I - Introduction par des exemples.

1) En itératif.

a)Cas d'une boucle répétitive indexée (boucle `for`).

Exemple : algorithme calculant $n!$:

```
   let factorielle1 n =
      let p = ref 1 in
         for k =1 to n do
            p := (!p) * k
         done;
     !p
;;
```

i)Preuve de correction.

Nous pouvons commencer par vérifier la correction de notre algorithme pour quelques jeux d'essai :

n	k	p
5	1	

Nous n'allons pas nous contenter de cette illustration, nous allons mettre en place une **preuve**.
Afin de montrer la correction de cette répétitive c'est-à-dire que les données connues avant la répétitive nous fournissent les résultats attendus, nous allons décrire la situation à la fin de chaque itération (la "situation" étant souvent l'état des différentes variables).

Dans notre exemple, nous voulons démontrer que, pour tout $i \in [|1;n|]$, à la fin de la $i^{\text{ème}}$ itération,

$$„ \left\{ \begin{array}{lll} k & \text{contient} & i \\ p & \text{contient} & i! \end{array} \right. " \qquad (\mathcal{H}_i)$$

Cette description est appelée **invariant de boucle**.

Montrer la correction de notre algorithme revient à montrer la correction de l'**invariant de boucle** par récurrence.

Remarque : attention à être encore plus rigoureux qu'en mathématiques :

— 1er cas : si $n = 0$ alors `factorielle1 0` nous conduit à la séquence
`p <-1`

et nous renvoit la valeur de `p` c'est-à-dire $1 = 0!$

— 2nd cas : si $n \in \mathbb{N}^*$, notons pour tout $i \in [|1; n|]$, $\mathcal{H}_i$: " $\begin{cases} k & \text{contient} & i \\ p & \text{contient} & i! \end{cases}$ "

• $\mathcal{H}_1$: p est initialisé à 1 (`p <- 1`).
Ensuite k prend la valeur 1 (`k <- 1`) et p prend la valeur $!p * k$ (`p <-1` $\times$ `1`) .
Et donc à la fin de la 1 ère itération :
" $\begin{cases} k & \text{contient} & 1 \\ p & \text{contient} & 1! \end{cases}$ " donc $(\mathcal{H}_1)$ est vraie.

• Soit $i \in [|2; n|]$, supposons $(\mathcal{H}_{i-1})$ donc avant de commencer la $i^{\text{ème}}$ itération, " $\begin{cases} k & \text{contient} & i-1 \\ p & \text{contient} & (i-1)! \end{cases}$ ",
et nous passons à l'itération suivante : La valeur de k est incrémentée de 1 (d'après la sémantique de "pour") donc `k <-i` ;
et p reçoit $!p * k$ donc `p <-(i-1)!` $\times$ `i = i!` donc $(\mathcal{H}_i)$ est vraie.

• Par le principe de récurrence, pour tout i appartenant à $[|1; n|]$.

A la fin de cette répétitive, soit après n itérations, $\begin{cases} k & \text{contient} & n \\ p & \text{contient} & n! \end{cases}$

Donc, dans les deux cas, après la répétitive, p contient $n!$.
Nous renvoyons alors le contenu de p, à savoir $n!$, le résultat attendu, donc notre algorithme est correct.

ii)Preuve de terminaison.

Seule une instruction répétitive attirera notre attention.
Nous sommes en présence d'une répétitive indexée donc notre algorithme se termine.

iii)Complexité.

Nous cherchons à évaluer la complexité temporelle de notre algorithme en dénombrant le nombre d'opérations effectuées.

Nous commençons par choisir une taille de données c'est-à-dire une ou plusieurs données représentatives du temps d'exécution de l'algorithme.
Ici nous choisissons n (la seule données).

Nous choisissons maintenant nos opérations fondamentales c'est-à-dire des opérations représentatives du travail réalisé par l'algorithme.
Nous choisissons ici la multiplication (tout ceci est très subjectif...).

Notons $T(n)$ la complexité de l'algorithme.

`let p= ref 1 in` n'amène aucune multiplication.

— 1er cas : si $n = 0$ alors la répétitive ne s'exécute pas donc aucune multiplication.

— 2nd cas : si $n \in \mathbb{N}^*$, étudions alors la répétitive :
Pour tout $k^i n[|1;n|]$, `p:=!p*k` amène 1 multiplication et donc la répétitive amène $\sum_{k=1}^{n} 1 = n$ multiplications.

Donc, à ce stade, nous avons effectué n multiplication ;
et l'instruction $!p$ n'amène aucune multiplication.

Conclusion : $\boxed{\text{T(n)=n}}$.
On conclut que $T(n) = O(n)$ c'est-à-dire que la complexité est linéaire.

Exercice : Ecrire itératif un algorithme de recherche d'un maximum parmi les valeurs d'un ensemble ordonné contenues dans un tableau.
Prouver sa correction sa terminaison et étudier sa complexité.

Correction :

```
let max t =
   let n= Array.length t and p=ref 0 in
      for k=1:(n-1) do
         if t.(k) > t.(!p) then
            p:=k
      done;
   (!p)
;;
```

b)Cas d'une répétitive conditionnelle (boucle `while`).

Exemple : algorithme de recherche d'un élément x dans un tableau :

```
let recherche a x =
   let n= Array.length a and i= ref 0 in
      while (!i<n) && (a.(!i) <> x) do
         i:=!i+1
      done;
   (!i<n)
;;
```

i)Preuve de correction.

Proposons un invariant de boucle : Soit $k \in [|1;n|]$,

"à la fin de la $k^{\text{ème}}$ itération, i contient k et x n'appartient pas aux valeurs

$\{a(0), ..., a(k-1)\}$." $(\mathcal{H}_k)$

• $(\mathcal{H}_1)$: si l'on fait une première itération, c'est que la condition $a.(0) \neq x$ est vraie donc x n'appartient pas à $\{a.(0)\}$.

• Soit $k \in [|2; n|]$, supposons $(\mathcal{H}_{k-1})$ c'est-à-dire au début de la $k^{\text{ème}}$ itération, $\begin{cases} i & \text{contient} & k-1 \\ x & \text{n'appartient pas à} & \{a(0), ..., a(k-2)\} \end{cases}$.

Alors, en supposant les tests vérifiés $\begin{cases} !i & < & n \\ a.(!i) & <> & x \end{cases}$,

$a.(k-1)$ est donc différent de x.
Et comme x n'appartient pas à $\{a(0), ..., a(k-2)\}$, alors x n'appartient pas à $\{a(0), ..., a(k-1)\}$,

et i reçoit $i+1$ donc i contient k.

Ainsi, à la fin de la $k^{\text{ème}}$ itération, $\begin{cases} i & \text{contient} & k \\ x & \text{n'appartient pas à} & \{a(0), ..., a(k-1)\} \end{cases}$
donc $(\mathcal{H}_k)$ est vraie.

• Par principe de récurrence, pour tout k appartenant à $[|1; n|]$, $(\mathcal{H}_k)$ est vraie et la condition `while (!i<n) && (a.(!i) <> x)` n'est plus vérifiée à partir d'un nombre fini d'itérations.

