. **Séquence** : I = {I1, I2} ;

   op(I) = op(I1) + op(I2)

2. **Branchement conditionnel** : I = <u>if</u> C {I1} <u>else</u> {I2} ;

   $op(I) \leq op(C) + Max(op(I1), op(I2))$

3. **Boucle** : $I = B_{m \leq i \leq n}$ (J, i), où J est le corps de la boucle ;

   $op(I) = \mathbf{\Sigma_{m \leq i \leq n} op(J,i)}$ [à écrire au tableau en extension]

= **(n-m+1) op(J)** si le corps de la boucle J est constant

Exemples d'algorithmes de complexité différente

```
def table1(n):
    for i in range(10):
        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)
```

Calcul du nbre d'op fond d'une boucle B (2/4)

Tri-sélection : Principe : les i premiers éléments sont triés en ordre croissant et on cherche parmi les n-i restants le plus petit à mettre à l'indice i

```
def triSelection(t) :
    for i in range(len(t)-1):
        petit = t[i]
        indice = i
        for j in range(i+1,len(t)):
            if t[j] < petit :
                petit = t[j]
                indice = j
        t[indice] = t[i]
        t[i] = petit
    return t
```

Boucle B(J,i)

Corps J de la boucle B

op fond = test entre éléments de tableau ; on suppose len(t) = n ;
C(n) =

Calcul du nbre d'op fond d'une boucle B (2/4)

Tri-sélection : Principe : les i premiers éléments sont triés en ordre croissant et on cherche parmi les n-i restants le plus petit à mettre à l'indice i

```
def triSelection(t) :
    for i in range(len(t)-1):
        petit = t[i]
        indice = i
        for j in range(i+1,len(t)):
            if t[j] < petit :
                petit = t[j]
                indice = j
        t[indice] = t[i]
        t[i] = petit
    return t
```

Boucle B(J,i)

Corps J de la boucle B

op fond = test entre éléments de tableau ; on suppose len(t) = n ;

C(n) = $\Sigma_{0 \le i \le n-2} (n-i-1) = \Sigma_{1 \le i \le n-1} (i) = n*(n-1) / 2$

Pour évaluer le premier Σ, écrire en extension le Σ dans un sens puis dans l'autre et prendre la demi-somme.

Calcul du nbre d'op fond d'une instruction I (3/4)

4. **Appel de fonction** :

(i) I = {I1 ; P; I2 } ;

op(I) = op(I1) + op(P) + op(I2)

(ii) Cas récursif : on essaie d'établir une équation de récurrence

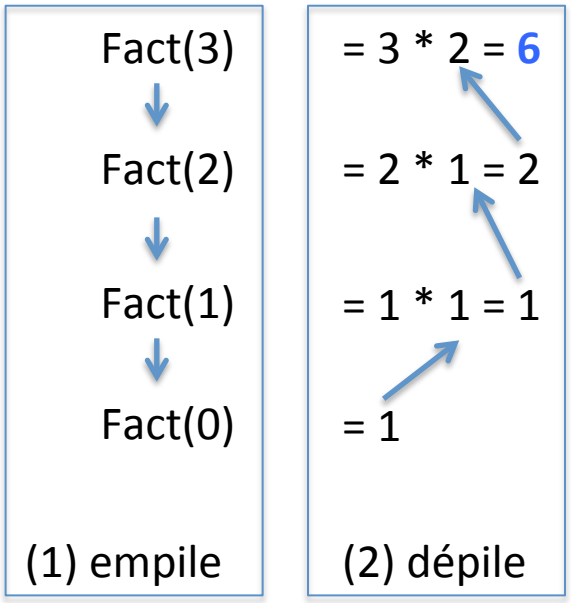
Fonction factorielle :

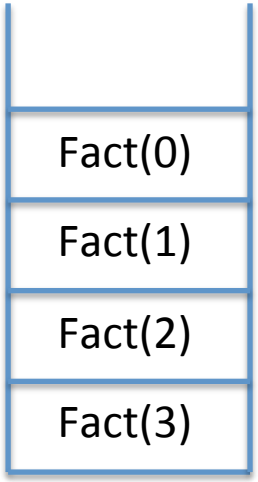
```
def fact(n):
#n supposé >=0
        if (n== 0):
                return (1)      condition d'arrêt
        return (n * fact(n-1))  appel récursif
```

Opération fondamentale : multiplication *

Pile des appels récursifs de fact(3)

Lors des appels récursifs successifs on empile les appels et lorsqu'on arrive à l'appel terminal, on dépile successivement et effectue les opérations





Pile des appels lors de l'appel de fact(0)

- Arbre des appels récursifs avec empilements puis dépilements, à construire au tableau
- Récursion quasi-terminale : pour obtenir fact(3), après avoir appelé fact(2), il reste encore à multiplier par 3, ce n'est donc pas terminal, mais c'est quasi-terminal car c'est facile à dé-récursifier.

Calcul du nbre d'op fond d'une instruction I (4/4)

- Equation de récurrence :

  $op(0) = 0$

  $op(n) = 1 + op(n-1)$ pour $n \geq 1$

  À résoudre …

REM : on peut glisser dans les programmes un compteur qui est incrémenté chaque fois que l'opération fondamentale est effectuée

Exercice

On considère la fonction secret suivante :

```
def secret(n):
    if n>0:
        if n==1:
            print('G', end=' ')
        else:
            print('A', end = ' ')
            secret(n-2)
            print('U', end=' ')
```

Que fait secret(5) ? Secret(6) ? Construire l'arbre des appels récursifs ? Que fait secret(n) en général ?
Quelle est la complexité de la fonction secret ?

Taille des données

- La complexité en temps d'un algorithme dépend de la ***taille*** des données. On appelle taille des données une (ou des) mesure significative du type de données utilisé par l'algorithme.

