TD 4 : Complexité des algorithmes

En cours, on a commencé à voir comment calculer la complexité remporelle des algorithmes : on détermine quelles opérations sont élémentaires, on les compte en fonction de la taille de l'entrée, et on l'exprime comme un grand-$\mathcal{O}$ suffisamment fin. On va voir d'abord une première façon de l'évaluer en exprimant la complexité avec une relation de récurrence : cela sera souvent la méthode naturelle pour les fonctions récursives.

Exercice 1 – Complexité d'algorithmes classiques

On reprend le code du tri par insertion sur les listes en OCaml (que vous avez noté quelque part et ramené en TD, évidemment).

▶ **Question 1** Déterminer une relation de récurrence vérifiée par la complexité temporelle de `insere`. On considèrera les comparaisons comme des opérations élémentaires.

▶ **Question 2** Déterminer formellement la complexité temporelle au pire et au mieux de la fonction `insere`.

Dans le pire cas, avec $c_1, c_2 > 0$, on a :

$$C(n) = \begin{cases} c_1 & \text{si } n = 0 \\ c_2 + C(n-1) & \text{sinon} \end{cases}$$

La première ligne viens du fait que pour une liste vide, un fait juste un `match` et on renvoie `[x]`, alors que pour la seconde ligne on a un nombre borné d'opérations élémentaires (`match`, la comparaison) plus un appel récursif sur une liste de taille $n-1$. Explicitement, cela donne donc $C_{pire}(n) = \mathcal{O}(n)$.
De la même manière, dans le meilleur cas, on aura $C_{mieux}(n) = \mathcal{O}(1)$.

▶ **Question 3** Soit u une suite réelle en $\mathcal{O}(n)$. Montrer que $v : k \mapsto \sum_{i=0}^{k} u_k = \mathcal{O}(n^2)$. Qu'obtiens-t-on si $u = \mathcal{O}(1)$?

On écrit la définition du grand-$\mathcal{O}$, puis on passe la constante A en facteur et on obtient les deux résultats simplement. De manière générale, on aura $v = \mathcal{O}(nu_n)$ si u_n est croissante.

▶ **Question 4** En déduire les complexités au pire et au mieux de `tri_insertion`.

On le montre simplement en utilisant la question précédente, dont on montre qu'on est dans son cas en explicitant les relations de récurrence suivantes :

$$D_{pire}(n) = \begin{cases} c_3 & \text{si } l = [] \\ C_{pire}(n-1) + D_{pire}(n-1) \leq An + D_{pire}(n-1) & \text{sinon} \end{cases}$$

$$D_{mieux}(n) = \begin{cases} c_3 & \text{si } l = [] \\ C_{mieux}(n-1) + D_{mieux}(n-1) \leq B + D_{mieux}(n-1) & \text{sinon} \end{cases}$$

Ce qui donne $D_{pire}(n) = \mathcal{O}(n^2)$ et $D_{mieux}(n) = \mathcal{O}(n)$.

Considérons maintenant que les comparaisons ne sont pas des opérations élémentaires et notons $m(l)$ un majorant de la complexité temporelle des comparaisons entre les éléments de `l`.

▶ **Question 5** Rappeler le nombre de comparaisons des fonctions `insere` et `tri_insertion`, dans le pire et dans le meilleur cas. Exprimer alors la complexité temporelle de `insere` et `tri_insertion` en fonction de $m(l)$ et $n = |l|$.

On modifie un peu notre calcul de complexité, en majorant toutes les opérations de comparaisons par $m(l)$: cela nous donne une complexité en $\mathcal{O}\big((n + nm(l))^2\big)$ au pire et $\mathcal{O}(n + nm(l))$ au mieux. On peut notamment en déduire que le coût des opérations de comparaisons est essentiel dans le coût des opérations.

▶ **Question 6** Déterminer la complexité temporelle de la fonction `tri_fusion`.

▶ **Question 7** Déterminer les complexités spatiales de `tri_insertion` et `tri_fusion`.

Exercice 2 – Étude de l'algorithme d'Euclide

L'algorithme d'Euclide calcule le PGCD de deux entiers naturels. Il s'écrit simplement :

```ocaml
let rec pgcd a b =
  if b = 0 then a else pgcd b (a mod b)
```
OCaml

On supposera dans la suite que $a \geq 0$ et $b \geq 0$. On admettra la correction partielle de cet algorithme (preuve en cours de maths 😅). On va étudier le nombre de divisions (dans ce cas, le nombre de fois qu'on appelle `mod`) effectuées par cette fonction lorsqu'elle est appelée sur a, b, que l'on notera $f(a, b)$.

▶ **Question 1** Démontrer la terminaison de cet algorithme et borner $f(a, b)$ en fonction de b.

