CAPES MATHS OPTION INFORMATIQUE

ALGORITHMIQUES CLASSIQUES

Recherche dans un tableau

a. Ecrire en Python la fonction qui recherche un élément e dans un tableau tab. La fonction retournera l'indice de l'élément s'il est présent, -1 sinon.

```
def rechercheTab(tab,e) :
    for i,val in enumerate(tab):
        if val == e : return i
    return -1
```

b. Ecrire en Python la fonction récurvive qui recherche un élément dans un tableau trié de façon dichotomique.

```
def rechercheDicho(tab,e) :
    return rechercheDansBornes(tab,e,0,len(tab)-1)

def rechercheDansBornes(tab, e, idep, ifin) :
    if ifin < idep : return -1
    ind_milieu = (ifin + idep) // 2
    if tab[ind_milieu] == e : return ind_milieu
    if e < tab[ind_milieu] : return rechercheDansBornes(tab,e,idep,ind_milieu-1)
    return rechercheDansBornes(tab,e,ind_milieu+1,ifin)
```

c. Quel est le coût d'une telle recherche ?

Au pire, on va trouver l'élément lorsque l'intervalle de recherche est de longueur 1, c'est-à-dire au bout de d subdivisions, avec $2^d = n$. C'est-à-dire $d=\log_2(n)$. Ce qui nous donne un coût en performance en $O(\log_2 n)$.
Le coût en espace est aussi de l'ordre de $\log_2 n$ car on doit empiler @tab, e, idep et ifin à chaque appel récursif (on en a d).

d. Donner une version de l'algorithme de recherche dichotomique qui est plus performante en termes d'espace mémoire.

Il suffit d'écrire l'algorithme de façon itérative et non récursive.

```
def rechercheDicho(tab,e) :
    idep,ifin = 0,len(tab)-1
    while ifin >= idep :
        ind_milieu = (ifin + idep) // 2
        if tab[ind_milieu] == e : return ind_milieu
        if e < tab[ind_milieu] : ifin = ind_milieu - 1
        else ideb = ind_milieu + 1
    return -1
```

Dans cette version, on a besoin de 2 variables (ideb et ifin, en plus de tab et e) et ce quelle que soit la taille du tableau. Cette complexité en mémoire est constante O(1).

Recherche d'un mot dans une chaîne

a. Ecrire une fonction en Python qui prend en paramètres un mot et une chaîne de caractères et détermine si le mot est dans la chaîne (sans utiliser la fonction in déjà existante en Python).

```
def appartient (mot, chaine) :
   n, p = len(chaine), len(mot)
   i = 0
   while i <= n-p :
      if chaine[i : i+p] == mot :
         return True
      i += 1
   return False
```

b. Estimer la complexité de cet algorithme dans le pire des cas.

Le pire des cas est celui où on fait le plus grand nombre de passages dans la boucle while, et cela se produit lorsque le mot n'est pas dans la chaîne. On commence par 3 affectations (n,p,i), puis à chaque passage dans la boucle il y a p comparaisons de caractères (de i à i+p-1) et une incrémentation de i (c-à-d 1 add+1 aff). Dans la condition du while il y a une comparaison et une soustraction (faits n-p+2 fois).

$$C(n,p) = 3 + 2 + \sum_{i=0}^{n-p} (p+2+2) = 5 + (n-p+1)(p+4)$$

La complexité est en $O(p \times (n-p))$.
Pour rappel (utile pour presque tout calcul de complexité) :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Insertion dans liste triée

a. Ecrire en Python la fonction qui permet d'ajouter un élément dans une liste triée. Cette fonction retournera la nouvelle liste. Vous ne pouvez pas utiliser la procédure insert de Python.

```
def inserer(e,l) :
    L.append(0)        # ajoute d'un élément quelconque en fin de liste
    i = len(L) - 2     # indice de l'avant dernier élément (après extension)
    while i > 0 and L[i] >= n :   # tant que dans liste et position pas trouvée
        L[i+1] = L[i]             # décalage vers la droite
        i -= 1
    L[i+1] = n         # insertion de n
    return L
```

b. Donner le coût de cette opération d'insertion.

Dans le pire des cas, on doit parcourir toute la liste pour insérer l'élément en début de liste. La complexité est donc en O(n).

c. Si l'élément était d'abord recherché par dichotomie aurait-on une meilleure complexité ?