Finalement, deux cas se présentent :
— soit $(!i < n)$ et $a.(!i) = x$:
  x apparaît alors dans a et le booléen $(!i < n)$ est évalué à `true`.

— soit $(!i \geq n)$ et à la fin de la $n^{\text{ème}}$ itération, $\begin{cases} i & \text{contient} & n \\ x & \text{n'appartient pas à} & \{a(0), ..., a(n-1)\} \end{cases}$
  ,
  donc x n'apparaît pas dans le tableau et le booléen $(!i < n)$ est évalué à `false`.

Donc Notre algorithme est correct .

ii)Preuve de terminaison.
La preuve est basée sur le fait qu'il n'existe pas de suite strictement décroissante d'entiers naturels.
Introduisons la suite des $(n-i)$:
$(n-i)$ est une suite d'entiers et la condition $(!i < n)$ suivie de $i :=!i+1$ majore les valeurs de i par n donc $n - i \in \mathbb{N}$.
Et $i_{\text{suivant}} < - i_{\text{précédent}} + 1$ donc $(n-i)_{\text{suivant}} - (n-i)_{\text{précédent}} = -i_{\text{suivant}} + i_{\text{précédent}} = -1$.

La suite $(n-i)$ est donc uns suite strictement décroissante d'entiers naturels .
Comme $(\mathbb{N}, \leq)$ est un ordre bien fondé, l'algorithme se termine en un nombre

fini d'étapes.

iii)Complexité.
Choisissons comme :
• taille de données : n
• opérations fondamentales : les comparaisons

Nous ne connaissons pas le nombre d'itérations, nous pouvons alors par exemple nous placer dans le pire des cas : complexité au pire (cf IV 3) Différents types de complexité).

Dans le pire des cas, x n'appartient pas au tableau, nous effectuons alors n itérations avec à chaque fois 2 comparaisons .
En notant $T(n)$ notre complexité, on obtient :

$\boxed{\text{T(n)= 2n= O(n)}}$: complexité linéaire

2) En récursif.
a)Premier exemple.
Choisissons comme exemple le calcul de la factorielle d'un entier naturel.

```
let rec factorielle2 n = match n with
   | 0 -> 1
   |_ -> n*(factorielle2 (n-1))
```

i)Preuve de correction.
Nous singeons la définition mathématique :

$$\begin{cases} 0! & = & 1 \\ \forall n \in \mathbb{N}^*,\ n! & = & n(n-1)! \end{cases}$$

Donc l'algorithme est forcément correct.
Nous développerons peu de preuves de correction d'algorithmes récursifs.

ii)Preuve de terminaison.
$\forall n \in \mathbb{N}^*$, `factorielle2 n` amène un appel récursif sur la donnée $(n-1)$ et $n-1 < n$.
Ainsi, l'ordre usuel sur $\mathbb{N}$ étant bien fondé (autrement dit, il n'existe pas de suite strictement décroissante), notre algorithme se termine.

iii)Complexité.
En choisissant n comme taille de données et la multiplication comme opération fondamentale, notons $C(n)$ la complexité de notre algorithme.

$$\begin{cases} C(0) & = & 0 \\ \forall n \in \mathbb{N}^*,\ C(n) & = & C(n-1)+1 \end{cases}$$

Suite arithmétique de raison 1 et de premier terme 0 donc $\boxed{\text{C(n)=n=O(n)}}$: complexité linéaire.

II - Preuves de terminaison.

1) Ensemble ordonné.

Définition 1

Soit E un ensemble non vide.
On appelle relation binaire sur E, toute application de $E \times E$ dans $\mathbb{B} = \{0; 1\}$.

Exemples

i. Soit E un ensemble non vide, la relation d'égalité sur E est une relation binaire sur E.
ii. La relation de comparaison usuelle $\leq$ est une relation binaire sur $\mathbb{R}$, notée $\leq$.
iii. La relation de divisibilité est une relation binaire sur $\mathbb{N}$ notée $|$.

Remarque : de manière générale, nous utiliserons la notation infixe pour une relation binaire.

Définition 2

Soit E un ensemble non vide, $\mathcal{R}$ une relation binaire sur E.

i. La relation $\mathcal{R}$ est dite réflexive sur E si $\forall x \in E$, $x\mathcal{R}x$.

ii. La relation $\mathcal{R}$ est dite anti-symétrique sur E si

$$\forall (x, y) \in E^2, \quad \left.\begin{array}{c} x\mathcal{R}y \\ y\mathcal{R}x \end{array}\right\} \Rightarrow x = y.$$

iii. La relation $\mathcal{R}$ est dite transitive sur E si

$$\forall (x, y, z) \in E^3, \quad \left.\begin{array}{c} x\mathcal{R}y \\ y\mathcal{R}z \end{array}\right\} \Rightarrow x\mathcal{R}z.$$

Dans le cas où $\mathcal{R}$ vérifie ces trois points, on dit que $\mathcal{R}$ est une relation d'ordre sur E et que $(E, \mathcal{R})$ est un ensemble ordonnée.

Exemples

i. $\mathbb{R}$ muni de la relation de comparaison usuelle $\leq$ est un ensemble ordonné.
ii. L'ensemble $\mathbb{N}^*$ muni de la relation de divisibilité est un ensemble ordonné.
Mais c'est faux dans $\mathbb{Z}$.
iii. Définissons sur $\mathbb{N}^2$ une relation binaire par :

$$\forall (a_1, a_2) \in \mathbb{N}^2,\ \forall (b_1, b_2) \in \mathbb{N}^2,\ (a_1, a_2) \leq (b_1, b_2) \Leftrightarrow \left\{\begin{array}{ccc} a_1 & < & b_1 \\ & \text{ou} & \\ a_1 = b_1 & \text{et} & a_2 \leq b_2 \end{array}\right.$$

Cette relation d'ordre est appelée l'ordre lexicographique sur $\mathbb{N}^2$, et se note $\leq_{\text{lex}}$.
Remarque : Python applique cet ordre sur les couples.

Définition 3

Soit $(E, \leq)$ un ensemble ordonné.
Nous définissons sur E un ordre strict noté $<$ par :

$$\forall (x,y) \in E^2,\ x < y \Leftrightarrow \left\{ \begin{array}{rcl} x & \leq & y \\ \text{et } x & \neq & y \end{array} \right.$$

2) Ordre total, ordre partiel.

Définition 4

Soit $(E, \leq)$ un ensemble ordonné.
On dit que x et y sont comparables si $x \leq y$ ou $y \leq x$.

Exemples

i. Dans l'ensemble $(\mathbb{N}, \leq)$, deux entiers naturels sont toujours comparables.
ii. Dans l'ensemble ordonné $(\mathbb{N}^*, |)$, 4 et 7 ne sont pas comparables.

Définition 5

Soit $(E, \leq)$ un ensemble ordonné.

i. On dit que $(E, \leq)$ est un ensemble totalement ordonné si deux éléments de E sont toujours comparables.
ii. On dit que $(E, \leq)$ est un ensemble partiellement ordonné s'il n'est pas totalement ordonné.

Exemples

i. $(\mathbb{N}, \leq)$ et $(\mathbb{N}^2, \leq_{\text{lex}})$ sont totalement ordonnés.
L'ordre lexicographique est un ordre total (les mots (à deux lettres ici) sont totalement ordonnés dans le dictionnaire).
ii. $(\mathbb{N}^*, |)$ est partiellement ordonné.