Pour $a, b \in \mathbb{N}$ avec $b > 0$, on sait que le reste de la division euclidienne de a par b (calculé par `a mod b`) est un entier compris entre 0 et $b-1$. Ainsi, le second argument du pgcd décroît strictement à chaque appel récursif, est minoré par 0 et la fonction termine quand $b = 0$. Ainsi, l'appel à la fonction termine et le nombre d'appels récursifs est majoré par b, donc $f(a, b) \leq b$.

▶ **Question 2** Écrire une fonction `etapes : int -> int -> int` telle que `etapes a b` calcule $f(a, b)$.

```ocaml
let rec etapes a b =
  if b = 0 then 0 else 1 + etapes b (a mod b)
```
OCaml

Soit $\phi : n \in \mathbb{N}^* \mapsto \max_{0 \leq k < n} f(n, k)$. On définit la suite $(F_n)_{n\in\mathbb{N}}$ par récurrence double :

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 2 \\ F_{n+2} &= F_n + F_{n+1} \quad \text{pour } k \geq 2 \end{aligned}$$

▶ **Question 3** Déterminer $f(F_{n+1}, F_n)$ pour $n \in \mathbb{N}$.

Montrons par récurrence que $f(F_{n+1}, F_n) = n + 1$.

Initialisation : Pour $n = 0$, on a $f(F_1, F_0) = f(2, 1) = 1$.

Hérédité : Supposons que pour $n \in \mathbb{N}$, on ait $f(F_{n+1}, F_n) = n + 1$. Alors on a :

$$F_{n+2} = F_{n+1} + F_n$$

Or F est positive et strictement croissante donc $F_{n+1} > F_n \geq 0$. Ainsi, l'égalité est en fait la division euclidienne de F_{n+2} par F_{n+1}, avec un quotient de 1 et un reste de F_n. Ainsi, l'appel à pgcd sur l'entrée (F_{n+2}, F_{n+1}) utilise l'appel récursif pgcd sur l'entrée (F_{n+1}, F_n), ce qui utilise un appel à `mod` de plus : on a donc $f(F_{n+2}, F_{n+1}) = 1 + f(F_{n+1}, F_n) = n + 2$ par hypothèse de récurrence.

▶ **Question 4** Montrer que pour tout $n \in \mathbb{N}$, si $1 \leq a < F_n$, alors $\phi(a) < n$.

Montrons-le par récurrence double sur $n \in \mathbb{N}$, et notons la propriété H_n.

Initialisation : Pour $n = 1$, on a $F_1 = 2$. Ainsi, il n'y a que le cas $a = 1$ de possible et on a $\phi(a) = f(1, 0) = 0 < 1$. Pour $n = 2$, on a $F_2 = 3$ et donc on a soit $a = 1$ (déjà traité), soit $a = 2$. Or $\phi(2) = \max(f(2, 0), f(2, 1)) = \max(0, 1) = 1 < 2$.

Hérédité : Soit $n \geq 2$. Supposons que H_n et H_{n-1} sont vrais. Soit $1 \leq a < F_{n+1}$. On a alors deux cas :

- Si $0 \leq a < F_n$, alors $\phi(a) < n < n + 1$ par H_n.
- Sinon $F_n \leq a < F_{n+1}$. Soit $1 \leq b < a$, alors on a deux cas :
  - Si $1 \leq b < F_n$, alors $f(a, b) = 1 + f(b, a \bmod b) \leq 1 + \phi(b)$. Ainsi, par H_n, on a $f(a, b) < 1 + n$.
  - Sinon, on a $F_n \leq b < a < F_{n+1}$. Alors, puisque l'on a $F_{n+1} = F_{n-1} + F_n \leq 2F_n$, on a $F_n \leq b < a < 2F_n$, donc le reste de la division euclidienne de a par b est $a - b$ (et le quotient 1). Ainsi, on a $a - b \leq F_{n+1} - F_n = F_{n-1}$ et donc :

$$\begin{aligned} f(a, b) &= 1 + f(b, a \bmod b) \\ &= 2 + f(a \bmod b, b \bmod (a \bmod b)) \\ &\leq 2 + \phi(a \bmod b) \\ &< 2 + n - 1 = n + 1 && \text{par } H_{n-1} \end{aligned}$$

▶ **Question 5** Montrer que pour tout $n \in \mathbb{N}$, $F_n \geq \left(\frac{3}{2}\right)^n$.

On va simplement le montrer par récurrence double sur $n \in \mathbb{N}$.

Initialisation : Pour $n = 0$, on a $F_0 = 1 = \left(\frac{3}{2}\right)^0$ et pour $n = 1$, on a $F_1 = 2 \geq \frac{3}{2} \geq \left(\frac{3}{2}\right)^1$.

