Algorithmique

Correction TD3 : Complexité et récursivité

1 Calculs de complexité

Évaluer la complexité des algorithmes suivants.

Algorithme 1 : Disjonction asymétrique

```
Entrées : n, m: entiers
pour i de 1 à n faire
    si i est pair alors
        OpérationElementaire()
    sinon
        pour j de 1 à m faire
            OpérationElementaire()
```

Algorithme 2 : Somme de puissances

```
Entrées : n: entier
i ← 1
tant que i ≤ n faire
    pour j de 1 à i faire
        OpérationElementaire()
    i ← i * 2
```

Algorithme 3 : Croissance très rapide

```
Entrées : n: entier
i ← 2
tant que i ≤ n faire
    OpérationElementaire()
    i ← i^2
```

Algorithme 4 : Condition coûteuse

```
Entrées : n:entier
i ← 0
tant que Puissance(2,i) < n faire
    i ← i + 1
retourner i
```

Algorithme 5 : Croissance non triviale

```
Entrées : n: entier
i ← 0, j ← 1
tant que i ≤ n faire
    OpérationElementaire()
    i ← i + j
    j ← j + 1
```

Algorithme 6 : Calcul corsé

```
Entrées : n: entier
i ← n, j ← 0
tant que i ≥ 1 faire
    pour k de 1 à i faire
        pour ℓ de 1 à j faire
            OpérationElementaire()
    i ← i/2
    j ← j + 1
```

Méthodologie

- On calcule en premier lieu la complexité interne de chacune des boucles, en commençant par la plus profonde. Cette complexité peut dépendre de variables internes de l'algorithme, auquel cas il faut évaluer leur valeur à chaque itération de boucle. Pour chaque variable interne `i`, on évalue la valeur i_k que prend `i` après k passages de boucles (et donc i_0 est sa valeur avant le début de la boucle). En général, cela revient à résoudre une formule de récurrence selon les opérations faites sur `i` à chaque passage de boucle.
- Une fois qu'on a une estimation $O(f(k))$ de la complexité du k-ème passage de la boucle, il faut estimer le nombre $k_{\max}$ de passages de boucle, en fonction d'une variable externe à la boucle (ici, le paramètre `n` de la fonction). Il s'agit du plus grand k tel que la condition de boucle est vérifiée (par exemple $i_k \leq n$).
- On obtient la complexité de la boucle en faisant la somme $\sum_{k=0}^{k_{\max}} O(f(k)) = O\left(\sum_{k=0}^{k_{\max}} f(k)\right)$.
- On répète pour chaque boucle en commençant par la plus profonde.

Correction

1. La complexité interne de la boucle `Pour` la plus profonde est $O(1)$ (une opération élémentaire). Cette boucle fait m itérations, et sa complexité est donc $O(m)$. En outre, la complexité du bloc `Alors` est $O(1)$ (un test et une opération élémentaire), tandis que celle du bloc `Sinon` est $O(1)$. Donc le bloc `Si` a un coût asymétrique en fonction de sa condition, i.e. de la parité de i. On observe qu'il y a une alternance entre ces deux complexités, et donc la complexité moyenne de chaque itération de la boucle `Pour` principale est $O(\frac{m+1}{2}) = O(m)$. Finalement, cette boucle fait n itérations, et sa complexité est donc $O(mn)$.

   Une autre approche consiste à séparer les itérations paires et impaires de la boucle `Pour` principale. Les premières ont chacune un coût $O(1)$, et les secondes un coût $O(m)$. Le coût total est alors

$$\lfloor n/2 \rfloor \times O(1) + \lceil n/2 \rceil \times O(m) = O(nm).$$

2. La complexité interne de la boucle `Pour` est $O(1)$ (une opération élémentaire). Le nombre d'itérations de la boucle `Pour` est donné par la valeur de la variable `i`. La complexité de la boucle `Pour` vaut donc $\sum_{j=1}^{i} O(1) = O(i)$.