- **Exemples** :

1) Recherche d'un élément dans une liste en mémoire centrale

Taille :

2) Tri d'une liste rangée dans un tableau

Taille :

3) Multiplication de deux matrices carrées

Taille :

Taille des données

- La complexité en temps d'un algorithme dépend de la ***taille*** des données. On appelle taille des données une (ou des) mesure significative du type de données utilisé par l'algorithme.

- **Exemples** :

1) Recherche d'un élément dans une liste en mémoire centrale

Taille : longueur de la liste

2) Tri d'une liste rangée dans un tableau

Taille : longueur de la liste

3) Multiplication de deux matrices carrées

Taille : nombre de colonnes

Complexités en moyenne, au pire et au mieux (1)

- La complexité en temps d'un algorithme peut ***dépendre ou non de la configuration des données***.

(i) La complexité de l'algorithme est indépendante de la configuration des données

Exemples :

Complexités en moyenne, au pire et au mieux (1)

- La complexité en temps d'un algorithme peut ***dépendre ou non de la configuration des données***.

(i) La complexité de l'algorithme est indépendante de la configuration des données

Exemples :

Pb1 : impression d'une liste de n entiers ;

Pb2 : impression de toutes les permutations d'un mot de n lettres

Pb3 : produit de 2 matrices carrées A et B de dimension n, avec résultat dans C, matrice carrée de dimension n

Produit de 2 matrices carrées A et B de dimension n

```
for i in range(n):
        for j in range(n):
                C[i][j] = 0
                for k in range(n):
                C[i][ j] = C[i][j] + A[i][k] *B[k][j]
```

Complexité :

opérations arithmétiques (algorithme « »)

REM : vers 1960, algorithme de Strassen O(n^2.81)

Produit de 2 matrices carrées A et B de dimension n

```
for i in range(n):
        for j in range(n):
                C[i][j] = 0
                for k in range(n):
                C[i][ j] = C[i][j] + A[i][k] *B[k][j]
```

Complexité : $\Sigma_{i=0..n-1}(\Sigma_{j=0..n-1}(\Sigma_{k=0..n-1}\ 2)) = 2n^3$
opérations arithmétiques (algorithme « **cubique** »)

REM : vers 1960, algorithme de Strassen O(n^2.81)

Complexités en moyenne, au pire et au mieux (2)

- (ii) La complexité de l'algorithme **dépend** de la configuration des données

  ➔ C'est le cas général.

Exemples :

Pb1 : recherche séquentielle d'un élément dans une liste de n éléments ; le nombre de comparaisons dépend de la place de l'élément cherché

Pb2 : tri d'une liste ; le nombre de déplacements d'éléments effectués varie selon que la liste initiale est déjà triée ou non.

Complexités au pire et au mieux

DÉFINITION

Soit A un algorithme ;

Soit D l'ensemble des données possibles pour A ;

Soit D_n, l'ensemble des données de D de taille n ;

Soit $d \in D_n$: on note **$coût_A(d)$** la complexité en temps de l'exécution de A sur d ;

- **Compl_Min(A, n)** = Min $\{coût_A(d) / d \in D_n\}$
  complexité au mieux
- **Compl_Max(A, n)** = Max $\{coût_A(d) / d \in D_n\}$
  complexité au pire

Complexités en moyenne (hors NSI)

- **Compl_Moy(A, n)** = $\Sigma_{d \in Dn}$ proba(d)* $coût_A(d)$ } où proba(d) est la probabilité d'avoir la donnée d en tant qu'entrée de D_n pour A

complexité en moyenne

Le calcul de la complexité en moyenne suppose donc que l'on dispose d'un modèle probabiliste.

On a :

Compl_Min(A, n) ≤ **Compl_Moy**(A, n) ≤ **Compl_Max**(A, n)

MAIS on peut avoir :

Compl_Max(A, n) ≥ **Compl_Max**(B, n)

alors qu'on a : **Compl_Moy**(A, n) < < **Compl_Moy**(B, n)

Ex :

Complexités en moyenne (hors NSI)

- **Compl_Moy(A, n)** = $\Sigma_{d \in Dn}$ proba(d)* coût$_A$(d) } où proba(d) est la probabilité d'avoir la donnée d en tant qu'entrée de D_n pour A

complexité en moyenne

Le calcul de la complexité en moyenne suppose donc que l'on dispose d'un modèle probabiliste.

On a :

Compl_Min(A, n) ≤ **Compl_Moy**(A, n) ≤ **Compl_Max**(A, n)

MAIS on peut avoir :

Compl_Max(A, n) >> **Compl_Max**(B, n)

alors qu'on a : **Compl_Moy**(A, n) ≤ **Compl_Moy**(B, n)

Ex : tri rapide (en moyenne en $n\log_2 n$, au pire en n^2) par rapport au tri fusion

Analyse d'algorithmes

1. Complexité
2. **Exemple d'analyse : recherche séquentielle**
3. Principes de conception d'un algorithme (illustré sur le tri par insertion)
4. Ordre de grandeur et échelle de fonctions
5. Tris

Analyse 1 :
Recherche séquentielle d'un élément dans une liste

➔ Il faut spécifier l'algorithme que nous proposons pour résoudre le problème

- Spécification 1 : {Cette procédure recherche la place d'un élément X dans une liste L de n éléments, représentée par un tableau. Si X n'est pas dans la liste, le résultat est -1.}

Etude d'un exemple

- **Recherche séquentielle d'un élément dans une liste**

➔ Il faut spécifier l'algorithme que nous proposons pour résoudre le problème

- Spécification 1 : {Cette procédure recherche la place d'un élément X dans une liste L de n éléments, représentée par un tableau. Si X n'est pas dans la liste, le résultat est -1.}
- On a quelques indications sur le type de données choisi
- Imprécisions : La liste peut-elle être vide ? Que fait-on s'il y a plusieurs occurrences de X ? Quels sont les données et les résultats ?

