Aix*Marseille université

Domaine Sciences et Technologies
Portail 1 René Descartes

Introduction à l'informatique : TD 6

Code UE : S14IN1I1

Année 2018-19

Algorithmes numériques

NOM : PRÉNOM :

Cette séance de TD est notée. Il vous est donc demandé de travailler de manière plus autonome. Vous pouvez poser des questions à l'enseignant qui y répondra comme à l'habitude mais les exercices ne seront pas corrigés au tableau. Vous devrez répondre aux différentes questions directement sur le sujet du TD, dans les cases prévues à cet effet. À la fin de la séance, vous devrez rendre votre travail, qui sera évalué et noté : il est donc important que vous rédigiez le mieux possible (de manière concise, formelle et non ambigüe) vos réponses aux questions.

Exercice 1 (Speed testing) Trouvez la bonne réponse à chacune des questions suivantes (on ne vous demande pas de longue justification) :

1. Considérons l'algorithme suivant :

```
fonction mystère(n): entier
    r := 1
    pour i de 2 à n faire
        r := r × i
    retour r
```

   Que renvoie l'appel à `mystère`(5) ?

   Solution : `mystère`$(5) = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

2. Décrivez en une phrase ce que calcule la fonction `mystère` pour une entrée n quelconque.

   Solution : La fonction `mystère` calcule la factorielle de l'entrée :

$$n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1 = \prod_{i=1}^{n} i$$

3. Donnez toutes les raisons pour lesquelles le morceau d'algorithme code suivant n'est pas un algorithme, au sens où nous l'avons vu dans le cours :

```
si (nombre d'étoiles dans l'univers = 250 × 10^9) alors
    a := 2 ou bien 3
sinon
    b := 4
```

   Solution : Le test `nombre d'étoiles dans l'univers` $= 250 \times 10^9$ n'est pas une opération basique (réalisable par un humain par exemple) puisqu'on ne connait pas exactement le nombre d'étoiles dans l'univers. Par ailleurs, l'affectation dans la variable a est ambigüe : on ne sait pas s'il faut affecter la valeur 2 ou la valeur 3. La description n'est donc pas assez précise.

4. Donnez toutes les raisons pour lesquelles le morceau de code suivant n'est pas un algorithme, au sens où nous l'avons vu dans le cours :

```
a := 0
tant que (a ≥ 0) faire
    a := a + 2
écrire(a)
```

   Solution : Ce code ne termine pas, puisqu'on ne sort jamais de la boucle.

Exercice 2 (Calcul de la puissance d'un entier et applications au codage RSA) L'élévation d'un nombre à une puissance entière positive est une opération de base cruciale lorsqu'on veut calculer. C'est même l'étape de base dans la méthode cryptographique RSA. Étant donné un nombre x (cela peut-être un entier ou un réel, qu'on représente en machine à l'aide d'un flottant) et un entier naturel n, on souhaite donc calculer le nombre $x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$. On a vu en cours une façon efficace de calculer x^n à l'aide de l'algorithme suivant :

```
fonction fastexp(x, n)
    a := 1
    b := x
    m := n
    tant que m > 0 faire
        si m mod 2 = 1 alors
            a := a × b
        b := b × b
        m := ⌊m/2⌋
    retour a
```

1. Exécutez cet algorithme lors du calcul de 2^{10}, en précisant la valeur des variables lors de chaque étape de la boucle.

   Solution :

	Début	Après 1 tour	Après 2 tours	Après 3 tours	Après 4 tours
m	10	5	2	1	0
a	1	1	$1 \times 4 = 4$	4	$4 \times 256 = 1\,024$
b	2	$2 \times 2 = 4$	$4 \times 4 = 16$	$16 \times 16 = 256$	$256 \times 256 = 65\,536$

2. Donnez une preuve du fait que l'algorithme termine toujours.

   Solution : Lors d'une itération, la valeur de m passe à $\lfloor m/2 \rfloor$. Si $m > 1$, alors $\lfloor m/2 \rfloor < m$ donc la valeur de m décroit strictement lors d'une itération. Elle finit donc par atteindre une valeur inférieure à 1, ce qui force alors à sortir de la boucle, puisque les valeurs inférieures à 1 sont tronquées à 0 par l'opération $\lfloor \cdot \rfloor$. Cela prouve que l'algorithme termine toujours.

