Cours 7 :
Complexité d'un algorithme

MPSI - Lycée Thiers

2014/2015

Pour bien programmer, il faut d'abord éviter les erreurs de syntaxe, écrire du code clair, bien commenté, décomposer son programme en fonctions, et éventuellement les regrouper dans des modules que l'on pourra ré-employer.

Mais un bon programme doit surtout appliquer les bons algorithmes. De façon évidente certains algorithmes seront meilleurs que d'autres.

Exemple : Ecrire en python une fonction qui prend en argument une chaîne de caractère et détermine si le caractère 'a' est présent dans la chaîne. Analysons plusieurs solutions.

• Première solution :

```
def contienta(chaine):
    k = 0; N = len(chaine)
    result = False
    while (result == False and k < N)
        if chaine[k] == 'a':
            result = True
        k += 1
    return result
```

• L'algorithme consiste à parcourir les caractères de la chaîne tant qu'on n'a pas trouvé le caractère 'a'. Le résultat est stocké dans une variable booléenne valant initialement `False` et passant à `True` dès que le caractère `'a'` est lu.

• Le temps d'exécution du programme est du même ordre de grandeur que la première occurrence de 'a' dans la chaine. Il existe des constantes a et b, dépendant du temps d'éxécution d'opérations élémentaires, comme l'affectation ou l'incrémentation, tels que si k est l'indice de la première occurrence de `'a'` dans `chaine`, le temps d'exécution sera $a.k + b$.

• Pour des chaines de taille N, le temps d'exécution sera au plus $a.N + b$ (lorsque le caractère 'a' est absent ou présent en dernière place). C'est le temps d'exécution dans le pire des cas pour des entrées de taille N. Il est ici linéaire au sens que $\lim_{N\mapsto\infty}(a.N + b)/N$ est finie et non-nulle (donc $\lim_{N\mapsto\infty} N/(a.N + b)$ est aussi finie).

• Dans le meilleur des cas, celui où `'a'` se trouve en début de chaine, le temps d'exécution sera `a+b`.

• Pour des chaines de taille N, le temps d'exécution sera en moyenne strictement inférieur à $a.N + b$. il est plus difficile à calculer. Il vaut ici $\frac{a}{2}.(N + 1) + b$: en effet la première occurrence de `'a'` a (pour une loi de probabilité uniforme) autant de chance d'apparaître à la position `i` qu'a la position `N-i`... Elle est aussi linéaire (dans le même sens : 'asymptotique').

• Deuxième solution :

```
def contienta(chaine):
    n = chaine.count('a')
    return bool(n)
```

• Le code est plus court ! Est-il meilleur ? Bien au contraire !

• On compte le nombre de fois où le caractère `'a'` apparaît dans `chaine` à l'aide de la méthode `.count()` des objets séquentiels. Il faut nécessairement lire tous les caractères de `chaine`, donc pour n'importe quelle chaîne de longueur N il faudra un temps d'exécution de la forme $c.N + d$ pour des constantes c et b.

• Dans le pire des cas le temps de calcul est encore linéaire en fonction de la taille des données.

• Mais la complexité dans le meilleur des cas et en moyenne est désormais aussi $c.N + d$. Elle était meilleure auparavant.

• Pour de grandes valeurs de N, c'est à dire pour des données de grande taille, notre précédent programme devrait en moyenne deux fois plus rapide.

• Troisième solution :

```
def contienta(chaine):
    return ('a' in chaine)
```

• Le code est encore plus court. Mais le temps d'exécution en moyenne et dans le pire des cas pour une entrée de taille N sera comme dans le première exemple, de même ordre de qualité, linéaire.

L'utilisation de fonctions préprogrammées spécifiques améliore le temps d'exécution. Mais ici ce sont les constantes a et b qui sont optimisées. L'instruction `in` ne peut que lire les éléments de la chaine l'un après l'autre tant qu'elle n'a pas trouvé 'a', puisqu'il n' y a aucun moyen de savoir à priori si `'a'` figure dans `chaine`.

L'algorithme employé est d'aussi bonne qualité que dans la première solution. On ne peut pas faire mieux.

• A l'exécution des deux programmes on ne verrait une différence pour des chaines de longueur petite, qu'en la mesurant (`%timeit` ou module `time`), l'exécution étant quasi-instantanée. Par contre pour des chaines de très grandes longueurs on pourrait percevoir une différence, et une relative lenteur de la deuxième solution par rapport aux deux autres.