3) Plus petit élément, élément minimal.

Définition 6

Soit $(E, \leq)$ un ensemble ordonné.

i. *On appelle plus petit élément de E tout élément m tel que :*

$$\begin{cases} m \in E \text{ et} \\ (\forall x \in E), \quad m \leq x \end{cases}$$

ii. *On appelle élément minimal de E tout élément m tel que :*

$$\begin{cases} m \in E \text{ et} \\ (\forall x \in E), \quad (x \leq m \Rightarrow x = m) \end{cases}$$

Exemple

Dans l'ensemble $(\mathbb{N} \setminus \{0;1\}, |)$, partons à la recherche des éléments minimaux.
Soit $m \in \mathbb{N}\setminus\{0;1\}$, m est un élément minimal de $\mathbb{N}\setminus\{0;1\}$ ssi $(\forall x \in \mathbb{N}\setminus\{0;1\})$, $(x|m \Rightarrow x = m)$
ssi $m \in \mathbb{N} \setminus \{0;1\}$ et $\mathcal{D}_m^+ = \{1; m\}$
ssi m est un nombre premier.

Donc les éléments minimaux de $\mathbb{N} \setminus \{0;1\}$ sont les nombres premiers.

Partons maintenant à la recherche d'un plus petit élément de $\mathbb{N} \setminus \{0;1\}$.
Soit $m \in \mathbb{N} \setminus \{0;1\}$,
m est un plus petit élément de $\mathbb{N} \setminus \{0;1\}$ ssi $\forall x \in \mathbb{N} \setminus \{0;1\}$, $m|x$.
recherche vaine : il n'existe pas de p.p.e. dans $\mathbb{N} \setminus \{0;1\}$.

Propriété 1

Soit $(E, \leq)$ un ensemble ordonné.

i. En supposant que E admet un plus petit élément m, alors m est un élément minimal de E.
ii. En supposant de plus $(E, \leq)$ totalement ordonné, alors tout élément minimal de E est un plus petit élément de E.

Preuve

i. m étant le plus petit élément de E : $\begin{cases} m \in E \text{ et} \\ (\forall x \in E), \quad m \leq x \end{cases}$,
D'une part, m appartient à E,
et d'autre part, s'il existe $x \in E$ tel que $x \leq m$ alors ,
$x \leq m$ et $m \leq x$ donc, par antisymétrie, $x = m$.
Donc m est un élément minimal de E.

Remarque : on peut prouver que s'il existe un plus petit élément alors il n'y a qu'un

seul élément minimal (le plus petit élément).
En revanche, s'il n'existe pas de plus petit élément, alors il peut y avoir plusieurs éléments minimaux.

ii. Soit m un élément minimal de E, alors m appartient à E.
Soit $x \in E$, l'ordre étant total sur E, on a $m \leq x$ ou bien $x \leq m$.
Dans le cas où $x \leq m$, m étant un élément minimal de E, alors $x = m$.
Donc dans tous les cas $m \leq x$..
Donc m est le plus petit élément de E.

4) Ensemble bien ordonné, ensemble bien fondé.

Définition 7

Soit $(E, \leq)$ un ensemble ordonné.

i. *On dit que $(E, \leq)$ est bien ordonné si toute partie non vide de E admet un*
ii. *On dit que $(E, \leq)$ est bien fondé si toute partie non vide de E admet un é minimal.*

Remarques

i. Soit E un ensemble totalement ordonné.
Alors $(E, \leq)$ est bien ordonné si et seulement si il est bien fondé.
ii. Un ensemble bien ordonné est aussi bien fondé.

Exemples

i. $(\mathbb{N}, \leq)$ est bien ordonné et bien fondé.

ii. $(\mathbb{N}^2, \leq_{\text{lex}})$ est bien ordonné et bien fondé.
iii. $(\mathbb{N} \setminus \{0; 1\}, |)$ est bien fondé.

Théorème 2

Soit $(E, \leq)$ un ensemble ordonné.
Les assertions suivantes sont équivalentes :

i. $(E, \leq)$ est bien fondé
ii. Il n'existe pas de suite strictement décroissante d'éléments de E.

Preuve

— $(i) \Rightarrow (ii)$:
Supposons $(E, \leq)$ bien fondé et effectuons un raisonnement par l'absurde :
Supposons qu'il existe une suite (u_n) strictement décroissante d'éléments de E.
Notons $\mathcal{U} = \{u_n; n \in \mathbb{N}\}$.
$\mathcal{U}$ est une partie non vide de E donc ; $(E, \leq)$ étant bien fondé, $\mathcal{U}$ admet un élément minimal m.

Comme m appartient à $\mathcal{U}$, il existe $p \in \mathbb{N}$ tel que $m = u_p$.
Or, comme la suite est strictement décroissante, alors $\underline{u_{p+1} < u_p}$ et $\left\{ \begin{array}{lll} u_{p+1} & \in & \mathcal{U} \text{ et} \\ u_{p+1} & \leq & u_p = m \end{array} \right.$
donc, m étant un élément minimal de $\mathcal{U}$, $\underline{u_{p+1} = u_p}$.
<u>contradiction</u>

— $(ii) \Rightarrow (i)$:
Supposons qu'il n'existe pas de suite strictement décroissante d'éléments de E.
Soit $\mathcal{X}$ une partie non vide de E.
Nous avons à montrer que $\mathcal{X}$ admet un élément minimal.
Raisonnons par l'absurde en supposant que $\mathcal{X}$ n'admet pas d'élément minimal c'est-à-dire :
$(\forall m \in \mathcal{X}),\ (\exists x \in \mathcal{X},\ \left\{ \begin{array}{lll} x & \leq & m \text{ et} \\ x & \neq & m \end{array} \right.)$
c'est-à-dire :
$(\forall m \in \mathcal{X}),\ (\exists x \in \mathcal{X},\ x < m)$.

$\mathcal{X} \neq \emptyset$, choisissons u_0 dans $\mathcal{X}$.
u_0 n'étant pas un élément minimal, il existe $u_1 \in \mathcal{X}$ tel que $u_1 < u_0$.
u_1 n'étant pas un élément minimal, il existe $u_2 \in \mathcal{X}$ tel que $u_2 < u_1$.
u_2 n'étant pas un élément minimal, il existe $u_3 \in \mathcal{X}$ tel que $u_3 < u_2$.
...etc...
Nous construisons ainsi une suite strictement décroissante (u_n) d'éléments de E.
<u>contradiction</u>

5) <u>La preuve de terminaison.</u>

Nous est donné un algorithme qui, à partir des données, nous fournit des résultats.
Nous espérons obtenir un résultat en un temps fini. Un algorithme étant une suite d'instructions, nous avons à montrer que cette suite est finie.

a)<u>En itératif.</u>

Les instructions simples, alternatives, sont comptées comme une seule instruction.
Seules les instructions répétitives attireront notre attention.
Afin de montrer qu'un répétitive correspond à un nombre fini d'instructions, nous allons construire une suite (u_n) strictement décroissante d'éléments d'un ensemble $(E, \leq)$ bien fondé telle que chaque itération de la répétitive nous fasse passer au terme suivant de la suite (u_n).
Et l'ordre sur E étant bien fondé, une telle suite ne peut comporter qu'un nombre fini de termes : notre répétitive amènera un nombre fini d'itérations.