3. Pour prouver que l'algorithme est correct, c'est-à-dire qu'il renvoie bien x^n, une méthode consiste à vérifier qu'à tout moment de l'algorithme, on a $x^n = a \times b^m$. Montrez que c'est vrai avant de rentrer dans la boucle, puis que si c'est vrai au début d'une itération de la boucle, alors c'est vrai à la fin de cette itération. Concluez alors à l'aide d'un raisonnement par récurrence.

   Solution : Avant de rentrer dans la boucle, $a = 1$, $b = x$ et $m = n$, donc on a bien $x^n = a \times b^m$. Supposons qu'au début d'une itération de la boucle, on a $x^n = a \times b^m$. Deux cas sont possibles :
   - **1er cas** : si m est pair, alors à la fin de la boucle la nouvelle valeur de la variable m vaut $m/2$, la variable a est restée inchangée, alors que la nouvelle valeur de la variable b est b^2. Or $a \times (b^2)^{m/2} = a \times b^m = x^n$. Donc l'égalité reste vraie à la fin de l'itération.
   - **2ème cas** : si m est impair, alors à la fin de la boucle la nouvelle valeur de la variable m vaut $(m-1)/2$, alors que la nouvelle valeur de la variable a est $a \times b$ et que la nouvelle valeur de la variable b est b^2. Or $a \times b \times (b^2)^{(m-1)/2} = a \times b \times b^{m-1} = a \times b^m = x^n$. Donc l'égalité reste vraie à la fin de l'itération.

   Par principe de récurrence, puisque c'est vrai au début et que cela reste vrai après un tour de boucle supplémentaire, c'est vrai à la fin de la boucle (on sait qu'on sort de la boucle, d'après la question précédente). À la fin, la valeur de m est 0 et on renvoie donc $a = a \times b^m = x^n$.

4. Les opérations élémentaires coûteuses d'un algorithme d'exponentiation (c'est le nom qu'on donne à l'*élévation à la puissance*) sont les multiplications. Combien de multiplications sont effectuées par l'algorithme lors du calcul de 2^{10}. Par souci de généralité, donnez un ordre de grandeur du nombre de multiplications effectuées pour calculer x^n par l'algorithme, en

fonction de n.

Solution : Lors du calcul de 2^{10}, on a effectué 6 multiplications. Plus généralement, la valeur de m est divisée par 2 au moins à chaque itération de la boucle. Puisqu'on initialise cette variable avec la valeur de n et qu'on s'arrête lorsque m vaut 0 (un coup après qu'elle vaille 1 donc), il y a au plus $\lfloor \log_2(n) \rfloor + 1$ itérations de la boucle. Chaque itération exécute au plus 2 multiplications, donc l'algorithme exécute dans le pire des cas $O(\log_2(n))$ multiplications.

5. Le système de cryptographie RSA consiste à calculer des *puissances modulaires*, c'est-à-dire $x^n \bmod k$, le reste de x^n dans la division euclidienne par k. Sachant que

$$\text{si } a \bmod k = b \bmod k \qquad \text{alors } a^n \bmod k = b^n \bmod k,$$

on a intérêt à calculer les puissances en prenant les restes dans la division euclidienne par k à chaque étape. Modifiez ainsi la fonction `fastexp`, en ajoutant un troisième argument k, afin qu'elle calcule $x^n \mod k$: on s'autorise à utiliser comme opération élémentaire supplémentaire le calcul de $y \mod k$, le reste dans la division euclidienne de y par k.

Solution :

```
fonction fastexpmod(x, n, k)
    a := 1
    b := x mod k
    m := n
    tant que m > 0 faire
       si m mod 2 = 1 alors
          a := a × b mod k
       b := b × b mod k
       m := ⌊m/2⌋
    retour a
```

Exercice 3 (Algorithme de Héron pour l'approximation de la racine carrée)

La méthode de Newton permet de calculer une approximation d'un zéro d'une fonction réelle dérivable f, c'est-à-dire une valeur proche d'un réel z tel que $f(z) = 0$. On rappelle que la tangente à la courbe de f en le point x_0 (dans son ensemble de définition) est la droite d'équation $y = f(x_0) + f'(x_0) \times (x - x_0)$. La méthode de Newton se base sur l'idée qu'on peut approcher la courbe d'une fonction par sa tangente : si x est suffisamment proche de x_0 alors $f(x)$ est proche de $f(x_0) + f'(x_0)(x - x_0)$.

En particulier, si on calcule l'intersection de la tangente à la courbe en x_0 avec l'axe des abscisses, on espère obtenir une nouvelle abscisse x_1 plus proche du zéro comme le montre la figure 1 située en haut de la page 4. On résout donc l'équation $0 = f(x_0) + f'(x_0)(x - x_0)$ afin de trouver, si la dérivée $f'(x_0)$ est non nulle, $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Plus généralement, on construit donc la suite $(x_n)_{n\in\mathbb{N}}$ par récurrence, à partir du point x_0 :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

tant que la dérivée $f'(x_n)$ est non nulle (sinon la méthode échoue).

