Boucle Tant que

Nous avons commencé à écrire un **algorithme** pour convertir un nombre entier en binaire

Dans cet algorithme certains passages compréhensibles par un humain doivent être précisées

Par exemple *Extraire la plus grande puissance de 2 contenue dans l'entier N*

Une des briques des langages de programmation nous permet de résoudre ce problème, c'est la boucle Tant que (while)

1 Boucle Tant que

La **donnée** de notre problème est un entier naturel `N` par exemple 124

Le **résultat** de notre algorithme est la plus grande puissance de 2 contenue dans `N` ici 64

Comment faire ça ?

On va initialiser une variable `P` à 1 et **Tant que** `P` reste inférieur ou égal à `N` on va exécuter `P` $\leftarrow$ `P` *2

On va mettre en forme cet algorithme

Algorithme 1 : La plus grande puissance de 2 contenue dans un entier

Données : un entier naturel `N`
Résultat : la plus grande puissance `P` de 2 contenue dans `N`

```
1 début
2 |  P ← 1
3 |  tant que  P <= N faire
4 |  |  P ← P *2
5 |  fin
6 |  P ← P //2
7 fin
```

Commentaires

1. **Il est très important de s'assurer qu'un programme contenant une boucle Tant que, s'arrête au bout d'un moment**

   Autrement dit un algorithme contenant une boucle **Tant que** nécessite plus de rigueur qu'un algorithme ne contenant pas de boucle **Tant que**

   Ici la boucle s'arrête car car à force de multiplier la valeur de `P` par 2, `P` va croitre et dépasser tout nombre entier fixé au préalable et dépasser la valeur de `N`.

2. Mais comment être sûr que `P` contient à la fin pour tous les `N` possibles la plus grande puissance de 2 contenue dans `N` ? On peut se convaincre en faisant quelques tests mais on aimerait avoir une sorte de **preuve** que c'est toujours vraie

3. En effet à la sortie de boucle P $>$ N donc en divisant par 2 on se retrouve avec la valeur de P juste inférieure ou égale à N

On traduit cet **algorithme** dans le langage Python par une fonction

```
def plus_grande_puissance(n):
    p = 1
    while p <= n:
        p = p *2
    return p // 2
```

Exercices

Ex 1

1. Ecrire un algorithme d'abord en **français** puis sous une forme un peu plus structurée, qui permet d'obtenir le nombre de chiffres d'un entier naturel écrit en base 10
2. Définir une fonction Python `nb_chiffres(n)` qui retourne le nombre de chiffres (on n'utilisera pas `len(str(n))`)

Ex 2

1. Ecrire un algorithme d'abord en **français** puis sous une forme un peu plus structurée, qui permet d'obtenir le nombre de bits contenue dans la représentation binaire d'un entier naturel écrit en base 10 par exemple si n = 7 alors $7 = (111)_2$ est représenté par 3 bits
2. Définir une fonction Python `nb_bits(n)` qui retourne le nombre de bits de la représentation binaire de n

Ex 3

Ecrire une fonction Python `exposant_2(n)` qui renvoie l'exposant de la plus grande puissance de 2 contenue dans l'entier naturel `n`

Par exemple `exposant_2(10)` renvoie 3

Ex 4

Si on multiplie un nombre $x > 1$ par lui-même "suffisamment de fois" alors on peut dépasser n'importe quel nombre A fixé au préalable

Autrement dit

Si $x > 1$ alors pour tout $A > 0$ il existe $n \in \mathbb{N}$ tel que $x^n > A$

Ecrire une fonction `seuil(x,A)` qui renvoie `n` étant donnés `x > 1` et `A > 0` avec $x^n > A$

Par exemple

```
>>> seuil(2, 1024)
11
```

Ex 5

Si on multiplie un nombre $x < 1$ par lui-même "suffisamment de fois" alors on obtenir une valeur inférieure à n'importe quel nombre A fixé au préalable

Autrement dit

Si $x < 1$ alors pour tout $0 < A < 1$ il existe $n \in \mathbb{N}$ tel que $x^n < A$

Ecrire une fonction `seuil2(x,A)` qui renvoie `n` étant donnés `x < 1` et `A > 0` avec $x^n < A$

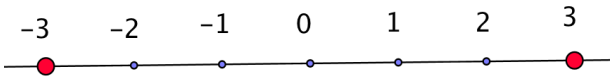
Par exemple

```
>>> seuil2(0.5, 0.001)
10
```

