Informatique Tronc Commun

Rappels : Preuves des algorithmes

Programme officiel

Notions	Commentaires
Algorithmes dichotomiques.	On met en évidence une accélération entre complexité linéaire d'un algorithme naïf et complexité logarithmique d'un algorithme dichotomique.
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine. On montre sur plusieurs exemples que la terminaison peut se démontrer à l'aide d'un variant de boucle. Sur plusieurs exemples, on explicite, sans insister sur aucun formalisme, des invariants de boucles en vue de montrer la correction des algorithmes.
Complexité.	On aborde la notion de complexité temporelle dans le pire cas en ordre de grandeur. On peut, sur des exemples, aborder la notion de complexité en espace.

Plan du cours

I Complexité

I.1 Définitions

L'exécution d'un programme utilise les ressources de l'ordinateur, que l'on peut caractériser par :

— le temps de calcul pour exécuter les opérations,

— la place mémoire nécessaire pour stocker les données et le programme en cours d'exécution.

Définitions : Complexités

On définit plusieurs types de complexités :

— **complexité en temps** : évaluation du nombre d'opérations nécessaires pour effectuer le calcul.

— **complexité en mémoire** : évaluation de l'espace mémoire nécessaire pour effectuer le calcul.

— On calculera souvent la **complexité dans le pire des cas et dans le meilleur des cas**.

La majorité du temps, on vous demandera de déterminer une complexité en temps (ce qui sera souvent sous entendu)

Pour évaluer le temps d'exécution d'un calcul, il faut évaluer le temps nécessaire pour réaliser chaque instruction et les sommer. La durée d'exécution d'un problème dépend de taille des instances d'entrée, de l'algorithme utilisé, de l'ordinateur utilisé, des autres processus qui tournent sur l'ordinateur en même temps, . . .

On se contentera de compter le **nombre d'opérations élémentaires**.

I.2 Ordres de grandeurs

On exprime la complexité d'un calcul comme une fonction des données nécessaires pour décrire le problème à l'ordinateur. Par exemple, si l'entrée est une liste, on exprimera la complexité en fonction du nombre d'éléments de la liste.

Les complexités seront données en utilisant la notation de Landau O (« grand O ») qui sert à comparer le comportement de deux suites (cf cours de maths).

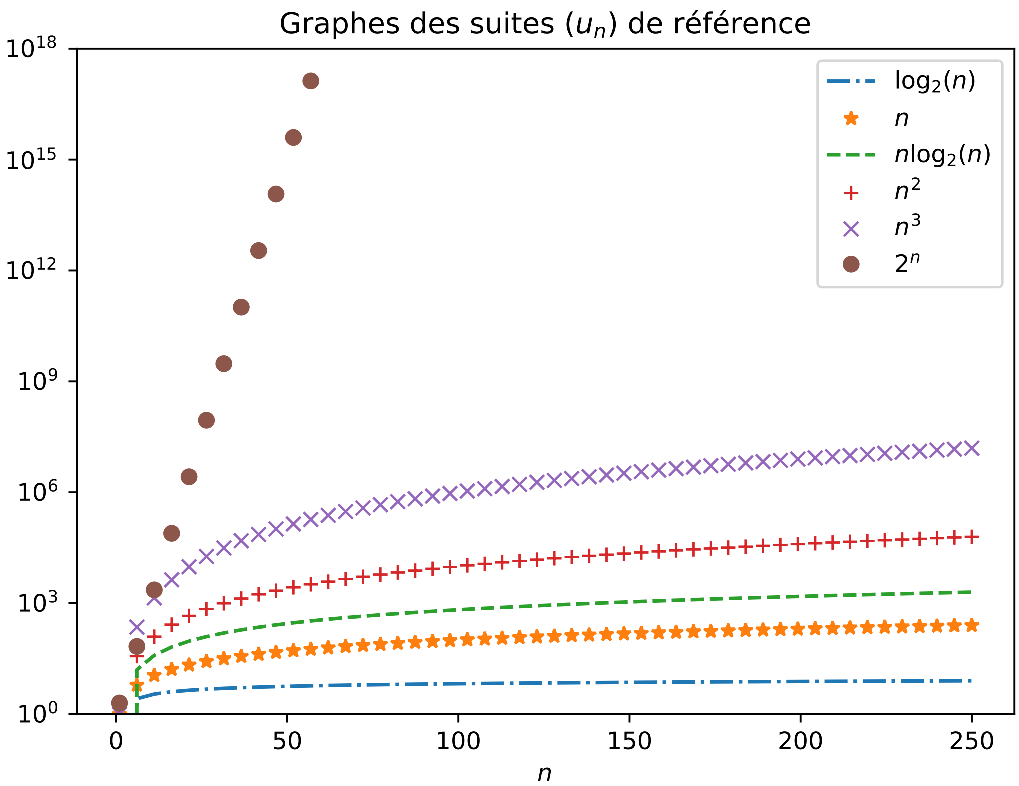
Exemples : $5n + 7 = O(n)$; $3n^2 + 4n + 2 = O(n^2)$

On dira par exemple que « la complexité est en $O(n^2)$ » quand la complexité croit aussi vite que n^2.

