AP2

Bases de l'algorithmique

Code *AP2*.1: Algorithme de calcul direct d'une factorielle

```
#####################################
# Calcul direct: n! = 1*2*3*..*(n-1)*n
#####################################
# Initialisation de la valeur à renvoyer
# Faire une boucle pour k variant de 1 à n
#     Multiplier la valeur stockée par k
# Renvoyer la valeur stockée à la fin de la boucle
```

Code *AP2*.2: Implémentation directe de la factorielle en Python

```
def factorielle_directe(n):
    """ (int) -> int
    Précondition: n >= 0
    Calcul direct de la valeur de n!

    >>> factorielle_directe(1)
    1
    >>> factorielle_directe(3)
    6
    >>> factorielle_directe(0)  # Cas particulier du 0
    1
    """
    fact = 1                # Initialisation de la valeur à renvoyer
    for k in range(1,n+1): # Faire une boucle pour k variant de 1 à n
        fact *= k           # Multiplier la valeur stockée par k
    return fact             # Renvoyer la valeur stockée à la fin de la boucle
```

Code *AP2*.3: Algorithme de calcul récursif d'une factorielle

```
##################################
# Calcul récursif: n! = n * (n-1)!
##################################
# Si n = 1 ou 0: renvoyer 1
# Sinon renvoyer n * (n-1)!
```

Code *AP2*.4: Implémentation récursive de la factorielle en Python

```
def factorielle_recursive(n):
    """ (int) -> int
    Précondition: n >= 0
    Calcul récursif de la valeur de n!

    >>> factorielle_recursive(1)
    1
    >>> factorielle_recursive(3)
    6
    >>> factorielle_directe(0)  # Cas particulier du 0
    1
    """
    if n == 1 or n == 0: return 1                # Si n = 1 ou 0: renvoyer 1
    else: return n * factorielle_recursive(n-1) # Sinon renvoyer n * (n-1)!
```

Code *AP2*.5: Algorithmes de recherche simple dans une liste

```
# Itérer sur les éléments de la liste en maintenant un compteur
#     Si on trouve l'élément cherché, on renvoie la valeur du compteur
# Sinon, on renvoie "rien"
```

Code *AP2*.6: Traduction de l'algorithme de recherche simple

```
def recherche_simple(element,liste):
    """ (?,list) -> int ou None
    Fonction permettant la recherche d'un élément dans une liste. Elle
    renvoie l'indice déterminant la (première) place de l'élément dans la
    liste si il existe et 'None' sinon.

    >>> L = [10,'a',5,2,['abra','cadabra']]
    >>> recherche_simple(5,L)
    2
    >>> recherche_simple('a',L)
    1
    >>> recherche_simple(['hokus','pokus'],L)
    >>> recherche_simple(['abra','cadabra'],L)
    4
    """
    # Itérer sur les éléments de la liste en maintenant un compteur
    for i in range(len(liste)):
        if liste[i] == element: # Si on trouve l'élément cherché,
            return i            # on renvoie la valeur du compteur
    return None                 # Si aucun ne marche, on renvoie "rien"
```

Code *AP2*.7: Algorithme de division euclidienne par soustraction

```
# Initialiser les valeurs
# Tant que le reste est supérieur au diviseur
#     Soustraire le diviseur du reste
#     et rajouter 1 au quotient
# Renvoyer quotient et reste en sortie de boucle
```

Code *AP2*.8: Implémentation de la division euclidienne par soustraction

```
1  def division_euclidienne(a,b):
2      """(int,int) -> tuple(int,int)
3      Precondition: a > 0 et b > 0
4      Fonction effectuant la division euclidienne de a par b en utilisant
5      l'algorithme par soustraction. Renvoie un doublet (quotient,reste), de
6      sorte que a = quotient*b + reste avec reste < b.
7
8      >>> division_euclidienne(10,2)
9      (5, 0)
10     >>> division_euclidienne(2,10)
11     (0, 2)
12     >>> division_euclidienne(7,3)
13     (2, 1)
14     """
15     q,r = 0,a      # Initialisation des valeurs
16     while r >= b: # Tant que le reste est supérieur au diviseur
17        r = r - b  # Soustraire le diviseur du reste
18        q = q + 1  # et rajouter 1 au quotient
19     return q,r
```

