Complexité d'un algorithme

Le problème : On veut savoir si un algorithme est performant du point du vue de sa rapidité d'exécution.

Exemple : Comparons les 2 programmes suivants de calcul du terme u_n de la suite de Fibonacci.

OCaml
```
let fib1 n = let u0 = ref 0 and
                 u1 = ref 1 in
                 for k = 2 to n do let c = !u1 in
                                   u1 := !u0 + !u1;
                                   u0 := c
                                done ;
             !u1;;

let rec fib2 = function
        | 0 -> 0
        | 1 -> 1
        | n -> fib2 (n-1) + fib2 (n-2);;

fib1 5;;
fib2 5;;

fib1 100;;
fib2 100;;
```

On constate que pour des grands arguments, le premier programme est beaucoup plus rapide que le second !

TOUS les programmes ne se valent pas !

I] Evaluation de la complexité d'un algorithme

Objectif : Estimer par un simple calcul la rapidité d'exécution d'un algorithme.

Pour cela :

- On s'intéresse aux grands arguments (les situations qui posent problème)
- On s'autorise des approximations
- On cherche un ordre de grandeur.

La démarche :

- **Taille de l'argument** : On définit un entier n qui représente la taille des arguments de l'algorithme.

  → n la valeur de l'argument
  → n la longueur du tableau ou de la liste en argument
  → ...

- **Unité de coût** : On choisit une *opération élémentaire* à dénombrer.

  → la multiplication, ou l'addition ou les 2
  → la comparaison de deux grandeurs
  → l'itération dans une boucle
  → ...

- **Complexité** : On détermine $c(n)$ le nombre d'opérations élémentaires effectuées par un algorithme pour traiter un argument de taille n.

  - On se place dans le pire des cas
  - On recherche cette quantité sous la forme d'un équivalent ou plus simplement d'un « O ».

La rédaction :

- Taille :
- Unité de coût :
- Complexité :

  →
  →
  →

$$\boxed{c(n) = \cdots = O(\dots)}$$

Exemple 1 : Recherche d'un élément e dans un tableau trié

Algorithme naïf

On teste les éléments du tableau en partant du premier.

- Taille : n la taille du tableau
- Unité de coût : la comparaison
- Complexité : On se place dans le pire des cas, c'est à dire dans le cas où on ne trouve pas e.

  On doit tester tous les éléments du tableau ce qui fait n comparaisons.

$$\boxed{c_1(n) = n}$$

Algorithme dichotomique

On teste l'élément médian, puis s'il diffère de e, on recommence avec l'une des deux moitiés du tableau.

- Taille : n la taille du tableau
- Unité de coût : la comparaison

- Complexité : On se place dans le pire des cas, c'est à dire dans le cas où on ne trouve pas e.

  Il y a autant de comparaion que de sous-tableaux considérés.
  Cherchons combien il y en a :

  → Le k ième tableau testé a une taille de $\frac{n}{2^k}$ puisqu'on divise le tableau en 2 après chaque test.

  → On s'arrête lorsque le tableau testé a une taille inférieure ou égale à 1.
  C'est à dire lorsque $\frac{n}{2^k} \leq 1$.

$$\frac{n}{2^k} \leq 1 \iff n \leq 2^k \iff \ln n \leq k \ln 2 \iff k \geq \frac{\ln n}{\ln 2} = \log_2 n$$

$$c_2(n) \leq \log_2 n \quad \text{et donc} \quad \boxed{c_2(n) = O(\log_2(n))}$$

Exemple 2 : Calcul du terme F_n de la suite de Fibonacci

Fibonacci en itératif

OCaml
```
let fib1 n = let u0 = ref 0 and
                 u1 = ref 1 in
                 for k = 2 to n do let c = !u1 in
                                   u1 := !u0 + !u1;
                                   u0 := c
                                done ;
             !u1;;
```