Exemple
L'appartenance d'un élément à un ensemble vu dans la partie I.

Remarques

i. Les répétitives <u>indexées</u> ne demandent pas de preuve de terminaison.
ii. En cas de <u>répétitives imbriquées</u>, le travail commencera sur la répétitive <u>intérieure</u>.

b)<u>En récursif.</u>

L'idée est semblable en construisant une suite strictement décroissante d'un ensemble ordonné $(E, \leq)$ telle que chaque appel récursif nous fasse passer au terme suivant.

Exemples

(a) Factorielle d'un entier naturel (cf partie I).

(b) Algorithme récursif du pgcd de deux entiers naturels :

```
let rec pgcd a b = match b with
   | 0 -> a
   |_ -> if (a>b) then pgcd (a-b) b
                  else pgcd b a
;;
```

Exemple
$12 \wedge 5 = 7 \wedge 5 = 2 \wedge 5 = 5 \wedge 2 = 3 \wedge 2 = 1 \wedge 2 = 2 \wedge 1 = 1 \wedge 1 = 0 \wedge 1 = 1 \wedge 0 = 1$

Nous proposons comme suite strictement décroissante la suite (b, a) dans $\mathbb{N}^2$ munie de l'ordre lexicographique.

Notons (a', b') le couple amené par l'appel récursif de la fonction pgcd sur $(a, b) \in \mathbb{N}^2$.
On a alors $(a', b') = \begin{cases} (a-b, b) & \text{si} \quad a \geq b \\ (b, a) & \text{si} \quad a < b \end{cases}$
donc $(b', a') = \begin{cases} (b, a-b) & \text{si} \quad a \geq b \\ (a, b) & \text{si} \quad a < b \end{cases}$

Se présentent alors deux cas :
— dans le cas où $a \geq b$ alors :
$b' = b$
$a' = a - b < b$ (car $b > 0$) donc $(b', a') < (b, a)$.
— Dans le cas où $a < b$:
$b' = a < b$ donc $(b', a') < (b, a)$.

Donc dans les deux cas $(b', a') <_{\text{lex}} (b, a)$.
La suite (b, a) est donc bien une suite strictement décroissante sur $\mathbb{N}^2$.
L'ordre lexicographique étant bien fondé sur $\mathbb{N}^2$, l'algorithme se termine.

III - Preuves de correction.

1) Le principe de récurrence.

Théorème 3 : récurrence simple

Soit $\mathcal{P}$ une propriété indexée sur $\mathbb{N}$ telle que :

$$\begin{cases} \mathcal{P}_0 & \text{est vraie} \\ \forall n \in \mathbb{N}, & \mathcal{P}_n \Rightarrow \mathcal{P}_{n+1} \end{cases}$$

Alors pour tout $n \in \mathbb{N}$, $\mathcal{P}_n$ est vraie.

Théorème 4 : principe de récurrence forte

Soit $\mathcal{P}$ une propriété indexée sur $\mathbb{N}$ telle que :

$$\begin{cases} \mathcal{P}_0 & \text{est vraie} \\ \forall n \in \mathbb{N}, & (\forall k \in [|0;n|],\ \mathcal{P}_k \text{ est vraie } \Rightarrow \mathcal{P}_{n+1} \text{ est vraie }) \end{cases}$$

Alors pour tout $n \in \mathbb{N}$, $\mathcal{P}_n$ est vraie.

Exemple

L'algorithme d'exponentiation rapide.
$x \in \mathbb{R}$,

(u_n) la suite définie par $\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, \quad u_n = \begin{cases} (u_{\frac{n}{2}})^2 & \text{si} \quad n \text{ est pair} \\ (u_{\frac{n-1}{2}})^2 \times x & \text{si} \quad n \text{ est impair} \end{cases} \end{cases}$:

Montrons que pour tout $n \in \mathbb{N}$, $u_n = x^n$.

Notons pour tout $n \in \mathbb{N}$,
$\mathcal{H}_n$: "$\forall k \in [|0;n|],\ u_k = x^k$".
• $u_0 = 1 = x^0$ donc $(\mathcal{H}_0)$ est vraie.
• Soit $n \in \mathbb{N}$, supposons $(\mathcal{H}_n)$ vraie.

— 1er cas : si $n+1$ pair alors $(u_{\frac{n+1}{2}})^2$ et notons $p = \frac{n+1}{2}$ appartient à $\mathbb{N}$ et $p - n = \frac{n+1}{2} - n = \frac{1-n}{2}$.
Or, $n+1$ étant pair, $n \neq 0$ donc $n \geq 1$ donc $p - n \leq 0$.
Ainsi, $(\mathcal{H}_n)$ étant vraie, comme on a vérifié que $p \in [|0;n|]$ alors $u_p = x^p$ donc : $u_{n+1} = (u_p)^2 = x^{2p} = x^{n+1}$.

— 2nd cas : si $n+1$ impair alors notons $p = \frac{n}{2} \Leftrightarrow n+1 = 2p+1$
Alors, $u_{n+1} = (u_p)^2 \times x$, et en vérifiant de même que $p \in [|0;n|]$, on obtient : $u_{n+1} = (x^p)^2 \times x = x^{2p+1} = x^{n+1}$.

2) Le principe d'induction.

Théorème 5 : induction simple

Soit $(E, \leq)$ bien fondé.
$A \neq \emptyset$ dans E. $\varphi : E \setminus A \to E$ telle que $\forall x \in E \setminus A$, $\varphi(x) < x$.
On considère alors un prédicat $\mathcal{P}$ vérifiant :

i. Initialisation : Pour tout $x \in A$, $P(x)$ est vraie.
ii. Hérédité : Pour tout $x \in E \setminus A$, $P(\varphi(x)) \Rightarrow P(x)$.

Alors, pour tout $x \in E$, $P(x)$ est vraie.

Remarque

Relions ce principe d'induction au principe de récurrence sur $\mathbb{N}$.
$E = \mathbb{N}$
$A = \{0\}$

$$\varphi : \begin{array}{ccc} \mathbb{N}^* & \to & \mathbb{N} \\ n & \mapsto & n-1 \end{array}$$

Preuve
Notons $X = \{x \in E; P(x) \text{ est fausse }\}$.
Supposons X non vide.
Alors, l'ordre sur $(E, \leq)$ étant <u>bien fondé</u>, X admet un élément minimal m.
Or, pour tout $a \in A$, $P(a)$ est vraie donc $m \in E \setminus A$.
Ainsi, par propriété de φ, $\varphi(m) < m$ donc $\varphi(m)$ n'appartient pas à X (car m élément <u>minimal</u> de X) et donc $P(\varphi(m))$ est vraie.
Mais alors, d'après la propriété (ii), <u>$P(m)$ est vraie</u>.
Mais <u>$m \in X$</u> : <u>contradiction</u>.
<u>Conclusion :</u> $X = \emptyset$.