Hérédité : Supposons que pour $n \in \mathbb{N}$, l'inégalité est vraie en n et en $n+1$. Alors :

$$\begin{aligned} F_{n+2} &= F_n + F_{n+1} \\ &\geq \left(\frac{3}{2}\right)^n + \left(\frac{3}{2}\right)^{n+1} && \text{par HI} \\ &\geq \frac{5}{2}\left(\frac{3}{2}\right)^n \\ &\geq \frac{9}{4}\left(\frac{3}{2}\right)^n \\ &\geq \left(\frac{3}{2}\right)^{n+2} \end{aligned}$$

▶ **Question 6** En déduire que $\phi(a) = \mathcal{O}(\ln a)$, *c'est-à-dire il existe $A > 0$ tel que pour tout $a \in \mathbb{N}$, $\phi(a) \leq A \ln a$ à partir d'un certain rang (ici 0 suffit).*

Soit $a \in \mathbb{N}$. Alors puisque la suite $\left(\left(\frac{3}{2}\right)^n\right)_{n\in\mathbb{N}}$ tend vers $+\infty$, il existe $n \in \mathbb{N}$ tel que $\left(\frac{3}{2}\right)^n \leq a < \left(\frac{3}{2}\right)^{n+1}$. Alors on a $a < F_{n+1}$ par la question précédente, donc $\phi(a) < n$. Or, $n = \mathcal{O}(\ln a)$ donc $\phi(a) = \mathcal{O}(\ln a)$. Pour conclure cela, il y a quand même un détail à connaître : le modulo est une opération élémentaire.

▶ **Question 7** Que peut-on en conclure sur la complexité temporelle de l'algorithme d'Euclide ?

Quand on regarde la fonction, on se rend compte qu'on fait $\mathcal{O}(\phi(a))$ appels récursifs, or l'on ne fait que des modulos et des appels récursifs, donc la complexité de la fonction est en $\mathcal{O}(\ln a)$.

Exercice 3 – Recherche dichotomique

On reprend les algorithmes classiques du cours et des TD/TPs.

Pseudo-code

```
fonction RECHERCHE(x, t) :
    deb, fin ← 0, n
    Tant que fin − deb > 0, faire :
        milieu ← (deb + fin)/2                ▷ division entière
        Si t_milieu = x alors
            Renvoyer milieu
        Sinon, si t_milieu < x
            deb ← milieu + 1
        Sinon
            fin ← milieu
        Fin si
    Fin tant que
    Renvoyer n
Fin fonction
```

▶ **Question 1** Déterminer la complexité de l'algorithme de recherche dichotomique ci-dessus. On ne comptera que les comparaisons.

▶ **Question 2** Pourquoi compter les comparaisons suffit ?

Exercice 4 – Maxima de tranches

On travaille sur un tableau $t = [t_0, \ldots, t_{n-1}]$. On cherche à calculer les *maxima par tranche de h* du tableau, définis comme suis pour $i \in [\![0, n-h]\!]$:

$$m_h(i) = \max(t\,[i\,:\,i+h])$$

▶ **Question 1** Écrire un algorithme naïf qui calcule ce tableau. On pourra s'aider d'un algorithme auxiliaire MAXTRANCHE(t, i, j) qui calcule le maximum de la tranche $t\,[i\,:\,j]$.

▶ **Question 2** Déterminer la complexité de cet algorithme.

▶ **Question 3** Il est possible d'écrire un algorithme de complexité en $\mathcal{O}(n)$ pour effectuer cette opération. Proposer une stratégie pour le faire.

▶ **Question 4** (À la maison) Implémenter cette stratégie en C.

Exercice 5 – Calcul de suite rapide

Le but de cet exercice est de calculer la suite définie par récurrence :

$$\begin{cases} u_0 = 1 \\ u_n = \dfrac{u_{n-1}}{1} + \dfrac{u_{n-2}}{2} + \cdots + \dfrac{u_0}{n} \quad \text{pour } n > 0 \end{cases}$$

▶ **Question 1** Implémenter cette suite en une fonction récursive `double u(int n)` traduisant directement la relation de récurrence.

▶ **Question 2** En comptant uniquement les divisions, déterminer une relation de récurrence vérifiée par la complexité temporelle de `suite`.

▶ **Question 3** Proposer une version améliorée de votre fonction de complexité linéaire. Est-ce que vous y voyez un désavantage par rapport à la première ?

Exercice 6 – Encore de la complexité

On considère la fonction suivante :

OCaml
```ocaml
let rec f = function
  | [] -> []
  | h :: t -> h + List.length t :: f t
```

▶ **Question 1** Déterminer la spécification de la fonction `f`.

▶ **Question 2** Déterminer la complexité temporelle et spatiale de `f`.

▶ **Question 3** Proposer une fonction de complexité spatiale et temporelle linéaire répondant à la même spécification que `f`. Montrer sa correction.