Preuve et Complexité algorihmique

Christian CYRILLE

22 juin 2017

"Le contrôle de la complexité est l'essence de la programmation informatique"
Brian W. Kernighan

1 Qualités d'un algorithme

Un algorithme doit être :

1. **juste :**
   Il doit bien répondre au problème posé. La preuve d'un algorithme est complexe et ce n'est qu'à la fin des années 1970 que l'on s'est posé le problème de la justesse d'un algorithme.
2. **efficace :**
   Il doit se terminer en un minimum de temps , en utilisant le minimum de place en mémoire. On peut être obligé par moments à faire un compromis entre espace et temps.

2 Complexité

Un des objectifs du calcul de la complexité des algorithmes est la comparaison de plusieurs algorithmes résolvant le même problème.
Par exemple, les problèmes de tris, le calcul du PGCD de deux entiers naturels , le calcul des puissances entières positives d'un entier naturel.
On veut pouvoir dire : *"Sur toute machine, quel que soit le langage de programmation , l'algorithmeA_1 est meilleur que l'algorithme A_2 pour les données de grande taille"*. Il faut donc évaluer les ressources nécessaires pour réaliser l'algorithme : place mémoire et temps d'exécution. Un algorithme peut être préféré à un autre pour sa clarté et sa lisibilité même s'il est moins performant.
Certains algorithmes ne sont pas performants en général mais peuvent l'être pour certaines configurations de données(exemple l' algorithme de tri insertion sur des listes déjà presque triées).

2.1 2 critères de base pour la complexité

1. C_1 : **la complexité en espace** : quel est l'encombrement mémoire ? (Nombre de variables, nombre d'instructions, taille des variables,...)
2. C_2 : **la complexité en temps** : quelle est la rapidité d'exécution de l'algorithme ?

Dans les 2 cas, il faudrait tenir compte de plusieurs **paramètres** :

- P_1 : **la taille des données manipulées** . Lorsque les données algorithmiques sont de grande taille, on se préoccupe de la croissance de cette complexité en fonction de cette taille.
- P_2 : **la nature et l'organisation des données.**
- P_3 : **les contraintes de temps maximal d'exécution.**

2.2 La complexité en espace

Il s'agit d'évaluer l'encombrement mémoire .

2.2.1 Programme en Turbo Pascal sur la permutation du contenu de deux variables

```
procedure echange1 (var A : integer; var B : integer);

var AUX : integer;
begin
AUX := A ;
A := B;
B := AUX;
end;
procedure echange2 (var A : integer; var B : integer);
begin
A := A + B;
B := A - B;
A := A - B;
end;
```

La deuxième procédure n'utilise que 2 cases-mémoires alors que la première en utilise 3 dont une auxiliaire AUX.

Mais la première procédure est plus sûre informatiquement car elle ne réalise que des échanges entre cases-mémoires. Il n'y a aucun calcul alors que dans la seconde, on peut avoir de grandes surprises en particulier des erreurs d'arrondi en utilisant par exemple une variable A très grande et une variable B très petite.

2.2.2 Programme Maple sur la suite de Fibonacci

Ici la structure de tableau mobilise n cases-mémoires en plus de la case-mémoire k :

```
fibotable :=proc(n :: nonnegint)
           local k, f;
           f[0]:=0;f[1]:=1;
             for k from 2 to n do
                                  f[k] :=f[k-1] + f[k-2];
                           od;
           return(f[n]);
          end;
```

On peut simplifier cette complexité spatiale : au lieu d'utiliser k plus les n cases du tableau f on utilise uniquement 4 cases -mémoire : k, a, b, c

```
fibo :=proc(n :: nonnegint)
           local a, b, c , k ;
              if n = 0 or n = 1
                 then return(n)
                 else
                  a:=0 ;
                  b :=1;
                  for k from 2 to n
                   do
                     c:= a + b;
                     a:= b;
                     b:= c ;
                   od;
                  return(c)
           fi;
          end;
```

3 Complexité en temps

Un calcul numérique doit être conduit de manière à ce que les résultats intermédiaires aient des valeurs sympathiques c'est-à-dire pas trop grandes et, si possible, sans erreur d'arrondi.
On suppose que l'algorithme est exécuté de façon séquentielle sur une machine et que les opérations sont effectuées les unes après les autres en mémoire centrale. En conséquence, les transferts de données seront considérés comme ayant un coût négligeable par rapport aux opérations.
Pour déterminer la complexité d'un algorithme on détermine **le nombre d'opérations fondamentales ou élémentaires**. La nature de ces opérations élémentaires diffère selon le problème qui est résolu.

3.1 Exemples d'opérations fondamentales

1. Recherche d'un élément x dans un tableau T :
   Opération fondamentale : comparaison entre cet élément et les autres éléments du tableau.
2. Tri d'un tableau d'un élément x dans un tableau T :
   Opérations fondamentales : comparaison entre éléments et déplacements d'éléments.
3. Multiplication de deux matrices :
   Opération fondamentale : additions, multiplications.
4. Parcours d'un arbre binaire :
   Opération fondamentale : visite d'un noeud.

3.2 Exemples de calculs du nombre d'opérations élémentaires

1. coût total(séquence d'instructions $I_1; I_2; \cdots; I_n$) = coût(I_1) + coût (I_2) + $\cdots$ + coût(I_n)
2. - coût (pour k allant de a à b faire I(k)) =coût I(a) +coût I(a + 1) + $\cdots$ +coûtI(b)
   - coût (tant que $C(p)$ faire $(I(k); k := k + 1)$ = coûtC(p) + coût $I(p_1)$ + coût $I(p_2)$ + $\cdots$ +coût$I(p_n)$
3. - coût maximum (si C alors I_1 sinon I_2) = coût (C) + Max(coût (I_1) ; coût(I_2))
   - coût minimum (si C alors I_1 sinon I_2) = coût (C) + Min(coût (I_1) ; coût(I_2))

3.3 Complexité dans le meilleur des cas, dans le pire des cas, en moyenne

Dans un algorithme A soit un ensemble de données D_n de taille n. On note coût$_A(d)$ la complexité en temps de l'algorithme A agissant sur la donnée d'entrée d.