- Taille : n l'argument
- Unité de coût : l'addition
- Complexité : Il y a autant d'addition que d'itérations :

$$\boxed{c_1(n) = n - 1 \sim n}$$

Fibonacci en récursif

OCaml
```
let rec fib2 = function
        | 0 -> 0
        | 1 -> 1
        | n -> fib2 (n-1) + fib2 (n-2);;
```

- Taille : n l'argument
- Unité de coût : l'addition
- Complexité : On regarde la formule de récursivité :

```
fib2 n = fib2 (n-1) + fib2 (n-2)
```

Pour obtenir `fi2 n` :

→ On commence par calculer `fib2 (n-1)` : $c_2(n-1)$ additions

→ Puis on calcule `fib2 (n-2)` : $c_2(n-2)$ additions

→ Et enfin on additionne les deux résultats obtenus : 1 addition

Nous avons donc :

$$\boxed{c_2(n) = c_2(n-1) + c_2(n-2) + 1}$$

Il s'agit d'une suite récurrente linéaire d'ordre 2 à coefficients constants avec 1 pour second membre...

Pour simplifier les calculs, nous pouvons « négliger » le 1.
En fait, effectuer cette approximation revient à sous-estimer la complexité réelle de l'algorithme.
On recherche donc la suite $c_2(n)$ vérifiant :

$$\boxed{c_2(n) = c_2(n-1) + c_2(n-2)}$$

L'équation caractéristique : $r^2 = r + 1$ admet deux racines réelles $r_1 = \dfrac{1+\sqrt{5}}{2}$ et $r_2 = \dfrac{1-\sqrt{5}}{2}$.
$c_2(n)$ est alors de la forme :

$$c_2(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n \sim A\left(\frac{1+\sqrt{5}}{2}\right)^n \quad \text{car} \quad \left|\frac{1-\sqrt{5}}{2}\right| < 1$$

Et donc

$$\boxed{c_2(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)}$$

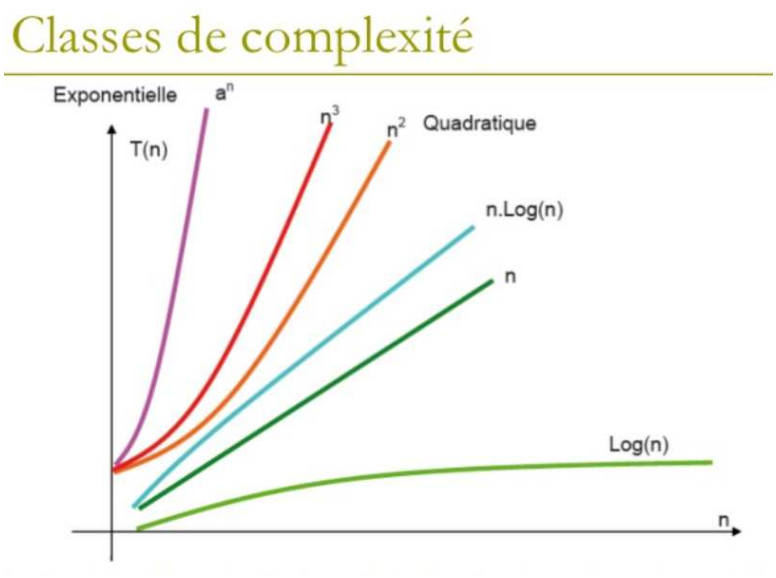
Bilan : Les types de complexité

Nous venons de rencontrer 3 types de complexité :

- Les complexités logarithmique : $c(n) = O(\ln n)$
- Les complexités linéaires : $c(n) = O(n)$
- Les complexités exponentielles : $c(n) = O(a^n)$ avec $a > 1$.