<u>Théorème 6 : induction forte</u>

Soit $(E, \leq)$ <u>bien fondé</u> et M l'ensemble de ses éléments minimaux.
Si un prédicat $\mathcal{P}$ sur E vérifie :

i. <u>Initialisation :</u> Pour tout $x \in M$, $P(x)$ est vraie.
ii. <u>Hérédité :</u> Pour tout $x \in E \setminus M$, $(\forall y < x,\ P(y)) \Rightarrow P(x)$.

Alors, pour tout $x \in E$, $P(x)$ est vraie.

Exemple
Montrons que tout entier naturel n supérieur ou égal à 2 admet une décomposition en produit de facteurs premiers.
Nous allons utiliser le principe d'induction sur $(\mathbb{N} \setminus \{0;1\}, |)$.

i. <u>Initialisation :</u> Soit p un élément minimal de $\mathbb{N} \setminus \{0;1\}$ c'est-à-dire un nombre premier alors $p = p^1$.

ii. <u>Hérédité :</u> Soit n appartenant à $\mathbb{N} \setminus \{0;1\}$.
Supposons que : $\forall x \in \mathbb{N} \setminus \{0;1\}$ tel que $\left\{ \begin{array}{ccc} x & | & n \\ x & \neq & n \end{array} \right.$, x admet une d.p.f.p. (hypothèse d'induction)

n admet un diviseur premier p, il existe donc r appartenant à $\mathbb{N}$ tel que $n = r \times p$; avec $r \in \mathbb{N}^*$, $r|n$ et $r < n$.

— <u>1er cas : si $r = 1$</u> alors $n = p = p^1$.

— <u>2nd cas : si $r \neq 1$</u> alors $r \in \mathbb{N} \setminus \{0;1\}$ et $\left\{ \begin{array}{c} r|n \\ r \neq n \end{array} \right.$ donc r admet une d.p.f.p., et donc $n = r \times p$ admet une décomposition en produit de facteurs premiers.

iii. <u>Conclusion :</u> par <u>principe d'induction</u>, pour tout n appartenant à $\mathbb{N} \setminus \{0;1\}$, n admet une d.p.f.p.

3) <u>Preuves de correction</u>.
Afin de valider notre algorithme, nous devons montrer qu'à partir des données fournies, nous obtenons les résultats escomptés.

a)En itératif.
Un algorithme itératif est une séquence finie d'instructions se succédant avec ou sans condition.
Les instructions simples et conditionnelles ne demandent pas réellement de preuve de correction, il nous suffit de les réaliser avec discussion éventuelle.

Exemple
Algorithme affichant le maximum de trois nombres :

```
let exemple a b c =
   let m= ref a in
      if (b < !m) then  m:=b;
      if (c < !m) then m:=c
   !m
;;
```

On réalise les instructions pour vérifier la correction de notre algorithme.

a	b	c	m
a	b	c	
			a
			$\min(\{a;b\})$
			$\min\{a;b;c\})$

Explications : m est initialisé à a.
Ensuite, nous comparons m (càd a) à b : dans le cas où $b < a$, alors m prend la valeur de b, sinon m conserve la valeur de a.
Ainsi, m contient $\min(\{a;b\})$.
Nous comparons ensuite m càd $\min(\{a;b\})$ à c ; et m contient après cette comparaison le minimum entre $\min(\{a;b\})$ et c.
Au final, m contient $\min(\{a;b;c\})$.

Reste les instructions répétitives qui elles vont demander une attention particulière.
Une instruction répétitive est une instruction qui se répète un nombre fini de fois, et, afin de valider cette instruction répétitive, nous allons valider chaque instruction qui se répète en décrivant , à chaque itération, l'état courant.
Nous adopterons le principe d'induction et la description de l'état courant sera appelé un invariant de boucle.

Bous avons deux approches de la théorie des invariants de boucle :

i. L'algorithme étant exact, à nous de mettre en place un invariant de boucle afin de prouver notre algorithme.
ii. Nous réfléchissons à l'invariant de boucle et nous en déduisons l'algorithme.

Exemples

i. Recherche d'un maximum dans un tableau :

```
let indice_maximum a =
   let n= Array.length a and i= ref 0 in
      for k=1 to (n-1) do
         if a.(k) > a.(!i) then
            i:=k
      done;
   !i
;;
```

Soit $i \in [|1; n-1|]$, nous proposons comme invariant de boucle :
"à la fin de la $j^{\text{ème}}$ itération, i appartient à $[|0; j|]$ tel que $a.(i) = \max(\{a.(0); a.(1); ...; a.(j)\})$".
(prouver cet invariant de boucle par récurrence : à la maison)

Après la répétitive, soit après la $(n-1)^{\text{ème}}$ itération, i appartient à $[|0; n-1|]$ et $a.(i) = \max(\{a.(0); a.(1); ...; a.(n-1)\})$.

ii. Recherche d'un mot dans un texte :

Nous sont données un texte a et un mot m.
Nous avons à écrire un algorithme qui détermine l'appartenance de m à a.

Nous choisissons de représenter a et m par des chaînes de caractères : de type `String` en Ocaml.

Nous respectons l'invariant de boucle suivant :

et nous comparons maintenant $a.[i]$ et $m.[j]$:

— soit $a.[i] = m.[j]$, alors sous les conditions que i soit inférieur strictement à longueur(texte) (`String.length a`) et que j soit inférieur strictement à longueur(mot) `String.length a`,
`i <- i+1`
`j <- j+1`

et dans le cas où $j = longueur(mot) - 1$ alors *mot* apparaît dans le *texte*, dans le cas où $j < longueur(mot) - 1$ et $i = longueur(texte) - 1$ et *mot* n'apparaît pas dans le *texte*.

— soit $a.[i] \neq m.[j]$, alors nous recommençons la recherche du mot au début mais décalé d'un cran dans le texte

```
i <- i-j+1
j <- 0
```

```
let recherche_mot_texte a m =
   let la = String.length a and lm = String.length m and i = ref 0 and j = ref
      while (!i <= la-lm) && (not trouve ) do
         while (!j < lm ) && (a.[!i]=m.[!j]) do
            i:=!i+1;
            j:=!j+1
         done;
         if (!j=lm)
            then
               trouve:=true
            else
               begin
                  i:=!i-!j+1;
                  j:=0
               end
     done
     !trouve
;;
```

b)En récursif.

La preuve de correction d'un algorithme récursif nous conduit à singer des preuves par récurrence ou des preuves par induction.

Exemple

Calcul récursif du pgcd :

```
let rec pgcd a b = match b with
   |0            -> a
   |b when a < b -> pgcd b a
   |_            -> pgcd (a-b) b
;;
```

L'algorithme est basé sur trois propriétés mathématiques :

$$\forall a \in \mathbb{N},\ a \wedge 0 = a$$

$$\forall (a,b) \in \mathbb{N}^2,\ a \wedge b = b \wedge a$$

$$\forall (a,b) \in \mathbb{N}^2,\ a \geq b \Rightarrow (a-b) \wedge b = a \wedge b$$

• Nous nous rappelons que nous avons fait une preuve de terminaison. En effet, le couple (b,a) constitue une suite de $\mathbb{N}^2$ strictement décroissante pour l'ordre lexicographique.