Recherche séquentielle d'un élément dans une liste

Spécification 2 :

- **Input** : X de type entier et L un tableau de n éléments de type entier qui représente une liste. Cette liste peut être vide (n=0).
- **Output** : L'algorithme retourne l'indice i dans le tableau de la première occurrence de X dans L, s'il y en a une, et sinon retourne -1.

Spécification formelle et principe

- Spécification 3 :
  - Card(L) = n ($n \geq 0$);
  - $X \notin L \Rightarrow i = -1$
  - $(X = L[k]$ pour $k \geq 0$ & $\forall j \in [0, k-1], X \neq L[j]) \Rightarrow i = k)$.

Principe : on parcourt successivement les éléments du tableau en partant de la case 0 et on compare chaque élément du tableau L avec l'élément X cherché. On s'arrête dès qu'on trouve X, s'il a au moins une occurrence dans L et sinon, on s'arrête à la fin du tableau et on retourne -1.

Algorithme de recherche séquentielle

```
def premOccur(L,x):
    for i in range (0,len(L)):
        if L[i] == x:
            return i
    return -1
```

Rem : en Python, -1 peut être considéré comme un indice, choisir de retourner len(t) + 1 ? False (mais alors la focntion retour un résultat qui peut être soit un entier soit un booléen)

Tests et complexité

- **Tests** :

- **Complexités :**

1) En place :

2) En temps :

Op fond : comparaison entre X et les éléments du tableau.

- **Compl_Min(A, n) =**

- **Compl_Max(A, n) =**

Tests et complexité

- **Tests** : liste vide, X en 1ère place, en dernière place, X non présent dans la liste, X a plusieurs occurrences ; X a une seule occurrence dans la liste.

- **Complexités : évaluées pour des listes de n éléments**

1) **En place** : de l'ordre de n (les n cases du tableau, l'élément X comptant pour 1).

2) En temps :

Op fond : comparaison entre X et les éléments du tableau.

- **Compl_Min(A, n) = 1** (cas où X se trouve à la première place)
- **Compl_Max(A, n) = n** (cas où se trouve à la dernière place ou est absent)

Recherche séquentielle : Complexité en moyenne (1)

Deux hypothèses :

- On suppose que les éléments sont tous distincts
- On suppose que si X est dans L, toutes les places sont équiprobables

Notations :

- Soit D_n l'ensemble des listes de taille n, d'éléments tous distincts. On partitionne l'ensemble D_n en regroupant les données de taille n, de même coût.
- $D_{n,i}$ = { listes L de D_n telles que X est le ième élément de L}, $1 \leq i \leq n$
- $D_{n,0}$ = { listes L de D_n ne contenant pas X }

Complexité en moyenne (2)

- Soit q la probabilité que X soit dans L :

  $q = p(D_{n,1}) + p(D_{n,2}) + \ldots + p(D_{n,n})$

  $\mathbf{p(D_{n,i})} = q/n \quad ; \mathbf{p(D_{n,0})} = 1 - q$

- On note **coût($D_{n,i}$)** le coût d'une donnée quelconque de $D_{n,i}$.

  coût($D_{n,0}$) = ; coût($D_{n,i}$) =

Complexité en moyenne (2)

- Soit q la probabilité que X soit dans L :

  $q = p(D_{n,1}) + p(D_{n,2}) + \ldots + p(D_{n,n})$

  $\mathbf{p(D_{n,i})} = q/n \quad ; \mathbf{p(D_{n,0})} = 1 - q$

- On note $\mathbf{coût(D_{n,i})}$ le coût d'une donnée quelconque de $D_{n,i}$.

  $coût(D_{n,0}) = n \quad ; coût(D_{n,i}) = i$

- $Compl\_Moy(A, n) = \Sigma_{i=0..n}\, p(D_{n,i}) * coût(D_{n,i})$

  $= (1-q) * n + \Sigma_{i=1..n}\, (q/n)*i.$

- $Compl\_Moy(A, n) = (1-q) * n + [(n*(n+1)/2)*(q/n)$
- $=(1-q) * n + ((n+1)/2)*q$

Complexité en moyenne : cas particuliers

- On sait que X est dans la liste :

  q=1 ;

  Compl_Moy(A, n) = (n+1)/2

- On sait que X a une chance sur 2 d'être dans la liste :

  q=1/2 ;

  Compl_Moy(A, n) = (n/2) +(n+1)/4 = (3n+1)/4

Analyse d'algorithmes

1. Complexité
2. Exemple d'analyse : recherche séquentielle
3. **Principes de conception d'un algorithme (illustré sur le tri par insertion)**
4. Ordre de grandeur et échelle de fonctions

Écriture d'un algorithme : « SPACTP »

SPÉCIFICATION (question : **Quoi ?**)

- Décrire ce que fait l'algorithme, sans détailler comment
- Préciser l'input, l'output, les préconditions à satisfaire pour que l'algorithme soit correct
- Indiquer les paramètres modifiés, les variables locales, les autres méthodes utilisées

PRINCIPE (question : **Comment ?**)