Code *AP2*.9: Procédure de test pour la division euclidienne par soustraction

```
1  # On fait quelques tests
2  def teste_division_euclidienne(listea,listeb):
3      """(list,list) -> str
4      Fonction qui teste division_euclidienne() avec une série de valeurs
5      pour a et b (les deux listes doivent avoir la même taille...).
6      Renvoie une chaîne de caractère qui devrait "parler" à l'utilisateur."""
7      stot = ''  # Initialisation de la chaîne à renvoyer au final
8      for i in range(len(listea)):         # On itère sur les listes
9          a,b = listea[i],listeb[i]        # On récupère les valeurs de a et b
10         q,r = division_euclidienne(a,b) # La division proprement dite
11         # Le rendu sous forme de chaîne de caractère
12         stot+= 'On a bien {:2d} = {:2d}*{:2d} + {:2d}'.format(a,q,b,r) + "\n"
13     return stot
14
15 listea = [2,4,5,1,9,83,45,75,43,28]
16 listeb = [4,2,5,2,3, 7, 8, 7,10, 3]
17
18 print(teste_division_euclidienne(listea,listeb))
```

Code *AP2*.10: Résultats des tests précédents

```
On a bien  2 =  0* 4 +  2
On a bien  4 =  2* 2 +  0
On a bien  5 =  1* 5 +  0
On a bien  1 =  0* 2 +  1
On a bien  9 =  3* 3 +  0
On a bien 83 = 11* 7 +  6
On a bien 45 =  5* 8 +  5
On a bien 75 = 10* 7 +  5
On a bien 43 =  4*10 +  3
On a bien 28 =  9* 3 +  1
```

Code *AP2*.11: Implémentation directe de la factorielle en Python (sans commentaires: c'est Mal !)

```
1  def factorielle_directe(n):
2      fact = 1
3      for k in range(1,n+1):
4          fact *= k
5      return fact
```

Code *AP2*.12: Algorithme à comprendre (dur, dur car il n'y a aucun commentaire...)

```
1  def exponentiation_rapide(a,n):
2      """n est un entier."""
3      A = a
4      N = n
5      R = 1
6      while N>0:
7          if N%2 == 0:
8              A = A*A
9              N = N/2
10         else:
11             R = R*A
12             N = N-1
13     return R
```

Code *AP2*.13: Algorithme de construction d'une liste par ajouts successifs

```
    ###################################
    # Algorithme quadratique en mémoire
    ###################################
    #
    # Itérer sur les éléments à rajouter
    #     Créer une nouvelle liste par concaténation
    #     de l'ancienne liste + le nouvel élément

    ###############################
    # Algorithme linéaire en mémoire
    ###############################
    #
    # Itérer sur les éléments à rajouter
    #     Utiliser append()...
```

Code *AP2*.14: Implémentation de la construction d'une liste par ajouts successifs

```
>>> ###################################
>>> # Algorithme quadratique en mémoire
>>> ###################################

>>> N = 10  # Initialisation du nombre d'éléments
>>> L = []  # Initialisation de la liste
>>> # Itérer sur les éléments à rajouter
>>> for k in range(N):
...     # Créer une nouvelle liste par concaténation
...     L = L + [k]
...     # Vérification de l'identité de L
...     id(L)
...
4557792464
4557792392
4557792320
4557792464
4557792392
4557792320
4557792464
4557792392
4557792320
4557792464
>>> ################################
>>> # Algorithme linéaire en mémoire
>>> ################################

>>> N = 10  # Initialisation du nombre d'éléments
>>> L = []  # Initialisation de la liste
>>> # Itérer sur les éléments à rajouter
>>> for k in range(1,N+1):
...     # Utiliser append qui rajoute juste "une case"
...     L.append(k)
...     # Vérification de l'identité de L
...     id(L)
...
4557792320
4557792320
4557792320
4557792320
4557792320
4557792320
4557792320
4557792320
4557792320
4557792320
```