On rencontrera des algorithmes de complexité :

- **constante**, quand elle est en $O(1)$, c'est-à-dire indépendante de la taille de l'entrée ;
- **logarithmique**, quand elle est en $O(\log_2(n))$;
- **linéaire**, quand elle est en $O(n)$;
- **quasi-linéaire**, quand elle est en $O(n\log_2(n))$;
- **quadratique**, quand elle est en $O(n^2)$;
- **exponentielle**, quand elle est en $O(\rho^n)$ pour un $\rho > 1$.

n	2	2^4	2^6	2^8
$\log_2(n)$	1	4	6	8
n	2	16	64	256
$n\log_2(n)$	2	64	384	2048
n^2	4	256	4096	65536
2^n	4	65536	$1,84 \times 10^{19}$	$1,15 \times 10^{77}$



I.3 Quelques exemples

Activité n°1– Recherche d'un élément dans une liste

On donne le programme Python de recherche d'un élément dans une liste :

```python
def recherche(L,x):
    for i in range(len(L)):
        if x==L[i]:
            return True
    return False
```

Q1. Quel est le pire des cas en terme de nombres d'opérations ?

Q2. Déterminer dans ce cas le nombre d'opérations. En déduire la complexité temporelle.

On propose l'algorithme de recherche d'un élément dans une liste triée, par dichotomie :

```python
def dichotomie(L,x):
    a=0
    b=len(L)-1
    while b-a>=0:
        c=(a+b)//2
        if x==L[c]:
            return True
        elif x<L[c]:
            b=c-1
        else:
            a=c+1
    return False
```

Q3. Déterminer la complexité temporelle. Commenter.

Activité n°2– Somme de factorielles

On souhaite écrire un programme renvoyant la somme des n premières factorielles. On propose d'écrire ce programme en deux fonctions :

```python
def factorielle(n):
    F=1
    for i in range(1,n+1):
        F=F*i
    return F
def somme_fact(n):
    S=0
    for j in range(0,n+1):
        S=S+factorielle(j)
    return S
```

Q1. Déterminer le nombre d'opérations élémentaires effectuées dans la fonction `factorielle`. En déduire la complexité temporelle.

Q2. Déterminer le nombre d'opérations élémentaires effectuées dans la fonction `somme_fact`. En déduire la complexité temporelle.

Q3. Proposer une amélioration du programme complet afin d'obtenir une complexité linéaire.

Activité n°3– Fonctions récursives

On considère la fonction récursive ci-dessous :

```
def factorielle(n):
    if n==0 or n==1:
        return 1
    return n*factorielle(n-1)
```

Q1. Quelle est la complexité temporelle de cet algorithme ?

Q2. On souhaite maintenant étudier la suite suivante : $u_0 = 3$ et $u_{n+1} = 5u_n^2 + 2u_n + 1$. On propose :

```
def suite(n):
    if n==0:
        return(3)
    else:
        return(5*suite(n-1)**2+2*suite(n-1)+1)
```

Déterminer la complexité en temps. Commenter.

Quelle est la meilleure fonction ?

On souhaite normaliser (cf TD n°4.IA – exercice n°3) les données d'un tableau selon la formule, valable pour chaque colonne : $d'_i = \dfrac{d_i - d_{\min}}{d_{\max} - d_{\min}}$, où $d_{\min}$ est la valeur minimale de la colonne et $d_{\max}$ sa valeur maximale.

La fonction `min_max(T,i)` renvoie une liste contenant la valeur minimale et maximale de la colonne `i` du tableau `T`.

Laquelle de ces deux fonctions est la plus efficace ? Pourquoi ? *On évaluera la complexité de chaque fonction.*

```
def fonction1(T):
    T_norm = [ [ 0 for i in range(len(T[0])) ] for j in range(len(T))]
    for i in range(len(T[0])) : # colonne
        m , M = min_max(T,i) # min, max de la colonne i
        for j in range(len(T)): # lignes de la colonne i
            T_norm[j][i] = ( T[j][i] - m ) / (M-m)
    return T_norm
```

```
def fonction2(T):
    T_norm = [ [ 0 for i in range(len(T[0])) ] for j in range(len(T))]
    for i in range(len(T)): # lignes
        for j in range(len(T[0])) : # colonne
            m , M = min_max(T,i) # min, max de la colonne j
            T_norm[j][i] = ( T[j][i] - m ) / (M-m)
    return T_norm
```

I.4 Complexités des algorithmes de tris

Activité n°5– Tri bulle

On considère une liste `L` à n éléments. Le tri à bulles consiste à faire remonter les éléments les plus grands en permutant successivement les éléments du tableau : on parcourt le tableau, et à chaque fois que l'élément de gauche est strictement supérieur à l'élément de droite, on les permute. À la fin de ce parcours, le plus grand élément du tableau est en dernière position. On recommence le parcours du tableau pour trier les $n-1$ éléments, puis les $n-2$ éléments, etc. Le nom « tri à bulles » vient du fait que les éléments les plus grands remontent plus vite, comme les bulles dans l'eau.