   Pour la boucle `Tq`, on a une variable interne `i` dont on doit évaluer la valeur. La valeur initiale de `i` est $i_0 = 1$, et sa valeur après $k \geq 1$ passages de boucle est $i_k = 2i_{k-1} = \underbrace{2 \times \cdots \times 2}_{k} \times i_0 = 2^k$. On reste dans la boucle `Tq` si $i_k \leq n$, et le plus grand entier k vérifiant cela est $k_{\max} = \lfloor \log_2 n \rfloor$. La complexité de la boucle `Tq` est donc

$$\sum_{k=0}^{k_{\max}} O(i_k) = O\left(\sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k \right) = O\left(\frac{2^{\lfloor \log_2 n \rfloor + 1} - 1}{2 - 1} \right) = O(n).$$

   On a ici utilisé la formule pour la somme d'une suite géométrique : $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$ pour tout $q > 1$. Si on ne sait pas calculer la somme qui intervient dans le calcul de complexité, on peut la borner à l'aide d'une intégrale.

   Si une fonction f est *monotone* sur l'intervalle $[a, b]$, alors

$$\min\{f(a), f(b)\} + \int_a^b f(x)\,dx \leq \sum_{i=a}^{b} f(i) \leq \max\{f(a), f(b)\} + \int_a^b f(x)\,dx.$$

   Dans notre cas, cela donnerait

$$\sum_{i=0}^{n} q^n \leq q^n + \int_0^n q^x\,dx = q^n + \left[\frac{q^x}{\ln q} \right]_0^n < \left(1 + \frac{1}{\ln q} \right) q^n.$$

3. La complexité interne de la boucle `Tq` est $O(1)$. Nous avons une variable interne `i` dont nous évaluons la valeur i_k après k passages de boucle. Nous avons $i_0 = 2$, et $i_k = i_{k-1}^2$ pour tout $k \geq 1$. Pour résoudre cette formule de récurrence, on peut remarquer qu'en posant $i'_k := \log_2 i_k$, on a $i'_0 = 1$ et $i'_k = 2i'_{k-1}$, ce qui implique que $i'_k = 2^k$ pour tout $k \geq 0$. On a donc $i_k = 2^{i'_k} = 2^{2^k}$ pour tout $k \geq 0$. La condition de boucle est $2^{2^k} \leq n$, et le plus grand entier k vérifiant cela est $k_{\max} = \lfloor \log_2 \log_2 n \rfloor$. La complexité totale est donc $O(\log \log n) \times O(1) = O(\log \log n)$.

4. La complexité interne de la boucle `Tq` est $O(1)$, et la complexité de la condition de boucle est $O(i)$. En posant i_k la valeur de i à la k-ème itération de la boucle, nous avons donc que cette k-ème itération a coût $O(i_k)$. Or $i_0 = 0$, et $i_k = i_{k-1} + 1$ pour tout $k \geq 1$, ce dont on déduit que $i_k = k$ pour tout $k \geq 0$. La

condition de boucle est $2^{i_k} < n$, et le plus grand entier k vérifiant cela est $k_{\max} = \lceil \log_2 n \rceil - 1 = O(\log n)$. La complexité totale est donc

$$\sum_{i=0}^{k_{\max}} O(\log i_k) = O\left(\sum_{i=1}^{k_{\max}} \log k\right).$$

Ici on a ignoré le premier terme de la somme pour ne pas avoir $\log 0$ qui n'est pas défini. On peut le faire car l'itération correspondante est en fait de coût $O(1)$, qui est négligeable dans le coût total. Pour borner cette somme, on passe par l'intégrale, ce qui donne

$$\begin{aligned}\sum_{i=1}^{k_{\max}} \log k \leq \log k_{\max} + \int_1^{k_{\max}} \log x dx = \log k_{\max} + \left[x \log x - x\right]_1^{k_{\max}} &= O(k_{\max} \log k_{\max}) \\ &= O(\log n \log\log n).\end{aligned}$$

