Complexité d'un algorithme
Application aux listes

Alix Munier-Kordon et Maryse Pelletier

LIP6
Sorbonne Université
Paris

LU2IN003 Initiation à l'algorithmique

Plan du cours

1. Complexité
   - Introduction
   - Complexité d'un algorithme
   - Exemples de calculs de complexité
   - Méthodologie
2. Listes
   - Définition, représentation, primitives
   - Complexité
3. Conclusion

Questions au sujet de l'évaluation d'un algorithme

1. Est-ce que l'algorithme résout le problème ?
   - terminaison
   - validité
2. Quelle est la complexité de l'algorithme ?
   - en temps de calcul
   - en taille mémoire

Objet de ce cours : la complexité en temps de calcul.

Taille de codage des paramètres d’un algorithme

Definition

La *taille de codage* d’un paramètre est une évaluation, la plus "raisonnable" possible, de la place nécessaire en mémoire pour le stocker.

Quelle est la taille de stockage d’un entier ? d’un tableau d’entiers ?

Complexité d’un algorithme

Definition

La complexité d’un algorithme est une évaluation du nombre d’instructions élémentaires[1] dans une exécution de l’algorithme.

On l’exprime en fonction de la taille de codage des paramètres.

On en calcule un *ordre de grandeur* (notations de Landau).

[1] Parfois on se concentre sur une instruction élémentaire représentative.

Pire cas, meilleur cas

Complexité pire cas : on évalue le nombre d'instructions dans le pire des cas (borne supérieure).

Complexité meilleur cas : on évalue le nombre d'instructions dans le meilleur des cas (borne inférieure).

Exemple : recherche séquentielle d'un élément x dans un tableau T de taille n.
Instruction élémentaire représentative : comparaison.

Pire cas : x en dernière place de T ou pas présent $\rightarrow$ n comparaisons

Meilleur cas : x en première place de T $\rightarrow$ 1 comparaison

Par pessimisme, on identifie la complexité d'un algorithme avec sa complexité dans le pire des cas.
La complexité de la recherche séquentielle est en $\mathcal{O}(n)$.

Comparaison de complexités

Avec une durée de 10^{-6} secondes par instruction, on obtient les durées suivantes (en secondes) pour 100 instructions[2] :

complexité	durée
$\log n$	$4,60 \times 10^{-6}$
n	10^{-4}
$n \log n$	$4,60 \times 10^{-4}$
n^2	10^{-2}
n^3	1
2^n	$\approx 1,27 \times 10^{24}$

Remarque : $1,27 \times 10^{24}$ secondes $\approx 4 \times 10^{16}$ ans !
Un algorithme de complexité logarithmique est meilleur qu'un algorithme de complexité linéaire, meilleur qu'un algorithme de complexité quadratique, etc.

Un algorithme de complexité exponentielle est à bannir.

[2] $\log n$: logarithme népérien

Somme des entiers

Pour $n \in \mathbb{N}$, on veut calculer la somme $Som(n)$ des entiers de 0 à n.
Autrement dit :

$$Som(n) = \sum_{0}^{n} i$$

Cette somme vaut 0 si $n = 0$.

Somme des entiers, en itératif

Algorithme itératif calculant la somme $Som(n)$:

```
def somIte(n):
    res = 0
    for i in range(1, n + 1):
        res = res + i
    return res
```

Complexité en nombre d'additions.

Soit c le nombre total d'additions et c_i le nombre d'additions dans le tour de boucle i.
Alors $c_i = 1$ pour tout $i \in \{1, \ldots, n\}$ et

$$c = \sum_{i=1}^{n} c_i = n$$

La complexité est en $\Theta(n)$, elle est *linéaire*.

Somme des entiers, en récursif

Remarquons que $Som(n) = Som(n-1) + n$ si $n > 0$ et $Som(0) = 0$.
Algorithme récursif calculant la somme $Som(n)$:

```
def somRec(n):
    if n == 0:
        return 0
    else:
        return n + somRec(n - 1)
```

Complexité en nombre d'opérations : tests et additions.

Soit u_n le nombre d'additions effectuées par l'appel `somRec(n)`.
Alors u_n est la suite récurrente définie par :

$$u_n = u_{n-1} + 2 \quad \text{si } n > 0 \quad \text{et} \quad u_0 = 1$$

Par substitution :

$$u_n = u_{n-1} + 2 = u_{n-2} + 4 = \ldots = u_0 + 2n = 2n + 1$$

La complexité est en $\Theta(n)$.

Fibonacci, en itératif

Algorithme itératif calculant F_n :

```
def fibIte(n):
    if (n == 0):
        return 0
    else:
        x = 0 ; y = 1
        for i in range(2, n + 1):
            z = x + y ; x = y ; y = z
        return y
```

Complexité en nombre d'additions.