• Mettons en place une preuve de correction par le principe d'induction :

i. Initialisation : Pour tout a appartenant à $\mathbb{N}$, $a \wedge 0 = a$ donc l'algorithme est correct pour tout couple $(a, 0)$, $a \in \mathbb{N}$.

ii. Hérédité : Soit (a, b) appartenant à $\mathbb{N}^2$, tel que pour tout couple $(x, y) \in \mathbb{N}^2$, tel que $(y, x) <_{\text{lex}} (b, a)$, l'algorithme est correct sur les données x et y.

— 1er cas : $a < b$, alors `pgcd a b` amène l'appel récursif `pgcd x y` où $\begin{cases} x & = & b \\ y & = & a \end{cases}$ et $(y, x) <_{\text{lex}} (b, a)$ donc `pgcd x y` renvoit bien $x \wedge y = b \wedge a$ or $a \wedge b = b \wedge a$ donc `pgcd a b` renvoit bien $a \wedge b$.

— 2nd cas : $a \geq b$, alors `pgcd a b` amène l'appel récursif `pgcd x y` où $\begin{cases} x & = & a - b \\ y & = & b \end{cases}$ et $(y, x) = (b, a - b) <_{\text{lex}} (b, a)$ donc `pgcd x y` renvoit bien $x \wedge y = (a - b) \wedge b$ or $(a - b) \wedge b = a \wedge b$ donc `pgcd a b` renvoit bien $a \wedge b$.

iii. Conclusion : par principe d'induction, pour tout $(a, b) \in \mathbb{N}^2$, `pgcd a b` renvoit bien $a \wedge b$.

IV - Complexité d'un algorithme.

1) Généralités.

La qualité d'un programme dépend essentiellement de deux choses :

✯ son temps d'exécution qui dépend de :
- de la structure des données du problème pour ce programme,
- de la qualité du code engendré par le compilateur,
- de la rapidité des instructions offerte par l'ordinateur,
- de l'efficacité de l'algorithme,
- de la qualité de la programmation.

✯ son utilisation de l'espace mémoire (qui influe aussi sur le temps d'exécution : les temps d'accès).

L'idée de la suite est de s'affranchir de considérations subjectives (matériel, programme, ...) et de nous centrer sur l'algorithme en lui-même.

Nous allons donc nous limiter dans un premier temps à :
- la structure des données ,
- l'efficacité de l'algorithme,
- l'espace mémoire utilisé.

Le mot **complexité** recouvre en fait deux réalités :
- la complexité des algorithmes : c'est l'étude de l'efficacité comparée d'algorithmes réalisant la même tâche, nous mesurons le temps d'exécution et aussi l'espace mémoire utilisé de chaque algorithme,
- la complexité des problèmes : c'est la classification des problèmes en fonction des performances des meilleurs algorithmes connus à ce jour qui les résolvent, citons le problème du voyageur de commerce qui a une telle complexité que nous pouvons le considérer comme non résolu.

Notre exposé se limite à la complexité des algorithmes et plus particulièrement à leur **complexité temporelle** (temps d'exécution) ,

même si comme nous l'avons déjà dit, la **complexité spatiale** (occupation mémoire) peut influer sur la complexité temporelle (temps d'accès) .

2) Mise en place de la complexité d'un algorithme.

La complexité temporelle de notre algorithme va dépendre des données manipulées, de leur structure et de leur taille.

Le choix de la structure des données se fera par bon sens. Par exemple, pour effectuer la division euclidienne de deux entiers, nous manipulerons des variables de type entier plutôt que des variables de type flottant ...

La structure des données manipulée ne sera pas évoquée dans la mise en place de la complexité de l'algorithme.

Leur taille nous semble plus importante. Nous définissons alors la notion de **taille de données** qui représentera la taille des objets manipulés.

Il est assez difficile de donner une définition théorique de taille de données. Citons plutôt quelques situations classiques afin de « sentir » cette notion :

algorithme	taille de données
opération sur les nombres	valeur maximale des nombres
calcul de sommes, de produits	nombre de termes de la somme, du produit
polynôme	degré du polynôme, valeur maximale des coefficients du polynôme
tableau	nombre de cases du tableau

La taille des données n étant choisie, nous allons maintenant chercher à évaluer le temps d'exécution $T(n)$ de notre algorithme en dénombrant tout simplement, le nombre d'opérations fondamentales que l'algorithme réalise.

Cette notion d'**opération fondamentale** est comme la notion de taille de données, difficile à définir et é associer aux types de problème étudié.

type de données	opération(s) fondamentale(s)
opérations sur les nombres	addition, multiplication, ...
recherche d'un élément dans un tableau	comparaison
tri des éléments d'un tableau	comparaison, échange

Une fois choisie(s), nous déciderons que les opérations fondamentales ont toutes le même coût.

3) Différents types de complexité.

Nous rencontrerons trois types de complexité :

— la complexité **au pire** : évaluation de la complexité dans le pire des cas, synonyme de sécurité,
— la complexité **au mieux** : évaluation de la complexité dans le meilleur des cas, inutile,
— la complexité **en moyenne** : évaluation de la moyenne des complexités dans tous les cas, sans doute représentative mais difficile à évaluer.

4) Différentes échelles de complexité.

Définition : Nous avons différents types de complexité, dans un ordre croissant pour une taille de données entière n :

— **logarithmique** : de l'ordre de $O(\text{Log}_2(n))$,
— **linéaire** : de l'ordre de $O(n)$,
— **quadratique** : de l'ordre de $O(n^2)$,
— **polynômiale d'ordre** p , $p \in \mathbb{N}$: de l'ordre de $O(n^p)$,
— **exponentielle** : de l'ordre de $O(a^n)$,
— **hyper-exponentielle** : comme $O(n!)$ par exemple .

Les ordinateurs personnels sont actuellement capables d'accomplir 10^{11} opérations par seconde et imaginons des algorithmes qui effectuent un traitement de données dans un temps $T(n)$.

Nous donnons ci-dessous un tableau de temps de traitement que l'on peut espérer pour n éléments suivant la complexité de l'algorithme,

où s, m ,h ,j , a désignent respectivement la seconde, la minute, l'heure, le jour, l'année et u l'âge estimé de l'univers (15 milliards d'années),

et où les nombres totalement inimaginables n'ont pas été affichés.