Code *AP2*.15: Algorithme de recherche d'un maximum dans une liste de nombres

```
##############################################################
# Algorithme naturel (non optimal: complexité quadratique)
##############################################################
#
# On itère sur tous les éléments de la liste
#     et on compare chaque élément à tous ceux de la liste
#         Si c'est le plus grand, on le stocke
# pour renvoi à la fin.
#
#################################################################
# Algorithme naturel avec break
# (non optimal: complexité quadratique dans le pire des cas)
#################################################################
#
# On itère sur tous les éléments de la liste
#     et on compare chaque élément à tous ceux de la liste,
#         jusqu'à en trouver un plus grand
#     On renvoie celui qui passe la comparaison jusqu'au bout.
#
###################################
# Algorithme à mémoire (linéaire)
###################################
#
# Initialisation de la mémoire au premier élément
# On itère sur tous les éléments de la liste
#      Si on trouve plus grand que l'élément en mémoire,
#      on le stocke et on continue
# Arrivé au bout de la liste, on renvoie l'élément en mémoire
```

Code *AP2*.16: Implémentation de l'algorithme naturel

```
############################################################
# Algorithme naturel (non optimal: complexité quadratique)
############################################################
def maximum_non_optimal(liste):
    """(list(float)) -> float
    Recherche du maximum d'une liste de nombres de manière naturelle mais
    non optimale, c'est-à-dire avec deux boucles imbriquées parcourues
    entièrement.

    >>> L = [1,5,0.2,-35.1,5.01]
    >>> maximum_non_optimal(L)
    5.01
    """
    for v1 in liste:            # On itère sur tous les éléments de la liste
        est_plus_grand = True   # Initialisation du booléen
        for v2 in liste:        # On compare chaque élément
            if v2 > v1:         # à tous ceux de la liste
                est_plus_grand = False
        if est_plus_grand:      # Si c'est le plus grand,
            maximum = v1        # on le stocke
    return maximum              # pour renvoi à la fin.
```

Code *AP2*.17: Implémentation de l'algorithme naturel avec `break` pour optimiser

```
##############################################################
# Algorithme naturel avec break
# (non optimal: complexité quadratique dans le pire des cas)
##############################################################
def maximum_non_optimal_break(liste):
    """Recherche du maximum d'une liste de nombres de manière naturelle,
    mais utilisant break pour sortir plus tôt des listes parcourues
    et gagner ainsi un peu de temps d'exécution dans les cas favorables.

    >>> L = [1,5,0.2,-35.1,5.01]
    >>> maximum_non_optimal_break(L)
    5.01
    """
    # On itère sur tous les éléments de la liste
    for v1 in liste:
        est_plus_grand = True  # Initialisation du booléen
        for v2 in liste:       # On compare chaque élément à tous ceux de la liste
            if v2 > v1:        # jusqu'à en trouver un plus grand
                est_plus_grand = False  # Dans ce cas, ce n'était pas le bon
                break          # et on peut s'arrêter
        if est_plus_grand:     # On renvoie le premier qui passe la
            return v1          # comparaison jusqu'au bout.
```

Code *AP2*.18: Implémentation de l'algorithme de maximum à mémoire (linéaire en temps)

```
############################################
# Algorithme à mémoire (complexité linéaire)
############################################
def maximum_a_memoire(liste):
    """Recherche du maximum d'une liste de nombres de manière "réfléchie",
    c'est-à-dire en ne parcourant la liste qu'une seule fois et en stockant le
    dernier maximum connu dans une mémoire, mise à jour au besoin.

    >>> L = [1,5,0.2,-35.1,5.01]
    >>> maximum_a_memoire(L)
    5.01
    """
    maximum = liste[0]     # Initialisation de la mémoire au premier élément
    for v in liste:        # On itère sur tous les éléments de la liste
       if v > maximum:     # Si on trouve plus grand que l'élément en mémoire,
           maximum = v     # on le stocke et on continue
    return maximum  # Arrivé au bout de la liste, on renvoie l'élément en mémoire
```

