Concours blanc n°1 : Corrigé

Exercice 1. **Multiplication du paysan russe.**

(1) On obtient les résultats suivants :

x	y	p
20	23	0
10	46	0
5	92	92
2	184	92
1	368	460

On obtient donc bien comme attendu $20 \times 23 = 460$.

(2) En appliquant l'algorithme donné on écrit naturellement le programme suivant :

```
def produit(x,y) :
  if x\%2 == 1 :
    p = y
  else :
    p = 0
  while x > 0 :
    x = x // 2
    y = 2 * y
    if x\%2 == 1 :
    p = p + y
  return p
```

(3) A chaque itération, x est divisé par deux, donc on peut en déduire un nombre d'itérations égal à $\log_2(x)$ (une manière de s'en convaincre est de voir que remplacer x par $2x$ ne rajoute qu'une seule itération).

(4) En adaptant le programme précédent, on obtient :

```
def produit2(x,y) :
  if x == 0 :
    return 0
  elif x\%2 == 1 :
    return (y + produit2(x//2,2*y))
  else :
    return produit2(x//2,2*y)
```

Cet algorithme est aussi efficace que l'algorithme itératif.

(5) Le programme itératif se termine puisque à chaque itération, x est remplacé par $x//2$ qui est un entier strictement inférieur. On s'assure donc ainsi que cette suite d'entiers strictement décroissante atteint nécessairement 0 qui est le cas de sortie de la boucle *while*.

Le programme récursif se termine exactement pour la mëme raison puisque que le premier argument de la fonction décroit strictement à chaque appel de la fonction, on tombera nécessairement sur le cas de base dans lequel cet argument est nul.

(6) Si l'on appelle n le nombre d'étapes et $\epsilon_k = x_k\%2$ le reste dans la division euclidienne du $k$ième membre de la suite x_n par deux, alors on a par construction de l'algorithme $p = p_n = \sum_{k=1}^{n} \epsilon_k y_k$.

De plus, on sait que pour tout k, $y_k = 2^{k-1}y$, donc $p = \sum_{k=1}^{n} \epsilon_k 2^{k-1} y$.

Toutefois, l'écriture en base 2 de x est telle que $x = \sum_{k=1}^{n} \epsilon_k 2^{k-1}$. En effet $x = x_1 = 2x_2 + \epsilon_1 = 2(2x_3 + \epsilon_2) + \epsilon_1 = \dots$.

On trouve donc bien que :

$$p = \sum_{k=1}^{n} \epsilon_k 2^{k-1} y = \left(\sum_{k=1}^{n} \epsilon_k 2^{k-1} \right) y = x \times y.$$

Exercice 2. **Génération aléatoire d'une configuration de particules.**

(1) La ligne 9 affecte à la variable `p` une valeur aléatoire entre 0 et `L`.

(2) Le paramètre `c` représente l'abscisse du centre d'une particule qu'on souhaite ajouter à la configuration.

(3) La ligne 3 sert à exclure une particule qui entrerait en collision avec les parois.

(4) Les lignes 4 et 5 vérifient que la particule de centre `c` n'entre en collision avec aucune des particules déjà placées. Pour cela, elle parcourt la liste des particules déjà placées et vérifie que la distance entre les centres des particules est supérieure ou égale à `2*R`.

(5) La fonction `possible` vérifie qu'on peut ajouter la particule de centre `c` à la configuration déjà construite (`res`) sans collisions avec les bords du récipient ou avec les particules déjà en place.

(6) On peut exclure les rejets de la ligne 3 en générant directement un nombre aléatoire entre `R` et `L-R` :

```
p = (L - 2*R)*random.random() + R
```

(7) Il est impossible de placer une nouvelle particule dans cette configuration. La fonction `placement` va continuellement proposer des positions pour la quatrième particule, mais celles-ci vont systématiquement être rejetées par la fonction `possible`. On va alors se retrouver dans une situation de boucle infinie.

(8) Si le nombre de particules à placer est très inférieur au nombre maximal théorique de particules pouvant être placées (autrement dit, chaque particule a beaucoup de place disponible), la probabilité de collision entre deux particules au moment du placement est très faible. La fonction `possible` renvoie presque toujours `True`, le nombre d'itérations dans la boucle `while` est donc approximativement N.

Par ailleurs, chaque appel de la fonction `possible` doit effectuer un parcours de la liste `res` contenant les particules déjà placées. Cette liste s'allonge au fur et à mesure de l'exécution de la fonction, depuis une liste vide jusqu'à une liste de longueur $N-1$.