Plus généralement : on distingue

Type de complexité	Définition
Complexité logarithmique	$c(n) = O(\ln n)$ ou $c(n) = O(\ln^k n)$ avec $k > 1$
Complexité quasi-linéaire	$c(n) = O(n \ln n)$
Complexité linéaire	$c(n) = O(n)$
Complexité polynômiale	$c(n) = O(n^k)$ avec $k > 1$
Complexité quadratique	$c(n) = O(n^2)$
Complexité exponentielle	$c(n) = O(a^n)$ avec $a > 1$
Complexité hyperexponentielle	$c(n)/a^n \to +\infty$, $\forall a > 1$ expl : $c(n) = n!$



Exemple 3 : Test usuel de primalité

On vérifie si n est divisible par l'un des entiers $d \in [\![2, \lfloor\sqrt{n}\rfloor]\!]$.
On teste alors chacun de ces entiers.

- Taille : n le nombre à tester
- Unité de coût : le test
- Complexité : On se place dans le cas où n est premier (le pire des cas).

  → Le nombre de tests sera alors $c(n) = \lfloor\sqrt{n}\rfloor - 1 \sim \sqrt{n}$.
  → Nous avons alors $c(n) = O(\sqrt{n})$ ou plus usuellement

$$\boxed{c(n) = O(n)}$$

Exemple 4 : Algorithme de tri par sélection

- Taille : n la longueur du tableau à trier
- Unité de coût : la comparaison
- Complexité :

  → A la première étape, on cherche le plus grand élément parmi n éléments : $n-1$ comparaisons
  Placer cet élément en fin de tableau ne correspond pas à une comparaison.
  → Puis, on recherche le plus grand élément parmi $n-1$ éléments : $n-2$ comparaisons
  → Et on recommence ainsi jusqu'à obtenir un tableau trié.

→ La complexité est donc :

$$c(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1)n}{2} \sim \frac{n^2}{2} \quad \text{et donc} \quad \boxed{c(n) = O(n^2)}$$

Exemple 5 : Algorithme de tri par insertion

- Taille : n la longueur du tableau
- Unité de coût : la comparaison
- Complexité :

→ Pour insérer un élément à sa place dans un tableau trié de taille k, il faut au plus k comparaisons

→ Dans notre algorithme :

- on insère t.(1) dans [t.(0)] (1 comparaison) puis,
- on insère t.(2) dans [t.(0) ;t.(1)] (2 comparaions)... etc...

→ En termes de comparaisons, nous avons donc :

$$c(n) = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} \sim \frac{n^2}{2} \quad \text{et donc} \quad \boxed{c(n) = O(n^2)}$$

Exemple 6 : Algorithme du tri à bulle

- Taille : n la longueur du tableau
- Unité de coût : la comparaison
- Complexité : on ne compte pas les échanges éventuels

→ On commence par placer le plus grand élément en fin de tableau : (n-1) comparaisons

→ Puis on place le second plus grand élément en avant dernière position : (n-2) comparaisons... etc...

En termes de comparaisons, nous avons donc :

$$c(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1)n}{2} \sim \frac{n^2}{2} \quad \text{et donc} \quad \boxed{c(n) = O(n^2)}$$

Dans certains cas, la complexité est fonction de plusieurs arguments ou d'une expression des arguments

Exemple 7 : La recherche d'un élément dans une matrice de taille $p \times q$

Dans le pire des cas, il faut parcourir les $p \times q$ coefficients de la matrice pour vérifier si l'un d'entre eux correspond à l'élément cherché.

- Taille : ?
- Unité de coût : la comparaison
- Complexité : Dans le pire des cas, il y a $p \times q$ comparaisons.

Donc :

$$c(p,\ q) = pq$$

Si on a avait pris $n = pq$ pour exprimer la taille de la donnée, on avait $c(n) = n$.
Ce résultat est plus simple à interpréter, mais moins précis.

