Exercice 1 : piles

1. On inverse l'ordre de la pile P dans la pile Q : sommet(8 - 5 - 2 -4)fond de la pile.

2.
```
def hauteur_pile(P):
    Q = creer_pile_vide()
    n = 0
    while not (est_vide(P)):
        n += 1
        x = depiler(P)
        empiler(Q, x)
    while not (est_vide(Q)):
        x = depiler(Q)
        empiler(P, x)
    return n
```

3.
```
def max_pile(P, i):
    Q = creer_pile_vide()
    j = 1
    j_maxi = 1
    maxi = depiler(P)
    empiler(Q, maxi)
    for _ in range(i-1):
        j += 1
        x = depiler(P)
        empiler(Q, x)
        if x > maxi:
            j_maxi = j
            maxi = x
    while not (est_vide(Q)):
        empiler(P, depiler(Q))
    return j_maxi
```

4.
```
def retourner(P, j):
    Q = creer_pile_vide()
    R = creer_pile_vide()
    for _ in range(j):
        empiler(Q, depiler(P))
    while not (est_vide(Q)):
        empiler(R, depiler(Q))
    while not (est_vide(R)):
        empiler(P, depiler(R))
```

5. Trier une pile de crêpes.

```
def trier(P):
    n = hauteur_pile(pile)
    for i in range(n-1):
        j_max = max_pile(P, n-i)
        retourner(P, j_max)
        retourner(P, n-i)
```

Exercice 2 : parcours de tableau

1. (a) Il faut descendre de 2 cases (n).

   (b) Seuls les déplacements vers la gauche ou vers le bas sont autorisés (pas de retour arrière), tous les chemins contiennent donc la case de départ plus 3 nouvelles cases lors des déplacements vers la droite et deux nouvelles cases lors des déplacements vers le bas (la cas d'arrivée étant atteinte), soit un total de (1+3+2) = 6 cases.

2. Les chemins possibles sont :
   — (0, 0) (0, 1) (0, 2) (0, 3) (1, 3) (2, 3) : somme = 11
   — (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (2, 3) : somme = 10
   — (0, 0) (0, 1) (0, 2) (1, 2) (2, 2) (2, 3) : somme = 14
   — (0, 0) (0, 1) (1, 1) (1, 2) (1, 3) (2, 3) : somme = 9
   — (0, 0) (0, 1) (1, 1) (1, 2) (2, 2) (2, 3) : somme = 13
   — (0, 0) (0, 1) (1, 1) (2, 1) (2, 2) (2, 3) : somme = 12
   — (0, 0) (1, 0) (1, 1) (1, 2) (1, 3) (2, 3) : somme = 10
   — (0, 0) (1, 0) (1, 1) (1, 2) (2, 2) (2, 3) : somme = 14
   — (0, 0) (1, 0) (1, 1) (2, 1) (2, 2) (2, 3) : somme = 13
   — (0, 0) (1, 0) (2, 0) (2, 1) (2, 2) (2, 3) : somme = 16 ; somme maxi !

3. (a) Tableau T' :

4	5	6	**9**
6	**6**	8	10
9	10	**15**	16

   (b) On restant sur la 1ère ligne ($i = 0$), on ne peut atteindre une case de colonne j qu'en venant de la case précédente à gauche ($j - 1$). Donc la somme des cases traversées en $(0, j)$ (càd $T'[0][j]$) est la valeur de la case $(0, j)$ (càd $T[0][j]$) plus la somme des cases atteintes à la cellule de gauche (càd $T[0][j - 1]$).
   $T'[0][j] = T[0][j] + T'[0][j - 1]$

4. De façon similaire, on ne peut atteindre une case (i, j) qu'en venant de la case à gauche $(i, j - 1)$ ou de la case supérieure $(i - 1, j)$. Donc la somme des cases traversées en (i, j) (càd $T'[i][j]$) est la valeur de la case (i, j) (càd $T[i][j]$) plus le maximum entre, d'une part la somme des cases atteintes à la cellule de gauche (càd $T[i][j - 1]$), et d'autre part la somme des cases atteintes à la cellule supérieure (càd $T[i - 1][j]$).
   $T'[i][j] = T[i][j] + max(T'[i - 1][j], T'[i][j - 1])$

   (a) Le cas de base est obtenu avec $i = j = 0$, la fonction renvoie alors directement la valeur de la case (0, 0).

(b) Dans les autres cas, on appelle récursivement la fonction en distinguant le cas particulier pour la 1ère ligne où $i = 0$, et le cas particulier pour la 1ère colonne où $j = 0$.

```python
def somme_max(T, i, j):
   if i == j == 0:
      return T[0][0]
   elif i == 0:
      return T[0][j] + somme_max(T, 0, j-1)
   elif j == 0:
      return T[i][0] + somme_max(T, i-1, 0)
   else:
      return T[i][j] + max(somme_max(T, i-1, j), somme_max(T, i, j-1))
```

(c) L'appel est `somme_max(T, n-1, p-1)` ;
Avec `n = len(T)` et `p = len(T[0]))` .