1. La complexité dans le pire des cas est $Max_A(n) = max(\text{coût}_A(d)/d \in D_n)$
2. La complexité dans le meilleur des cas est $Min_A(n) = min(\text{coût}_A(d)/d \in D_n)$
3. La complexité en moyenne est $Moy_A(n) = \sum_{d \in D_n} p(d)(\text{coût}_A(d)/d \in D_n)$ où $p(d)$ est la probabilité d'obtention de la donnée d en entrée de l'algorithme A.
4. $Min_A(n) \leq Moy_A(n) \leq Max_A(n)$

3.4 Trois Algorithmes de calcul du coefficient binomial

1. **Méthode 1** : $a = n!$; $b = p!$; $c = (n-p)!$; $d = b \times c$; $\binom{n}{p} = \frac{a}{d}$
2. **Méthode 2** : $a_1 = \frac{n}{p}$; $a_2 = \frac{a_1 \times (n-1)}{p-1}$; $a_3 = \frac{a_2 \times (n-2)}{p-2}$; $\cdots$
   jusqu'à obtenir $\binom{n}{p} = a_p = \frac{a_{p-1} \times (n-p+1)}{p}$
3. **Méthode 3** : $a_1 = \frac{n}{1}$; $a_2 = \frac{a_1 \times (n-1)}{2}$; $a_3 = \frac{a_2 \times (n-2)}{3}$; $\cdots$
   jusqu'à obtenir $\binom{n}{p} = a_p = \frac{a_{p-1} \times (n-p+1)}{p}$

1. Combien y-a-t-il de multiplications dans le calcul itératif de $n!$?
2. Calculer $\binom{9}{4}$ par les 3 méthodes. Pour chacune des 3 méthodes vous indiquerez le nombre total d'additions éventuelles, de soustractions éventuelles, de multiplications éventuelles , de divisions éventuelles.Expliquer pourquoi la méthode 3 est meilleure que les deux autres
3. Voici deux algorithmes. Indiquez pour chaque algorithme la méthode 1, 2 ou 3 qu'il représente.
   (a) **Début Algorithme 1**
   $A := 1$;$B := 0$; $P := p$;$N := n$;
   Répéter
   $A := A \times \frac{N-B}{B+1}$;
   $B := B + 1$;
   jusqu'à $B \geq P$
   Afficher A
   fin.
   (b) **Début Algorithme 2**
   $A := 1$;$P := p$; $N := n$;
   Pour $B := 1$ à P faire
   début
   $A := A \times \frac{N-B+1}{B}$;
   fin ;
   Afficher A
   fin.

3.4.1 Corrigé

Voici trois méthodes systématiques (programmables) de calcul de $\binom{n}{p}$

1. **Méthode 1** : $a = n!$; $b = p!$; $c = (n-p)!$; $d = b \times c$; $\binom{n}{p} = \frac{a}{d}$
2. **Méthode 2** : $a_1 = \frac{n}{p}$; $a_2 = \frac{a_1 \times (n-1)}{p-1}$; $a_3 = \frac{a_2 \times (n-2)}{p-2}$; $\cdots$
   jusqu'à obtenir $\binom{n}{p} = a_p = \frac{a_{p-1} \times (n-p+1)}{p}$
3. **Méthode 3** : $a_1 = \frac{n}{1}$; $a_2 = \frac{a_1 \times (n-1)}{2}$; $a_3 = \frac{a_2 \times (n-2)}{3}$; $\cdots$
   jusqu'à obtenir $\binom{n}{p} = a_p = \frac{a_{p-1} \times (n-p+1)}{p}$

1. $n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$ pour $n \geq 2$.
2. En prenant en considération la multiplication initiale par 1 il y a exactement $n-1$ multiplications dans ce calcul itératif de $n!$ pour $n \geq 2$.
3. Calculer $\binom{9}{4}$ par les 3 méthodes. Pour chacune des 3 méthodes nous indiquerons le nombre d'additions éventuelles, de soustractions éventuelles, de multiplications éventuelles , de divisions éventuelles puis le nombre total d'opérations éventuelles et expliquerons pourquoi la méthode 3 est meilleure que les deux autres
   (a) Méthode 1 :

	$*$	$/$	$+$	$-$	*Total*
$a = 9!$	8				8
$b = 4!$	3				3
$c = (9-4)! = 5!$	4			1	5
$d = b \times c$	1				1
$\frac{a}{d}$		1			1
Total	16	1		1	18

   En tout 18 opérations élémentaires.

(b) Méthode 2 :

	$*$	$/$	$+$	$-$	*Total*
$a_1 = \frac{9}{4}$		1			1
$a_2 = \frac{a_1 \times (9-1)}{(4-1)} = \frac{9}{4} \times \frac{8}{3}$	1	1		2	4
$a_3 = \frac{a_2 \times (9-2)}{(4-2)} = \frac{9}{4} \times \frac{8}{3} \times \frac{7}{2}$	1	1		2	4
$a_4 = \frac{a_3 \times (9-4+1)}{(4-3)} = \frac{9}{4} \times \frac{8}{3} \times \frac{7}{2} \times \frac{6}{1}$	1	1	1	2	5
Total	3	4	1	6	14

En tout 14 opérations élémentaires.

(c) Méthode 3 :

	$*$	$/$	$+$	$-$	*Total*
$a_1 = \frac{9}{1}$		1			1
$a_2 = \frac{a_1 \times (9-1)}{2} = \frac{9}{1} \times \frac{8}{2}$	1	1		1	3
$a_3 = \frac{a_2 \times (9-2)}{3} = \frac{9}{1} \times \frac{8}{2} \times \frac{7}{3}$	1	1		1	3
$a_4 = \frac{a_3 \times (9-4+1)}{4} = \frac{9}{1} \times \frac{8}{2} \times \frac{7}{3} \times \frac{6}{4}$	1	1	1	1	4
Total	3	4	1	3	11

En tout 11 opérations élémentaires. Elle comporte moins d'opérations élémentaires que les deux autres donc elle est la meilleure.

4. Voici deux algorithmes.

   (a) **Début Algorithme 1**
   $A := 1$;$B := 0$; $P := p$;$N := n$;
   Répéter
   $A := A \times \dfrac{N - B}{B + 1}$;
   $B := B + 1$;
   jusqu'à $B \geq P$
   Afficher A
   fin.

A	B	P	N	*Ecran*
1	0	4	9	
$\frac{9}{1}$				
	1			
$\frac{9}{1} \times \frac{8}{2}$				
	2			
$\frac{9}{1} \times \frac{8}{2} \times \frac{7}{3}$				
	3			
$\frac{9}{1} \times \frac{8}{2} \times \frac{7}{3} \times \frac{6}{4}$				
	4			
				$\frac{9}{1} \times \frac{8}{2} \times \frac{7}{3} \times \frac{6}{4}$

   Cet algorithme représente la méthode 3.

(b) **Début Algorithme 2**
$A := 1$;$P := p$; $N := n$;
Pour $B := 1$ à P faire
début
$A := A \times \frac{N - B + 1}{B}$;
fin ;
Afficher A
fin.

A	B	P	N	*Ecran*
1		4	9	
	1			
$1 \times \frac{9}{1}$				
	2			
$1 \times \frac{9}{1} \times \frac{8}{2}$				
	3			
$1 \times \frac{9}{1} \times \frac{8}{2} \times \frac{7}{3}$				
	4			
$1 \times \frac{9}{1} \times \frac{8}{2} \times \frac{7}{3} \times \frac{6}{4}$				
				$1 \times \frac{9}{1} \times \frac{8}{2} \times \frac{7}{3} \times \frac{6}{4}$

Cet algorithme représente aussi la méthode 3.

