Informatique 5 : Complexité

I Terminaison et correction d'un algorithme

On rappelle la définition :

Définition 1 (*Algorithme*)

Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant (à un ordinateur) de résoudre un problème.

Remarque 2

Définition 3

La **terminaison** d'un algorithme est l'assurance que l'algorithme terminera en un temps fini.

Remarque 4

Définition 5

Si un algorithme se termine, on dit qu'il vérifie la condition de **correction** lorsque celui-ci se termine en donnant une solution conforme au résultat demandé.

Remarque 6

Exemple 7

Déterminer dans chaque cas le fonctionnement de la fonction et déterminer si la fonction Python présentée se termine.

1.
```
def affiche_carres(n):
    L = []
    for k in range(1,n):
        L.append(k**2)
    return L
```
Explications :

2.
```
def ppe(M):
    k=0
    while exp(-k) < M:
        k=k+1
    return k
```

Explications :

Remarque 8

II Complexité

1 Comparaison de suites

Définition 9

Soit (u_n), (v_n) deux suites réelles non nulles à partir d'un certain rang. On note

1. $u_n = O(v_n)$ lorsque

2. $u_n = \Theta(v_n)$ lorsque

Remarque 10

Exemple 11

On donne quelques exemples :

2 Notion de complexité

Chaque opération effectuée lors de l'exécution d'un algorithme nécessite un temps de traitement. Les opérations élémentaires effectuées au cours d'un algorithme sont :

1. l'affectation ;
2. les comparaisons ;
3. les opérations sur les données numériques (multiplications, additions notamment).

Évaluer la complexité en temps revient alors à sommer tous les temps d'exécution des différentes opérations effectuées lors de l'exécution d'un algorithme. La durée d'exécution des différentes opérations dépend de la nature de l'opération : une multiplication sur un flottant demande par exemple plus de temps qu'une addition sur un entier. Ce problème est complexe et pour simplifier ce problème, on se place dans le cas d'un modèle simple :

Définition 12 (*Modèle à coût fixe*)

On appelle **modèle à coût fixe** d'un algorithme le fait de supposer que la durée des opérations ne dépend pas de la taille des objets (notamment des entiers), et que les opérations sur les flottants sont de durée similaire aux opérations semblables sur les entiers.

Remarque 13

Définition 14 (*Complexité*)

La complexité en temps d'un algorithme est une fonction C dépendant de la taille n des données (éventuellement de plusieurs variables s'il y a en entrée des objets pouvant être de tailles différentes) et estimant le temps de réponse en fonction de n.

Exemple 15

Déterminons la complexité $C(n)$ de la fonction `facto(n)`

```
def facto(n):
    P=1
    for k in range(1,n+1):
        P = P*k
    return P
```

Le fait de déterminer précisément la fonction de complexité n'est pas le but mais plutôt d'estimer asymptotiquement la comportement de cette fonction, d'en avoir un ordre de grandeur. Un algorithme dont la fonction de complexité sera n sera plus efficace qu'un algorithme dont la fonction de complexité sera n^2 : si n est grand, on préférera faire n opérations plutôt que n^2.
La complexité est donc une mesure **d'efficacité d'un algorithme**, du **coût** de l'algorithme en termes de ressources de l'ordinateur, et on cherchera seulement à avoir une estimation de la fonction de complexité de la forme suivante qui utilise la notation Θ vue plus haut :

Relation	Nom
$C(n) = \Theta(1)$	Coût constant
$C(n) = \Theta(\ln n)$	Coût logarithmique
$C(n) = \Theta(n)$	Coût linéaire
$C(n) = \Theta(n \ln n)$	Coût quasi-linéaire
$C(n) = \Theta(n^2)$	Coût quadratique
$C(n) = \Theta(n^3)$	Coût cubique
$C(n) = \Theta(n^\alpha)$, $\alpha > 0$	Coût polynomial
$C(n) = \Theta(x^n)$, $x > 1$	Coût exponentiel

3 Une approche de la complexité avec Python

Python permet de connaitre le temps mis pour effectuer une série d'instructions avec la package `time`. Une fois ce package importé, l'instruction `time.time()` permet d'afficher le temps. On donne un exemple. Si on exécute dans l'éditeur :

```
T1 = time.time()
5*5
T2 = time.time()
print(T2-T1)
```

Cette série d'instructions affiche le temps que Python a mis pour calculer 5×5.