Exemple 8 : Résolution de $f(x) = 0$ sur $[a,\ b]$ à la précision ε par dichotomie

- Taille : ?
- Unité de coût : le nombre d'itérations
- Complexité :

→ L'intervalle est divisé par 2 à chaque itération et aura pour longueur $\frac{b-a}{2^k}$ à la kième itération

→ L'algorithme s'arrête lorsque cette longueur est inférieure à ε, c'est à dire $\frac{b-a}{2^k} \leq \varepsilon$.

$$\frac{b-a}{2^k} \leq \varepsilon \iff \cdots \iff k \geq \log_2 \frac{b-a}{\varepsilon}$$

→ L'algorithme s'arrête donc au bout de $k = \lfloor \log_2 \frac{b-a}{\varepsilon} \rfloor + 1$ itérations.
Nous avons donc :

$$c(a,\ b,\ \varepsilon) = O(\log_2 \frac{b-a}{\varepsilon})$$

Exemple 9 : Algorithme d'Euclide $PGCD(a,\ b)$ avec $b \leq a$

- Taille : a ou b ?
- Unité de coût : la division euclidienne
- Complexité : Il s'agit de compter le nombre de divisions euclidiennes effectuées, ou d'en trouver un majorant.

→ Idée 1 : Dans le pire des cas, la suite des restes diminue de 1 à chaque division et on obtient b divisions euclidiennes :

$$\boxed{c(a,\ b) = b}$$

→ Idée 2 : Notons $a = bq + r$ la division euclidienne de a par b avec $b \leq a$.

Remarquons que $r \leq \frac{a}{2}$.
En effet :

- Si $b \leq \frac{a}{2}$ alors $r < b \leq \frac{a}{2}$
- Si $b > \frac{a}{2}$ alors $r = a - b \leq \frac{a}{2}$.

Par conséquent, r_{2k} le 2kième reste de la division euclidienne vérifie $r_{2k} \leq \frac{a}{2^k}$ (simple récurrence).
L'algorithme s'arrête quand $r_{2k} = 0$ ($2k$ divisions euclidiennes) ce qui sera vérifié lorsque $\frac{a}{2^k} < 1$ ce qui nous donne $k \geq \log_2(a)$.
Ainsi, nous sommes certains que les itérations seront terminées lorsque $k = \lfloor \log_2(a) \rfloor + 1$.
C'est à dire après :

$$2k = 2\lfloor \log_2(a) \rfloor + 2 \sim 2\log_2(a) \quad \text{itérations}$$

Nous avons donc :

$$\boxed{c(a,\ b) = O(\log_2(a))}$$

II] Complexité des algorithmes récursifs

Nous traitons ici les cas simples où la fonction récursive f admet :

- 1 ou plusieurs appels récursifs sur des argments de taille $n-1$:
- 1 ou plusieurs appels récursifs sur des argments de taille $n-1$ et $n-2$

D'autres cas seront traités dans le cours sur les algorithmes de type Diviser Pour Régner.

Cas 1 : Un ou plusieurs appels récursifs sur des argments de taille $n-1$

Dans ce cas, la complexité vérifie une relation de récurrence de la forme :

$$\boxed{c(n) = a.c(n-1) + f(n) \quad \text{où} \quad \begin{cases} a \text{ est le nombre d'appels récursifs} \\ f(n) \text{ est le coût des opérations complémentaires} \end{cases}}$$

→ Lorsque $a = 1$:

On a $c(k) - c(k-1) = f(k)$ et en sommant pour $k \in [\![1, n]\!]$ on obtient : $\boxed{c(n) = c(0) + \sum_{k=1}^{n} f(k) \sim \sum_{k=1}^{n} f(k)}$.

Remarque : on obtient une complexité au moins LINEAIRE.

Exemple 10 : Calcul récursif de la longueur d'une liste

La formule de récursivité est : $f(x :: q) = 1 + f(q)$

- Taille : n la longueur de la liste
- Unité de coût : l'addition
- Complexité :

→ La complexité vérifie la relation de récurrence : $c(k) = 1 + c(k-1)$ c'est à dire

$$c(k) - c(k-1) = 1$$

→ En additionnant, on obtient :

$$\boxed{c(n) = c(0) + \sum_{k=1}^{n} 1 = c(0) + n \sim n}$$

→ Lorsque $a \geq 2$:

On divise $c(k) = a.c(k-1) + f(k)$ par a^k et on pose $u_k = \dfrac{c(k)}{a^k}$.