	$\text{Log}_2(n)$	n	$n\text{Log}_2(n)$	n^2	n^4	2^n	4^n	$n!$
$n = 5$	$2,3 \cdot 10^{-11}\,s$	$5 \cdot 10^{-11}\,s$	$1,1 \cdot 10^{-10}\,s$	$2,5 \cdot 10^{-10}\,s$	$6 \cdot 10^{-9}\,s$	$3,2 \cdot 10^{-10}\,s$	$1 \cdot 10^{-8}\,s$	$1,2 \cdot 10^{-9}\,s$
$n = 10$	$3,3 \cdot 10^{-11}\,s$	$1 \cdot 10^{-10}\,s$	$3,3 \cdot 10^{-10}\,s$	$1 \cdot 10^{-9}\,s$	$1 \cdot 10^{-7}\,s$	$1 \cdot 10^{-8}\,s$	$3 \cdot 10^{-5}\,s$	$3,6 \cdot 10^{-5}\,s$
$n = 15$	$3,9 \cdot 10^{-11}\,s$	$1,5 \cdot 10^{-10}\,s$	$5,8 \cdot 10^{-10}\,s$	$2,2 \cdot 10^{-9}\,s$	$5 \cdot 10^{-7}\,s$	$3,2 \cdot 10^{-7}\,s$	$1 \cdot 10^{-2}\,s$	$13\,s$
$n = 20$	$4,3 \cdot 10^{-11}\,s$	$2 \cdot 10^{-10}\,s$	$8,6 \cdot 10^{-10}\,s$	$4 \cdot 10^{-9}\,s$	$1,6 \cdot 10^{-6}\,s$	$1 \cdot 10^{-5}\,s$	$11\,s$	$281\,j$
$n = 30$	$4,9 \cdot 10^{-11}\,s$	$3 \cdot 10^{-10}\,s$	$1,4 \cdot 10^{-9}\,s$	$9 \cdot 10^{-9}\,s$	$8,1 \cdot 10^{-6}\,s$	$1 \cdot 10^{-2}\,s$	$133\,j$	$6470\,u$
$n = 50$	$5,6 \cdot 10^{-11}\,s$	$5 \cdot 10^{-10}\,s$	$2,8 \cdot 10^{-9}\,s$	$2,5 \cdot 10^{-8}\,s$	$6,2 \cdot 10^{-7}\,s$	$3,1\,h$	$30u$	
$n = 100$	$6,6 \cdot 10^{-11}\,s$	$1 \cdot 10^{-9}\,s$	$6,6 \cdot 10^{-9}\,s$	$1 \cdot 10^{-7}\,s$	$1 \cdot 10^{-3}\,s$	$30\,u$		
$n = 500$	$8,9 \cdot 10^{-11}\,s$	$5 \cdot 10^{-9}\,s$	$4,4 \cdot 10^{-8}\,s$	$2,5 \cdot 10^{-6}\,s$	$6,2 \cdot 10^{-1}\,s$			
$n = 1000$	$1 \cdot 10^{-10}\,s$	$1 \cdot 10^{-8}\,s$	$1 \cdot 10^{-7}\,s$	$1 \cdot 10^{-5}\,s$	$10\,s$			
$n = 5000$	$1,2 \cdot 10^{-10}\,s$	$5 \cdot 10^{-8}\,s$	$6,1 \cdot 10^{-7}\,s$	$2,5 \cdot 10^{-4}\,s$	$1,7\,h$			
$n = 10000$	$1,3 \cdot 10^{-10}\,s$	$1 \cdot 10^{-7}\,s$	$1,3 \cdot 10^{-6}\,s$	$1 \cdot 10^{-3}\,s$	$1\,j$			
$n = 50000$	$1,6 \cdot 10^{-10}\,s$	$5 \cdot 10^{-7}\,s$	$7,8 \cdot 10^{-6}\,s$	$2,5 \cdot 10^{-2}\,s$	$2\,a$			

5) Calculs de complexité.

a)Sommes usuelles.

Propriété 7

Pour tout $n \in \mathbb{N}^*$,

$$\sum_{k=1}^{n} 1 = n = O(n), \qquad \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = O(n^2), \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3$$

b)Etudes de suites.

i)Suites arithmétiques.

(u_n) étant une suite arithmétique de raison r, alors,

$$u_n = u_0 + nr \text{ et } u_n = O(n)$$

ii)Suites arithmético-géométriques.
(u_n) étant définie par : $\forall n \in \mathbb{N}$, $u_{n+1} = au_n + b$; où $a > 1$ (une complexité est toujours croissante) , alors $u_n = O(a^n) = O(e^{n\ln(a)})$.
La complexité est exponentielle : pas bon du tout !

iii)Suites dichotomiques.
Nous considérons $p \in \mathbb{N}^*$, $\lambda \in \mathbb{R}_+$, $\gamma \in \mathbb{R}_+$, et la suite (u_n) vérifiant la relation :

$$\forall n \in \mathbb{N}^*, \quad u_n = pu_{\lfloor \frac{n}{2} \rfloor} + \lambda n^\gamma$$

6) Exemples. Nous obtiendrons ces suites lors des calculs de complexité des algorithmes récursifs utilisant le paradigme "diviser pour régner".
Par exemple, u_n est la complexité d'un algorithme récursif lorsque l'on divise la liste en 2 à chaque appel récursif : cf tri fusion , ...etc...

Considérons $p \in [|1; +\infty|[$, λ et $\gamma \in \mathbb{R}_+$ et la suite u définie par $\begin{cases} u_0 & \in \mathbb{R}_+ \\ \forall n \in \mathbb{N}^*, & u_n = pu_{\lfloor \frac{n}{2} \rfloor} + \lambda n^\gamma \end{cases}$

i. Nous limitons dans un premier temps l'étude aux entiers n de la forme 2^k, $k \in \mathbb{N}$:

   Introduisons alors pour tout $k \in \mathbb{N}$, $v_k = u_{2^k}$.
   Ainsi, pour tout $k \in \mathbb{N}^*$, $v_k = u_{2^k} = pu_{2^{k-1}} + \lambda 2^{k\gamma} = pv_{k-1} + \lambda 2^{k\gamma}$.

   Afin d' "absorber" le p, on introduit à nouveau une suite auxiliaire w définie par : $\forall k \in \mathbb{N}$, $w_k = \frac{v_k}{p^k}$.
   Alors $w_k = \frac{v_k}{p^k} = \frac{pv_{k-1}+\lambda 2^{k\gamma}}{p^k} = w_{k-1} + \lambda \frac{2^{k\gamma}}{p^k}$.

$$\begin{aligned}
\text{Soit } k \in \mathbb{N}^*, w_k &= w_{k-1} + \lambda \frac{2^{k\gamma}}{p^k} \\
w_{k-1} &= w_{k-2} + \lambda \frac{2^{(k-1)\gamma}}{p^{k-1}} \\
w_{k-2} &= w_{k-3} + \lambda \frac{2^{(k-2)\gamma}}{p^{k-2}} \\
&. \\
&. \\
&. \\
w_2 &= w_1 + \lambda \frac{2^{2\gamma}}{p^2} \\
w_1 &= w_0 + \lambda \frac{2^{1\gamma}}{p^1}
\end{aligned}$$

On somme ces égalités, par télescopage on obtient :

$$\begin{aligned}
\text{Soit } w_k &= w_0 + \lambda \sum_{j=1}^{k} \frac{2^{j\gamma}}{p^j} \\
&= w_0 + \lambda \sum_{j=1}^{k} \left(\frac{2^\gamma}{p}\right)^j
\end{aligned}$$

— <u>1er cas :</u> $\frac{2^\gamma}{p} = 1$ c-à-d $p = 2^\gamma \Leftrightarrow \log_2(p) = \gamma$.
Ainsi,

$$\begin{aligned}
\forall k \in \mathbb{N},\ w_k &= w_0 + \lambda k \\
\text{donc } \forall k \in \mathbb{N},\ \frac{u_{2k}}{p^k} &= w_0 + \lambda k \\
\text{d'où } \forall k \in \mathbb{N},\ u_{2k} &= (w_0 + \lambda k) p^k
\end{aligned}$$

Soit $k \in \mathbb{N}$, en notant $n = 2^k$, on a :

$$u_n = (w_0 + \lambda k) p^k$$

où $k = \log_2(n) = \frac{\ln(n)}{\ln(2)}$
et $p^k = e^{k \ln(p)} = e^{\frac{\ln(n)}{\ln(2)} \ln(p)} = n^{\log_2(p)}$.

