Corrigé

Piles

Exercice 1 On utilise deux piles, l'une destinée à recevoir les nombres pairs, l'autre les nombres impairs. On commence par vider la pile initiale dans ces deux piles, puis on transfère de nouveau leur contenu dans la première pile, d'abord celle contenant les nombres pairs, puis celle contenant les nombres impairs.

```
def trier(p):
    pairs = pile()
    impairs = pile()
    while not estVide(p):
        x = pop(p)
        if x % 2 == 0:
            push(x, pairs)
        else:
            push(x, impairs)
    while not estVide(pairs):
        push(pop(pairs), p)
    while not estVide(impairs):
        push(pop(impairs), p)
```

Exercice 2

a) La solution consiste à ranger dans une pile auxiliaire les éléments de p qui sont supérieurs à x, puis à placer x dans la pile, et enfin à faire revenir les éléments qui ont été stockés dans la pile auxiliaire.

```
def insere(x, p):
    aux = pile()
    while not estVide(p):
        y = pop(p)
        if x < y:
            push(y, aux)
        else:
            push(y, p)
            break
    push(x, p)
    while not estVide(aux):
        push(pop(aux), p)
```

Dans le pire des cas, tous les éléments de la pile p sont déplacés dans la pile auxiliaire donc la fonction `insere` a un coût spatial en $O(n)$ où n est la taille de la pile p.

b) On commence par transférer les éléments de p dans une pile auxiliaire, puis chacun d'eux est de nouveau transféré dans p en utilisant la fonction `insere`.

```
def tri(p):
    aux = pile()
    while not estVide(p):
        push(pop(p), aux)
    while not estVide(aux):
        insere(pop(aux), p)
```

Le transfert de la pile p vers la pile auxiliaire a une complexité en $O(n)$ puis la fonction `insere` est utilisée n fois donc la complexité temporelle de la fonction `tri` est en $O(n) + n \times O(n) = O(n^2)$: la complexité est quadratique.

Exercice 3 La solution consiste à empiler les parenthèses, crochets et accolades ouvrants dans une pile, et à chaque fois qu'on rencontre un symbole fermant, vérifier qu'il correspond au sommet de la pile, qui est alors supprimé.

```
def parenthese(S):
    p = pile()
    for c in S:
        if c in '([{':
            push(c, p)
        elif c == ')':
            if estVide(p) or pop(p) != '(':
                return False
        elif c == ']':
            if estVide(p) or pop(p) != '[':
                return False
        elif c == '}':
            if estVide(p) or pop(p) != '{':
                return False
    return estVide(p)
```

Exercice 4

a) La première permutation est engendrable par la suite d'opérations : EEEDEEDEEDDEDDEDDEDD.

La seconde ne peut être générée complètement : les opérations EEEEDEEDDDEEDD permettent de générer les entiers 4 6 5 3 8 7 mais il n'est pas possible de générer 1 car l'entier 2 se trouve situé au dessus de lui dans la pile.

b) Passons en revue les éléments de s à générer. Si l'élément $s[i]$ ne se trouve pas dans la pile lorsque son tour vient de sortir, il suffit de faire entrer tous les éléments en attente jusqu'à l'atteindre. C'est le rôle de la variable x, qui représente le premier élément en attente devant la pile.

Lorsqu'on est certain qu'il se trouve dans la pile, de deux choses l'une : ou bien il se trouve au sommet de la pile, auquel cas le processus peut se poursuivre, ou bien il ne s'y trouve pas, et dans ce cas cela montre que la permutation n'est pas engendrable.

Ceci conduit à la version suivante :

```
def genere(s):
    p = pile()
    x = 1                   # le premier élément de la séquence d'entrée
    for i in range(len(s)):
        while x <= s[i]:    # s[i] ne se trouve pas encore dans la pile
            push(x, p)
            x += 1
        y = pop(p)
        if y != s[i]:       # si s[i] n'est pas au sommet de la pile, le processus est bloqué
            return False
    return True             # tous les éléments ont pu être générés
```

Exercice 5 Il suffit de suivre pas-à-pas l'algorithme tel qu'il est décrit.

```
def evalue(lst):
    p = pile()
    for t in lst:
        if t in op_bin:
            y = pop(p)
            x = pop(p)
            push(t(x, y), p)
        elif t in op_uni:
            x = pop(p)
            push(t(x), p)
        else:
            push(t, p)
    return pop(p)
```