La recherche de la position d'insertion peut en effet se faire en O(log n) par dichotomie puisque la liste est triée. Par contre l'insertion demande de recopier les éléments pour faire l'insertion, et dans le pire des cas tous les éléménts. Donc toujours en O(n). Ceci est dû à l'implémentation des listes grâce à des tableaux en Python et non en listes chaînées (qui ferait une insertion en O(1)). Le choix des tableaux permet de faire les accès L[i] en O(1) ce qui est en général préféré.

Conversion décimal-binaire

L'objectif est de créer un programme qui code un nombre entier naturel, donné en écriture décimale, en binaire. Les nombres binaires seront représentés par des chaînes de caractères. Pour plus de lisibilité, on insérera un espace

tous les quatres caractères en partant du dernier. Ainsi, le nombre décimal 525 sera traduit par la chaîne "10 0000 1101" en binaire.

a. Ecrire une fonction conversion en Python, qui prend en paramètre un nombre entier naturel en décimal et renvoie la chaîne de caractères composée uniquement de 0 et de 1 correspondant à son écriture binaire. On ne cherchera pas encore à insérer d'espaces.

Pour rappel :

```
6 | 2                                   7 | 2
0 | 3 | 2                               1 | 3 | 2
  1 | 1 | 2                               1 | 1 | 2
    1 | 0                                   1 | 0
```

On lit le binaire de droite à gauche : 0110 et 0111

On vérifie avec l'opération inverse : $0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 4 + 2 = 6$

Et $0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 4 + 2 + 1 = 7$

```
def conversion (n) :
   bin = str(n % 2)             # premier reste dans la chaine resultat
   q = n // 2                   # premier quotient
   while q != 0 :               # tant que le quotient est non nul
     n , q = q , q // 2         # maj n et q
     bin = str (n % 2) + bin    # ajout à gauche
   return bin
```

b. Ecrire une fonction prenant en paramètre une chaîne de caractères et réalisant l'insertion d'un espace tous les quatres caractères en partant du dernier.

```
def insererEspace (ch) :
   i = 0              # compteur
   M = ''             # chaine resultat
   n = len(ch)
   while i <= n-4 :
      M = ' ' + ch[n-i-4:n-i] + M    # insertion espace tous les 4 caractères
      i += 4
   M = ch[0:n-i] + M      # ajout des caractères restants (de gauche)
   return M
```

c. Modifier la fonction conversion pour prendre en compte l'insertion des espaces

```
def conversion (n) :
   bin = str(n % 2)
   q = n // 2
   while q != 0 :
     n , q = q , q // 2
     bin = str (n % 2) + bin
   return insererEspace(bin)
```

Exponentielle rapide

On s'intéresse à calculer une exponentielle de la manière la plus efficace possible étant donné un réel n et un exposant entier positif a : n^a.

a. Ecrire une première version de la fonction exponentielle, la plus simple et intuitive possible.

```
def exponentielle (n,a) :
    res = 1
    for i in range(a):
        res = res * n
    return res
```

b. Indiquer la complexité de cette fonction.

La fonction exécute la boucle a fois, donc O(a).

Dans une deuxième version, on propose d'utiliser le principe d'exponentiation rapide reposant sur le constat que :

$$n^a = \begin{cases} (n^2)^{\frac{a}{2}} & \text{si a est pair} \\ n \times n^{a-1} & \text{si a est impair} \end{cases}$$

c. Ecrire en Python la fonction `exponentielleRapide` qui retourne la valeur de n^a suivant le principe ci-dessus.

```
def exponentielleRapide(n,a):
    N = n
    A = a
    R = 1
    while A > 0 :
        if A % 2 == 0:
            N = N * N
            A = A // 2
        else :
            R = R * N
            A = A - 1
    return R
```

d. Donner la complexité de cette fonction.

Cette fonction est de complexité $\log_2 a$. Si a est pair, on itère avec un a divisé par 2. Si a est impair, on décrémente a. Mais si on décrémente un nombre impair on obtient un nombre pair. A la fin, on arrive forcément avec a qui vaut 1 (impair) puis 0 ce qui fait sortir de la boucle. Le fait de diviser a par 2 quand il est pair et de faire n-1/2 après deux itérations s'il est impair (i.e. au pire on le divise par 2 toutes les deux itérations), nous donne la complexité en log a.

