

Explication de code Exercice 3

Sanna Thomas, L3STI

September 16, 2024

1 Introduction

L'entier (int) tel qu'on le connaît a une taille de 32 bits, et un booléen sollicite 8 bits (alors que l'information peut théoriquement rentrer dans un seul bit).

L'objectif de cet exercice est de proposer une solution pour stocker un ensemble de trois nombres et deux booléens dans un seul entier, en sachant que les trois nombres sont compris entre 0 et 500; puis une solution pour lire les informations contenues dans un tel entier. On utilisera exclusivement les opérateurs bitwise.

On rédigera un algorithme, qu'on mettra en oeuvre en Python.

2 Proposition de solution

- Pour stocker les trois nombres, comme ceux-ci sont compris entre 0 et 500, on peut les stocker sur 9 bits chacun car $2^9 = 512 \leq 500$.
On peut donc stocker les trois nombres sur 27 bits.
- Pour les deux booléens, on peut les stocker sur 1 bit chacun. On peut donc stocker les deux booléens sur 2 bits.
- On a donc besoin de 29 bits pour stocker les cinq informations.

3 Proposition d'algorithme en Python

Variables

- `n1`, `n2`, `n3`: Trois nombres compris entre 0 et 500.
- `b1`, `b2`: Deux booléens (True ou False).

Sérialisation

L'algorithme de sérialisation utilise les décalages de bits pour encoder les valeurs dans un seul entier.

```
1 def serialisation(n1, n2, n3, b1, b2):
2     """
3     Sérialise trois nombres et deux booléens en un seul
4         ↪ entier.
5
6     Args:
7         n1 (int): Un nombre entier compris entre 0 et 500.
8         n2 (int): Un nombre entier compris entre 0 et 500.
9         n3 (int): Un nombre entier compris entre 0 et 500.
10        b1 (bool): Un booléen (True ou False).
11        b2 (bool): Un booléen (True ou False).
12
13    Returns:
14        int: Un entier type décimal représentant les
15            ↪ valeurs sérialisées.
16    """
17    # gestion d'erreurs
18    assert 0 <= n1 <= 500, "n1 doit être compris entre 0 et
19        ↪ 500"
20    assert 0 <= n2 <= 500, "n2 doit être compris entre 0 et
21        ↪ 500"
22    assert 0 <= n3 <= 500, "n3 doit être compris entre 0 et
23        ↪ 500"
24    assert isinstance(b1, bool), "b1 doit être un booléen"
25    assert isinstance(b2, bool), "b2 doit être un booléen"
26
27    b1 = int(b1) # si b1 est True, on le convertit en 1,
28        ↪ sinon en 0
29    b2 = int(b2)
30
31    # Sérialisation des valeurs en un seul entier
32    serialisation = (n1 << 20) | (n2 << 11) | (n3 << 2) |
33        ↪ (b1 << 1) | b2
34    return serialisation
```

Listing 1: Fonction de Sérialisation

Désérialisation

L'algorithme de désérialisation utilise les décalages de bits et les opérations AND bit à bit pour extraire les valeurs encodées.

```
1 def désérialisation(serialisation):
2     """
3     Désérialise un entier en trois nombres et deux booléens.
4
5     Args:
6         serialisation (int): Un entier représentant les
7             ↪ valeurs sérialisées.
8
9     Returns:
10        tuple: Un tuple contenant trois nombres (n1, n2,
11            ↪ n3) et deux booléens (b1, b2).
12    """
13    # Extraction des valeurs à partir de l'entier sérialisé
14    n1 = (serialisation >> 20) & 0b11111111 # on prend les
15        ↪ 9 bits de poids faible
16    n2 = (serialisation >> 11) & 0b11111111
17    n3 = (serialisation >> 2) & 0b11111111
18    b1 = (serialisation >> 1) & 1
19    b2 = serialisation & 1
20
21    return n1, n2, n3, bool(b1), bool(b2)
```

Listing 2: Fonction de Désérialisation

Exemple d'utilisation

Voici un exemple d'utilisation des fonctions de sérialisation et de désérialisation.

```
1 def main():
2     """
3     Fonction principale pour tester la sérialisation et la
4         ↪ désérialisation.
5     """
6     # Définition des valeurs à sérialiser
7     n1 = 5
8     n2 = 500
9     n3 = 0
10    b1 = True
11    b2 = False
12
13    # Affichage des valeurs initiales
14    print(n1, n2, n3, b1, b2)
15
16    # Sérialisation des valeurs
17    serialized_value = serialisation(n1, n2, n3, b1, b2)
18    print(serialized_value)
19
20    # Désérialisation des valeurs
21    deserialized_values = deserialisation(serialized_value)
22    print(deserialized_values)
23
24    if __name__ == "__main__":
25        main()
```

Listing 3: Exemple d'Utilisation

```
1 >>> 5 500 0 True False
2 >>> 6266882 # entier sérialisé (en décimal)
3 >>> (5, 500, 0, True, False)
```

Listing 4: Résultat de l'exemple d'utilisation

Explication des décalages de bits

```
1 n1: 9 bits (0 à 500)
2 n2: 9 bits (0 à 500)
3 n3: 9 bits (0 à 500)
4 b1: 1 bit (True ou False)
5 b2: 1 bit (True ou False)
6
7 Structure de l'entier encodé (29 bits au total):
8 -----
9 | n1 (9 bits) | n2 (9 bits) | n3 (9 bits) | b1 (1 bit) | b2
   ↪ (1 bit) |
```

```

10 -----
11
12 Décalages de bits:
13 - n1 << 20: n1 est décalé de 20 bits vers la gauche pour
14   ↳ occuper les bits 20 à 28.
15 - n2 << 11: n2 est décalé de 11 bits vers la gauche pour
16   ↳ occuper les bits 11 à 19.
17 - n3 << 2 : n3 est décalé de 2 bits vers la gauche pour
18   ↳ occuper les bits 2 à 10.
19 - b1 << 1 : b1 est décalé de 1 bit vers la gauche pour
20   ↳ occuper le bit 1.
21 - b2      : b2 occupe le bit 0.
22
23 Exemple:
24 Supposons que n1 = 100, n2 = 200, n3 = 300, b1 = True, b2 =
25   ↳ False.
26
27 1. Conversion en binaire:
28   n1 = 100  -> 0001100100 (9 bits)
29   n2 = 200  -> 0011001000 (9 bits)
30   n3 = 300  -> 0100101100 (9 bits)
31   b1 = True -> 1 (1 bit)
32   b2 = False -> 0 (1 bit)
33
34 2. Décalages de bits:
35   n1 << 20: 0001100100 << 20 ->
36   ↳ 00011001000000000000000000000000
37   n2 << 11: 0011001000 << 11 ->
38   ↳ 00000000000011001000000000000000
39   n3 << 2 : 0100101100 << 2 ->
40   ↳ 000000000000000000000001001011000000
41   b1 << 1 : 1 << 1 ->
42   ↳ 0000000000000000000000000000000010
43   b2      : 0 ->
44   ↳ 0000000000000000000000000000000000
45
46 3. Combinaison avec OR bit à bit (|):
47   00011001000000000000000000000000
48   | 00000000000011001000000000000000
49   | 000000000000000000000001001011000000
50   | 0000000000000000000000000000000010
51   | 0000000000000000000000000000000000
52   -----
53   00011001000110010001001011000010
54
55 4. Résultat final:
56   Encodé = 00011001000110010001001011000010 (en binaire)
57   Encodé = 10569634 (en décimal)

```

Listing 5: Explication des décalages de bits