Au final, la complexité de la fonction `placement` est

$$O(0+1+\cdots+N-1) = O(N^2).$$

(9)

```
7     res = []
8     while len(res) < N:
9         p = (L - 2*R)*random.random() + R
10        if possible(p): res.append(p)
11        else: res = []
12    return res
```

(10) Une façon (pas la seule) dont on peut implémenter cet algorithme est de trier la liste des particules virtuelles afin de les placer de la gauche vers la droite. De cette façon, chaque particule réelle que l'on place est immédiatement positionnée à son emplacement définitif.

Si on place N particules sur un segment de longueur L, l'espace disponible est $L-2NR$ (puisque chaque particule occupe une longueur $2R$). On peut, dès le départ, tenir compte du fait que toutes les abscisses doivent être augmentées de R car la plus petite abscisse possible est R.

Ensuite, pour chaque particule virtuelle en commençant par celle la plus à gauche, on la transforme en particule réelle, puis on ajoute $2R$ à l'abscisse de toutes les particules virtuelles restantes.

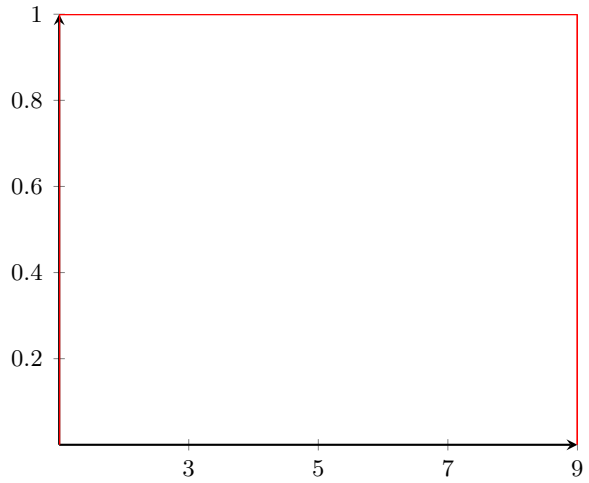
```
1    def placementrapide(N:int, R:float, L:float) -> list:
2        l = L - 2*N*R
3        if l<0: raise ValueError("Espace insuffisant")
4        resvirt = []     # liste des particules virtuelles
5        res = []         # liste des particules réelles
6        for k in range(N):
7            resvirt.append(l*random.random() + R)
8        resvirt.sort()
9        while resvirt != []:
10           res.append(resvirt.pop(0))
11           for k in range(len(resvirt)):
12               resvirt[k] = resvirt[k] + 2*R
13       return res
```

(11) La complexité de la fonction `placementrapide` est $O(N^2)$. En effet, pour chaque particule que l'on souhaite placer, il faut faire

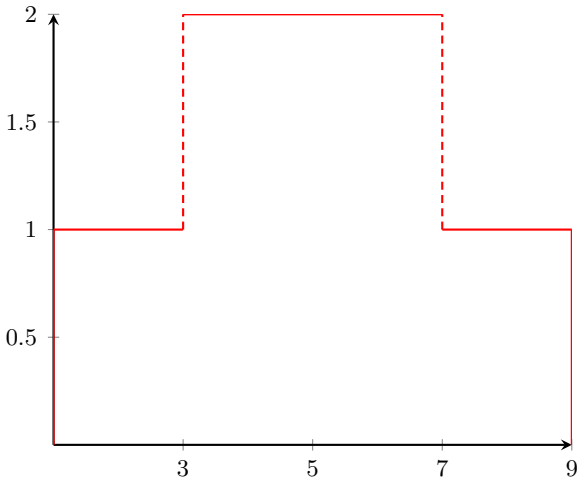
un passage dans la liste des particules virtuelles dont la longueur varie de 0 à N. En toute rigueur, il faudrait tenir compte de la complexité de la fonction de tri, mais le coût de celle-ci ne dépasse pas $O(N^2)$ (on verra en classe qu'elle est en $O(N \log N)$).

On pourrait en conclure que, le coût étant essentiellement le même que celui de la fonction `classement`, le nom « placement rapide » semble quelque peu usurpé. Ce serait oublier que la complexité de la fonction `classement` augmente nettement (au point de frôler la boucle infinie dans certains cas) dès que N approche de N_{max}. C'est dans ce type de situations que la fonction `classementrapide` se révèle comparativement la plus performante.

(12) Pour $N = 1$, on a une unique particule dont l'abscisse peut occuper uniformément l'intervalle $[1; 9]$ (impossible d'aller plus à gauche ou plus à droite sans entrer en collision avec les bords).



Pour $N = 2$, il faut voir que la particule située la plus à gauche ne peut pas dépasser l'abscisse 7 puisqu'il faut laisser la place pour une autre particule plus à droite. De même, la particule de droite possède pour abscisse minimale 3.



Pour $N = 5$, il n'y a en fait qu'une seule configuration possible, celle où les particules occupent les abscisses 1, 3, 5, 7 et 9. Notons que c'est typiquement un cas où la fonction `placement` mettrait un temps très long à répondre.