Crible d'Eratosthène

On se propose de calculer tous les nombres premiers plus petits qu'un entier n donné. La méthode consiste à calculer pas à pas ces nombres en utilisant la règle suivante : si un entier k n'est divisible par aucun nombre premier plus petit que k alors il est lui-même premier.

Ecrire en Python la fonction `eratosthene` qui retourne le tableau des nombres premiers plus petits que n passé en paramètre.

```
def eratosthene(n):
    # Préconditions : n > 1
    # Résultat : le tableau contenant les nombres premiers <= n
    t = [2]
    for i in range(3,n+1) :     # indices 3 à n
        k = 0
        divisible = False
        while k < len(t) and not divisible :
            if i % t[k] == 0 :
                divisible = True
            else :
                k = k + 1
        if (not divisible) :
            t.append(i)
    return t
```

Analyse de la complexité d'un algorithme

On considère le pseudo-code suivant, comportant deux « tant que » imbriqués. On cherche à mesurer la complexité de cette imbrication en fonction de n. Pour cela, on utilise la variable « compteur », qui est incrémentée à chaque passage dans le « tant que » interne.

```
Variables :
  n : entier
  compteur : entier
  i, j : entiers
Début
  Afficher(« Quelle est la valeur de n ? »)
  Saisir(n)
  compteur ← 0
  i ← 1
  Tant que (i < n) Faire
      j ← i + 1
      Tant que (j ≤ n) Faire
          compteur ← compteur + 1
          j ← j + 1
      Fin tantque
      i ← i * 2
  Fin tantque
  Afficher(compteur)
Fin
```

a. Quelle est la valeur finale du compteur dans le cas où n = 16 ?

Pour i=1, j varie de 2 à 16 inclus, on fait donc 15 incrémentations du compteur.
Pour i=2, j varie de 3 à 16 inclus, on fait donc 14 incrémentations du compteur.
Pour i=4, j varie de 5 à 16 inclus, on fait donc 12 incrémentations du compteur.
Pour i=8, j varie de 9 à 16 inclus, on fait donc 8 incrémentations du compteur.
Ensuite, i vaut 16, donc on sort du « Tant que (i < n) ».
Au total, on a donc fait 15+14+12+8 = 49 incrémentations du compteur. Donc compteur vaut 49 en sortie du programme.

b. Considérons le cas particulier où n est une puissance de 2 : on suppose que $n = 2^p$ avec p connu. Quelle est la valeur finale du compteur en fonction de p ? Justifiez votre réponse.

i prend successivement les valeurs suivantes : 2^0, 2^1, 2^2, ... 2^{p-1}, soit 2^k avec k variant de 0 à (p-1). Pour chacune de ces valeurs, on fait (n-i) incrémentations, soit (2^p - 2^k) incrémentations. Ensuite i vaut 2^p, ce qui provoque la sortie du « Tant que (i < n) ». On ne fait pas d'incrémentations du compteur pour cette dernière valeur de i.

$$\sum_{k=0}^{p-1}(2^p - 2^k) = p \times 2^p - \sum_{k=0}^{p-1} 2^k = p \times 2^p - (2^p - 1) = (p-1) \times 2^p + 1$$

Pour rappel :

$$\sum_{k=0}^{p-1} 2^k = 2^0 + \cdots + 2^{p-1} = 2^p - 1$$

Ainsi, la valeur finale du compteur est (p-1)×2^p+1.

c. Réexprimez le résultat précédent en fonction de n.

$(\log_2 n - 1) \times n + 1$ (rappel : $n = 2^p \rightarrow p = \log_2 n$). On a donc une compléxité en $O(n \log n)$.

Calcul d'un polynôme en un point

Soit un polynôme *P* tel que $P(x) = \sum_{k=0}^{n} a_k x^k$.

On veut écrire la fonction qui retourne la valeur de $P(x)$ pour une valeur de *x* passée en paramètre, les coefficients du polynôme étant également passés en paramètre dans un tableau. Pour cela, une méthode efficace est la méthode de Horner, qui utilise la réécriture suivante de *P*(*x*) :

$$P(x) = \Big(\big(\ldots\big((a_n x + a_{n-1})x + a_{n-2}\big)x + \cdots\big)x + a_1\Big)x + a_0$$

La méthode consiste donc à multiplier le coefficient de plus haut degré par *x* et à lui ajouter le coefficient suivant. On multiplie alors le nombre obtenu par *x* et on lui ajoute le troisième coefficient, etc., jusqu'à avoir ajouté le coefficient constant.