Code *AP2*.19: Procédures annexes pour les tests d'efficacité des différents algorithmes

```
def affichage_resultats_tests(N,non_optimal,non_optimal_break,memoire):
    """Procédure d'affichage des résultats pour différentes tailles de la
    liste dans laquelle déterminer le maximum pour les trois algorithmes
    testés. On essaie de faire simple et efficace mais un minimum joli.
    Renvoie une chaîne de caractère contenant le tableau de valeurs."""
    # L'entête détermine la largeur
    entete = " Nb d'éléments | Algo naturel | Algo break | Algo mémoire |"
    largeur= len(entete)
    s = entete + "\n" + '-'*largeur + "\n" # On rajoute une ligne de '-'
    # Pour plus de facilité, on se définit une liste des listes
    listes = [N,non_optimal,non_optimal_break,memoire]
    # Et le split de l'entête suivant le séparateur
    splitted = entete.split('|')
    # On boucle sur le nombre de tests
    for i in range(len(N)):
        for j in range(4):
            # On formatte à la bonne taille
            s += str(listes[j][i]).rjust(len(splitted[j])-1) + ' |'
        s += "\n"                # Le saut de ligne
#         s += '-'*largeur + "\n" # et la ligne de séparation
    return s

import time # Pour pouvoir déclencher le chronomètre

def fait_tests(listes_a_tester):
    """Procédure de tests. Renvoie un quadruplet comprenant le nombre
    d'éléments et les résultats (en ms) de chaque algorithme sur la liste en
    question."""
    # Initialisations diverses
    resultats = [[],[],[]]
    algos = [maximum_non_optimal,maximum_non_optimal_break,maximum_a_memoire]
    for liste in listes_a_tester:       # On itère sur les listes
        for i in range(len(algos)):     # Et sur les algorithmes
            t1 = time.clock()           # Départ
            algos[i](liste)             # Boulot, boulot...
            t2 = time.clock()           # Arrivée
            # On stocke le résultat avec 3 chiffres après la virgule
            resultats[i].append('{}'.format(round((t2-t1)*1e3,3)))
    return resultats[0],resultats[1],resultats[2]  # et on renvoie le tout.
```

Code *AP2*.20: Tests à proprement parler

```
import random  # Pour la génération de listes aléatoires

def tests_et_affichage(N):
    """Jeu complet d'instructions permettant les tests sur listes triées,
    aléatoires et inversées pour les trois algorithmes définis précédemment."""
    stot = "Dans tous les tests, le temps est donné en millisecondes\n\n"
    stot+= "On commence par les listes triées:\n\n"
    triees = [list(range(n)) for n in N]                # Définition
    n,b,m = fait_tests(triees)                          # Tests
    stot+= affichage_resultats_tests(N,n,b,m) + "\n\n" # Affichage
    stot+= "On continue avec les listes inversées:\n\n"
    inversees = triees[:]               # Copie des listes
    # NB: en fait on change les listes de départ mais ce n'est pas grave à
    # condition de faire l'inversion avant les listes aléatoires.
    for l in inversees: l.reverse()     # Inversion des listes *in place*
    n,b,m = fait_tests(inversees)                       # Tests
    stot+= affichage_resultats_tests(N,n,b,m) + "\n\n" # Affichage
    stot+= "On termine sur les listes aléatoires:\n\n"
    aleas = triees[:]                   # Copie des listes
    for l in aleas: random.shuffle(l)  # Mélange des listes *in place*
    n,b,m = fait_tests(aleas)                           # Tests
    stot+= affichage_resultats_tests(N,n,b,m) + "\n\n" # Affichage
    return stot  # Renvoie de la chaîne de résultats.

N = [3*10**i for i in range(4)]

print(tests_et_affichage(N))
```