3.5 Terminaison et complexité de l'algorithme d'Euclide

Si a est un entier naturel et b un entier naturel non nul alors il existe un couple unique (q, r) d'entiers naturels tels que $a = bq + r$ avec $0 \leq r < b$.
q s'appelle le quotient de la division euclidienne de a par b
r s'appelle le reste de la division euclidienne de a par b.
Comme $a = bq + r$ alors tout diviseur commun à a et b est un diviseur commun à a , à b et à bq donc est un diviseur commun à b et à $a - bq$ donc est un diviseur commun à b et à r.
Réciproquement, tout diviseur commun à b et à r est un diviseur commun à b, à bq et à r donc est un diviseur commun à b et à $bq + r$ c'est-à-dire un diviseur commun à a et à b
Théorème : l'ensemble des diviseurs communs à a et à b est l'ensemble des diviseurs communs à b et à r.

Le calcul du pgcd $a \wedge b$ de 2 entiers a et b ($a > b > 0$) par l'algorithme d'Euclide se fait par divisions successives.
Exemple : $a = 44$; $b = 18$

$$44 = 2 \times 18 + 8$$
$$18 = 2 \times 8 + 2$$
$$8 = 4 \times 2 + 0$$
$$44 \wedge 18 = 2$$

Si on désigne par $l(a, b)$ la longueur de l'algorithme, c'est-à-dire le nombre de divisions nécessaires pour aboutir au résultat, nous avons ici :

$$l(44, 18) = 3$$

L'objet de cet exercice est de majorer $l(a, b)$. Pour cela, on note

$$r(1), r(2), r(3), \cdots, r(n)$$

les restes des n divisions successives . On a donc $l(a, b) = n$ et $r(n) = 0$.
On note par convention $a = r(-1)$ et $b = r(0)$.

1. Montrer que l'algorithme d'Euclide se termine.
2. Montrer que pour $1 \leq k \leq n - 1$, on a $r(k - 2) \geq r(k - 1) + r(k)$
3. Soit $(F(n))$ la suite de Fibonacci définie par :

   $$F(0) = F(1) = 1$$
   $$F(n) = F(n - 2) + F(n - 1) \ pour \ n \geq 2$$

   Montrer qu'en posant $\alpha = \dfrac{1 + \sqrt{5}}{2}$, on a $\alpha^{n-1} \leq F(n)$
4. Montrer que $\forall k \in \mathbb{N}$, on a
   $$r(n - k - 1) \geq F(k)$$
   et en déduire que n vérifie une majoration de la forme :
   $$n \leq A \ln(a) + B$$
5. Vérifier cette majoration sur l'exemple $a = 44$ et $b = 18$. On pourra prendre $B = 1$ et $A < 2,1$

3.5.1 Algorithme d'Euclide de recherche du PGCD - Capes interne Maths 2000

1. Le calcul du pgcd $a \wedge b$ de 2 entiers a et b ($a > b > 0$) par l'algorithme d'Euclide se fait par divisions successives.
   Exemple : $a = 44$; $b = 18$

   $$44 = 2 \times 18 + 8$$
   $$18 = 2 \times 8 + 2$$
   $$8 = 4 \times 2 + 0$$

   donc le $pgcd(44; 22) = 44 \wedge 18 = 2$ = le dernier reste non nul = 2.
   Si on désigne par $l(a, b)$ la longueur de l'algorithme, c'est-à-dire le nombre de divisions nécessaires pour aboutir au résultat, nous avons ici :

   $$l(44, 18) = 3$$

2. Posons $n = L(a; b)$. On note $r(1), r(2), \cdots, r(n)$ la suite des restes des n divisions successives.
   Donc $r(n) = 0$. Par convention, on pose $r(-1) = a$ et $r(0) = b$.

3. L'algorithme d'Euclide se termine car :

   $$\begin{cases} a = r(-1) = q(0)\, b + r(1) = q(0)r(0) + r(1)\ avec\ 0 \leq r(1) < r(0) \\ b = r(0) = q(1)r(1) + r(2)\ avec\ 0 \leq r(2) < r(1) \\ r(1) = q(2)r(2) + r(3)\ avec\ 0 \leq r(3) < r(2) \\ \cdots \\ r(n-3) = q(n-2)r(n-2) + r(n-1))\ avec\ 0 \leq r(n-1) < r(n-2) \\ r(n-2) = q(n-1)r(n-1) + r(n)\ avec\ 0 = r(n) < r(n-1) \end{cases}$$

   La suite des restes $r(1), r(2), \ldots\ldots\ldots, r(n)$ est une suite strictement décroissante d'entiers naturels donc au bout d'un nombre n fini d'opérations on aura $r(n) = 0$.
   Par conséquent **l'algorithme d'Euclide se termine**. **De plus, cet algorithme est valide** car le dernier reste non nul obtenu $r(n-1)$ est bien le $pgcd(a, b)$:
   - En effet, soit d un diviseur commun à a et à b alors il divise $r(1)$ car $r(1) = a - b\, q(0)$; de même puisqu'il divise b et $r(1)$ il divisera $r(2)$ car $r(2) = b - r(1)\ q(1)$ et ainsi de suite, il finira par diviser $r(n-1)$. Par conséquent tout diviseur de a et de b divise donc $r(n-1)$.
   - Réciproquement, on démontre de même, qu'un diviseur de $r(n-1)$ divise $r(n-2)$ (à cause de la dernière division) ; s'il divise $r(n-1)$ et $r(n-2)$ ill divisera $r(n-3)$ à cause de l'avant-dernière division , etc ..., il divisera donc b et donc a. Par conséquent, le plus grand diviseur de $r(n-1)$ est le plus grand commun diviseur de a et de b. Or le plus grand diviseur de $r(n-1)$ est $r(n-1)$ lui-même.
   - Ainsi $pgcd(a, b) = r(n-1)$

4. Soit Fib la suite de Fibonacci définie par $Fib(0) = 1$; $Fib(1) = 1$ et $Fib[n] = Fib[n-1] + Fib[n-2]$ pour tout entier $n \geq 2$

   (a) On pose Φ = le nombre d'Or = $\dfrac{1+\sqrt{5}}{2}$ alors $\forall n \in \mathbb{N}\ \Phi^{n-1} \leq Fib(n)$.
   Démontrons-le :
   Comme Fib est une suite doublement récurrente, on considère son équation caractéristique $q^2 = q + 1$ qui a pour solutions Φ et et $\Psi = \dfrac{-1}{\Phi} = \dfrac{1-\sqrt{5}}{2}$.

alors pour tout entier naturel n l'on a $Fib(n) = a\Phi^n + b\Psi^n$. Comme $Fib(0) = 1$ et $Fib(1) = 1$ alors a et b vérifient le système

$$\begin{cases} 1 = a\Phi^0 + b\Psi^0 \\ 1 = a\Phi^1 + b\Psi^1 \end{cases}$$

donc

$$\begin{cases} 1 = a + b \\ 1 = a\Phi + b\Psi \end{cases}$$

d'où

$$\begin{cases} b = 1 - a+ \\ 1 = a\Phi + (1-a)\Psi \end{cases}$$

On a donc

$$\begin{cases} b = 1 - a \\ a = \dfrac{1-\Psi}{\Phi + \Psi} \end{cases}$$

d'où

$$\begin{cases} b = 1 - a \\ a = \dfrac{1-\Psi}{\sqrt{5}} \end{cases}$$

On en tire que

$$\begin{cases} b = \dfrac{\Phi - 1}{\sqrt{5}} \\ a = \dfrac{1-\Psi}{\sqrt{5}} \end{cases}$$