Soit c le nombre total d'additions et c_i le nombre d'additions dans le tour de boucle i.
Alors $c_i = 1$ et

$$c = \sum_{i=2}^{n} c_i = n - 1$$

La complexité en $\Theta(n)$.
Remarque : la complexité en nombre d'affectations est aussi linéaire.

Fibonacci, en récursif

Algorithme récursif calculant F_n :

```
def fibRec(n):
    if (n == 0) or (n == 1):
        return n
    else:
        return fibRec(n - 1) + fibRec(n - 2)
```

Pour simplifier, on évalue la complexité en nombre d'additions.

Soit u_n le nombre d'additions effectuées par l'appel `fibRec(n)`.
Alors u_n est la suite récurrente définie par :

$$u_n = u_{n-1} + u_{n-2} + 1 \quad \text{si } n > 1 \quad \text{et} \quad u_0 = u_1 = 0$$

Fibonacci, en récursif

$$u_n = u_{n-1} + u_{n-2} + 1 \quad \text{si } n > 1 \quad \text{et} \quad u_0 = u_1 = 0$$

Theorem

$u_n = F_{n+1} - 1$.

Preuve par récurrence.

Theorem

$F_n \geq \frac{1}{\sqrt{5}}(\varphi^n - 1)$ où φ est le nombre d'or : $\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618$.

Preuve par récurrence.

La complexité est en $\Omega(\varphi^n)$. Comme $\varphi > 1$, elle est exponentielle.
Elle croît très vite, par exemple : $\varphi^{100} > 7,9 * 10^{20}$.

Calcul de la puissance : un premier algorithme

Algorithme basé sur la définition récursive *naturelle* de x^n :

$$x^n = x * x^{n-1} \quad \text{si} \quad n > 0 \quad \text{et} \quad x^0 = 1$$

```
def puissSeq(x, n):
    if (n == 0)):
        return 1
    else:
        return x * puissSeq(x, n - 1)
```

La complexité, en nombre de multiplications, est en $\Theta(n)$.

Calcul de la puissance par dichotomie

On peut faire un calcul de x^n par *dichotomie* :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^{n \div 2})^2 & \text{si } n \text{ pair et } n > 0 \\ (x^{n \div 2})^2 \times x & \text{si } n \text{ impair} \end{cases}$$

où $n \div 2$ est le résultat de la division entière de n par 2.

Calcul de la puissance par dichotomie : algorithme

Algorithme calculant x^n par dichotomie :

```
def puissDicho(x, n):
    if (n == 0):
        return 1
    else:
        if n % 2 == 0:
            return carre(puissDicho(x, n // 2))
        else:
            return carre(puissDicho(x, n // 2)) * x
```

où la fonction `carre` est ainsi définie :

```
def carre(x):
    return x * x
```

Calcul de la puissance par dichotomie : complexité

Soit u_n le nombre de multiplications effectuées par l'appel `puissDicho(n)`.

$$u_n = \begin{cases} 0 & \text{si } n = 0 \\ u_{n \div 2} + 1 & \text{si } n \text{ pair et } n > 0 \\ u_{n \div 2} + 2 & \text{si } n \text{ impair et } n > 0 \end{cases}$$

Dans tous les cas, pour $n > 0$, on a $u_n \leq u_{n_1} + 2$ où $n_1 = n \div 2$.

$$\begin{array}{lll} u_n & \leq & u_{n_1} + 2 \quad \text{avec } n_1 = n \div 2 \\ & \leq & u_{n_2} + 4 \quad \text{avec } n_2 = n_1 \div 2 = n \div 2^2 \\ & \leq & u_{n_3} + 6 \quad \text{avec } n_3 = n_2 \div 2 = n \div 2^3 \\ & \leq & \ldots \\ & \leq & u_{n_k} + 2 * k \quad \text{avec } n_k = n \div 2^k \end{array}$$

Les calculs s'arrêtent lorsque $n_k = 0$, c'est-à-dire lorsque $k = \lfloor log_2(n) \rfloor + 1$.
Donc $u_n \leq 2 * \lfloor log_2(n) \rfloor + 2$.
La complexité est en $\mathcal{O}(\log_2(n))$, elle est logarithmique.

Un peu de méthodologie

Complexité d'un algorithme itératif :

- évaluer la complexité c_i du tour de boucle i,
- calculer la somme des c_i pour tous les tours de boucle.