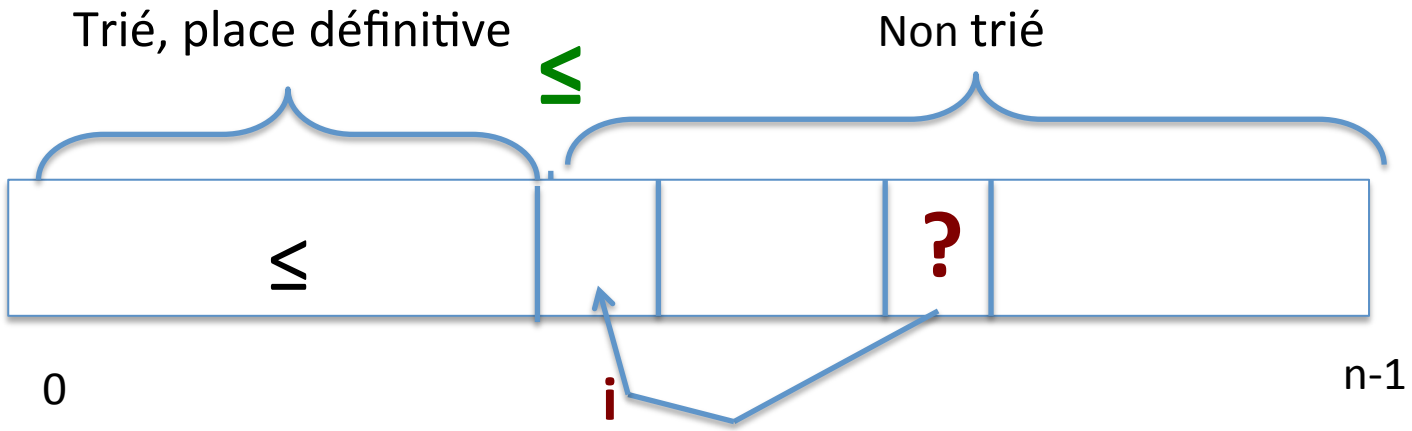
- Décrire comment l'algorithme fait ce qu'il doit faire
- Donner les grandes étapes de l'algorithme : à écrire en langage naturel
- Prouver la correction de l'algorithme (difficile) ou en tout cas justifier son principe en tout cas

Exemple d'algorithme : tri sélection

- **Spécification** :
  - L'algorithme de tri sélection prend en entrée une liste de n entiers rangés dans un tableau T d'entiers, entre les indices 0 et n-1, et
  - donne en sortie un tableau T comportant entre les indices 0 et n-1 la liste des n entiers rangés par ordre croissant.
- **Principe** :
  - Deux parties dans le tableau : à gauche les éléments triés à leur place définitive ; à droite la partie qui n'est pas encore triée, composée d'éléments qui sont tous plus grands que les éléments de la partie déjà triée.
  - A chaque étape on agrandit la partie gauche en y ajoutant, juste à sa droite, le plus petit élément de la partie droite.
  - Pour cela on échange le plus petit élément de la partie droite avec le premier élément de cette partie droite.

Principe du tri par sélection

- Choisir ***le plus petit élément de la partie non encore triée ;*** le placer en place i (c'est sa place définitive) et continuer sur le reste du tableau :



SPACTP (suite)

ALGORITHME

- Préciser les structures de données choisies et justifier les choix faits
- Donner un ensemble d'instructions simples, à exécuter, pour résoudre un problème ou calculer une fonction, à regrouper en modules (écriture modulaire de l'algorithme)
- Donner les détails de chaque procédure / fonction / méthode : penser à réutiliser ce qui a été déjà défini
- A écrire éventuellement en pseudo-code, indépendamment de tout langage de programmation
- Commenter judicieusement par étapes
- Choisir un ensemble de noms de variables, méthodes, etc. parlants

Algorithme du tri sélection

Programme Python

```
def triSelection(t):
    for i in range(len(t)-1):
        petit = t[i]
        indice = i
        for j in range(i+1,len(t)) :
            if t[j] < petit :
                petit = t[j]
                indice = j
  # petit est le plus petit élément de t à droite de i, i inclus; on le place en i
        t[indice] = t[i]
        t[i] = petit
    return t
```

SPACT (suite)

COMPLEXITE

- À évaluer en temps (et en espace)

- Choisir pour la complexité en temps les opérations fondamentales qui seront calculées

- Calculer la complexité au pire (resp. au mieux) et indiquer les configurations de données d'entrée qui causent cette complexité au pire (resp. au mieux).

- Etablir des équations de récurrence pour les fonctions récursives

Ex : algorithme du tri sélection

Complexité

En nombre d'opérations effectuées (comparaisons et affectations d'éléments de tableau) pour un tableau de n éléments.

- **Comparaisons** : de l'ordre de n^2 comparaisons dans le pire des cas, et au mieux (et donc en moyenne). Algorithme « quadratique » pour le nombre de comparaisons
- **Affectations** : dans tous les cas, 3*(n-1) ou encore (n-1) échanges d'éléments de tableau. Algorithme linéaire pour le nombre d'affectations

SPACTP (fin)

TESTS ***(cf bloc 1)***

- Tester ***les cas de la spécification*** : cas normaux (cas généraux) et cas limites, voire cas hors des conditions de la spécification mais proches ; choisir un jeu de tests couvrant toutes les instructions

- Faire du ***test fonctionnel*** : préciser les données (entrées/input) choisies et les résultats (sorties/ output) attendus, et confronter aux résultats obtenus.

PREUVE de CORRECTION : voir autre chapitre

EX : tri sélection

Tests

Que se passe-t-il si le tableau a 0 ou 1 élément ?

Si le tableau est déjà trié ?

Trié en ordre décroissant ?

Si tous les éléments sont égaux ?

Que donne l'algorithme sur un tableau quelconque ?

Etc…

Remarques

- Complexité au mieux : on ne la calcule pas sur une liste de 0 éléments !
- Si Compl_Min(A, n) et Compl_Max(A, n) sont du même ordre de grandeur, cela donne l'ordre de grandeur de Compl_Moy(A, n)
- Mais :

Compl_Moy(A, n) ≠ Compl_Min(A, n) + Compl_Max(A, n) /2

(en général)

Complexité d'un problème vs d'un algorithme

Problème de grande complexité : Imprimer toutes les permutations d'un mot de n lettres ➔ n! opérations print

Pour d'autres problèmes, il peut exister un algorithme simple à concevoir mais de complexité très grande, alors qu'on peut trouver un algorithme de complexité bien moindre ➔ cas du calcul des nombres de Fibonacci

Complexité d'un problème vs d'un algorithme

Fibonacci : mathématicien ***Léonard de Pise dit Fibonacci – Italien*** fin 12^e – début 13e siècle ; reproduction des lapins ; Nombre d'or présent dans la nature et dans les oeuvres d'art

On définit ici les nombres de Fibonacci Fibo(n) comme suit :

Fibo(0) = 0 ; Fibo(1) = 1 et

Fibo(n) = Fibo(n-1) + Fibo(n-2) pour n > 2

C'est le nombre de façons de monter un escalier de n marches, en montant une ou deux marches d'un coup

1) Ecrire un programme récursif qui calcule Fibo(n). Quelle est sa complexité ?
2) Ecrire un programme itératif pour calculer Fibo(n). Quelle est sa complexité en temps ? Qu'en concluez-vous ?