Par récurrence, l'on prouve alors que $\forall n \in \mathbb{N}\ \Phi^{n-1} \leq Fib(n)$**.**

- Etape 1 : Cette propriété est vraie pour $n = 0$
  car $\Phi^{0-1} = \dfrac{1}{\phi} = \dfrac{2}{1+\sqrt{5}} \leq Fib(0) = 1$
  en effet $\sqrt{5} \approx 1,623$ donc $1 + \sqrt{5} \approx 2,623$ donc $\dfrac{2}{1+\sqrt{5}} \leq 1$
- Etape 2 : soit n un entier naturel fixé, supposons que pour tout $k \leq n$ l'on a $\Phi^{n-1} \leq Fib(n)$, démontrons qu'alors $\Phi^n \leq Fib(n+1)$
  Comme Φ vérifie l'équation caractéristique $\Phi^2 = \Phi + 1$ donc $\Phi^2\Phi^{n-2} = \Phi\Phi^{n-2} + 1\Phi^{n-2}$ donc $\Phi^n = \Phi^{n-1} + \Phi^{n-2} \leq Fib(n) + Fib(n-1)$ donc $\Phi^n \leq Fib(n+1)$.
- Etape 3 : d'après étape 1 et étape 2, on peut conclure que $\forall n \in \mathbb{N}\Phi^{n-1} \leq Fib(n)$

(b) **Démontrons par récurrence sur** k **que** $r(n-k-1) \geq Fib(k)$

- Etape 1 : Cette propriété est vraie pour $k = 0$ car $r(n-1) \geq Fib(0) = 1$
  en effet $r(n-1)$ est le dernier reste non nul donc $r(n-1) \geq 1$
- Etape 2 : supposons que pour $r(n-1) \geq Fib(0)$ et $r(n-2) \geq Fib(1)$ et $\cdots$ et $r(n-k-1) \geq Fib(k)$
  Démontrons qu'alors $r(n-k-2) \geq Fib(k+1)$
  Or $r(n-k-2) = q(n-k-1)r(n-k-1) + r(n-k)$ avec $0 \leq r(n-k) < r(n-k-1)$
  donc $r(n-k-2) = q(n-k-1)r(n-k-1) + r(n-k) \geq q(n-k-1)Fib(k) + Fib(k-1)$
  Comme les quotients ne peuvent s'annuler car alors 2 restes seraient identiques

ce qui ne se peut car la suite des restes est strictement décroissante alors $q(n-k-1) \geq 1$
Donc $r(n-k-2) \geq Fib(k) + Fib(k-1)$ donc $r(n-k-2) \geq Fib(k+1)$

- Etape 3 : d'après étape 1 et étape 2, on peut conclure que $\forall k \in \mathbb{N}\ r(n-k-1) \geq Fib(k)$

(c) On en déduit que $n \leq Aln(a) + B$.
Comme $\forall k \in \mathbb{N}\ r(n-k-1) \geq Fib(k)$ donc pour $k = n$ l'on a $r(-1) \geq Fib(n)$ donc $a \geq Fib(n)$ Or $Fib(n) \geq \Phi^{n-1}$ donc $a \geq \Phi^{n-1}$ donc $ln(a) \geq (n-1)ln(\Phi)$ donc $n - 1 \leq \frac{1}{ln(\phi)} ln(a)$
Donc $n \leq 1 + \frac{1}{ln(\phi)} ln(a)$
Par exemple pour $a = 44$ on a théoriquement $n \leq 1 + \frac{1}{ln(\phi)} ln(44) \approx 8,86$ alors qu'en pratique $n = 3$

$$L(a;b) \leq Aln(a) + B$$

donc la complexité en temps de cet algorithme au pire est en $O(ln(a))$

3.6 Complexité du calcul de la valeur d'un polynôme en une valeur

> On dit que deux fonctions f et g de $\mathbb{N}^*$ dans $\mathbb{R}^+$ ont même ordre de grandeur asymptotique (ce que l'on note $f(n) = \Theta(g(n))$ lorsqu'il existe un réel $C > 0$ et un réel $D > 0$ ainsi qu'un entier n_0 tel que pour tout entier $n \geq n_0$ l'on ait
>
> $$Cg(n) \leq f(n) \leq Dg(n)$$

3.6.1 Méthode classique

Soit le polynôme $P(x) = 3x^4 - 5x^3 + 6x^2 - 2x + 4$
Pour calculer $P(a)$ il faut normalement 4 additions et 10 multiplications.
Pour le polynôme de degré n : $P(x) = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1} + a_nx^n$ le coût maximum du calcul de $P(a)$ en opérations élémentaires(additions et multiplications) est de :
n additions et de $1 + 2 + \cdots . + n$ c'est-à-dire $\dfrac{n(n+1)}{2}$ multiplications.
Le coût total maximum est $MAX_1(n) = n + \dfrac{n(n+1)}{2} = \dfrac{n^2+3n}{2}$.
$\lim_{n\mapsto+\infty} MAX_1(n) = \lim_{n\mapsto+\infty} n + \dfrac{n(n+1)}{2} = \lim\limits_{n\mapsto+\infty} \dfrac{n^2}{2}$
De plus dès que $n \geq 1$ on a $n \leq n^2$ donc $0 \leq \dfrac{3n}{2} \leq \dfrac{3n^2}{2}$
donc $0 + \dfrac{n^2}{2} \leq \dfrac{3n+n^2}{2} \leq \dfrac{3n^2}{2} + \dfrac{n^2}{2}$
d'où $\dfrac{1}{2}n^2 \leq MAX_1(n) \leq 2n^2$ donc $MAX_1(n) = \Theta(n^2)$.
Cet algorithme classique de calcul de $P(a)$ a pour ordre de grandeur n^2.