5. La complexité interne de la boucle `Tq` est $O(1)$. Nous avons deux variables internes `i` et `j` dont nous évaluons les valeurs. Les valeurs initiales de `i` et `j` sont respectivement $i_0 = 0$ et $j_0 = 1$, et leurs valeurs après $k \geq 1$ passages de boucle sont respectivement $i_k = i_{k-1} + j_{k-1}$ et $j_k = j_{k-1} + 1 = \underbrace{1 + \cdots + 1}_{k} + j_0 = k + 1$. On a donc

$$i_k = j_{k-1} + j_{k-2} + \cdots + j_0 = \sum_{j=1}^{k} j = \frac{k(k+1)}{2}.$$

Ici encore, si on ne sait pas calculer cette somme, on peut la borner à l'aide d'une intégrale, ce qui donne

$$i_k = \sum_{j=1}^{k} j \geq \int_0^k x\, dx = \left[\frac{x^2}{2}\right]_0^k = \frac{k^2}{2}.$$

Ici, on veut une borne inférieure sur i_k, car la condition de sortie de la boucle `Tq` est $i_k > n$, qui est donc vérifiée en particulier si $\frac{k^2}{2} > n$, c'est-à-dire $k > \sqrt{2n}$. On fait donc $O(\sqrt{n})$ passages de boucle, chacun de coût $O(1)$, ce qui signifie que la complexité totale est $O(\sqrt{n}) \times O(1) = O(\sqrt{n})$.

6. La complexité de la k-ème itération de la boucle `Tq` est $O(i_{k-1} j_{k-1})$, où $i_k = n/2^k$ et $j = k$ sont les valeurs respectives de `i` et `j` après k itérations de boucle. La condition de sortie est atteinte quand $k > \log_2 n$. La complexité totale est donc

$$\sum_{k=0}^{\lfloor \log_2 n \rfloor} O(kn/2^k) = O\left(n \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k}\right) < O(n) \times \sum_{k=0}^{\infty} \frac{k}{2^k}.$$

Or $S := \sum_{k=0}^{\infty} k2^{-k} = 2$. La complexité totale est donc $O(n)$. Pour prouver que cette somme infinie vaut $O(1)$, on peut encore une fois passer par une intégrale, en utilisant une intégration par parties. La fonction $f : x \mapsto x2^{-x}$ est décroissante sur l'intervalle $[2, +\infty[$, et donc

$$\begin{aligned}\sum_{k=2}^{\infty} k2^{-k} &\leq \frac{1}{2} + \int_2^{+\infty} x2^{-x}\, dx = \frac{1}{2} + \left[-\frac{x2^{-x}}{\ln 2}\right]_2^{+\infty} - \int_2^{+\infty} -\frac{2^{-x}}{\ln 2}\, dx \\ &\leq \frac{1}{2} + \frac{1}{2\ln 2} - \left[\frac{2^{-x}}{(\ln 2)^2}\right]_2^{+\infty} = \frac{1}{2} + \frac{1}{2\ln 2} + \frac{1}{2(\ln 2)^2}.\end{aligned}$$

En rajoutant les deux premiers termes de la somme S, on obtient $S \leq 0 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2\ln 2} + \frac{1}{2(\ln 2)^2} = O(1)$.

2 Recherche dichotomique

Algorithme 7 : EstPrésent

Entrées : x: valeur, `tab`: tableau trié croissant de taille n
si $x <$ *tab[0] ou* $x >$ *tab[n-1]* **alors**
  retourner *Faux*
début $\leftarrow 0$, fin $\leftarrow n-1$
tant que *fin* $\geq$ *début* **faire**
  milieu $\leftarrow \lfloor$(début+fin)/2$\rfloor$
  si *tab[milieu]* $= x$ **alors**
    retourner *Vrai*
  si *tab[milieu]* $> x$ **alors**
    fin $\leftarrow$ milieu $- 1$
  sinon
    début $\leftarrow$ milieu $+ 1$
retourner *Faux*