Code *AP2*.21: Résultats des tests

```
Dans tous les tests, le temps est donné en millisecondes

On commence par les listes triées:

 Nb d'éléments | Algo naturel | Algo break | Algo mémoire |
-------------------------------------------------------------
             3 |        0.002 |      0.001 |        0.001 |
            30 |        0.034 |      0.016 |        0.002 |
           300 |        3.022 |      1.103 |         0.01 |
          3000 |      301.023 |    111.746 |        0.105 |


On continue avec les listes inversées:

 Nb d'éléments | Algo naturel | Algo break | Algo mémoire |
-------------------------------------------------------------
             3 |        0.001 |      0.001 |          0.0 |
            30 |        0.067 |      0.003 |        0.003 |
           300 |        4.948 |      0.013 |        0.012 |
          3000 |      294.695 |      0.076 |        0.074 |


On termine sur les listes aléatoires:

 Nb d'éléments | Algo naturel | Algo break | Algo mémoire |
-------------------------------------------------------------
             3 |        0.002 |      0.002 |        0.001 |
            30 |        0.039 |      0.006 |        0.003 |
           300 |        3.294 |      0.077 |        0.007 |
          3000 |      325.503 |      0.387 |        0.075 |
```

Code *AP2*.22: Implémentation récursive de la suite de Fibonacci

```python
def fibo_rec(n):
    """ (int) -> int
    Precondition: n > 0
    Implémentation récursive de la suite de fibonacci

    >>> fibo_rec(1),fibo_rec(2),fibo_rec(3),fibo_rec(4),fibo_rec(5),fibo_rec(6)
    (1, 1, 2, 3, 5, 8)
    """
    if n == 0 or n == 1: return n         # Conditions d'arrêt
    return fibo_rec(n-1) + fibo_rec(n-2) # Sinon, on réappelle la fonction
```

Code *AP2*.23: Implémentation directe de la suite de Fibonacci (sans commentaires...)

```
def fibo_dir(n):
    """ (int) -> int
    Precondition: n > 0
    Implémentation directe de la suite de fibonacci. Sans commentaires,
    donc plus énigmatique... À vous de comprendre comment cela marche.

    >>> fibo_dir(1),fibo_dir(2),fibo_dir(3),fibo_dir(4),fibo_dir(5),fibo_dir(6)
    (1, 1, 2, 3, 5, 8)
    """
    a,b,i = 0,1,2
    while i<= n :
        a,b,i = b,a+b,i+1
    return b
```

Code *AP2*.24: Temps (en ms) pour le calcul du n^{e} terme de Fibonacci par les deux méthodes

```
 n | direct | récursif | t[i]/t[i-1] pour récursif |
---------------------------------------------------
10 |  0.002 |    0.026 |                        NA |
11 |  0.001 |    0.040 |                      1.54 |
12 |  0.002 |    0.215 |                      5.38 |
13 |  0.003 |    0.106 |                      0.49 |
14 |  0.002 |    0.171 |                      1.61 |
15 |  0.002 |    0.277 |                      1.62 |
16 |  0.001 |    0.448 |                      1.62 |
17 |  0.001 |    0.726 |                      1.62 |
18 |  0.003 |    1.197 |                      1.65 |
19 |  0.003 |    1.897 |                      1.58 |
20 |  0.003 |    3.070 |                      1.62 |
21 |  0.003 |    4.984 |                      1.62 |
22 |  0.003 |    8.076 |                      1.62 |
23 |  0.002 |   13.041 |                      1.61 |
24 |  0.002 |   21.144 |                      1.62 |
25 |  0.003 |   34.078 |                      1.61 |
26 |  0.003 |   55.500 |                      1.63 |
27 |  0.004 |   89.230 |                      1.61 |
28 |  0.004 |  141.352 |                      1.58 |
29 |  0.003 |  228.671 |                      1.62 |
30 |  0.003 |  374.576 |                      1.64 |
31 |  0.005 |  601.111 |                       1.6 |
32 |  0.004 |  973.593 |                      1.62 |
```