3.6.2 Méthode de Horner

HORNER (mathématicien et physicien anglais 1786-1837) a mis en forme une méthode proposée 150 ans plus tôt par Newton Il propose l'algorithme suivant :
$3x^4 - 5x^3 + 6x^2 - 2x + 4 = (((3x - 5)x + 6)x - 2)x + 4$
On se retrouve avec toujours 4 additions mais seulement 4 multiplications.
De façon générale, $a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1} + a_nx^n = (((((a_nx + a_{n-1})x + a_{n-2})x + \ldots\ldots + a_1)x + a_0$
L'algorithme de Horner coûte au maximum n additions et n multiplications donc le coût total maximum est $MAX_2(n) = n + n = 2n$
$\lim_{n\mapsto+\infty} MAX_2(n) = \lim_{n\mapsto+\infty} 2n$
Pour tout entier naturel n, l'on a $n \leq 2n \leq 3n$ donc $MAX_2(n) = \Theta(n)$
donc **l'algorithme de Horner de calcul de $P(a)$ a pour ordre de grandeur n.**

Il est plus performant donc que l'algorithme classique.
Reprenons l'exemple $P(x) = 3x^4 - 5x^3 + 6x^2 - 2x + 4$ ici $a_4 = 3$; $a_3 = -5$; $a_2 = 6$; $a_1 = -2$; $a_0 = 4$ d'après Horner $P(x_0) = (((3x_0 - 5)x_0 + 6)x_0 - 2)x_0 + 4$ On pose :

- $h_4 = a_4$
- $h_3 = h_4 x_0 + a_3$
- $h_2 = h_3 x_0 + a_2$
- $h_1 = h_2 x_0 + a_1$
- $h_0 = h_1 x_0 + a_0$

donc $P(x_0) = h_0$
On peut disposer les calculs selon le schéma suivant appelé shéma de Horner

a_4	a_3	a_2	a_1	a_0
	$+x_0h_4$	$+x_0h_3$	$+x_0h_2$	$+x_0h_1$
$= h_4$	$= h_3$	$= h_2$	$= h_1$	$= h_0 = P(x_0)$

Pour notre exemple, prenons $x_0 = 2$

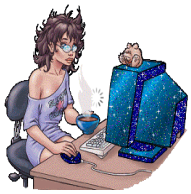
3	-5	6	-2	4
	$+2 \times 3$	$+2 \times 1$	$+2 \times 8$	$+2 \times 14$
$= 3$	$= 1$	$= 8$	$= 14$	$= 32 = P(2)$

1. **Théorème de Horner**
   $P(x) = (x - x_0)(h_4x^3 + h_3x^2 + h_2x + h_1) + h_0$
2. **Corollaire** Si x_0 est une racine de $P(x)$ c'est-à-dire que $P(x_0) = 0 = h_0$
   alors $P(x) = (x - x_0)(h_4x^3 + h_3x^2 + h_2x + h_1)$

3.6.3 Algorithme de calcul classique de $P(x)$

```
debut
              lire(degre);
              lire(X);
              afficher(''Entrez le coefficient P[0] =');
              lire(P[0]);
              valeurpolynome := P[0];
              si degre <>0
              alors
                   pour I variant de (degre - 1) à 0
                      faire
                            debut
                                afficher('Entrez le coefficient P[', I, '] =');
                                lire(P[I]);
                                produit := 1;
                                pour J variant de 1 à I
                                   faire
                                      début
                                           produit := produit *X;
                                  finpour;
                                  valeurpolynome := valeurpolynome +P[I] * produit;
                           finpour
               finsi
               afficher(valeurpolynome)
fin.
```

Codage de l'algorithme en Turbo-Pascal :



```
program QUATORZE;
     uses WinCrt;
     const nmax= 6;
     type tableau = array[1..NMAX] of word;
     var     n, i ,x : word;
          P : real;
              a : tableau;

     begin (* programme principal *)
         clrscr;
         (* entrée des données *)
         write('Veuillez entrer le degré du polynôme);
         readln(n);
          write('Veuillez entrer un réel strictement positif x = ');
         readln(x);
         writeln('Veuillez entrer les coefficients du polynôme du tableau :  ');
         for i := 0 to n do
             begin
                  write('Entrez A[',i,'] =');
                  readln(A[i]);
             end;
         (* traitement *)
         P :=A[0];
         for i := 1 to n do
             begin
                  P := P  + a[I) * exp(I * ln(x));
             end;
         (* affichage des résultats *)
          writeln('P[ ', x , '] = ', P :10 : 2);
     end.
```

3.6.4 Algorithme de calcul de $P(x)$ par la méthode de Horner

```
debut
           lire(degre);
           lire(X);
           afficher('Quel est le coefficient dominant ?');
           lire(P[degre]);
           valeurpolynome := P[degre];
           pour I variant de (degre - 1) à 0
           faire
                  debut
                     afficher('Entrez le coefficient P[', I, '] =');
                     lire(P[I]);
                     valeurpolynome := P[I] + X * valeurpolynome;
            finpour
          afficher(valeurpolynome)
    fin.
```

Codage de l'algorithme en Turbo-Pascal :

```
program QUINZE;
     uses WinCrt;
     const NMAX = 6;
     type tableau = array[1..NMAX] of word;
     var     n, I ,x : word;
          P : real;
              A : tableau;

     begin (* programme principal *)
         clrscr;
         (* entrée des données *)
         write('Veuillez entrer le degré du polynôme);
         readln(n);
          write('Veuillez entrer le réel x = ');
         readln(x);
         writeln('Veuillez entrer les coefficients du polynôme du tableau :  ');
         for I := 1 to n do
             begin
                  write('Entrez A[',I,'] =');
                  readln(A[I]);
             end;
         (* traitement *)
         P :=A[N];
         for I := 1 to n do
             begin
                  P := P * x + a[n - I);
             end;
         (* affichage des résultats *)
          writeln('P[ ', x , '] = ', P :10 : 2);
     end.
```

3.7 Complexité du pivot de Gauss - Centrale MP 2008

Soit un entier $n \geq 2$. Soit $\mathcal{M}_n$ l'ensemble des matrices carrées à n lignes, à coefficients réels. On note $\mathcal{TS}_n \subset \mathcal{M}_n$ l'ensemble des matrices triangulaires supérieures
Soit la matrice $A = (a_{ij}) \in \mathcal{TS}_n$.
On considère le système linéaire (1) : $Au = w$ où $w =^t (w_1, w_2, \cdots, w_n)$ est donné et $u =^t (u_1, u_2, \cdots, u_n)$ est l'inconnue.
On suppose que $det(A) \neq 0$ donc (1) admet une unique solution $u =^t (u_1, u_2, \cdots, u_n) \in \mathbb{R}^n$.

