Dépassement sur les entiers

Gérer la détection de dépassement sur les **entiers naturels** (non signés, dans ℕ) est assez facile en addition. Il suffit de vérifier que a + b donne bien un résultat supérieur à a ou à b :

```
def detection_naturels(a,b) :
      if (a+b) < a :
            return False
      else :
            return True
```

Pour les **entiers relatifs** (signés, ceux dans ℤ), c'est un peu plus délicat.

I – Addition et dépassement de capacité : cas des entiers positifs

Le problème vient du fait qu'avec un entier signé (sur un octet par exemple), on peut avoir dépassement des capacités de cacul sans pour autant avoir de retenue hors de l'octet.

Pour rappel, on peut compter jusqu'au plus grand nombre encodable sur un octet : **0111 1111**, soit _____.

1) Exemple de dépassement de capacité de calcul SANS dépassement de l'octet en lui-même

Ainsi, si on calcule **0111 1111**$_2$
+ **0000 0001**$_2$, on obtient **1000 0000**$_2$

Le bit de signe de la somme est à 1, alors qu'on additionne deux entiers signés positifs !

Qu'encode N = **1000 0000** si on considère un entier signé sur un octet ?

→ Bit de signe à 1 : c'est un nombre _____.

→ Pour le trouver : Inversion des bits : **0111 1111** puis rajout de **1**

→ On obtient 1000 0000 qu'on doit lire comme un non signé : +128.

→ CONCLUSION : On a donc encodé **-128** par N = **1000 0000**

Si utilise des integers signés sur un octet, on se retrouve donc avec 127 + 1 qui donne -128 !

2) Exemple de dépassement de capacité de calcul SANS dépassement de l'octet en lui-même

	Signe	64	32	16	8	4	2	1
Retenue ?								
+65	0	**1**	0	0	0	0	0	**1**
+66	0	**1**	0	0	0	0	**1**	0
Total ?								

On constate donc qu'on peut détecter un dépassement des capacités de calcul assez facilement en tant qu'humain : sur un octet, la somme des a+b ne doit pas dépasser +127.

3) Conclusion :

Mais comment la machine peut-elle faire facilement pour détecter un dépassement ?

II – Addition et dépassement de capacité : cas des entiers négatifs

1) Exemple de dépassement de l'octet SANS dépassement des capacités de calcul

Pour les entiers relatifs négatifs, c'est encode pire ! L'addition provoque nécessairement un dépassement hors de l'octet, dépassement qu'on met d'ailleurs de côté puisqu'il est inhérent à l'addition de nombres négatifs.

	Signe	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Retenue ?								
-6	1	1	1	1	1	0	1	0
-6	1	1	1	1	1	0	1	0
Total ?								

Vérifions que cela donne bien -12 :

→ Bit de signe à 1 : c'est un nombre _____.

→ Pour le trouver : Inversion des bits : _____ puis rajout de **1**

→ On obtient _____ qu'on doit lire comme un non signé : _____

→ CONCLUSION : On a donc encodé _____ par N = `1111 0100`

La vérité sur le complément à 2 (en réalité, complément à 2^N) : Hors programme ?
On nomme cette technique Complément à 2 car pour on peut trouver le nombre négatif en calculant

2^N – nombre lu comme un entier non signé.

Exemple : `1111 0100` sur un octet (8 bits) représente en entier non signé 128+64+32+16+4, soit 244.
Si on calcule 2^8 – 244 = 256 – 244, on obtient 12. L'octet `1111 0100` encode donc **-12** !

2) Exemple de dépassement de l'octet AVEC dépassement des capacités de calcul

	Signe	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Retenue ?								
-6	1	1	1	1	1	0	1	0
-126	1	0	0	0	0	0	1	0
Total ?								

3) Conclusion :

Mais comment la machine peut-elle faire facilement pour détecter un dépassement ?