1. Directement sur la figure 1 de la page 4, exécutez les deux itérations suivantes de l'algorithme afin de trouver x_2 et x_3.

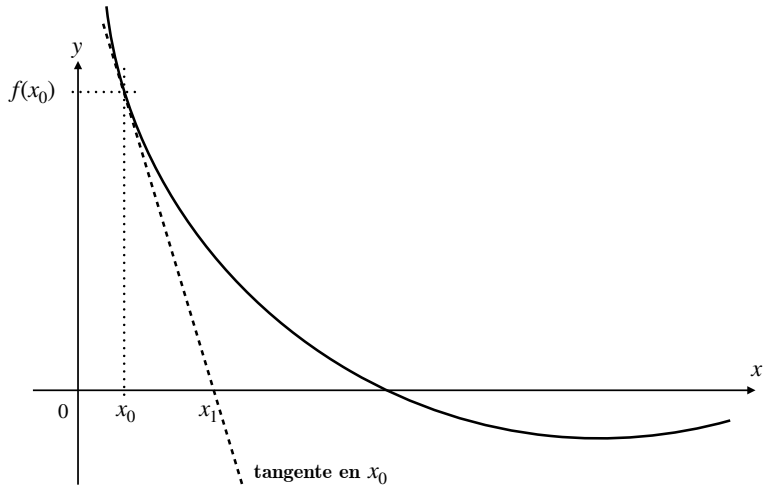
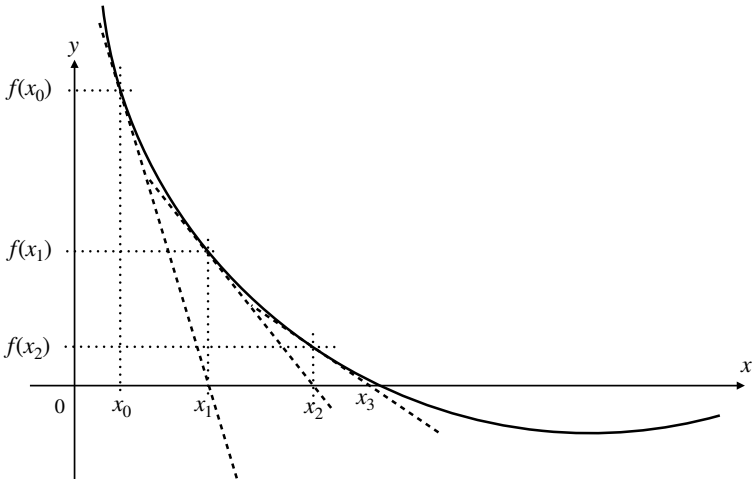


FIGURE 1 – Dessin de la tangente en le point d'abscisse x_0 de la fonction $f : \mathbb{R} \to \mathbb{R}$ représentée.

Solution :

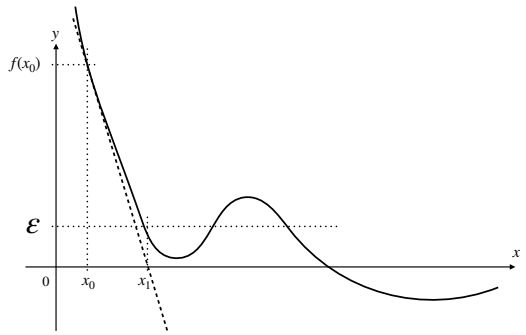


2. On a vu en cours comment décrire l'algorithme de Newton :

```
fonction approx_zero(f, f', x_0, ε)
    x_tmp := x_0
    tant que (|f(x_tmp)| > ε) faire
        x_tmp := x_tmp − f(x_tmp)/f'(x_tmp)
    retour x_tmp
```

Il prend en entrée une description des fonctions f et f', l'abscisse initiale x_0 ainsi qu'un paramètre d'erreur [1] ε (par exemple $\varepsilon = 0,001$) permettant d'écrire la condition d'arrêt de la boucle : ici, on arrête les itérations dès qu'on trouve une abscisse x_{tmp} telle que $|f(x_{\text{tmp}})| \leq \varepsilon$. Malheureusement, cet algorithme n'est pas correct en général : il se peut que l'algorithme termine et renvoie un résultat qui ne soit pas "proche" d'un zéro de la fonction f. C'est le cas dans l'exemple ci-dessous où l'algorithme s'arrête dès la fin de la première itération :

1. On utilise traditionnellement la lettre grecque ε pour décrire ce paramètre d'erreur, car c'est la lettre correspondant à la lettre e.