1. Calculer u_n puis pour tout $k \in [|1; n-1|]$ exprimer u_{n-k} en fonction de $u_n, u_{n-1}, \cdots, u_{n-k+1}$
   Travaillons d'abord sur un exemple $n = 3$.

$$\begin{cases} a[1,1]u_1 + a[1,2]u_2 + a[1,3]u_3 = w_1 \\ a[2,2]u_2 + a[2,3]u_3 = w_2 \\ a[3,3]u_3 = w_3 \end{cases}$$

$$\iff \begin{cases} a[1,1]u_1 + a[1,2]u_2 + a[1,3]u_3 = w_1 \\ a[2,2]u_2 + a[2,3]u_3 = w_2 \\ u_3 = \dfrac{1}{a[3,3]} w_3 \end{cases}$$

$$\iff \begin{cases} a[1,1]u_1 + a[1,2]u_2 + a[1,3]u_3 = w_1 \\ a[2,2]u_2 = w_2 - a[2,3]u_3 \\ u_3 = \dfrac{1}{a[3,3]} w_3 \end{cases}$$

$$\iff \begin{cases} a[1,1]u_1 + a[1,2]u_2 + a[1,3]u_3 = w_1 \\ u_2 = \dfrac{1}{a[2,2]}(w_2 - a[2,3]u_3) \\ u_3 = \dfrac{1}{a[3,3]} w_3 \end{cases}$$

$$\iff \begin{cases} a[1,1]u_1 = w_1 - a[1,2]u_2 - a[1,3]u_3 \\ u_2 = \dfrac{1}{a[2,2]}(w_2 - a[2,3]u_3) \\ u_3 = \dfrac{1}{a[3,3]} w_3 \end{cases}$$

$$\iff \begin{cases} u_1 = \dfrac{1}{a[1,1]}(w_1 - a[1,2]u_2 - a[1,3]u_3) \\ u_2 = \dfrac{1}{a[2,2]}(w_2 - a[2,3]u_3) \\ u_3 = \dfrac{1}{a[3,3]} w_3 \end{cases}$$

- Pour $u[3]$ on a 1 division, 0 soustractions et 0 multiplications
- Pour $u[2]$ on a 1 division, 1 soustractions et 1 multiplications

- Pour $u[1]$ on a 1 division, 2 soustractions et 2 multiplications
- donc en tout on a 3 divisions, $0+1+2$ soustractions et $0+1+2$ multiplications

On peut donc généraliser pour n quelconque.
Les solutions du système sont :

$$\begin{cases} u_n = \dfrac{1}{a[n,n]} w_n \\ \forall k \in [|1; n-1|]\ u_{n-k} = \dfrac{1}{a[n-k,n-k]} (w_{n-k} - \displaystyle\sum_{i=n-k+1}^{n} a[n-k,i]u_i) \end{cases}$$

2. On peut écrire ceci sous forme d'un algorithme de résolution codé en Maple

```
   n:=3;
w[1] := 44;w[2] := 26;w[3] := 30;
a[1,1] :=1;a[1,2] := 3; a[1,3] := 5; a[2,2] := 2 ; a[2,3] := 3;a[3,3] := 5;
    u[n]= (1/a[n,n])*w[n];
 for k from 1 to n - 1
 do
  u[n - k] =(1/a[n-k,n - k])[w[n-k] - sum(a[n - k, i] * u[i], i = n - k + 1..n)];
 print(u[n-k]);
od;
```

Ce programme affiche $u[1] = 2; u[2] = 4; u[3] = 6$ On a bien $\begin{cases} 1 \times 2 + 3 \times 4 + 5 \times 6 = 44 \\ 2 \times 4 + 3 \times 6 = 26 \\ 5 \times 6 = 30 \end{cases}$

3. Algorithme en pseudo-code

```
  début
   (* entrée des données *)
 lire au clavier n;
pour i variant de 1 à n
           lire au clavier w[i]
fin pour
pour i variant de 1 à n
           pour j variant de 1 à n
                    lire au clavier a[i , j]
           fin pour
 fin pour
(* traitement et affichage des résultats *)
    u[n] reçoit  (1/a[n , n])*w[n];
  afficher( u[n])
     Pour k variant de 1 à  n - 1
           S reçoit 0
           pour i variant de (n - k + 1) à n
                  S reçoit S + a[n - k , i]*u[i]
           fin pour
           u[n - k] =(1/a[n - k , n - k])*[w[n-k] - S];
           afficher(u[n-k]);
      fin pour;
 fin.
```

4.
   - Pour $u[n]$ on a 1 division, 0 soustractions et 0 multiplications
   - Pour $u[n-1]$ on a 1 division, 1 soustractions et 1 multiplications
   - Pour $u[n-2]$ on a 1 division, 2 soustractions et 2 multiplications
   - $\cdots$
   - Pour $u[1]$ on a 1 division,$n-1$ soustractions et $n-1$ multiplications
   - donc en tout on a :
     n divisions, $0+1+2+\cdots+(n-1) = \dfrac{(n-1)n}{2}$ soustractions et $0+1+2+\cdots+(n-1) = \dfrac{(n-1)n}{2}$ multiplications

   Le coût total de l'algorithme du pivot de Gauss est donc :
   $n + 2\dfrac{(n-1)n}{2} = n + n^2 - n = n^2$.
   Cet algorithme est donc un algorithme dont l'ordre de grandeur asymptotique est n^2.

3.8 Complexité d'un algorithme - Edhec 1996

Partie 1

On considère f la fonction numérique d'une variable réelle définie par :
$f(x) = x - \ln(x)$

1. Etudier f et résumer cette étude par un tableau de variations.
2. Etudier le signe de $f(x) - x$ pour $x > 0$

Partie 2

On considère l'algorithme suivant :

```
program iter;
var   n , k : integer ;
      a , u , p : real ;
 function f(x : real): real;
      begin
            if x > 0 then f := x - ln(x) ;
      end;
 begin
   (* entrée des données *)
        readln(n);
        readln(a);

   (* traitement *)
       u := a ;
       p := a;
       for k := 1 to n do     begin
                                   u := f(u) ;
                                   p:= p* u;
                               end;

   (* affichage des résultats *)
       writeln(u);
       writeln(p);
 end.
```

Dans le cas particulier où $n = 3$ et $a = 2$

1. Donner les écritures des valeurs exactes des contenus de la variable u à chaque tour de boucle et à la fin de l'algorithme.
2. On donne $2 - \ln(2) \approx 1,306$
   $2 - \ln(2) - \ln(2 - \ln(2)) \approx 1,039$
   $2,612 * 1,039 \approx 2,714$
   $2 - \ln(2) - \ln(2 - \ln(2)) - \ln(2 - \ln(2) - \ln(2 - \ln(2))) \approx 1,00074$
   $2,715 * 1,00074 \approx 2,716$
   Donner les valeurs approchées des contenus successifs de la variable p à chaque tour de boucle et à la fin de l'algorithme. Que peut-on conjecturer pour cette variable p ?

Partie 3

Dorénavant, dans le cas général, on note u_n et p_n les contenus respectifs des variables u et p à la fin de l'algorithme lorsque leur calcul est possible.