• De nombreux systèmes informatiques manipulent des données de très grandes : Météorologie, gestion du trafic aérien, statistiques de la population, traitement d'image, serveurs, réseau sociaux, etc...

• Soit N_{max} la taille maximale des données pour que l'exécution d'un programme s'exécute efficacement dans un temps d'exécution acceptable. De quelle capacité informatique faudrait-il se doter pour doubler la capacité de traitement $2N_{max}$, pour la multiplier par 10 : $10N_{max}$?

• Si l'algorithme employé a un temps d'exécution linéaire, $a.N + b \approx a.N$, il suffit approximativement respectivement de doubler, ou de multiplier par 10 son parc informatique en pratiquant du calcul en parallèle, ou parallélisme.
• Si l'algorithme employé a un temps d'exécution quadratique $a.N^2 + b.N + c \approx a.N^2$, il suffit de multiplier approximativement son parc informatique par 4, ou 100 !

• Si l'algorithme employé a un temps d'exécution exponentiel, par exemple $a.2^N$ il faudrait élever les capacités du parc informatique au carré, ou à la puissance 10.

En effet : $2^{N_{max}} \leqslant K_{max} \implies 2^{2N_{max}} \leqslant K_{max}^2$ et $2^{10N_{max}} \leqslant K_{max}^{10}$.

C'est pourquoi une solution algorithme de complexité de temps de calcul linéaire en fonction de la taille des données présente de considérables avantages par rapport à une solution de complexité quadratique, pire : exponentielle, etc...

Comparons les temps d'exécution sur un ordinateur à 10 GHz (10 milliards d'opérations à la seconde) dans le pire des cas pour divers tailles de données et différentes fonction de complexités. La taille des données est N.

	$N = 20$	$N = 50$	$N = 100$	$N = 200$
$1000.N$	0.002s	0.005s	0.01s	0.02s
$100.N^2$	0.004s	0.025s	0.1s	0.4s
$10.N^3$	0.002s	0.1s	1s	6s
2^N	0.1s	3.6 années	-	-
3^N	5.8 min.	2.10^{10} années	-	-
$N!$	7710 années	-	-	-

- : supérieur à 100 milliards d'années

• On l'a vu puisque la taille des données numériques est bornée par leur représentation, le temps d'exécution de l'addition de 2 nombres, de leur multiplication, etc..., sont bornés par une constante dépendant de la machine et de l'interpréteur. Il en est de même si l'on effectue n'importe quelle opération mathématiques usuelle, comme prendre une puissance, une racine carrée, etc...

• On appelle opérations élémentaires toutes les opérations consistant en :

1. une comparaison, un test, une opération logique,
2. une opération arithmétique, une fonction mathématiques,
3. l'accès à un élément d'une structure de donnée (liste, chaine, etc..), sa création, sa modification,
4. l'affectation de variable,
5. saisie/impression/retour d'une donnée à l'écran, ou dans un fichier.
6. Toute instruction ou opération prédéfinie, basique, s'effectuant sur une donnée de taille fixée par le système, et dont le temps d'exécution est majoré par une constante ne dépendant que du système.

• Par exemple :

1. Lir ou modifier un élément dans une liste est une opération élémentaire.
2. Le calcul de la longueur d'une liste est une opération élémentaire (l'interpréteur la connait à priori).
3. Les instructions `liste.count('a')` ou `sum(liste)` ne sont pas des opération élémentaires. Leur temps d'exécution dépend de la longueur de `liste`.

• Donné un algorithme ou un programme prenant en paramètre des données de taille N variable.

• Sa **Fonction de complexité (en temps) dans le pire des cas** de l'algorithme, est l'application $F_{max} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{max}(N)$ est le nombre maximum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus défavorable).

• Sa **Fonction de complexité (en temps) dans le meilleur des cas** de l'algorithme, est l'application $F_{min} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{min}(N)$ est le nombre minimum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus favorable).

• Sa **Fonction de complexité (en temps) en moyenne** de l'algorithme, est l'application $F_{moy} : \mathbb{N} \longrightarrow \mathbb{R}$ définie par $F_{moy}(N)$ est le nombre moyens d'opérations élémentaires effectuées lorsque l'on exécute le programme sur les entrées de taille N.