Exercice 3 arbre

1. La taille est le nombre de nœuds : $n = 9$. La hauteur est « le nombre d'étages » (avec la convention que la hauteur vaut 1 s'il y a seulement la racine) : $h = 4$.
2. (a) $D = 101$, $G = 1010$.
   (b) $13_{10} = 1101_2$: c'est le nœud I.
   (c) Pour une profondeur p, un nœud s'écrit sur p bits :
   Exemple : $p = 1 => A = 1$: 1 bit ; $p = 2 => B = 10$, $C = 11$: 2 bits.
   Donc pour les nœuds « en bas », la profondeur est égale à la hauteur et donc les nœuds sont numérotés avec h bits.
   (d) Le nombre de nœuds, càd la taille n, est forcément au moins égale à la hauteur h car on ne peut pas mettre moins de 1 nœud pour chaque profondeur (un seul fils pour chaque nœud) : donc $n_{mini} = h$.

   Par ailleurs, le nombre maximum de nœuds qu'on peut avoir pour une hauteur h donnée est obtenu en mettant le maximum de nœuds à chaque profondeur p (2 fils pour tous les nœuds). Dans cette situation, le dernier nœud en bas à droite s'écrit comme une suite de 1 et il y en a h (cf question précédente) : sa valeur est donc $2^h - 1$. Or, sa valeur est justement la taille de l'arbre n : on a donc dans ce cas : $n_{maxi} = 2^h - 1$.

   En combinant les 2 résultats précédents, on a bien montré que $h \leq n \leq 2^h - 1$.
3. (a) L'arbre est : [15, 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
   (b) Le père du nœud d'indice i a pour indice $i//2$ (pour i pair comme impair).

4. Recherche dans un arbre binaire :

```python
def recherche(arbre, element):
    i = 1
    while i <= arbre[0]:
        if arbre[i] == element:
            return True
        elif arbre[i] > element:
            i = 2 * i
        else:
            i = 2 * i + 1
    return False
```

Exercice 4 : base de données

1. (a) `num_eleve` est une clef primaire qui permet d'identifier de manière unique chaque enregistrement de la table.

   (b) `INSERT INTO seconde (langue1, langue2, classe)`
   `VALUES ("anglais", "espagnol", "2A");`

   (c) `UPDATE seconde SET langue1 = "allemand" WHERE num_eleve = "156929JJ";`

2. (a) `SELECT num_eleve FROM seconde ;` renvoie la liste de tous les `num_eleve` de la table.

   (b) `SELECT COUNT(num_eleve) FROM seconde ;` renvoie le nombre d'élèves de seconde.

   (c) `SELECT COUNT(num_eleve) FROM seconde`
   `WHERE langue1 = "allemand" OR langue2 = "allemand";`

3. (a) La clef étrangère `num_eleve` permet de référencer les enregistrements de la table `seconde`. On peut ainsi s'assurer qu'un élève décrit par sa civilité dans la table `eleve` correspond bien à un seul élève décrit par sa classe et options dans la table `seconde`.

   (b) `SELECT nom, prenom, datenaissance`
   `FROM eleve JOIN seconde USING(num_eleve)`
   `WHERE classe = "2A";`

4. Table `coordonnees` :

coordonnees
`num_eleve` (clef primaire, clef étrangère de la table `eleve`)
`adresse`
`codepostal`
`ville`
`mail`

Exercice 5 : routage

1. (a) Route de A à G avec le protocole RIP : A - C - F - G.

   (b) Table de routage de G :

Table de routage du routeur G		
Destination	Routeur suivant	Distance
A	E	3
B	E	3
C	E	2
D	E	2
E	E	1
F	F	1

2. Table de routage de A si le routeur C est en panne :

Table de routage du routeur A		
Destination	Routeur suivant	Distance
B	B	1
D	D	1
E	D	2
F	D	4
G	D	3

3. (a) Débit entre A et B : $d = 10$ Gb/s $= 10^{10}$ b/s

   Donc le coût est $c = \dfrac{10^8}{d} = \dfrac{10^8}{10^{10}} = 10^{-2} = 0,01$.

   (b) Débit entre B et D : $d = \dfrac{10^8}{c} = \dfrac{10^8}{5} = 2.10^7$ b/s $= 20$ Mb/s.

4. Route de A à G avec le protocole OSPF :
   A - D - E - G, dont le coût total est 1,011. C'est la route qui emprunte les liaisons avec les meilleurs débits ce qui permet de minimiser le coût total.
   Par exemple, la route A - C - E - G a un coût de 13, et la route A - C - F - G a une coût de 12.
   Une route passant par B est forcément plus longue que celle passant plus directement par D.