1. *Donner un invariant de boucle vérifié par* `EstPrésent` *qui permet de garantir que l'algorithme est correct.*

2. *Donner un invariant de boucle vérifié par* `EstPrésent` *qui permet de garantir que l'algorithme est de complexité* $O(\log n)$*, puis justifier le calcul de cette complexité.*

   L'algorithme vérifie l'invariant de boucle suivant : à la k-ème itération de boucle, on a

   (i) $\texttt{fin} - \texttt{début} \leq n/2^k$

   (ii) `x` ne se trouve pas en dehors des positions contenues dans l'intervalle $[\texttt{début}, \texttt{fin}]$ dans `tab`.

   Cet invariant se prouve par récurrence, avec une initialisation triviale. Supposons l'invariant vrai à la k-ème itération, et notons $\ell \leq n/2^k$ la valeur de $\texttt{fin} - \texttt{début}$ à cette itération. Si on n'a pas retourné `Vrai` à la $(k+1)$-ème itération, alors dans le pire cas (le bloc `sinon`) la valeur de $\texttt{fin} - \texttt{début}$ est $\lceil t/2 \rceil - 1 \leq t/2 \leq t/2^{k+1}$, ce qui prouve le point (i). Le point (ii) se prouve aisément par une analyse de cas.

   Le point (i) de l'invariant nous indique qu'au bout de $\lceil \log_2 n \rceil + 1$ itérations, on a $\texttt{fin} - \texttt{début} < 1$, ce qui implique que l'on a soit $\texttt{fin} = \texttt{début}$ (et on atteindra la condition de sortie de boucle à l'itération suivante), soit on a déjà atteint la condition de sortie de boucle. La complexité de l'algorithme est donc $O(\log n)$ puisque chaque itération de boucle a coût $O(1)$ (elle ne contient que des opérations élémentaires).

   Le point (ii) de l'invariant nous indique que lorsque l'on sort de la boucle, l'ensemble des positions dans lesquelles `x` peut se trouver est vide. L'algorithme ne renvoie donc `Faux` que lorsque `x` est absent de `tab`, et il est évident qu'il ne renvoie `Vrai` que lorsque `x` est présent dans `tab` (à la position donnée par la variable `milieu`).

3. *Ecrire un algorithme* `PlusProche(tab,x)` *qui renvoie la valeur contenue dans* `tab` *la plus proche de* `x` *(i.e. la valeur* $y \in$ `tab` *telle que* $|y - x|$ *est minimum). Celui-ci devra reposer sur une recherche dichotomique, et donc être de complexité* $O(\log n)$*.*

   L'objectif est de trouver deux cases consécutives de `tab` qui encadrent la valeur de x. Pour cela, nous allons préserver l'invariant $\texttt{tab[début]} \leq x \leq \texttt{tab[fin]}$, tout en faisant se rapprocher `début` et `fin` de façon classique. Lorsque l'on aura $\texttt{fin} = \texttt{début} + 1$, alors la valeur la plus proche de x dans `tab` sera nécessairement soit `tab[début]`, soit `tab[fin]`, et il ne restera plus qu'à renvoyer la meilleure des deux valeurs.

Algorithme 8 : PlusProche

Entrées : x: valeur, `tab`: tableau trié croissant de taille n

si $x \leq$ *tab[0]* **alors**
  retourner *tab[0]*

si $x \geq$ *tab[n-1]* **alors**
  retourner *tab[n-1]*

début $\leftarrow 0$, fin $\leftarrow n-1$

tant que *fin* $-$ *début* ≥ 2 **faire**
  milieu $\leftarrow \lfloor$(début+fin)/2$\rfloor$
  si *tab[milieu]* $\geq x$ **alors**
    fin $\leftarrow$ milieu
  sinon
    début $\leftarrow$ milieu $+ 1$

si $|$*tab[fin]* $- x| < |$*tab[début]* $- x|$ **alors**
  retourner *tab[fin]*

sinon
  retourner *tab[début]*

4. *Ecrire une fonction `ISqrt(n)` qui prend en entrée un entier n, et renvoie $\lfloor\sqrt{n}\rfloor$, en utilisant une recherche dichotomique (et uniquement des opérations arithmétiques : addition, soustraction, multiplication, division entière).*