Fibonacci

```
def fibo_rec(n) :
# cas de base, n supposé >=0
    if (n <= 1) :
        if (n==0):
            return 0
        else:
            return 1
    return(fibo_rec(n-1) + fibo_rec(n-2))
```

Complexité en nombre d'additions : exponentielle de l'ordre de F(n)

Arbre des appels pour fibo_rec(5)

Fibonacci itératif

```
def fibo_iter(n) :
# cas de base, n supposé >=0
    if (n <= 1) :
        if (n==0):
            return 0
        else:
            return 1
#n>=2 :
    f0=0
    f1=1
    for i in range(2,n+1):
      f2=f0+f1
      f0=f1
      f1=f2
    return f2
```

Complexité en nombre d'additions : **linéaire**

Exo : Modifier la fonction pour qu'elle commence avec n'importe quelles deux valeurs initiales données, pour qu'elle compte le nombre d'additions

Fibo récursif amélioré

- Dans ce cas la récursivité mène à un algorithme exponentiel alors que la version itérative est linéaire !

- Il est possible de remédier à ce problème en mémorisant les valeurs déjà calculées (memoization), MAIS on occupe alors une mémoire linéaire en n alors qu'elle est de taille constante dans la version itérative.

- Compromis temps vs espace

Analyse d'algorithmes

1. Complexité
2. Exemple d'analyse : recherche séquentielle
3. Principes de conception d'un algorithme (illustré sur le tri par insertion)
4. **Ordre de grandeur et échelle de fonctions**
5. Tris

Ordre de grandeur (1/2)

- *Ce qui nous intéresse est plus **l'ordre de grandeur** de la complexité* que sa valeur exacte, par ailleurs souvent très difficile à estimer avec précision.

- Exemple : Soient 4 algorithmes A, B, C, D dont la complexité en fonction de la taille N de l'entrée est :

$$\text{complexité}_A(N) = N-1,$$
$$\text{complexité}_B(N) = 2*N - 3,$$
$$\text{complexité}_C(N) = N^2 + N - 5,$$
$$\text{complexité}_D(N) = N^3 + N.$$

Calculons les valeurs de ces fonctions pour quelques valeurs de N:

- Si N est petit, ces valeurs sont assez différentes, mais restent petites.

Ordre de grandeur (1/2)

- *Ce qui nous intéresse est plus **l'ordre de grandeur** de la complexité* que sa valeur exacte, par ailleurs souvent très difficile à estimer avec précision.

- Exemple : Soient 4 algorithmes A, B, C, D dont la complexité en fonction de la taille N de l'entrée est :

$complexité_A(N) = N-1$, **linéaire**

$complexité_B(N) = 2*N - 3$, **linéaire**

$complexité_C(N) = N^2 + N - 5$, **quadratique**

$complexité_D(N) = N^3 + N$, **cubique**

Calculons les valeurs de ces fonctions pour quelques valeurs de N:

- Si N est petit, ces valeurs sont assez différentes, mais restent petites.
- Par contre, **si N est grand…**

Ordre de grandeur (2/2)

- ***f est dominée asymptotiquement par g***, noté,

***f* = O(g)** si et seulement s'il existe une constante K positive (indépendante de n) tq, pour tout n assez grand (à partir d'un certain rang), on ait : $f(n) < K*g(n)$.

Par exemple : $5475*n - 17 = O(3*n^2)$ et $3*n^2 = O(4*n^2)$ mais on n'a pas $3*n^2 = O(5475*n)$

- ***f et g sont de même ordre de grandeur asymptotique***, noté ***f* = Θ(g)** si et seulement si $f = O(g)$ et $g = O(f)$.

Par exemple : $3*n^3 = \Theta(487*n^3)$ mais on n'a pas $3547*n^2 = \Theta(2*n^4)$

Propriétés **

- Si $\lim_{n\to+\infty} f(n)/g(n) = a$ (avec $a \neq 0$) alors $f = \Theta(g)$
- Si $\lim_{n\to+\infty} f(n)/g(n) = 0$ alors $f = O(g)$ (***f est négligeable devant g***)
- Si $\lim_{n\to+\infty} f(n)/g(n) = \infty$ alors $g = O(f)$ et g n'est pas $\Theta(f)$

- *Quelques formules utiles aux calculs :*
- $1 + 2 + \ldots + n = n*(n+1)/2$
- si $a \neq 1$, $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1) / (a - 1)$
- $a_n x^n + \ldots + a_1 x^1 + a_0 x^0 = O(x^n) = \Theta(x^n)$
- $n! = (2 \pi n)^{1/2} (n/e)^n (1 + \varepsilon)$ (formule de Stirling)
- $\log(n!) \approx \Theta(n*\log(n))$

Exercice : Echelle de fonctions

Regrouper en classes d'équivalence pour Θ ie classes « même ordre de grandeur » les fonctions suivantes :

n, 2^n, $n \log n$, $n - n^3 + 7n^5$, $n^2 + \log n$, n^2, $\log_2 n$, n^3, $\sqrt{n} + \log_2 n$, $\log_2 n^2$, $n!$, $\log n$

Ordres de grandeur du temps d'exécution

On considère des données de taille 10^6 et des algorithmes de différentes complexités exécutés sur ces données sur un ordinateur effectuant 10^9 opérations par seconde (d'après Wack et al. Informatique pour tous en CPGE, Eyrolles)