Exemples :

- `somIte(n)` : $c_i = 1$ pour $i \in \{1, \ldots, n\}$ et il y a n tours de boucle
- `fibIte(n)`, pour $n \geq 2$: $c_i = 1$ pour $i \in \{2, \ldots, n\}$ et il y a $n - 1$ tours de boucle.

Un peu de méthodologie

Complexité d'un algorithme récursif :

- évaluer la complexité des cas de base,
- établir une relation de récurrence permettant de calculer c_n. c_n est exprimé en fonction des complexités des appels récursifs et de la complexité $b(n)$ des autres calculs.

Exemples :

- `somRec(n)` : $c_0 = 1$, $c_n = c_{n-1} + b(n)$ pour $n > 0$, avec $b(n) = 2$
- `fibRec(n)` : $c_0 = c_1 = 1$, $c_n = c_{n-1} + c_{n-2} + b(n)$ pour $n > 1$, avec $b(n) = 1$
- `puissDicho(x, n)` : $c_0 = 0$, $c_n = c_{n \div 2} + b(n)$ pour $n > 0$, avec $b(n) \leq 2$.

Notion de liste

Definition

Une liste $L = (a_0, \cdots, a_n)$ est une succession d'éléments.

```
jour=['lundi', 'mardi', 'mercredi']
corbeille=[56, 'jeudi', 45, 67, 'coucou']
```

Quels points communs (différences) voyez-vous entre listes et ensembles ?

Definition

La liste L est *homogène* si tous ses éléments sont de même type.

Représentation des listes

Une liste peut être implémentée par :

- un tableau,
- une liste simplement chaînée,
- une liste circulaire doublement chaînée.

Primitives et opérations sur les listes

- $L[i]$: renvoie l'élément en position i dans la liste L (positions numérotées à partir de 0);
- $L[i:j]$: renvoie la sous-liste de L composée des éléments situés entre la position i et la position $j-1$ comprises;
- L.append(x) : insertion de l'élément x en queue de la liste L;
- L. insert(i, x) : insertion de l'élement x en i-ème place;
- L. pop(i) : renvoie l'élément en i-ème position et le supprime de la liste;
- L.remove(x) : détruit la première instance de l'élément x dans la liste L;
- len(L) : renvoie le nombre d'éléments de L;
- L.index(x) : indice du premier élément de valeur x dans L;
- L. count(x) : nombre d'occurrences de x dans L;
- $L_1 + L_2$: renvoie la concaténation des deux listes;
- $L \star k$: crée une liste de k occurrences de L.

Complexité des primitives

Primitive	Tableau	Liste simpl. chaînée	Liste doubl. circulaire
$L[i]$	$\Theta(1)$	$\Theta(i)$	$\Theta(i)$
L.append(x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
L.insert(i,x)	$\Theta(n-i)$	$\Theta(i)$	$\Theta(n-i)$
L.pop(i)	$\Theta(n-i)$	$\Theta(i)$	$\Theta(i)$
L.remove(x)	$\Theta(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
L.index(x)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
L.count(x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$L_1 + L_2$	$\Theta(n_1 + n_2)$	$\Theta(n_1)$	$\Theta(1)$
$L \star k$	$\Theta(n \times k)$	$\Theta(n \times k)$	$\Theta(n \times k)$
len(L)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$

$n = |L|$, $n_1 = |L_1|$, et $n_2 = |L_2|$.

Exemple : fonction miroir

```
def swapp (tab, i, j):
    aux = tab[i]; tab[i]=tab[j]; tab[j]=aux

def miroir (tab):
    n = len(tab)
    j = n // 2
    if (n%2 == 0):    # Si n est pair
        i = n // 2 - 1
    else:
        i = n // 2
    while (j<n):
        swapp(tab, i, j)
        i = i -1;   j = j +1
```

Complexité de miroir en fonction de la représentation de *tab*

	$len(tab)$	$swapp(tab, i, j)$	$miroir(tab)$
Tableau	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Liste simpl. chaînée	$\Theta(n)$	$\Theta(max(i, j))$	$\Theta(n^2)$
Liste doubl. circulaire	$\Theta(n)$	$\Theta(max(i, j))$	$\Theta(n^2)$

$n = |tab|$.

Conclusion

Sur la complexité

- Un même problème peut être résolu par différents algorithmes.
- Il est important de connaître un ordre de grandeur de la complexité de chaque algorithme.
- Il faut proscrire les algorithmes de complexité exponentielle.

Sur les listes

- Plusieurs représentations possibles des listes avec des primitives d'accès et de gestion de complexité différentes.
- Quand on évalue la complexité d'un algorithme, il faut connaître précisement la complexité de toutes les primitives associées aux structures de données.
- Choisir la structure de données en fonction des traitements à réaliser.