Ainsi, $u_n = (w_0 + \lambda \log_2(n)) n^{\log_2(p)} = O\left(\log_2(n) n^\gamma\right)$

— <u>2nd cas :</u> $\frac{2^\gamma}{p} \neq 1$
Alors, pour tout $k \in \mathbb{N}$,

$$\begin{aligned}
w_k &= w_0 + \lambda \times \frac{2^\gamma}{p} \frac{\left(\frac{2^\gamma}{p}\right)^k - 1}{\frac{2^\gamma}{p} - 1} \\
&= w_0 + \frac{\lambda 2^\gamma}{2^\gamma - p}\left(\left(\frac{2^\gamma}{p}\right)^k - 1\right) \\
\text{donc } \forall k \in \mathbb{N},\ v_k &= p^k w_k \\
&= w_0 p^k + \frac{\lambda 2^\gamma}{2^\gamma - p}\left((2^\gamma)^k - p^k\right) \\
&= \left(w_0 - \frac{\lambda 2^\gamma}{2^\gamma - p}\right) p^k + \frac{\lambda 2^\gamma}{2^\gamma - p}(2^\gamma)^k
\end{aligned}$$

Donc, il existe α et β constantes telles que, pour tout $k \in \mathbb{N}$, en notant $n = 2^k \Leftrightarrow \log_2(n) = k$,

$$u_n = u_{2k} = v_k = \alpha p^{\log_2(n)} + \beta 2^{\gamma \log_2(n)}$$

On distingue donc 2 cas :

— <u>dans le cas où $p > 2^\gamma \Leftrightarrow \log_2(p) > \gamma$</u> alors

$$u_n = O\left(p^{\log_2(n)}\right)$$

et $p^{\log_2(n)} = e^{\frac{\ln(n)}{\ln(2)}\ln(p)} = n^{\log_2(p)}$.
Ainsi,

$$u_n = O\left(n^{\log_2(p)}\right)$$

— <u>dans le cas où $p < 2^\gamma \Leftrightarrow \log_2(p) < \gamma$</u> alors

$$u_n = O\left(2^{\gamma \log_2(n)}\right)$$

et $2^{\gamma \log_2(n)} = e^{\gamma \frac{\ln(n)}{\ln(2)}\ln(2)} = e^{\gamma \ln(n)} = n^\gamma$.
Ainsi,

$$u_n = O\left(n^\gamma\right)$$

ii. Généralisons maintenant nos résultats à un entier $n \in \mathbb{N}^*$ quelconque :
Il existe un unique entier k ($k = \lfloor \log_2(n) \rfloor$) tel que

$$2^k \leq n < 2^{k+1}$$

Or, la suite u_n étant une complexité, elle est forcément croissante (preuve plus rigoureuse si dessous) , donc :

$$u_{2^k} \leq u_n < u_{2^{k+1}}$$

Ainsi, nous pouvons dominer la suite (u_n) par la suite (v_k) et ainsi étendre nos résultats) tout entier naturel non nul.

CQFD

iii. Prouvons que la suite (u_n) est croissante (optionnel) :
Rappelons que

$$\forall n \in \mathbb{N}^*, \quad u_n = pu_{\lfloor \frac{n}{2} \rfloor} + \lambda n^\gamma$$

donc que

$$\forall n \in \mathbb{N}^*, \quad u_{n+1} - u_n = p\left(u_{\lfloor \frac{n+1}{2} \rfloor} - u_{\lfloor \frac{n}{2} \rfloor}\right) + \lambda \underbrace{\left((n+1)^\gamma - n^\gamma\right)}_{\geq 0}$$

Nous allons donc faire une récurrence forte :
Notons pour tout $n \in \mathbb{N}$,

$$(\mathcal{H}_n) \ : \ \text{"}\forall k \in [|0;n|] \ \ u_k \leq u_{k+1}\text{"}$$

— $u_1 = pu_0 + \lambda 1^\gamma \geq u_0$ car $p \geq 1$ et $\lambda \geq 0$ donc $(\mathcal{H}_0)$ est vraie.

— Soit $n \geq 2$, supposons $(\mathcal{H}_{n-1})$,
$u_{n+1} - u_n = p\left(u_{\lfloor \frac{n+1}{2} \rfloor} - u_{\lfloor \frac{n}{2} \rfloor}\right) + \lambda \underbrace{\left((n+1)^\gamma - n^\gamma\right)}_{\geq 0}$,
or $\lfloor \frac{n+1}{2} \rfloor$ et $\lfloor \frac{n}{2} \rfloor$ appartiennent à $[|0;n-1|]$ donc $u_{\lfloor \frac{n+1}{2} \rfloor} \geq u_{\lfloor \frac{n}{2} \rfloor}$ donc $u_{n+1} - u_n \geq 0$ donc $(\mathcal{H}_n)$ est vraie.

— Par principe de récurrence, (u_n) est croissante.

Nous obtenons donc le théorème suivant :
Théorème 8 : (à connaître !)

Soit $p \in \mathbb{N}^*$, λ et $\gamma \in \mathbb{R}_+$, considérons la suite (u_n) définie par $\begin{cases} u_0 & \in \mathbb{R}_+ \\ \forall n \in \mathbb{N}^*, & u_n = pu_{\lfloor \frac{n}{2} \rfloor} + \lambda n \end{cases}$

— <u>1er cas :</u> si $\log_2(p) = \gamma$ alors

$$u_n = O\left(n^\gamma \log_2(n)\right)$$

— <u>2ème cas :</u> si $\log_2(p) > \gamma$ alors

$$u_n = O\left(n^{\log_2(p)}\right)$$

— <u>3ème cas :</u> si $\log_2(p) < \gamma$ alors

$$u_n = O\left(n^\gamma\right)$$

Nous pouvons généraliser ainsi le théorème (admis)

Théorème 9 : complexité des algorithmes utilisant le paradigme "diviser pour rég

Soient a et $b \in \mathbb{N}^*$, k un entier naturel ≥ 2 (on prend souvent $k = 2$) et $\gamma \in \mathbb{R}_+$, considérons la suite (u_n) définie par $\begin{cases} u_0 & \in \mathbb{R}_+ \\ \forall n \in \mathbb{N}^*, & u_n = au_{\lfloor \frac{n}{k} \rfloor} + bu_{\lceil \frac{n}{k} \rceil} + O\left(n^\gamma\right) \end{cases}$,

— <u>1er cas :</u> si $\log_k(a+b) = \gamma$ alors

$$u_n = O\left(n^\gamma \log(n)\right)$$

— <u>2ème cas :</u> si $\log_k(a+b) > \gamma$ alors

$$u_n = O\left(n^{\log_2(a+b)}\right)$$

— <u>3ème cas :</u> si $\log_k(a+b) < \gamma$ alors

$$u_n = O\left(n^\gamma\right)$$