• On ne s'intéresse pas au détail de la fonction de complexité, mais à son comportement asymptotique, sa croissance en fonction de N au voisinage de $+\infty$. Précisément on s'intéresse à l'ordre de la fonction de complexité :

Définitions

• Soient f et g deux applications de $\mathbb{N}$ dans $\mathbb{R}$. On note $f = O(g)$ si il existe une constante réelle C et $n_0 \in \mathbb{N}$ tels que pour tout $n \geqslant n_0$: $f(n) \leqslant Cg(n)$.

• $f = \Theta(g)$, et on dit que f et g ont même ordre si $f = O(g)$ et $g = O(f)$.

C'est le cas notamment lorsque f/g a une limite finie $\neq 0$ en ∞.

Par abus de notation on a coutume de noter :

1. $O(1)$ l'ordre d'une application bornée.
2. $\Theta(\ln(n))$ l'ordre de $\ln(n)$.
3. $\Theta(n)$ l'ordre des applications linéaires non nulles.
4. $\Theta(n^2)$ l'ordre des applications quadratiques kn^2 , $k > 0$.
5. $\Theta(f)$ l'ordre de f, c.à d. ensemble des application de même ordre que f.

Si un algorithme a une fonction de complexité (respectivement dans le pires des cas, dans le meilleur des cas, en moyenne) $T(n)$ on dira que l'**algorithme est de complexité** (respectivement dans le pire des cas, dans le meilleur des cas, en moyenne) :

1. **bornée** si $T(n)$ est majorée (ordre $O(1)$).
2. **logarithmique** si $T(n)$ a pour ordre $\Theta(\ln(n))$.
3. **linéaire** si $T(n)$ a pour ordre $\Theta(n)$.
4. **quadratique** si $T(n)$ a pour ordre $\Theta(n^2)$.
5. **polynomiale** si $\exists k \in N$, $T(n)$ a pour ordre $\Theta(n^k)$.
6. **exponentielle** si $\exists a > 1$, $T(n)$ a pour ordre $\Theta(a^n)$.

• Cas d'école :
Compter le nombre d'occurrences d'un élément dans une liste de longueur variable N :

```
def compte(liste,a):
    compteur = 0
    for x in liste
        if x == a:
            compteur += 1
    return compteur
```

Pour une entrée de taille N le nombre d'opérations élémentaires est égal à $T(N) = 2 + 2.N + k$ où k est le nombre d'occurrence de a dans `liste`, $0 \leqslant k \leqslant N$.

La fonction de complexité dans le pire des cas est $F_{max}(N) = 2 + 3N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

La fonction de complexité dans le meilleur des cas est $F_{min}(N) = 2 + 2N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

Il n'est pas possible de faire mieux : pour compter le nombre d'occurrence de a il faut lire chaque élément de la liste soit au moins N opérations élémentaires.

La fonction de complexité en moyenne est plus délicate à calculer (exercice).

Tout algorithme de la forme :

```
for x in liste :
    suite d'opérations élémentaires sans boucle, test ou sortie anticipée : break
```

est de complexité dans le pire des cas, le meilleur des cas et en moyenne linéaire.

• Autres cas d'école :
Recherche d'un élément dans une liste :

```
def find(liste,a):
    for x in liste :
        if x == a:
            return True
    return False
```

est de complexité dans le pire des cas, linéaire.

Il n'est pas possible de faire mieux (dans le pire des cas, celui où l'élément est absent, tous les élément doivent être lus au moins une fois, puisqu'on n'a aucune information à priori sur la position que pourrait occuper a, ainsi au moins N opérations).

• Recherche d'un élément dans un tableau de taille $n \times p$;

```
def find(tableau,a):
    for ligne in tableau:  # on décrit les lignes
        for element in ligne:  # on décrit les élément de chaque ligne
            if element == a :
                return True
    return False
```

est de complexité dans le pire des cas : $n \times p$ = nombre de lignes $\times$ nombre de colonnes du tableau.
La taille des données cette fois-ci est un couple.

de façon générale lorsque l'entrée est de taille N :

```
for i in [[1,N]]:
    for j in [[1,N]]:
        que des opérations élémentaires sans test, boucle ou sortie
```

est de complexité quadratique N^2.

Exemple : Calcul d'une somme double :

$$\sum_{0 \leqslant i,j \leqslant N} a_{i,j}$$

```
S = 0
for i in range(n):
    for j in range(n):
        S += a[i][j]
print(S)
```

est de complexité quadratique en moyenne et dans le pire et le meilleur des cas.

