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.

```python
def serialisation(n1, n2, n3, b1, b2):
    """
    Sérialise trois nombres et deux booléens en un seul entier.

    Args:
        n1 (int): Un nombre entier compris entre 0 et 500.
        n2 (int): Un nombre entier compris entre 0 et 500.
        n3 (int): Un nombre entier compris entre 0 et 500.
        b1 (bool): Un booléen (True ou False).
        b2 (bool): Un booléen (True ou False).

    Returns:
        int: Un entier type décimal représentant les valeurs sérialisées.
    """
    # gestion d'erreurs
    assert 0 <= n1 <= 500, "n1 doit être compris entre 0 et 500"
    assert 0 <= n2 <= 500, "n2 doit être compris entre 0 et 500"
    assert 0 <= n3 <= 500, "n3 doit être compris entre 0 et 500"
    assert isinstance(b1, bool), "b1 doit être un booléen"
    assert isinstance(b2, bool), "b2 doit être un booléen"

    b1 = int(b1) # si b1 est True, on le convertit en 1, sinon en 0
    b2 = int(b2)

    # Sérialisation des valeurs en un seul entier
    serialisation = (n1 << 20) | (n2 << 11) | (n3 << 2) | (b1 << 1) | b2
    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.

```python
def deserialisation(serialisation):
    """
    Désérialise un entier en trois nombres et deux booléens.

    Args:
        serialisation (int): Un entier représentant les
            ↪ valeurs sérialisées.

    Returns:
        tuple: Un tuple contenant trois nombres (n1, n2,
            ↪ n3) et deux booléens (b1, b2).
    """
    # Extraction des valeurs à partir de l'entier sérialisé
    n1 = (serialisation >> 20) & 0b111111111 # on prend les
        ↪ 9 bits de poids faible
    n2 = (serialisation >> 11) & 0b111111111
    n3 = (serialisation >> 2) & 0b111111111
    b1 = (serialisation >> 1) & 1
    b2 = serialisation & 1

    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.

```python
def main():
    """
    Fonction principale pour tester la sérialisation et la
        ↪ désérialisation.
    """
    # Définition des valeurs à sérialiser
    n1 = 5
    n2 = 500
    n3 = 0
    b1 = True
    b2 = False

    # Affichage des valeurs initiales
    print(n1, n2, n3, b1, b2)

    # Sérialisation des valeurs
    serialized_value = serialisation(n1, n2, n3, b1, b2)
    print(serialized_value)

    # Désérialisation des valeurs
    deserialized_values = deserialisation(serialized_value)
    print(deserialized_values)

if __name__ == "__main__":
    main()
```

Listing 3: Exemple d'Utilisation

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

Listing 4: Résultat de l'exemple d'utilisation

Explication des décalages de bits

```
n1: 9 bits (0 à 500)
n2: 9 bits (0 à 500)
n3: 9 bits (0 à 500)
b1: 1 bit (True ou False)
b2: 1 bit (True ou False)

Structure de l'entier encodé (29 bits au total):
-------------------------------------------------
| n1 (9 bits) | n2 (9 bits) | n3 (9 bits) | b1 (1 bit) | b2
    ↪ (1 bit) |
```

```
------------------------------------------------

Décalages de bits:
- n1 << 20: n1 est décalé de 20 bits vers la gauche pour occuper les bits 20 à 28.
- n2 << 11: n2 est décalé de 11 bits vers la gauche pour occuper les bits 11 à 19.
- n3 << 2 : n3 est décalé de 2 bits vers la gauche pour occuper les bits 2 à 10.
- b1 << 1 : b1 est décalé de 1 bit vers la gauche pour occuper le bit 1.
- b2      : b2 occupe le bit 0.

Exemple:
Supposons que n1 = 100, n2 = 200, n3 = 300, b1 = True, b2 = False.

1. Conversion en binaire:
   n1 = 100  -> 0001100100 (9 bits)
   n2 = 200  -> 0011001000 (9 bits)
   n3 = 300  -> 0100101100 (9 bits)
   b1 = True -> 1 (1 bit)
   b2 = False -> 0 (1 bit)

2. Décalages de bits:
   n1 << 20: 0001100100 << 20 -> 00011001000000000000000000000000
   n2 << 11: 0011001000 << 11 -> 00000000000110010000000000000000
   n3 << 2 : 0100101100 << 2  -> 00000000000000000001001011000000
   b1 << 1 : 1 << 1           -> 00000000000000000000000000000010
   b2      : 0                -> 00000000000000000000000000000000

3. Combinaison avec OR bit à bit (|):
   00011001000000000000000000000000
 | 00000000000110010000000000000000
 | 00000000000000000001001011000000
 | 00000000000000000000000000000010
 | 00000000000000000000000000000000
 ----------------------------------
   00011001000110010001001011000010

4. Résultat final:
   Encodé = 00011001000110010001001011000010 (en binaire)
   Encodé = 10569634 (en décimal)
```

Listing 5: Explication des décalages de bits