Complexité	nom	Temps	Commentaire
O(1)	Temps constant	1 ns	Plutôt rare
O(log n)	logarithmique	10 ns	Quasi-instantané
O(n)	linéaire	1 ms	Temps > 1mn pour données de très grande taille, mais alors la gestion de la mémoire pose pb
O(n log n)		10 ms	
O(n^2)	quadratique	1/4h	Acceptable pour n< 10^6
O(n^3)	cubique	30 ans	inefficace
O(n^k)	polynomial	…	…
O(2^n)	exponentiel	10^300 000 années	impraticable sauf si n < 50

Analyse d'algorithmes

1. Complexité
2. Exemple d'analyse : recherche séquentielle
3. Principes de conception d'un algorithme (illustré sur le tri par insertion)
4. Ordre de grandeur et échelle de fonctions
5. **Tris**

TRIS

- Soit L une liste de n éléments. A chaque élément est associée une clé telle que les clés soient toutes comparables entre elles. On va trier les éléments selon leurs clés. En pratique, on confond les éléments avec les clés.
- On stocke L dans un tableau et on trie sur place en faisant des échanges ou des déplacements, en fonction des résultats des comparaisons des clés.

- **Complexité en temps** :
  - Nombre de déplacements d'éléments,
  - Nombre de comparaisons entre clés.

- Résultat **d'optimalité** pour le nombre de comparaisons : il n'y a pas d'algorithme qui opère par comparaisons et transferts, ***sans autre information sur les clés***, qui dans le pire des cas soit de complexité en nombre de comparaisons d'ordre strictement plus petit que $\Theta(n \log n)$.

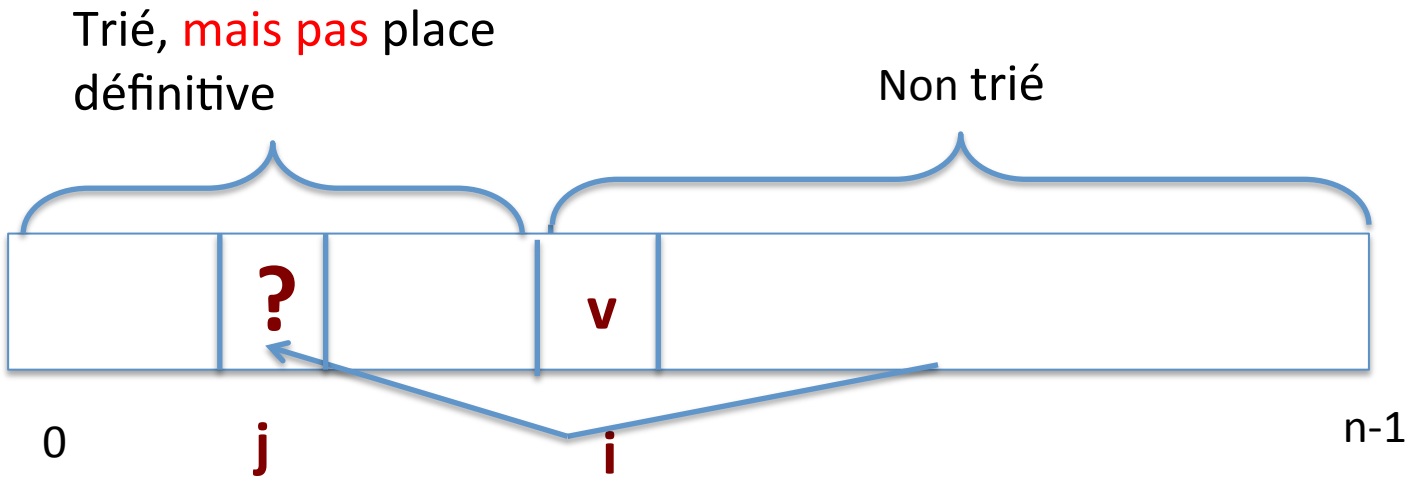
➔ Dans le cas de l'algorithme du drapeau tricolore (cf TD1), on trouve une complexité en nombre de comparaisons linéaire, car on a une information sur les éléments : ils ne peuvent prendre que 3 couleurs.

Tri insertion

Spécification : La fonction prend en entrée un tableau de n éléments comparables entre eux et le retourne trié en ordre croissant

Principe : méthode du joueur de cartes (ou du professeur ramassant et triant les copies)

A la ième étape, on insère l'élément t[i] à sa place j parmi les éléments à sa gauche qui sont triés entre eux, mais pas forcément à leur place définitive. Pour cela on décale successivement d'une place vers la droite, les éléments déjà triés, de la place i-1 à la place j.



Faire « tourner à la main » sur un exemple

Tri insertion

```
def TriInsertion(t):
    for i in range (1, len(t)):
        v = t[i]
        j = i -1
        while (j >=0 and t[j]>v):
            t[j+1] = t[j]
            j = j -1
        t[j+1] = v
```

Exemple :

Tri du tableau t : [4,2,5,1,3] ; len(t) = 5

En italique : cartes non encore distribuées

0	1	2	3	4	indices de t
4	**2**	*5*	*1*	*3*	**i = 1, v = 2**
4	4	*5*	*1*	*3*	**j = 0**
2	4	*5*	*1*	*3*	j = -1
2	4	**5**	*1*	*3*	**i = 2, v = 5**
2	4	5	*1*	*3*	**j = 1**
2	4	**5**	*1*	*3*	j = 1
2	4	5	**1**	*3*	**i = 3, v = 1**
2	4	5	**5**	*3*	**j = 2**
2	4	**4**	5	*3*	**j = 1**
2	**2**	4	5	*3*	**j = 0**
1	2	4	5	*3*	j = -1
1	2	4	5	**3**	**i = 4, v = 3**
1	2	4	5	**5**	**j = 3**
1	2	4	**4**	5	**j = 2**
1	2	4	4	5	**j = 1**
1	2	**3**	4	5	j = 1

Tri insertion : complexité

```
def TriInsertion(t):
    for i in range (1, len(t)):
        v = t[i]
        j = i -1
        while (j >=0 and t[j]>v):
            t[j+1] = t[j]
            j = j -1
        t[j+1] = v
```

Comparaisons comptées

Affectations comptées

Q : Comment savoir ce que fait l'algo pendant l'exécution ? Q

Exemple :

Tri du tableau t : [4,2,5,1,3] ; len(t) = 5

En italique : cartes non encore distribuées

Complexité du tri insertion

Nombre de comparaisons : on compte les tests t[j] > v
Nombre d'affectations : on compte toutes les affectations entre éléments de tableaux

Meilleur des cas :

Pire des cas :

➔ **Algorithme quadratique dans le pire des cas**

Rem : pour chercher la place j où doit venir l'élément t[i] on pourrait procéder par recherche dichotomique, puisque le sous-tableau t à gauche de i est trié ➔ ***tri insertion dichotomique***.