On obtient alors $u_k = u_{k-1} + \dfrac{f(k)}{a^k}$ et d'après le premier cas :

$$u_n = u_0 + \sum_{k=1}^{n} \frac{f(k)}{a^k} \sim \sum_{k=1}^{n} \frac{f(k)}{a^k} \quad \text{et donc} \quad \boxed{c(n) \sim a^n . \sum_{k=1}^{n} \frac{f(k)}{a^k}}$$

Remarque : on obtient une complexité au moins EXPONENTIELLE.

Question :

Que pensez-vous de cet algorithme ?

OCaml
```
let rec minim = function
                | [x] -> x
                | x::q -> if x < minim q then x
                                         else minim q;;
```

Exemple 11 : Les tours de Hanoï

L'appel récursif est :
- On déplace les $n-1$ disques supérieurs sur le second poteau
- On déplace le disque restant sur le 3ème poteau
- On déplace les $n-1$ disques du second poteau vers le 3ème poteau.

- Taille : le nombre de disques à déplacer sur le 3ème poteau.
- Unité de coût : le déplacement d'un disque
- Complexité :

→ La complexité vérifie la relation de récurrence : $c(k) = 1 + 2c(k-1)$.
En posant $u_k = \frac{c(k)}{2^k}$ on obtient $u_k = u_{k-1} + \frac{1}{2^k}$.

→ Cela nous donne : $u_n \sim \displaystyle\sum_{k=1}^{n} \frac{1}{2^k} \sim \frac{1}{2} . \frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} \sim 1.$

→ Par conséquent :

$$\boxed{c(n) \sim 2^n.1}$$

Cas 2 : Un ou plusieurs appels récursifs sur des arguments de taille $n-1$ et $n-2$

Dans ce cas, la complexité vérifie une relation de récurrence de la forme :

$$c(n) = a.c(n-1) + b.c(n-2) + f(n) \text{ où } \begin{cases} a \text{ est le nombre d'appels récursifs sur des arguments de taille } n-1 \\ b \text{ est le nombre d'appels récursifs sur des arguments de taille } n-2 \\ f(n) \text{ est le coût des opérations complémentaires} \end{cases}$$

Dans cette situation, on décide de simplifier les calculs en « négligeant » la quantité $f(n)$.
Cela revient en fait à sous-estimer la complexité réelle de l'algorithme.
Cette complexité vérifie alors la relation de récurrence linéaire d'ordre 2 à coefficients constants :

$$c(n) = a.c(n-1) + b.c(n-2)$$

Méthode : On recherche les racines de l'équation caractéristique $r^2 = ar + b$.

En général on a $\Delta > 0$:

On obtient deux racines réelles $r_1 \neq r_2$ et dans ce cas :

$$\boxed{c(n) = Ar_1^n + B.r_2^n \sim Ar_1^n} \quad \text{si} \quad |r_2| < |r_1|$$

La complexité est donc EXPONENTIELLE.

Les cas suivants se produisent très rarement !

• Si $\Delta = 0$, on obtient une racine réelle r et dans ce cas :

$$c(n) = (An + B)r^n \sim Anr^n = O\left((2r)^n\right)$$

On obtient de nouveau une complexité EXPONENTIELLE.

• Si $\Delta < 0$, on obtient des racines complexes $r_1 = \rho.e^{i\theta}$ et $r_2 = \overline{r_1}$ et dans ce cas :

$$c(n) = \rho^n \left(A\cos(\theta n) + B\sin(\theta n)\right)$$

Exemple 12 : Découpage d'une liste en 2 listes

La formule de récursivité est, pour une liste `x::y::q`.

- On applique la fonction à `q` et on obtient deux listes $l1$ et $l2$
- On renvoie alors `x::l1 , y::l2`

- Taille : n la longueur de la liste initiale
- Unité de coût : l'ajout d'un élément en tête d'une liste
- Complexité :

→ La relation de récurrence : $c(n) = 2 + c(n-2)$.
Ici on ne peut pas négliger le 2 car sinon, on obtiendrait une complexité constante ce qui est faux.