Exemple : calcul de $4x^3 - 7x^2 + 3x - 5 = \big((4x - 7)x + 3\big)x - 5$
pour *x* = 2 :

- première étape : 4*2 – 7 = 1
- deuxième étape : 1*2 + 3 = 5
- troisième étape : 5*2 – 5 = 5

Voici le code Python de la fonction qui calcule $P(x)$ selon cette méthode :

```
def valeur_polynome (x, coefs, n) :
# Précondition : coefs contient les (n+1) coefficients du polynôme, ainsi coef[0]
#                contient a0, coef[1] contient a1, etc.
# Résultat : retourne P(x), la valeur du polynôme au point x
1   y = 0.0
2   k = n
3   while k >= 0 :
4       y = y * x + coefs[k]
5       k = k - 1
6   return y
```

a. Combien de fois passe-t-on par les lignes 4 et 5 si le polynôme est de degré *n* ?

On fait (n+1) passages dans la boucle.

b. Complétez le tableau suivant en fonction de *n*, le degré du polynôme. Vous prendrez bien sûr en compte le nombre de fois que l'on passe sur chaque ligne lors de l'exécution complète de la fonction.

Ligne	Nombre d'affectations	Nombre d'additions ou de soustractions	Nombre de multiplications	Nombre de comparaisons
1	1	0	0	0
2	1	0	0	0
3	0	0	0	n+2
4	n+1	n+1	n+1	0
5	n+1	n+1	0	0
6	0	0	0	0
Total	2n+4	2n+2	n+1	n+2

c. Déduisez-en le temps d'exécution $T(n)$ de la fonction en microsecondes, en supposant qu'une affectation prend t_{aff} microsecondes, une addition ou une soustraction t_{add} microsecondes, une multiplication t_{mult} microsecondes, et une comparaison t_{comp} microsecondes.

$T(n) = (2n+4)t_{aff} + (2n+2)t_{add} + (n+1)t_{mult} + (n+2)t_{comp}$
$T(n) = (2t_{aff} + 2t_{add} + t_{mult} + t_{comp})n + (4t_{aff} + 2t_{add} + t_{mult} + t_{comp})$

d. De quelle fonction mathématique de n ce temps d'exécution $T(n)$ est-il « grand O » ?

T(n) est en O(n).

e. Complétez l'invariant de boucle de l'algorithme de Horner : « Au début de chaque itération de la boucle while (juste avant d'exécuter la ligne 4), on sait que (complétez les pointillés) : $y = \sum_{i=0}^{n-(k+1)} a_{\ldots\ldots} x^i$ ».

On sait que $y = a_{k+1} + a_{k+2}x + a_{k+3}x^2 + \cdots + a_n x^{n-(k+1)}$. On a donc $y = \sum_{i=0}^{n-(k+1)} a_{k+1+i} x^i$

f. A l'aide la propriété de terminaison de cet invariant, montrez que l'algorithme de Horner permet bien d'obtenir la valeur du polynôme au point x.

En sortie de boucle, k vaut -1. En remplaçant k par -1 dans l'invariant de boucle, on obtient $y = \sum_{i=0}^{n} a_i x^i$, ce qui est bien la définition de la valeur de P(x).

g. Ecrivez en langage Python la version « naïve » de la fonction valeur_polynome, qui calcule $P(x)$ selon la formule $P(x) = \sum_{k=0}^{n} a_k x^k$, c'est-à-dire en calculant pour chaque terme la puissance de x via autant de multiplications que nécessaire (on n'utilisera donc pas la fonction `pow()` ou l'opérateur `**`).

```
def valeur_polynome (x, coefs, n) :
# Précondition : coefs contient les (n+1) coefficients du polynôme,
#                ainsi coef[0] contient a0, coef[1] contient a1, etc.
# Résultat : retourne P(x), la valeur du polynôme au point x
   y, i, k = 0, 0, 0
   while k <= n :
      xpuissk = 1
      i = 1
      while i <= k :
         xpuissk = xpuissk * x
         i = i + 1
       y = y + coefs[k] * xpuissk
       k = k + 1
   return y
```

h. Combien de multiplications fait-on dans cette version naïve, si le polynôme est de degré n ?