Complexité du tri insertion

Nombre de comparaisons : on compte les tests t[j] > v
Nombre d'affectations : on compte toutes les affectations entre éléments de tableaux

Meilleur des cas : tableau trié en ordre croissant ; on n'entre jamais dans la boule while
On fait 1 ***comparaison*** par passage dans la boucle for : n-1 comparaisons en tout
On fait 2 ***affectations*** par passage dans la boucle for : 2*(n-1) affectations en tout

Pire des cas : tableau trié en ordre décroissant ; on parcourt entièrement la boucle while à chaque fois :
Comparaisons : $\Sigma_{i=1..(n-1)}(\Sigma_{j=(i-1)..0}\ 1) = \Sigma_{i=1..(n-1)}(i) = n(n-1)/2$
Affectations : on en fait 2 de plus que de comparaisons pour un i donné : soit
$\Sigma_{i=1..(n-1)}(i+2) = (n-1)(n+4)/2$
➔ **Algorithme quadratique dans le pire des cas**

Rem : pour chercher la place j où doit venir l'élément t[i] on pourrait procéder par recherche dichotomique, puisque le sous-tableau t à gauche de i est trié ➔ ***tri insertion dichotomique***.

CORRIGES du TD 1

Exercice 1 : drapeau tricolore

On considère n éléments rangés dans un tableau T ($n \geq 1$). Chaque élément a une clé qui ne peut prendre que l'une des trois valeurs : vert (V), bleu (B) ou rouge (R). On souhaite trier le tableau de sorte qu'on ait à gauche tous les éléments verts (s'il y en a), puis ensuite tous les éléments bleus (s'il y en a), et à droite tous les éléments rouges (s'il y en a).

OUTPUT attendu :

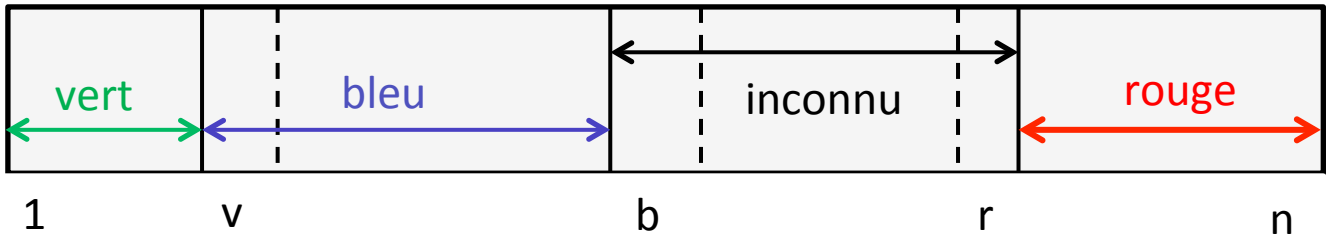
Donner un algorithme qui trie le tableau en testant au plus deux fois la couleur de chaque élément.

Principe du drapeau tricolore

Principe : on part des deux extrémités, comme dans la procédure partition du tri rapide. On s'intéresse à la case courante b du tableau T, dont on teste la couleur, et selon le résultat on procède à des échanges.

Principe du drapeau tricolore

Indication : utiliser une variable v, indice de la première case après la zone verte connue; une variable b, indice de la première case après la zone bleue connue et une variable r, indice de la première case avant la zone rouge connue.



Principe : on part des deux extrémités (on fait monter v et b vers la droite, et on fait descendre r vers la gauche), comme dans la procédure partition du tri rapide. On s'intéresse à la case courante b du tableau T, dont on teste la couleur, et selon le résultat on procède à des échanges.

Algorithme du drapeau tricolore

Enum couleur = {bleu, vert, rouge}
Procedure drapeau-tricol (T : tab[1..n] de couleur (IN/OUT)
/* Algorithme de drapeau-tricol prend en entrée un tableau d'éléments de trois couleurs rouge, vert et bleu, et donne en sortie le tableau de ces éléments tels que les verts sont avant les bleus, qui sont avant les rouges*/

Lexique : v, b, r : entier ;
Début

```
v := 1; b := 1 ; r := n
tant que b ≤ r faire
        si T[b] = bleu alors b := b+1
        sinon si T[b] = vert alors
                {échanger T[v] et T[b] ; /* T[v]  <--> T[b] */
                v := v+1 ; b := b+1 }
                sinon /* T[b] = rouge */
                {échanger T[b] avec T[r] ; /* T[b]  <--> T[r] */
                r:= r-1} finsi;
        finsi;
fintantque ;
```

Fin
Attention : b ≤ r nécessaire ; prendre n = 2, T[1] = bleu et T[2] vert

Programme Python

```
def drapeau_tricolore(t):
#cpt_coul compte le nombre de tests de couleurs
        v=0
        b=0
        r=len(t)-1
       cpt_coul=0
       cpt_ech=0
        while (b<=r):
                cpt_coul= cpt_coul+1
                if t[b]=='B':
                                b=b+1
                else:
                                cpt_ech= cpt_ech+1
                                if t[b]=='V':
                                                cpt_ech= cpt_ech+1
                                                t[v], t[b] = t[b], t[v]
                                                v=v+1
                                                b=b+1
                                else:
                                                cpt_ech= cpt_ech+1
                                                t[r], t[b] = t[b], t[r]
                                                r=r-1
```