- On cherche une solution particulière sous la forme $\tilde{c}(n) = \lambda n$.
  On remarque que $\tilde{c}(n) = n$ convient.
- On remarque alors que $c(n) = 2 + c(n-2) \iff (c - \tilde{c})(n) = (c - \tilde{c})(n-2)$ et donc que :

$$c(n) - \tilde{c}(n) = \lambda \quad \text{d'où} \quad \boxed{c(n) = \lambda + n \sim n} \quad (\text{ avec } \lambda \text{ qui dépend de la parité de } n)$$

Exemple 13 : Fibonacci en récursif

La formule de récursivité est : $f(n) = f(n-1) + f(n-2)$.

- Taille : n la valeur de l'indice du terme à calculer
- Unité de coût : l'addition
- Complexité : $c(n) = c(n-1) + c(n-2) + 1$.

  → On néglige la valeur 1 pour se ramener à : $c(n) = c(n-1) + c(n-2)$.

  → L'équation caractéristique : $r^2 = r + 1$ admet deux racines réelles $r_1 = \dfrac{1+\sqrt{5}}{2}$ et $r_2 = \dfrac{1-\sqrt{5}}{2}$.

  → $c(n)$ est alors de la forme :

$$c(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n \sim A\left(\frac{1+\sqrt{5}}{2}\right)^n \quad \text{car} \quad \left|\frac{1-\sqrt{5}}{2}\right| < 1$$

  Et donc

$$\boxed{c(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)}$$

Pour aller plus loin !

Nous n'avons pas traité tous les cas qui peuvent se présenter en programmation récursives...
Nous verrons en particulier dans le chapitre « Algorithmes de type Diviser Pour Régner » que la complexité $c(n)$ peut vérifier des relations de récurrence de la forme :

$$c(n) = p.c(\frac{n}{q}) + f(n) \quad \text{où} \quad \begin{cases} p \text{ est le nombre d'appels récursifs} \\ \frac{n}{q} \text{ est la taille des arguments dans les appels récursifs} \\ f(n) \text{ est le coût des opérations complémentaires} \end{cases}$$

Exemple 14 : Algorithme de tri fusion

- Taille : n la longueur de la liste
- Unité de coût : la comparaison
- Complexité :

  → On commence par couper la liste en deux parties de « même » taille $\frac{n}{2}$: $f_1(n)$ comparaions

  → Puis, on tri chacune des deux listes obtenues : $c(\frac{n}{2}) + c(\frac{n}{2})$ comparaisons

  → Puis, on fusionne ces deux listes : $f_2(n)$ comparaisons

  En termes de comparaisons, nous avons donc :

$$c(n) = 2c(\frac{n}{2}) + f_1(n) + f_2(n)$$

  Reste à déterminer $c(n)$ en fonction de n...

. *La méthode sera vue dans le cours sur les algorithmes de type « diviser pour régner » .*

Nous verrons alors une méthode mathématiques permettant de déterminer $c(n)$ en fonction de n...

A vous de jouer !

Déterminer la complexité des algorithmes suivants :

1. Calcul de $n!$
2. Recherche dichotomique d'un élément dans un tableau trié.
3. Trigonalisation d'un système de cramer par la méthode de Gauss.
4. Résolution d'un système linéaire triangulaire.
5. Calcul du nombre de chiffres d'un entier naturel à l'aide de divisions par 10 successives.
6. Recherche d'une relation de récurrence vérifiée par la complexité de l'algorithme d'exponentiation rapide.
7. test pour savoir si une liste est triée.
8. Décomposition d'un entier n en base 2.
9. Calcul du terme F_n de la suite de fibonacci à l'aide de l'algorithme suivant :

OCaml
```
let fib n = let rec f = function
                          |0 -> (1,0)
                          |z -> let (a,b) = f(z-1) in (a + b, a)
        in let a,c = (f (n-1))
        in a;;
```

Vous pouvez également vous entraîner à prouver sa validité...