Exemple 16

Créer une fonction `temps_facto(n)` qui renvoie le temps que mis par `facto(n)` pour calculer $n!$ (attention celle-ci ne doit pas renvoyer $n!$).

III L'exponentiation rapide

Toute cette partie va permettre de présenter un cas où la stratégie algorithmique choisie va influer sur la rapidité d'obtention du résultat.

1 Présentation générale

Dans cette partie, on va donner une méthode qui va permettre de calculer 2^n de façon efficace sans utiliser la commande `**` de Python. Dans un premier temps, pour calculer 2^n, on peut faire le programme `expo(n)` ≪ naif ≫ suivant

```
def expo(n):
   P = 1
   for i in range(1,n+1):
      P = P * 2
   return P
```

Exemple 17

Calculons la complexité C de la fonction `expo(n)` en fonction n.

On va désormais donner une méthode qui va permettre de faire moins d'opérations. On commence par donner un exemple avec le calcul de 2^9.

Méthode classique :

Méthode rapide :

2 Rappels sur la récursivité

Définition 18

Un algorithme **récursif** est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.

Remarque 19

Une fonction écrite récursivement s'appelle elle-même.

Exemple 20

Écrivons deux fonctions `bin(n)` et `bin_recur(n)` qui, à partir d'un entier n renvoie une liste de son écriture en binaire. *Rappels théoriques :*

```
def bin(n):
```

```
def bin_recur(n):
```

3 Exponentiation rapide

On va expliquer plus en détail la méthode pour calculer 2^n avec une exponentiation rapide :

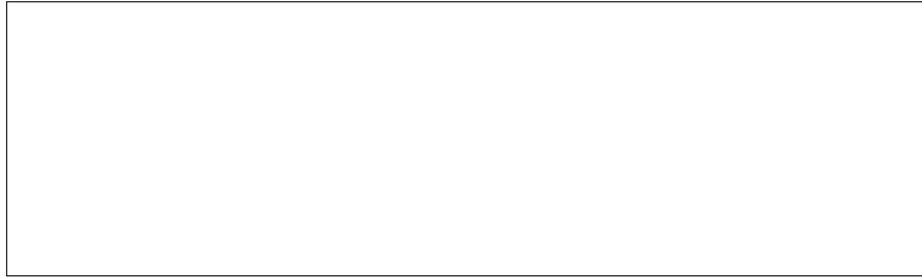
On écrit désormais une fonction Python `expo_r(n)` qui calcule 2^n par exponentiation rapide.

4 Complexité

Écrivons une fonction Python `comparaison(n)` qui va renvoyer une liste à deux termes contenant les temps de calculs pour les fonctions

```
def expo(n):
    P = 1
    for i in range(1,n+1):
        P = P * 2
    return P
```

et de `expo_r(n)` écrite précédemment.



Estimons désormais les complexité en fonction de n de `expo(n)` et de `expo_r(n)` :

Conclusion :

IV Diviser pour régner

1 Principe

Définition 21 (*Algorithme du type « diviser pour régner »*)

Un algorithme du type **diviser pour régner** est un algorithmique consistant à diviser récursivement le problème à traiter en plusieurs sous-problèmes jusqu'à arriver à des problèmes simples qu'il est possible de résoudre directement

Remarque 22

Exemple 23

Remarque 24

2 Complexité

Considérons un algorithme du type diviser pour régner et déterminons une majoration de sa complexité $C(n)$ où n est la taille des objets manipulés en supposant que

$$C(n) = C(n/2) + g(n)$$

où g est une fonction bornée. On constate notamment que C est croissante.