III – Logarithme base 10 (hors nsi)

La fonction logarithme base 10 peut être vue en première approximation comme une fonction qui renvoie la puissance de 10 d'un nombre qu'on écrirait sous la forme `N = 10 ^ X`.

```
>>> import math
>>> math.log10(100)
2.0
```

Ainsi :

$\log_{10}(0{,}01) = \log_{10}(10^{-2}) =$ _____ $\log_{10}(10) = \log_{10}(10^{1}) =$ _____

$\log_{10}(0{,}1) = \log_{10}(10^{-1}) =$ _____ $\log_{10}(100) = \log_{10}(10^{2}) =$ _____

$\log_{10}(1) = \log_{10}(10^{0}) =$ _____ $\log_{10}(1000) = \log_{10}(10^{3}) =$ _____

La fonction logarithme est définie dans $\mathbb{R}^*_+$, c'est à dire l'intervalle `] 0 ; +∞ ]`.

On peut également rechercher des valeurs qui ne tombent pas juste. Par exemple pour 200 :

$\log_{10}(200) =$ **2.3010**… d'où $200 = 10^{2.3010\ldots}$

En Python, on peut donc obtenir le nombre de chiffres nécessaires à l'écrire d'un nombre en base 10 :

```
>>> import math
>>> x = 200
>>> chiffres = int(1 + math.log10(x) )    # int permet ici d'arrondir à l'inf
>>> chiffres
3
```

IV – Logarithme base 2 (hors nsi)

La fonction logarithme base 2 peut être vue en première approximation comme une fonction qui renvoie la puissance de 2d'un nombre qu'on écrirait sous la forme `N = 2 ^ X`.

```
>>> import math
>>> math.log2(8)
3.0
```

Ainsi :

$\log_2(0{,}25) = \log_2(2^{-2}) =$ _____ $\log_2(2) = \log_2(2^{1}) =$ _____

$\log_2(0{,}5) = \log_2(2^{-1}) =$ _____ $\log_2(4) = \log_2(2^{2}) =$ _____

$\log_2(1) = \log_2(2^{0}) =$ _____ $\log_2(8) = \log_2(2^{3}) =$ _____

La fonction logarithme est définie dans $\mathbb{R}^*_+$, c'est à dire l'intervalle `] 0 ; +∞ ]`.

En Python, on peut donc obtenir le nombre de chiffres nécessaires à l'écrire d'un nombre en base 2 :

```
>>> import math
>>> x = 9
>>> bin(x)
'0b1001'
>>> chiffres = int(1 + math.log2(x) )
>>> chiffres
4
```

→ VOIR LE COURS EN LIGNE (https://www.infoforall.fr/act/donnees/depassement-des-entiers-relatifs/)
pour obtenir un code permettant de tracer des courbes.

V – Prévisions des dépassements par multiplication

Nombre de bits nécessaires pour stocker **`a x b`**

Soit N_A, le nombre de bits nécessaires au stockage du nombre a.
Soit N_B, le nombre de bits nécesaires au stockage du nombre b.

Dans le cas le plus défavorable, il faudra N = _____ bits pour stocker correctement `a x b`.

Répondre (avec JUSTIFICATIONS) aux questions suivantes :

Combien faut-il de bits pour stocker la valeur `a = 100` en base 2 ? Répondre en non signé et en signé.

Combien faut-il de bits pour stocker la valeur `b = 50` en base 2 ? Répondre en non signé et en signé.

Si on considère qu'il faut 7 bits pour stocker la valeur 100 et 6 bits pour stocker la valeur 50, la multiplication risque-t-elle de créer un problème de dépassement sur 16 bits dans le cas d'un entier non signé ? dans un entier signé ?

Conclusion :

Mais comment la machine peut-elle faire facilement pour détecter un dépassement ?

C'est le moment de signaler qu'il y a un ensemble d'équivalences de mots anglais-français dans la partie NSI, dans les articles.

Retenue en français donne **Carry** en anglais.

Dépassement en français donne **Overflow**.