Exercice 1

On peut le voir de deux façons :

Ou bien on prend la relation de récurrence telle qu'elle est donnée, et les indices de la boucle doivent varier de 0 à $n-1$:

```
def u(n):
    x = 1
    for k in range(n):
        x = 2*x + k
    return x
```

Ou bien on écrit que $u_n = 2u_{n-1} + (n-1)$, et on utilise alors une boucle dont l'indice varie de 1 à n :

```
def u(n):
    x = 1
    for k in range(1, n+1):
        x = 2*x + (k-1)
    return x
```

Exercice 2

On utilise une boucle `while` et une variable `cpt` qui compte le nombre de fois qu'il faut appliquer ln.

```
def logstar(n):
    cpt = 0
    x = n
    while x > 1:
        x = log(x)
        cpt += 1
    return cpt
```

Exercice 3

Le code suivant convient :

```
def sommeSurProd(L):
    s = 0
    p = 1
    for v in L:
        s += v
        p *= v
    return s / p
```

Exercice 4

On peut construire la liste à l'aide d'une boucle et de `append` :

```
def carres(n):
    cr = []
    for k in range(n):
        cr.append((n-k)**2)
    return cr
```

Ou bien la construire en une ligne par compréhension :

```
def carres(n):
    return [(n-k)**2 for k in range(n)]
```

Exercice 5

Il suffit de regarder si $L[k] = L[k+1]$ est vérifié pour tout $k \in [\![0, n-2]\!]$ (n étant la longueur de la liste). Si cette condition n'est pas vérifiée pour un k lors de la boucle, la liste n'est pas constante et on quitte la boucle en renvoyant `False`. La fonction ne renvoie `True` que si la boucle est arrivée à son terme.

```python
def estConstante(L):
    for k in range(len(L)-1):
        if L[k] != L[k+1]:
            return False
    return True
```

Exercice 6

On stocke les 10 compteurs dans une liste `C` qu'on initialise avec des 0. Puis on parcourt L (ici par valeurs, mais on peut aussi utiliser des indices). Si une valeur v de L est un chiffre, on incrémente le compteur $C[v]$.

```python
def compteChiffres(L):
    C = [0] * 10
    for v in L:
        if 0 <= v <= 9:
            C[v] += 1
    return C
```

Exercice 7

1. (a) Il s'agit d'une simulation (classique) d'une loi binomiale de paramètres N et p.

```python
def simulX1(N, p):
    X1 = 0
    for _ in range(N):
        if random() <= p:
            X1 += 1
    return X1
```

(b) Pour $k \in [\![0, N]\!]$, la loi conditionnelle de X_2 sachant $(X_1 = k)$ est la loi binomiale de paramètres k et p.

```python
def simulX2(N, p):
    k = simulX1(N, p)
    X2 = 0
    for _ in range(k):
        if random() <= p:
            X2 += 1
    return X2
```

On peut aussi réutiliser la simulation de X_1, mais avec un paramètre k :

```python
def simulX2(N, p):
    k = simulX1(N, p)
    return simulX1(k, p)
```

2. (a) Il s'agit d'une simulation (classique) de la loi géométrique de paramètre $q = 1 - p$.

```python
def simulTk(p):
    rg = 1
    while random() <= p:  # ou bien while random() >= 1-p
        rg += 1
    return rg
```

(b) L'algorithme permettant de déterminer un maximum est le suivant : on stocke dans une variable `maxtmp` la meilleure valeur connue. On initialise `maxtmp` à 0 (on fera forcément mieux). Puis on simule les variables $T_1, \ldots, T_N$. À chaque fois qu'une de ces simulation dépasse `maxtmp`, elle remplace `maxtmp`.

```python
def simulT(N, p):
    maxtmp = 0
    for _ in range(N):
        Tk = simulTk(p)
        if Tk > maxtmp:
            maxtmp = Tk
    return maxtmp
```

(c) On calcule la moyenne de *nbsim* simulations de T :

```python
def estimEspT(N, p, nbsim):
    s = 0
```

```python
    for _ in range(nbsim):
        s += simulT(N, p)
    return s / nbsim
```