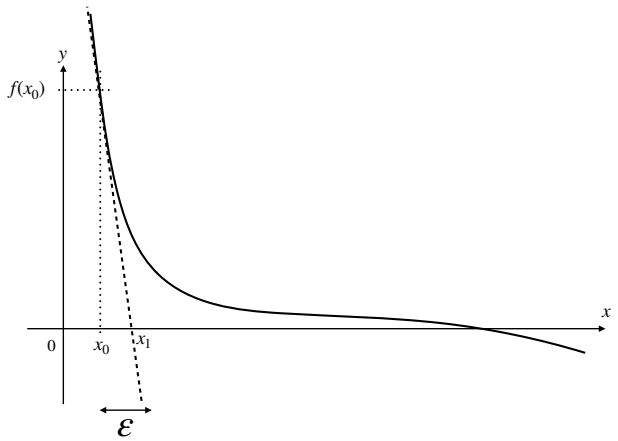
On propose une autre possibilité en remarquant que lorsqu'on s'approche du zéro de f, les abscisses x_n deviennent de plus en plus proche : précisément, la distance de x_n à x_{n+1} tend alors vers 0 lorsque n tend vers l'infini. Modifiez l'algorithme pour que la condition d'arrêt porte ainsi sur cette distance entre deux abscisses successives.

Solution : Pour pouvoir comparer les deux valeurs successives, il faut la conserver en mémoire dans une variable :

```
fonction approx_zero_bis(f, f', x_0, ε)
    x_tmp := x_0
    x_tmp2 := x_tmp − f(x_tmp)/f'(x_tmp)
    tant que (|x_tmp − x_tmp2| > ε) faire
        x_tmp := x_tmp2
        x_tmp2 := x_tmp − f(x_tmp)/f'(x_tmp)
    retour x_tmp
```

3. Trouvez un exemple montrant que ce nouvel algorithme est également incorrect. Cela revient à dessiner le graphe d'une fonction f et de choisir x_0 et un paramètre d'erreur tels que le nouvel algorithme renvoie un résultat "éloigné" du zéro de la fonction f.

Solution :



4. On cherche désormais à appliquer l'algorithme de Newton pour calculer la racine carrée d'un nombre r positif. Pour cela, on utilise la fonction f qui à tout réel x associe $x^2 - r$. Décrivez explicitement la suite récurrente $(x_n)_{n\in\mathbf{N}}$ de la méthode de Newton appliquée

à cette fonction. Cette estimation de la racine carrée $\sqrt{r}$ d'un nombre positif est appelée *algorithme de Héron*.

Solution : On réécrit la formule $x_{n+1} = x_n - f(x_n)/f'(x_n)$ avec la fonction $f : x \mapsto x^2 - r$ de dérivée $f' : x \mapsto 2x$:

$$\forall n \in \mathbb{N},\ x_{n+1} = x_n - \frac{{x_n}^2 - r}{2x_n} = \frac{x_n}{2} + \frac{r}{2x_n}.$$

5. Déduisez-en un algorithme (n'utilisant pas la fonction `approx_zero`) qui donne une approximation de la racine carrée d'un nombre r donné en argument, avec une précision ε donnée en argument également. On souhaite que cet algorithme renvoie un résultat a vérifiant $|a - \sqrt{r}| \leq \varepsilon$, mais attention on ne peut pas utiliser la valeur de $\sqrt{r}$...

Solution : On choisit $x_0 = r$ par exemple. Pour obtenir une approximation a de $\sqrt{r}$ qui vérifie $|a - \sqrt{r}| \leq \varepsilon$, il faut et il suffit que :

$$-\varepsilon \leq a - \sqrt{r} \leq \varepsilon,$$

c'est-à-dire (si ε est inférieur à a) :

$$0 \leq a - \varepsilon \leq \sqrt{r} \leq a + \varepsilon.$$

Par croissance de la fonction carrée, cela est vrai si et seulement si :

$$(a - \varepsilon)^2 \leq r \leq (a + \varepsilon)^2.$$

On en déduit la fonction suivante :

```
fonction approx_zero_ter(r, ε)
    a := r
    tant que r < (a − ε)^2 ou r > (a + ε)^2 faire
        a := (a + r/a)/2
    retour a
```