Pour une valeur de k donnée, on fait k passages dans le while interne, donc k multiplications. On doit traiter les valeurs de k allant de 0 à n inclus. On réalise donc :

$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ multiplications dans le while interne.

Il faut ajouter à cela la multiplication par le coefficient, qui est effectuée (n+1) fois.
On réalise donc au total $\frac{n(n+1)}{2} + n + 1 = \frac{n^2}{2} + \frac{3n}{2} + 1$ multiplications. On est donc clairement moins efficace qu'avec la méthode de Horner.

Points fixes

Soit $n \in \mathbb{N}^*$. On s'intéresse aux points fixes de fonctions $f : E_n \to E_n$, où $E_n = [0; n-1]$. On représente une fonction par une liste L de taille n telle que pour tout x, $L[x] = f(x)$. Pour tout k entier naturel, on note f^k l'itérée k de f, c'est-à-dire $f^k = f \circ f \circ \ldots \circ f$.

a. Ecrire une fonction admetPointFixe qui prend en argument une liste L de taille n et renvoie vrai si la fonction f représentée par L admet un point fixe, et faux sinon.

```
def admetPointFixe (L) :
    n = len(L)
    for i in range(n) :
        if L[i] == i :
            return True
    return False
```

b. Ecrire une fonction nbPointFixe qui prend en argument une liste L de taille n et renvoie le nombre de points fixes de la fonction f représentée par L.

```
def nbPointFixe (L) :
    n = len(L)
    ptF = 0
    for i in range(n) :
        if L[i] == i :
            ptF += 1
    return ptF
```

c. Ecrire une fonction itere qui prend en premier argument une liste L de taille n représentant une fonction f, en deuxième et troisième arguments un entier x et un entier k, et renvoie $f^k(x)$.

```
def itere (L,x,k) :
    for i in range(k) :
        x = L[x]
    return x
```

d. Ecrire une fonction nbPointFixeItere qui prend en premier argument un liste L de taille n représentant une fonction f, en deuxième argument un entier naturel k et renvoie le nombre de points fixes de f^k.

```
def nbPointFixeItere (L,k) :
    u = []
    n = len(L)
    for i in range(n) :
        u.append(itere(L,i,k))
    return nbPointFixe(u)
```

Calcul d'intégrale par la méthode de Monte-Carlo

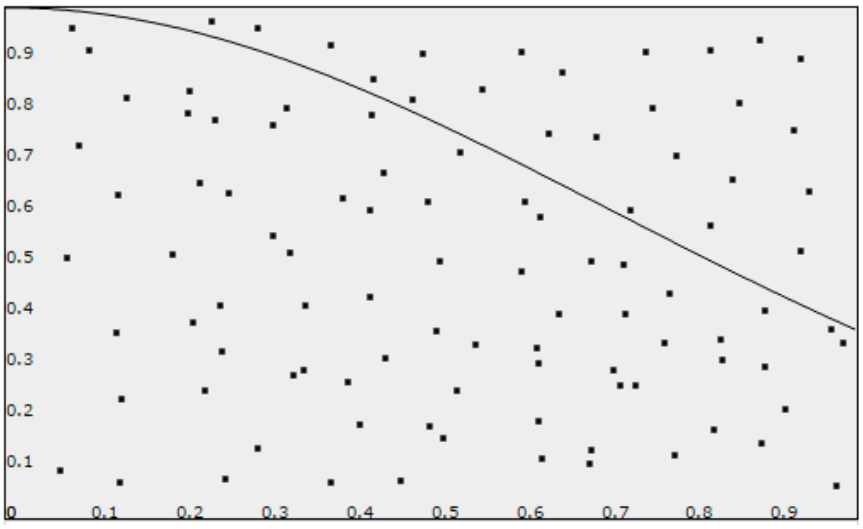
On souhaite calculer une valeur approchée de $I = \int_0^1 e^{-x^2}\,dx$ c'est-à-dire de l'aire sous la courbe de la fonction positive f définie par $f(x) = e^{-x^2}$ entre $x = 0$ et $x = 1$.
Cette fonction f est continue, positive et strictement décroissante sur [0 ;1]. La valeur maximale prise par f sur [0 ;1] est $f(0) = 1$. Soit $(0;\vec{\imath},\vec{\jmath})$ un repère orthonormal. On considère les points : $A(0,1), B(1,1), C(1,0), D(0,0)$. On note D le domaine défini par la courbe représentative de f, la droite d'équation $x = 0$, celle d'équation $x = 1$ et l'axe des abscisses.
Soit l'expérience aléatoire suivante : on tire au hasard un point de coordonnées (x,y) dans le rectangle ABCD. On associe à cette expérience aléatoire la variable aléatoire X qui prend la valeur 1 en cas de succès (appartenance du point au domaine D) et 0 en cas d'échec. On est donc en présence d'une épreuve de Bernoulli. On note p la probabilité de succès de cette expérience aléatoire.