Exemple 25

Théorème 26

Avec les définitions précédentes, on a

$$C(n) = O(\ln n).$$

Démonstration :

3 Recherche du point d'annulation d'une fonction : méthode de dichotomie

Dans cette partie, on considère deux réels $\alpha < \beta$, et f une fonction de $\mathbb{R}$ dans $\mathbb{R}$ continue et strictement croissante sur $[\alpha, \beta]$ avec $f(\alpha) < 0$ et $f(\beta) > 0$.

Proposition 27

Il existe un unique réel $x_0 \in]\alpha, \beta[$ tel que $f(x_0) = 0$.

Démonstration :

□

Position du problème :

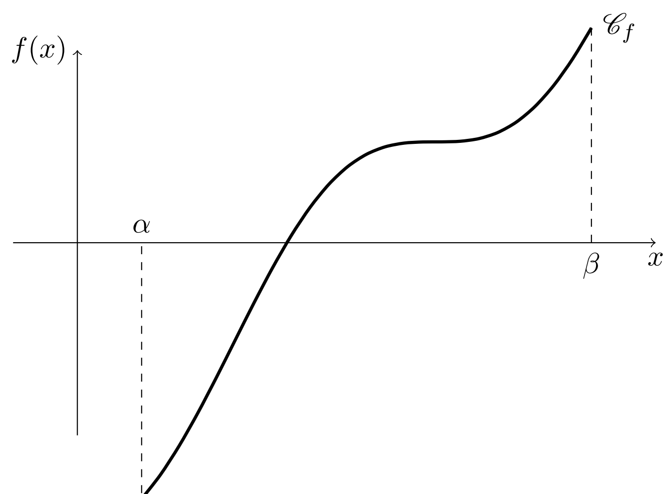
Théorème 28

Il existe deux suites adjacentes (a_n) et (b_n) vérifiant pour tout $n \in \mathbb{N}$

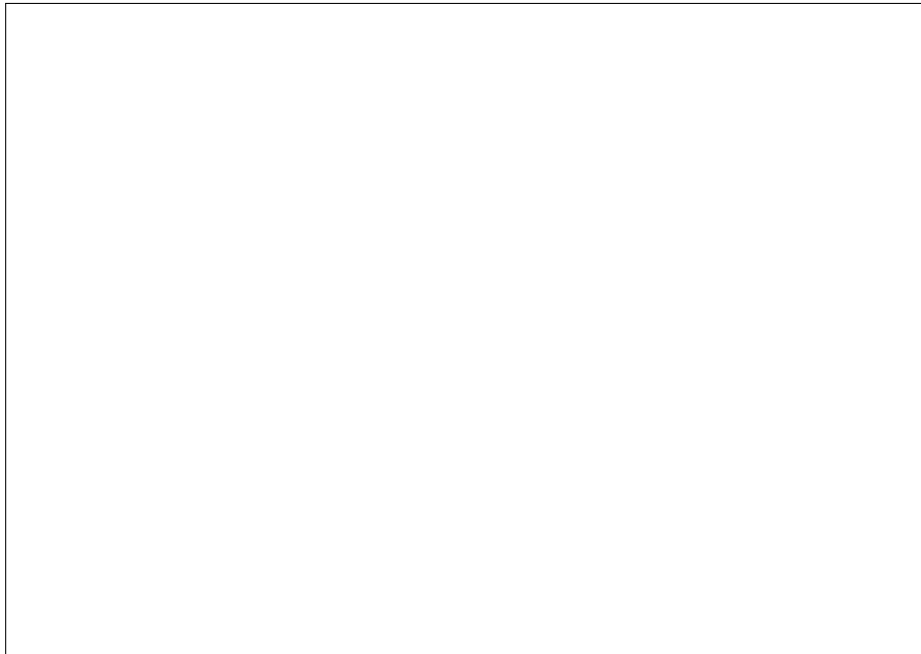
$$a_n \leq x_0 \leq b_n \quad \text{et} \quad b_n - a_n = \frac{b-a}{2^n}.$$

Remarque 29

Figure 1 (*Illustration de la démonstration*)



On va maintenant écrire une fonction Python `dicho(f,a,b,e)` qui renvoie une approximation du point d'annulation de f sur $[a, b]$ à e près.



Majoration de la complexité de la méthode de dichotomie :