DS5_24.py

```
001| # ================================
002| # Pré-chauffage théorique
003| # ================================
004| # Q1. Écriture binaire du nombre 9 : "1001".
005| # Q2. Les entiers positifs codés sur 8 bits vont
de 0 à 255.
006| # Q3. Un variant de boucle est une grandeur qui
diminue (ou augmente) à chaque itération, garantissant
ainsi la terminaison de la boucle.
007| # Q4. Un invariant de boucle est une propriété
vraie au début et à la fin de chaque itération, ce qui
aide à démontrer la correction de l'algorithme.
008|
009| # ================================
010| # 1. Opérations sur les entiers
011| # ================================
012|
013| def mystere(N):
014|     """
015|     La fonction mystere calcule la décomposition
en base 2 de N.
016|     Q5. Exemple d'exécution avec N=6 :
017|          Itération 1: n = 6, L = []         =>
L.append(6 % 2) = 0, n devient 3.
018|          Itération 2: n = 3, L = [0]         =>
L.append(3 % 2) = 1, n devient 1.
019|          Itération 3: n = 1, L = [0, 1]        =>
L.append(1 % 2) = 1, n devient 0.
020|          Résultat final: L = [0, 1, 1]
(représentation en "little-endian").
021|     """
022|     n = N
023|     L = []
024|     while n > 0:
025|         L.append(n % 2)
026|         n = n // 2
027|     return L
028|
029| # Q6. Invariant de boucle :
030| # À chaque itération, on a :
031| #      N = (∑_{i=0}^{len(L)-1} L[i]*2^i) +
```

```
n*2^(len(L))
032| # initialisation : L = [], len(L)=0 donc N = n
033| # propagation : on suppose vrai :
034| #   N = (∑_{i=0}^{len(L)-1} L[i]*2^i) +
n*2^(len(L))
035| #   à l'itération suivante, n//2 et L[i]=n%2, o,
vérifie bien que :
036| #   N = (∑_{i=0}^{len(L)-1} L[i]*2^i) +
n*2^(len(L))
037| # Ce qui permet de "reconstruire" N à partir de
la partie déjà traitée (L) et de la partie restante
(n).
038|
039|
040| # Q7. Variant de boucle :
041| # On peut prendre le variant "n". En effet, n est
toujours positif et diminue strictement à chaque
itération
042| # (puisque n = n // 2). Ainsi, n est un entier
naturel qui atteint 0, garantissant ainsi la
terminaison.
043|
044| def mystere_variant(N):
045|     """
046|     Version annotée montrant le variant n.
047|     Variante: n qui diminue strictement jusqu'à
0.
048|     """
049|     n = N
050|     L = []
051|     # Invariant initial : N = 0 + n*2^0 = n.
052|     while n > 0:  # variant n > 0, diminue à
chaque tour.
053|         L.append(n % 2)
054|         n = n // 2
055|     # À la fin, n == 0 et donc N = somme des bits
de L pondérée par 2^i.
056|     return L
057|
058| # Q8. Conclusion sur l'algorithme :
059| # Lorsque la boucle se termine, n == 0 et
l'invariant donne :
060| #      N = ∑_{i=0}^{len(L)-1} L[i]*2^i.
```

```
061| # Ainsi, L contient les coefficients de la
décomposition binaire de N (bien que dans l'ordre
inverse,
062| # ce qui correspond à une représentation dite en
"little-endian").
063|
064| # Q9. Modification pour obtenir une liste de 8
éléments avec le bit de poids fort en premier.
065| def mystere (N):
066|     """
067|     Version annotée montrant le variant n.
068|     Variante: n qui diminue strictement jusqu'à
0.
069|     """
070|     n = N
071|     L = []
072|     # Invariant initial : N = 0 + n*2^0 = n.
073|     while n > 0:  # variant n > 0, diminue à
chaque tour.
074|         L.append(n % 2)
075|         n = n // 2
076|     # À la fin, n == 0 et donc N = somme des bits
de L pondérée par 2^i.
077|     return (8-len(L))*[0] + L[::-1]
078|
079| # ================================
080| # 1 - Sommes d'éléments (Addition binaire)
081| # ================================
082|
083| # Q10. On a :
084| # [0,0,0,0,0,1,1,0] représente 6 et
[0,0,0,0,0,1,1,1] représente 7.
085| # Leur somme donne 13 qui se code sur 8 bits par
[0,0,0,0,1,1,0,1].
086|
087| def somme(L1, L2):
088| # On ignore le dépassement (overflow) pour rester
sur le même nombre de bits.
089|     n = len(L1)
090|     result = [0] * n
091|     retenue = 0
092|     for i in range(n-1, -1, -1):
093|         total = L1[i] + L2[i] + retenue
```

```
094|         if total <2:
095|             result[i] = total
096|             retenue =0
097|         elif total==2:
098|             result[i] = 0
099|             retenue = 1
100|         elif total==2:
101|             result[i] = 1
102|             retenue = 1
103|     return result
104|
105| # ================================
106| # 2 - Soustraction (Codage en complément à deux)
107| # ================================
108|
109| # Q12. Pour -3 :
110| #[1, 1, 1, 1, 1, 1, 0, 1]
111|
112| # Q13
113| def base2rel(N):
114|     if N>=0: return mystere(N)
115|     else :
116|         L = mystere(-N)
117|         # on inverse tous les bits
118|         L_inv = [(1-x) for x in L]
119|         # on ajoute 1 :
120|         L_final = somme(L_inv, 7*[0]+[1])
121|         return L_final
122|
123| # Q14 et Q15 :  [0, 0, 0, 0, 0, 1, 1, 1]
124| # problème de dépassement, c'est un probleme de
representation des nombres
125|
126|
127|
128| # ================================
129| # 3 -
130| # ================================
131| # Q16
132| def est_puissance_de2(n):
133|     L = mystere(n)
134|     s = 0
135|     for x in L :
```

```
136|             s+=x
137|         return s==1
138|
139|  # Q17
140|  def est_palindromique(L):
141|         n = len(L)
142|         for i in range(n//2):
143|             if L[i]!=L[n//2-i]:return False
144|         return True
145|  # ================================
146|  # 4 - Checksum
147|  # ================================
148|
149|  # Q18. Pour L = [1,0,0,1,1,0,1,0]:
150|  # Les 7 premiers bits (données) font la somme
1+0+0+1+1+0+1 = 4 (paire).
151|  # Le bit de contrôle attendu est donc 0, ce qui
est le dernier élément de L.
152|  # La fonction devrait donc renvoyer True.
153|
154|
155|
156|  # Q19. Étendue des nombres :
157|  # - Pour un codage avec 7 bits de données (et 1
bit de contrôle), les entiers non signés vont de 0 à
2^7 - 1, c'est-à-dire de 0 à 127.
158|  # - Pour le codage signé (en complément à deux
sur 7 bits), la plage est de -64 à 63.
159|
160|  # 20. La fonction controle_parite ci-dessus est
déjà une écriture simple vérifiant le critère.
161|  def controle_parite(L):
162|         """
163|         Vérifie la parité : pour une liste L où les 7
premiers éléments sont les données
164|         et le 8ème est le bit de parité. La parité
est correcte si la somme des 7 premiers bits est paire
et
165|         le bit de parité vaut 0, ou impaire et le bit
vaut 1.
166|         """
167|         valeurs = L[:-1]
168|         parity_bit = L[-1]
```

```
169|      s= 0
170|      for v in valeurs : s+=v
171|      return (s % 2) == parity_bit
```