1. Pour quelles valeurs de a peut-on définir cette suite (u_n) de premier terme $u_0 = a$ et dont le terme général est calculé par l'algorithme précédent ?
2. Pour les valeurs de a trouvées ci-dessus, donner en fonction de n :
   - le nombre total d'appels de la fonction f utilisée dans l'algorithme
   - le nombre total de soustractions nécessaires aux calculs de u et de p
   - le nombre total de multiplications nécessaires aux calculs de u et de p
   - le nombre total d'affectations nécessaires aux calculs de u et de p. a

   NB - On admettra que le symbole := utilisé dans l'écriture $for\ k := 1\ to\ n$ ne sera pas considéré comme une affectation mais que chaque appel de fonction nécessite une affectation $(f :=)$ et une soustraction.
3. Lorsque la suite (u_n) est bien définie, écrire , pour tout entier naturel n, la relation liant u_{n+1} et u_n.
4. Démontrer que si $a = 1$ alors la suite (u_n) est constante.
5. Démontrer que si la suite (u_n) est constante alors $a = 1$
6. On suppose dans cette question que $a > 1$.
   (a) Montrer que $\forall n \in \mathbb{N}$, l'on a : $u_n > 1$
   (b) Etudier les variations de la suite (u_n)
   (c) En déduire que la suite (u_n) converge puis déterminer sa limite.
7. On suppose dans cette question que $0 < a < 1$.
   (a) Montrer que $\forall n \in \mathbb{N}^*$, l'on a : $u_n > 1$
   (b) Etudier les variations de la suite (u_n) sur $\mathbb{N}^*$
   (c) En déduire que la suite (u_n) converge puis déterminer sa limite.

Partie 4

1. Démontrer par récurrence que $\forall n \in \mathbb{N}^*\ p_n = au_1u_2 \cdots u_n$
2. En considérant $\ln(p_n)$, montrer que $\forall n \in \mathbb{N}^*\ p_n = e^{a-u_{n+1}}$
3. En déduire $\lim\limits_{n\to+\infty} p_n$
4. Peut-on alors justifier la conjecture de la question 2 de la partie 2 lorsque a valait 2 ?
5. Ecrire alors un nouvel algorithme en Turbo-Pascal permettant le calcul de p_n.
   Cet algorithme ne doit contenir aucune multiplication.
   On rappelle que la fonction exponentielle notée exp est prédéfini dans le langage Turbo-Pascal.

3.8.1 Corrigé

Dans le cas particulier où $n = 3$ et $a = 2$

1. **Donner les écritures des valeurs exactes des contenus de la variable u à chaque tour de boucle et à la fin de l'algorithme.**

contenus successifs des cases	k	n	a	u	p
entree des donnees		3			
			2		
avant entree boucle				2	
					2
tour de boucle	1			2 - ln(2)	
					2 * (2 - ln(2))
tour de boucle	2			2 - ln(2) - ln(2 - ln(2))	
					2*(2 - ln(2) * (2 - ln(2) - ln(2 - ln(2)))
tour de boucle	3			2 - ln(2) - ln(2 - ln(2))) - ln(2 - ln(2) - ln(2 - ln(2))))	
					[2*(2 - ln(2) * (2 - ln(2) - ln(2 - ln(2)))] * [2 - ln(2) - ln(2 - ln(2))) - ln(2 - ln(2) - ln(2 - ln(2))))]
sortie de boucle				affichage du contenu de cette case	
					affichage du contenu de cette case

2. **On donne** $2 - \ln(2) \approx 1,306$ **;** $2 - \ln(2) - \ln(2 - \ln(2)) \approx 1,039$ **;** $2,612 * 1,039 \approx 2,714$ **;** $2 - \ln(2) - \ln(2 - \ln(2)) - \ln(2 - \ln(2) - \ln(2 - \ln(2))) \approx 1,00074$ **;** $2,715 * 1,00074 \approx 2,716$ **; Donner les valeurs approchées des contenus successifs de la variable p à chaque tour de boucle et à la fin de l'algorithme. Que peut-on conjecturer pour cette variable p ?**

valeur approchee de	u	p
	2	2
$k = 1$	1,306852819	2,613705639
$k = 2$	1,039231001	2,716243928
$k = 3$	1,000749983	2,718281065

On peut donc conjecturer que la suite (p_n) semble converger vers $e \approx 2,718$

Partie 3

Dorénavant, dans le cas général, on note u_n et p_n les contenus respectifs des variables u et p à la fin de l'algorithme lorsque leur calcul est possible.

1. **Pour quelles valeurs de a peut-on définir cette suite (u_n) de premier terme $u_0 = a$ et dont le terme général est calculé par l'algorithme précédent ?** A condition de prendre $a > 0$ Le programme simule l'affichage du nième terme des deux suites :
   - (u_n) définie par $u_0 = a$ et $\forall n \in \mathbb{N}\ u_{n+1} = f(u_n)$
   - (p_n) définie par $p_0 = a$ et $\forall n \in \mathbb{N}\ p_{n+1} = u_n p_n$
2. **NB - On admettra que le symbole $:=$ utilisé dans l'écriture $for\ k := 1\ to\ n$ ne sera pas considéré comme une affectation mais que chaque appel de fonction nécessite une affectation $(f :=)$ et une soustraction.**
   Pour les valeurs de a trouvées ci-dessus, donner en fonction de n :
   - **le nombre total d'appels de la fonction f utilisée dans l'algorithme**
     C'est n car à chacun des n tours de boucle $for\ k := 1\ to\ n$, on appelle une seule fois la fonction f par $u := f(u)$.
   - **le nombre total de soustractions nécessaires aux calculs de u et de p**
     C'est n car à chacun des n tours de boucle, on réalise une seule soustraction située à l'intérieur de la fonction f.
   - **le nombre total de multiplications nécessaires aux calculs de u et de p**
     C'est n car à chacun des n tours de boucle, on réalise une seule multiplication $p := p * u$.
   - **le nombre total d'affectations nécessaires aux calculs de u et de p.**
     C'est $2 + 3n$ car il y a deux affectations avant d'entrer dans la boucle $u := a$ et $p := a$ puis à chacun des n tours de boucle, on réalise 3 affectations la première est $u := f(u)$; la seconde est située dans le calcul de $f(u)$ et la troisième est $p := p * u$.
3. **Lorsque la suite (u_n) est bien définie, écrire , pour tout entier naturel n, la relation liant u_{n+1} et u_n.**
   (u_n) définie par $u_0 = a$ et $\forall n \in \mathbb{N}\ u_{n+1} = f(u_n) = u_n - \ln(u_n$
4. **Démontrer que si $a = 1$ alors la suite (u_n) est constante.**
   Démontrons par récurrence que $\forall n \in \mathbb{N}\ u_n = 1$:
   - cette propriété est vraie pour $n = 0$ car $u_0 = a = 1$
   - soit k un entier naturel. supposons que $u_k = 1$ alors $u_{k+1} = f(u_k) = u_k - ln(u_k) = 1 - \ln(1) = 1$
   - La propriété étant initialisée en 0 et héréditaire alors elle est vraie pour tout entier naturel n.
5. **Démontrer que si la suite (u_n) est constante alors $a = 1$**
   (u_n) constante $\Leftrightarrow \forall n \in \mathbb{N}\ u_{n+1} = u_n \Leftrightarrow \forall n \in \mathbb{N}\ u_n - \ln(u_n) = u_n \Leftrightarrow \forall n \in \mathbb{N}\ \ln(u_n) = 0$
   $\Leftrightarrow \forall n \in \mathbb{N}\ u_n = 1$
   Donc si la suite (u_n) est constante alors $u_0 = 1$ donc $a = 1$
6. **On suppose dans cette question que $a > 1$.**
   (a) **Montrer que $\forall n \in \mathbb{N}$, l'on a : $u_n > 1$**
   Démontrons par récurrence que $\forall n \in \mathbb{N}\ u_n > 1$:
   - cette propriété est vraie pour $n = 0$ car $u_0 = a > 1$
   - soit k un entier naturel. supposons que $u_k > 1$ alors $f(u_k) > f(1)$ car f est croissante sur $]1; +\infty[$. Or $u_{k+1} = f(u_k)$ et $f(1) = 1$ donc $u_{k+1} > 1$

- La propriété étant initialisée en 0 et héréditaire alors elle est vraie pour tout entier naturel n.