Parcours d'un tableau triangulaire :

```
for k in [[1,N]]:
    for j in [[1,k]]:
        que des opérations élémentaires sans test, boucle ou sortie
```

est de complexité quadratique N^2.

En effet supposons qu'il y ait C opérations élémentaires dans la boucle.

La fonction de complexité vaut dans chaque cas $C + 2C + 3C + \cdots NC = C\frac{N(N-1)}{2} = \frac{C}{2}(N^2 - N)$ qui est d'ordre quadratique.

Ainsi les fonctions de complexité en moyenne, dans le meilleur et dans le pire des cas sont toutes quadratiques.

• Exemple : recherche d'un nombre dans une liste triée.

Nous voyons en TD comment rechercher par dichotomie un élément dans une liste triée dans le sens croissant (ou décroissant).

On peut toujours appliquer l'algorithme de recherche d'un élément dans une liste déjà vu, ou appelé par l'instruction `x in liste`. Mais il ne met pas à profit l'information donnée : la liste est ordonnée. Sa complexité est linéaire dans le pire des cas.

Comparons avec la méthode par dichotomie :

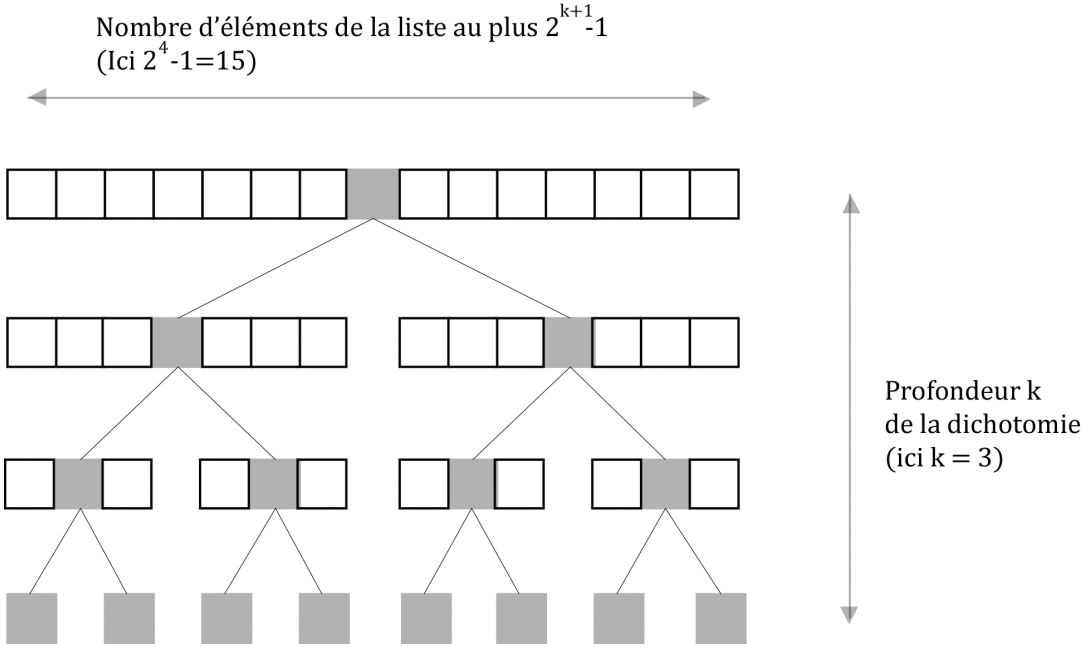
```
def dich_search(l,e):
# Recherche dichotomique dans une liste croissante
    Imin, Imax = 0, len(l)-1
    while Imax - Imin >= 0:
        Imed = (Imin + Imax)//2
        if l[Imed] == e:        # Cas de succés
            return True
        elif l[Imed] < e:
            Imin = Imed + 1         # dichotomie à droite
        else:
            Imax = Imed - 1         # dichotomie à gauche
    return False            # Cas d'échec
```

• Pour une liste de taille N le nombre d'opérations élémentaires pour rechercher un élément par dichotomie est du même ordre que la profondeur de la dichotomie (chaque étape consistant en 1 calcul, 1 affectation, au plus 2 tests, et 1 opération élémentaire).