a. Exprimer p en fonction de I et de l'aire du rectangle ABCD.

L'aire du domaine D est égale à I donc on a $p = \frac{I}{aire(ABCD)} = I$

On cherche à estimer au mieux p à partir de la répétition de l'expérience aléatoire. On répète donc n fois cette expérience et on note N_n la variable aléatoire qui prend pour valeur le nombre de succès obtenus lors de cette répétition de n expériences indépendantes et identiques.

Ci-dessous est représenté le tirage aléatoire, de manière indépendante, de 100 points dans le rectangle ABCD.



b. Par lecture graphique, déterminer la valeur prise par la variable aléatoire N_n à l'issue de ces 100 tirages. En déduire une estimation de p et de I, et comparer ces résultats avec la valeur exacte de I, en admettant que $I \approx 0.746824132812$.

Par lecture graphique, pour ce tirage, on a N_n=100-24=76 Et donc I=p≈0.76

On généralise la méthode précédente, appelée méthode de Monte-Carlo, à toute fonction f continue, positive et décroissante sur un intervalle [a;b]. On note D le domaine défini par la courbe représentative de f, la droite d'équation $x = a$, celle d'équation $x = b$, et l'axe des abscisses.

c. Quelles sont les coordonnées des points A, B, C et D dans ce cas ? Quelle est l'aire du rectangle ABCD ?

Les points ont les coordonnées suivantes : A(a,f(a)) B(b,f(a)) D(a,0) C(b,0) L'aire du rectangle ABCD est (b-a)f(a)

d. Comment caractériser l'appartenance d'un point de coordonnées (x,y) au domaine D ?

Un point de coordonnées (x,y) apparient à D si et seulement si : $a \leq x \leq b$ et $0 \leq y \leq f(x)$

e. On rappelle que l'on dispose de la fonction random du module random pour générer un nombre aléatoire pris entre 0 et 1. Quelle instruction permet d'obtenir un nombre aléatoire pris entre a et b ?

```
random()*(b-a)+a
```

f. Ecrire la fonction monteCarlo qui prend en paramètres la fonction continue, décroissante et positive f, les nombres réels a et b, et l'entier n non nul, et qui renvoie une valeur approchée de l'intégrale I par la méthode de Monte-Carlo.

```
def monteCarlo (f,a,b,n) :
   r = 0                  # init compteur de success
   max = f(a)             # max de la fonction
   for i in range(n) :
      x = a + random() * (b-a)       # tirage abscisse entre a et b
      y = random() * max             # tirage ordonnee entre 0 et f(a)
      if y <= f(x) :                 # si (x,y) appartient au domaine D
         r += 1                         # incrément compteur
   return r * (b-a) * max / n     # retourne valeur approchée de I
```

g. Donner des valeurs approchées de I obtenues pour $n = 100, n = 1000, n = 10000, n = 100000, n = 1000000$.

On a les valeurs (différentes à chaque exécution) :

n	valeur approchée de I	erreur
100	0.75000	0.00317
1 000	0.73899	0.00782
10 000	0.74280	0.00402
100 000	0.74841	0.00159
1 000 000	0.74636	0.00046

Contrairement à d'autres méthodes d'intégration numérique, cette méthode est stochastique (aléatoire). Elle retourne un résultat différent à chaque appel de la fonction monteCarlo.

h. Comment modifier l'algorithme si f est continue, positive et croissante ?

Si f est continue, positive et croissante, il suffit de remplacer l'instruction max=f(a) par max=f(b).