(b) **Etudier les variations de la suite** (u_n)
$\forall n \in \mathbb{N}\ u_{n+1} - u_n = -\ln(u_n) < 0$ car $\forall n \in \mathbb{N}$, l'on a : $u_n > 1$ donc $\ln(u_n) > 0$. Par conséquent la suite (u_n) est décroissante strictement sur $\mathbb{N}$

(c) **En déduire que la suite** (u_n) **converge puis déterminer sa limite.**
- La suite (u_n) est décroissante et minorée par 1 donc elle converge vers L.
- Comme $\forall n \in \mathbb{N}\ u_n > 1$ donc $L > 1$
- Comme $L > 1$ et que f est continue sur $]1; +\infty[$ alors f est continue en L
- Comme $u_{n+1} = f(u_n)$
- A cause de l'unicité de la limite, L vérifie donc l'équation suivante : $L = f(L)$ donc $L - \ln(L) = L$ donc $\ln(L) = 0$ donc $L = 1$

7. **On suppose dans cette question que** $0 < a < 1$**.**

(a) **Montrer que** $\forall n \in \mathbb{N}^*$ **, l'on a :** $u_n > 1$ Démontrons par récurrence que $\forall n \in \mathbb{N}^*$ $u_n > 1$:
- cette propriété est vraie pour $n = 1$ car $u_1 = f(u_0) = f(a) > 1$ car $0 < a < 1$ et que f est décroissante sur $]0; 1[$ et que $f(1) = 1$
- soit k un entier naturel non nul. Supposons que $u_k > 1$ alors $f(u_k) > f(1)$ car f est croissante sur $]1; +\infty[$. Or $u_{k+1} = f(u_k)$ et $f(1) = 1$ donc $u_{k+1} > 1$
- La propriété étant initialisée en 1 et héréditaire alors elle est vraie pour tout entier naturel n non nul.

(b) **Etudier les variations de la suite** (u_n) **sur** $\mathbb{N}^*$ $\forall n \in \mathbb{N}^*\ u_{n+1} - u_n = -\ln(u_n) < 0$ car $\forall n \in \mathbb{N}^*$, l'on a : $u_n > 1$ donc $\ln(u_n) > 0$. Par conséquent la suite (u_n) est décroissante strictement sur $\mathbb{N}^*$

(c) **En déduire que la suite** (u_n) **converge puis déterminer sa limite.**
- La suite (u_n) est décroissante sur $\mathbb{N}^*$ et minorée par 1 donc elle converge vers L.
- Comme $\forall n \in \mathbb{N}^*\ u_n > 1$ donc $L > 1$
- Comme $L > 1$ et que f est continue sur $]1; +\infty[$ alors f est continue en L
- Comme $u_{n+1} = f(u_n)$
- A cause de l'unicité de la limite, L vérifie donc l'équation suivante : $L = f(L)$ donc $L - \ln(L) = L$ donc $\ln(L) = 0$ donc $L = 1$

Partie 4

1. Démontrer par récurrence que $\forall n \in \mathbb{N}^*\ p_n = au_1u_2 \cdots u_n$
- Avant d'entrer dans la boucle u contient a et p contient a, donc $u_0 = a$ et $p_0 = a$. Puis au premier tour de boucle u reçoit $f(u_0) = u_1$ puis la case p reçoit $p * u$ donc si $n = 1$ on a $p_1 = p_0u_1 = au_1$
- soit k un entier naturel. Supposons que $p_k = au_1u_2 \cdots u_k$ alors au tour de boucle suivant $k + 1$ la case u contient u_{k+1} alors l'instruction $p := p * u$ crée alors $p_{k+1} = (au_1u_2 \cdots u_k)(u_{k+1})$
- La propriété étant initialisée en 1 et héréditaire alors elle est vraie pour tout entier naturel n non nul.

2. **En considérant** $\ln(p_n)$**, montrer que** $\forall n \in \mathbb{N}^*\ p_n = e^{a-u_{n+1}}$
Comme $\forall n \in \mathbb{N}^*\ p_n = au_1u_2 \cdots u_n$ alors $\forall n \in \mathbb{N}^* \ln(p_n) = \ln(au_1u_2 \cdots u_n) = \ln(a) + \ln(u_1) + \cdots + \ln(u_n)$

Donc

$$\left|\begin{array}{rl} u_{n+1}-u_n = & -\ln(u_n) \\ u_n-u_{n-1} = & -\ln(u_{n-1}) \\ u_{n-1}-u_{n-2} = & -\ln(u_{n-2}) \\ & \\ u_3-u_2 = & -\ln(u_2) \\ u_2-u_1 = & -\ln(u_1) \\ u_1-u_0 = & -\ln(u_0) \end{array}\right.$$

Alors par sommation télescopique on obtient :
$u_{n+1}-u_0 = -\ln(p_n)$ donc $u_{n+1}-a = -\ln(p_n)$ donc $\ln(p_n) = a-u_{n+1}$ On en déduit que $\forall n \in \mathbb{N}^*\ p_n = e^{a-u_{n+1}} = \dfrac{e^a}{e^{u_{n+1}}}$

3. **En déduire** $\lim\limits_{n\to+\infty} p_n$
   Quand $n \to +\infty$ u_{n+1} tend vers 1 donc $a-u_{n+1}$ tend vers $a-1$ donc $e^{a-u_{n+1}}$ tend vers $e^{a-1} = \dfrac{e^a}{e}$
4. **Peut-on alors justifier la conjecture de la question 2 de la partie 2 lorsque a valait 2 ?**
   Lorsque $a = 2$ alors $a-1 = 1$ donc $\lim\limits_{n\to+\infty} p_n = e \approx 2,718$
5. **Ecrire alors un nouvel algorithme en Turbo-Pascal permettant le calcul de p_n.**
   Cet algorithme ne doit contenir aucune multiplication.

```
program iter;
var   n , k : integer ;
      a , u , p : real ;
 function f(x : real): real;
      begin
            if x > 0 then f := x - ln(x) ;
      end;
 begin
   (* entrée des données *)
        readln(n);
        readln(a);

   (* traitement *)
       u := a ;
       for k := 1 to n + 1 do  begin
                                              u := f(u) ;
                                  end;
       p :=exp(a)/exp(u);
   (* affichage des résultats *)
       writeln(u);
       writeln(p);
 end.
```