• Alors pour une liste de N éléments combien est au plus la profondeur de la dichotomie, pour trouver un élément, exprimée comme une fonction de N ?

• Il est plus facile de calculer la fonction réciproque : Si pour une liste au plus k profondeurs de dichotomies sont nécessaires pour chercher un élément, combien d'éléments contient cette liste approximativement ?
- A chaque dichotomie on sépare la liste en 2 listes de tailles égales ± 1.
- Donc à l'étape i la liste contient entre approximativement 2 fois plus d'éléments, ± 1, qu'à l'étape $i+1$.
- A la dernière étape, dans le pire des cas, la liste ne contient plus qu'un élément.
- Ainsi la liste était initialement de taille comprise entre 2^k et au plus $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$.

• Donc le nombre d'éléments N a pour ordre $O(2^k)$ et en composant par $\log_2$, la profondeur de dichotomie k a pour ordre $O(\log_2(N))$.

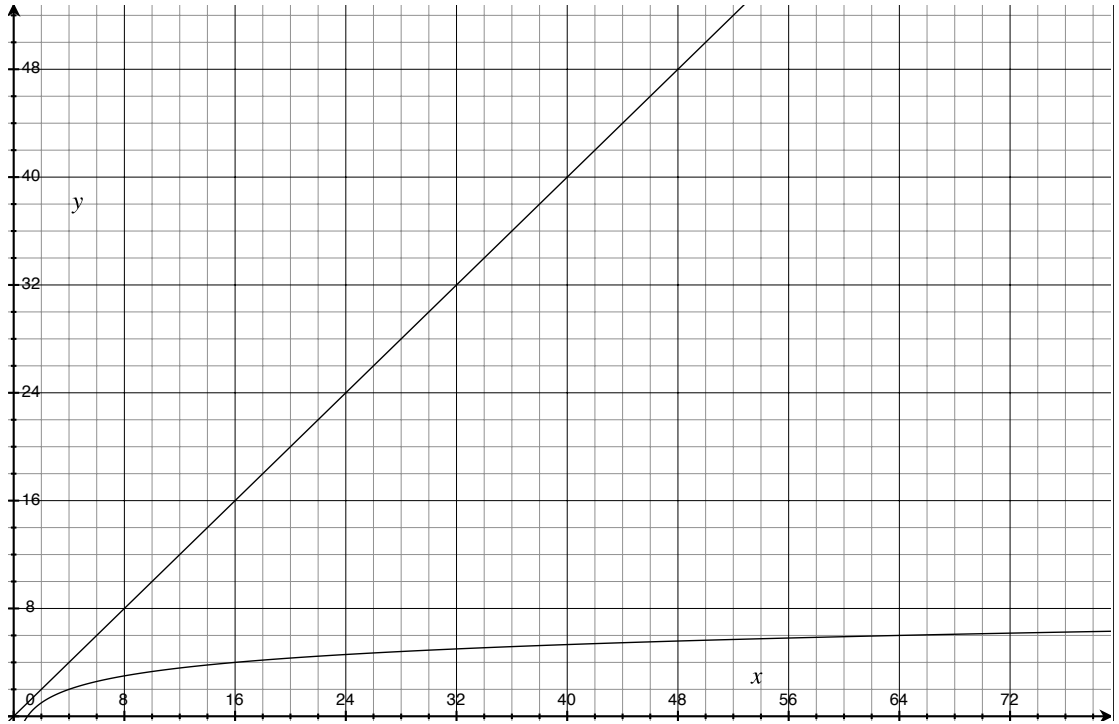


• Les cases grisées représentent les seuls éléments susceptibles d'être comparés avec l'élément recherché aux diverses étapes de la dichotomie.

Lors d'une implémentation, seules celles sur une branche (jusqu'en bas dans le pire des cas) seront comparées, alors que l'algorithme naïf peut comparer avec tous les éléments de la liste.

• La complexité d'une recherche par dichotomie dans une liste triée dans le sens croissant est dans le pire des cas :

$\boxed{O(\log_2(N))}$ **de complexité logarithmique**



• C'est beaucoup plus rapide par dichotomie que par une recherche naïve, linéaire :

Par exemple à partir de données de tailles $N = 100$ c'est de l'ordre de plus de 10 fois plus rapide, De taille $N = 1000$ de l'ordre de 100 fois plus rapide ($2^{10} = 1024 \implies \log_2(1000) < 10$)..