   Le but est de trouver deux entiers consécutifs a et b tels que $a^2 \leq n < b^2$. Alors la solution à retourner sera a. Il suffit de commencer par encadrer de façon naïve la valeur de $\lfloor\sqrt{n}\rfloor$ pour initialiser les valeurs de a et b, que l'on fera ensuite converger par le principe de la recherche dichotomique.

Algorithme 9 : ISqrt

Entrées : n : entier strictement positif

$a \leftarrow 1$

$b \leftarrow n + 1$

tant que $b - a \geq 2$ **faire**
  $m \leftarrow \lfloor(a+b)/2\rfloor$
  si $m^2 \leq n$ **alors**
    $a \leftarrow m$
  sinon
    $b \leftarrow m$

retourner a

L'algorithme vérifie bien l'invariant de boucle $a^2 \leq n < b^2$, ce qui garantit qu'il est correct.

3 Fibonacci

Les nombres de Fibonacci sont définis par $F_0 = F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour tout $n \geq 2$. On considère l'algorithme suivant.

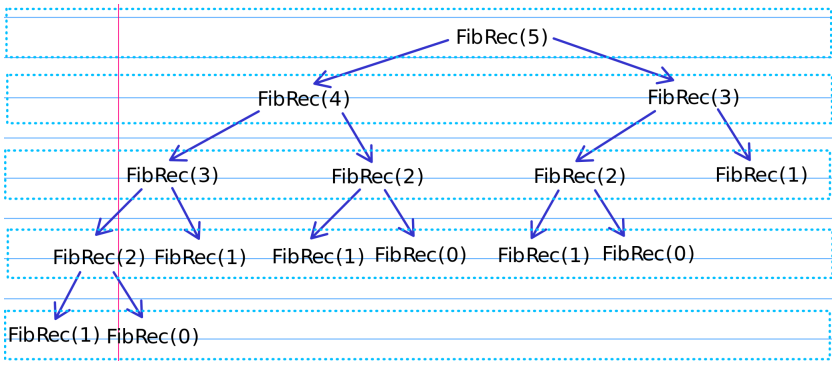
Algorithme 10 : FibRec

Entrées : $n \geq 0$: entier

si $n \leq 1$ **alors**
  retourner *1*

sinon
  retourner *FibRec(n-1)+FibRec(n-2)*

1. *Énumérer les appels récursifs de* `FibRec(5)` *(sous la forme d'un arbre). Que remarque-t-on ?*



On remarque qu'on fait des appels récursifs redondants. Par exemple, on fait 5 fois appel à `FibRec(1)`. En fait, le nombre d'appels récursifs de `FibRec(n)` est supérieur au n-ème nombre de Fibonacci F_n, qui est exponentiel en n.

2. *La complexité de l'algorithme* `FibRec` *est exponentielle, à cause de redondances dans les appels récursifs. Pour pallier cela, on peut implémenter un algorithme récursif* `Fibo` *qui prend comme paramètre un entier n, et qui retourne la paire (F_n, F_{n+1}). Écrire un tel algorithme.*

Algorithme 11 : Fibo

Entrées : $n \geq 0$: entier
si $n = 0$ **alors**
  retourner $(1, 1)$
sinon
  $(x, y) \leftarrow$ `Fibo(n-1)`
  retourner $(y, x + y)$

3. *Quelle est la complexité de* `Fibo` *?*

`Fibo` a complexité linéaire : on fait $n + 1$ appels récursifs sur une entrée n, et le corps de la fonction a pour complexité $O(1)$.