Q1. Déterminer la complexité en temps, puis en mémoire de l'algorithme itératif.

```python
def tri_bulles(L):
    for i in range(len(L)):#n=len(L) étapes
        for j in range(len(L)-i-1): #on parcourt L du rang 0 au rang n-i-2 ; on trie L[:n-i-1], les i+1 derniers éléments sont déjà triés
            if L[j+1]<L[j]:
                L[j],L[j+1]=L[j+1],L[j]
    return L  # inutile : fonction avec effet de bord, L est modifiée
```

Q2. Déterminer la complexité en temps et en mémoire de l'algorithme récursif.

```python
def tri_bulles_rec(L):
    if len(L)<=1: #condition d'arrêt
        return L
    for j in range(len(L)-1):
        if L[j+1]<L[j]: #on fait remonter l'élément le plus grand
            L[j],L[j+1]=L[j+1],L[j]
    return tri_bulles_rec(L[:len(L)-1])+[L[len(L)-1]] #appel récursif : on applique la fonction sur la liste L sans le dernier élément, et on concatène avec le dernier élément
```

Q3. Que peut-on dire de la complexité en mémoire ?

Complexité	En temps (nb de comparaisons)		
	Pire des cas	Meilleur des cas	En moyenne
Tri bulle	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Tri par insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Tri fusion	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$
Tri rapide	$\mathcal{O}(n^2)$	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$

II Terminaison et correction d'un algorithme

II.1 Terminaison d'un algorithme

Définition : Variant de boucle

Un variant de boucle est une suite qui :

- dépend des variables de la boucle,
- à valeurs entières positives,
- strictement décroissante à chaque itération de la boucle.

Méthode : Comment montrer la terminaison d'un algorithme ?

Il est nécessaire de prouver la terminaison des boucles while et des fonctions récursives. Pour prouver la terminaison d'un algorithme, si cela est possible, il suffit souvent de prouver que les boucles se terminent et donc de :

1. trouver un variant de boucle (suite d'entiers, positifs, strictement décroissante),
2. montrer que le variant est minoré, qu'il franchit nécessairement une valeur limite liée à la condition d'arrêt.

II.2 Correction d'un algorithme

Définition : Correction d'un algorithme

La correction d'un algorithme est sa capacité à :

1. se terminer,
2. produire un résultat correct, conforme à ce qu'on attend de lui, quelles que soient les entrées.

On dit que la correction est :

- **partielle** si le résultat est correct lorsque l'algorithme se termine,
- **totale** si elle est partielle et que l'algorithme se termine.

Méthode : Comment montrer la correction d'un algorithme ?

La démarche générale de démonstration de la correction des boucles est une forme de **démonstration par récurrence**. La propriété à démontrer est nommée invariant de boucle. On procède donc en trois phases : l'initialisation de l'invariant, l'hérédité et la conclusion.

Définition : Invariant de boucle

Un invariant de boucle est une **propriété** liée aux variables d'un algorithme qui :

1. est vraie avant la boucle,
2. est invariante par les instructions de la boucle à chaque itération,
3. donne le résultat escompté si la condition de boucle est invalidée.

Méthode : Cas des algorithmes récursifs

La terminaison et la correction d'un algorithme récursif se montrent simultanément **par récurrence**. $\mathscr{P}(n)$: « La fonction `fonction(n)` termine et renvoie .... »

II.3 Quelques exemples

Activité n°6– Terminaison et correction d'un programme itératif

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

Q1. Donner une preuve formelle de terminaison de cette fonction.

Q2. On note r_k la valeur de r à la fin de la k^{e} itération, avec $r_0 = 2$.
Démontrer que la propriété « $r_k = 2^{2^k}$ » est un invariant de boucle pour cette fonction.

Q3. Conclure sur ce que calcule cette fonction.

Activité n°7– Terminaison et correction d'un programme récursif

Montrer la terminaison et la correction de l'algorithme suivant qui renvoie $n!$.

```python
def factorielle(n):
    if n==0:
        return 1
    else :
        return n*factorielle(n-1)
```

Activité n°8– Puissance rapide

L'algorithme de puissance rapide repose sur la propriété suivante :

pour a réel et n entier naturel non nul, $a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & \text{si } n \text{ est impair} \end{cases}$

Q1. Écrire une fonction récursive `puissance_rapide` de paramètres a et n qui renvoie la valeur de a^n en utilisant la propriété précédente.

Q2. Montrer la terminaison et la correction de `puissance_rapide` (par récurrence forte).

Q3. Déterminer la complexité temporelle.