Ex 6

Un jeton part de l'origine et se déplace aléatoirement ainsi : Si le lancer d'une pièce a donné **Pile** alors le jeton se déplace d'une unité vers la droite sinon il se déplace d'une unité vers la gauche et le déplacement s'arrête lorsque l'abscisse du jeton est 3 ou -3. Il s'agit de compter **le nombre moyen** de lancers de la pièce par déplacement

1. Compléter l'algorithme suivant (on dira que Pile est traduit par 0 et Face par 1)



Algorithme 2 : Déplacement du jeton

Données : Un jeton à l'origine, un intervalle gradué [-3 ;3] et un générateur de nombres aléatoires

Résultat : Le nombre de lancers de la pièce

```
1  début
2      position ← 0
3      nbLancers ← 0
4      tant que ..... faire
5          piece ← entierAleatoire(0,1)
6          .....
7          .....
8          .....
9          .....
10     fin
11     afficher(nbLancers)
12 fin
```

2. Traduire l'algorithme par une fonction Python `nb_lancers()`
3. Exécuter un "grand nombre de fois" , par exemple 1000 fois la fonction précédente et calculer la moyenne des nombres de lancers.

   N'est ce pas remarquable ?Faire une **conjecture** et tester la

Ex 7

Dans le Jeu **Devine un nombre** le joueur doit deviner un nombre choisi au hasard par l'ordinateur entre 0 et 1000

A chaque tour le joueur propose une solution et l'ordinateur répond si la solution proposée est plus petite ou plus grande que le nombre tiré au hasard au départ

Ecrire un programme permettant de jouer à ce jeu

Ex 8

Pour pouvoir écrire en Python une fonction `binaire(n)` qui renvoie l'écriture binaire de `n` sous la forme d'une chaîne de caractères il nous faut travailler **la concaténation de chaînes de caractères**

1. Exécutez à la console

```
>>> chaine = "1"
>>> chaine = chaine + "2"
>>> chaine
?
```

   Qu'obtenez vous ?

2. Exécutez à la console

```
>>> chaine = "1"
>>> chaine = "2" +chaine
>>> chaine
?
```

   Qu'obtenez vous ?

3. Ecrire une fonction `binaire(n)` qui renvoie l'écriture binaire de `n` sous la forme d'une chaîne de caractères

Ex 9

Etant donné un nombre entier naturel a et un nombre entier naturel strictement positif b il existe un unique entier naturel q et un unique entier naturel $0 \leqslant r < b$ tel que

$a = bq + r$ (division euclidienne de a par b)

Voici un algorithme pour calculer q et r étant donnés a et b

Algorithme 3 : Division euclidienne

Données : un entier naturel a, un entier naturel $b > 0$
Résultat : deux entiers q et r tel que $a = bq + r$

```
1 début
2 |  q ← 0
3 |  r ← a
4 |  tant que  r >= b faire
5 |  |  r ← r - b
6 |  fin
7 |  q ← q + 1
8 fin
```

Compléter la fonction Python suivante `div(a,b)` (que renvoie la fonction ?)

```
def div(a,b):
    q = 0
    r = a
    .........
    return q,r
```

Ex 10

Quelles sont les briques élémentaires d'un langage de programmation ?

Ex 11

Dès l'antiquité on connaissait un algorithme pour calculer la racine carrée d'un nombre a > 1

Cet algorithme est de nature géométrique. On part d'un rectangle de longueur a et de largeur 1. Ce rectangle est d'aire a. Ensuite on va créer une suite de rectangles "de plus en plus carrés", toujours d'aire a. Autrement dit un des côtés de ces rectangles, lorsqu'on a suffisamment calculé, aura une valeur proche de $\sqrt{a}$

Algorithme 4 : Racine carrée de a > 0

Données : a un nombre positif, un seuil
Résultat : Une valeur approchée de la racine carrée de a

```
1  début
2      // On part d'un rectangle d'aire a
3      // de longueur L, de largeur w
4      // et tel que Lxw = a
5      L ← a
6      w ← 1
7      tant que  abs(L-w) > seuil faire
8          // la largeur w prend pour valeur
9          // la moyenne de w et de L
10         w ← (L + w) / 2
11         // L fois w est égal à a
12         L ← 2 / w
13     fin
14 fin
```

Ecrire une fonction `racine(a)` qui renvoie une valeur approchée de la racine carrée de a

Ex 11

La boucle tant que est elle vraiment nécessaire ? Est il possible de toujours traduire un programme avec une boucle while en un programme avec une boucle for et un if ?