On a rajouté un compteur de test de couleurs et un compteur d'échanges d'éléments de tableau pour confronter à la complexité théorique trouvée

Pour bien voir pourquoi il faut b≤r (avec égalité), considérer le tableau de 2 éléments ['B', 'V']

Drapeau tricolore : tests et complexité

Tests :

- n = 1 ;
- n > 1 et les éléments sont tous d'une seule couleur ; deux couleurs seulement ; trois couleurs déjà triées ; trois couleurs en ordre inverse ; trois couleurs en ordre quelconque

Complexité en temps : nombre de tests de couleurs

- Au mieux : tous les éléments sont bleus : n tests
- Au pire : tous les éléments sont verts (ou rouges) : 2n tests

Rem : quelle que soit la couleur de T[b], après un test, la zone T[b..r] diminue d'une case.

Drapeau tricolore : nombre d'échanges

A chaque fois on fait un échange de moins que de comparaison

On fait n passages dans la boucle tant que.

Au mieux : 0 échange

Au pire : n échanges

Drapeau tricolore : complexité en moyenne

Hypothèse probabiliste : toutes les configurations sont équiprobables

Soit C(n) le nombre moyen de tests de couleurs sur un tableau de n éléments dont on ne connaît pas la couleur.
On va écrire une **équation de récurrence** en raisonnant sur la case b du tableau.

Drapeau tricolore : complexité en moyenne

Hypothèse probabiliste : toutes les configurations sont équiprobables

Soit C(n) le nombre moyen de tests de couleurs sur un tableau de n éléments dont on ne connaît pas la couleur. On va écrire une **équation de récurrence** en raisonnant sur la case b du tableau.

Au départ, on teste T[1] : la case a une probabilité de 1/3 d'être bleue (resp. verte, resp. rouge).

- Si T[1] est bleue, on a fait un test et on est ramené au tableau T[2..n] (b est incrémenté de 1) de n-1 éléments, sans information sur leur couleur.
- Si T[1] est verte, on a fait 2 tests et on est ramené au tableau T[2..n] (b est incrémenté de 1) de n-1 éléments, sans information sur leur couleur.
- Si T[1] est rouge, on a fait 2 tests et on est ramené au tableau T[1..n-1] (r est décrémenté de 1) de n-1 éléments, sans information sur leur couleur.

Drapeau tricolore : complexité en moyenne

Hypothèse probabiliste : toutes les configurations sont équiprobables

Soit C(n) le nombre moyen de tests de couleurs sur un tableau de n éléments dont on ne connaît pas la couleur. On va écrire une **équation de récurrence** en raisonnant sur la case b du tableau.

Au départ, on teste T[1] : la case a une probabilité de 1/3 d'être bleue (resp. verte, resp. rouge).

- Si T[1] est bleue, on a fait un test et on est ramené au tableau T[2..n] (b est incrémenté de 1) de n-1 éléments, sans information sur leur couleur.
- Si T[1] est verte, on a fait 2 tests et on est ramené au tableau T[2..n] (b est incrémenté de 1) de n-1 éléments, sans information sur leur couleur.
- Si T[1] est rouge, on a fait 2 tests et on est ramené au tableau T[1..n-1] (r est décrémenté de 1) de n-1 éléments, sans information sur leur couleur.

D'où: $C(n) = 1/3\ (1 + C(n-1)) + 1/3\ (2 + C(n-1)) + 1/3\ (2 + C(n-1))$

Soit : **$C(n) = 5/3 + C(n-1)$ pour $n \geq 2$; $C(1) = 5/3$ (cas $b = 1 = r = n$)**

On obtient : $C(n) = 5/3\ (n-1) + 5/3 = 5n/3$

Recherche d'un mot dans un texte, représentés chacun par un tableau

```
def recherche_mot(m,t):
#renvoie l'indice de la première lettre de la première occurrence du mot m dans le tableau t, si elle existe, et renvoie -1 sinon
        for i in range(1+len(t)-len(m)):
                j=0
                while j < len(m) and m[j] == t[i+j]:
# and : évaluation paresseuse
                        j = j+1
                if j ==len(m):
                        return(i)
        return -1
```

Recherche d'un mot dans un texte : complexité

Au mieux : le mot m se trouve au début du texte ; on fait len(m) comparaisons de caractères.

Au pire : il faut parcourir tout le texte, et à chaque fois, on parcourt tout le mot. Par ex, on cherche AAAAB dans un texte qui ne contient que des A.
La complexité est alors len(m) * (len(t)-len(m)+1) comparaisons de caractères.

Palindrome

```
def pal(s):
        n=len(s)
        if n==0:
                return False
        for i in range(n//2):
                if s[i]!=s[n-i-1]:
                return False
        return True
```

Comment savoir si on a testé l'élément du milieu ?

Distance de Hamming

```
def hamming(u,v):
        c=0
        if len(u) != len(v):
                return -1
        for i in range(len(u)):
                if u[i] != v[i]:
                        c+=1
        return c
#test hamming("plage", "place")
```

Algorithme récursif de recherche séquentielle

```
def premOcRec(L,x,ind):
        if ind < len(L):
                if L[ind] == x:
                        return ind
                return premOcRec(L,x,ind+1)
        return -1
```

Premier appel : premOcRec(L,x,0)

Recherche dichotomique récursive

```
def rechdichorec(t,v,g,d):
# t supposé trié
        if g > d:
                return -1
        mil = (g+d)//2
        if t[mil] == v:
                return mil
        elif t[mil] > v:
                return rechdichorec(t,v,g,mil-1)
        else:
                return rechdichorec(t,v,mil+1,d)
# premier appel : rechdichorec(t,v,0, len(t)-1)
```