Correction du DS 0

Julien Reichert

Exercice 1

```
def pos_max(l):
    assert len(l) != 0
    rep = 0
    for i in range(1, len(l)):
        if l[i] > l[rep]:
            rep = i
    return rep
```

Exercice 2

```
def pos_dicho(l, e):
    deb, fin = 0, len(l)-1
    while deb <= fin:
        mil = (deb + fin) // 2
        if l[mil] == e:
            return mil
        elif l[mil] > e:
            fin = mil - 1
        else:
            deb = mil + 1
    raise ValueError("Introuvable")
```

Exercice 3

Solution proposée : tri par sélection en place (linéaire en affectations, quadratique en comparaisons).

```
def selection(l):
    for i in range(len(l)-1):
        indmin = i
        for j in range(i+1, len(l)):
            if l[j] < l[indmin]:
                indmin = j
        if indmin != i:
            l[i], l[indmin] = l[indmin], l[i]
```

Exercice 4

Pour prouver qu'un programme termine, on identifie les boucles conditionnelles et les récursions (s'il n'y en a pas il n'y a rien à faire), et pour chacune d'elles, on exhibe une expression, appelée *variant*, qui s'évalue en un entier naturel (il y a d'autres possibilités en théorie, mais elles ne sont pas vraiment au programme du tronc commun) **strictement** décroissant à chaque tour de boucle / appel récursif imbriqué.

Attention aux boucles `for` de Python qui peuvent être infinies si on mute l'objet parcouru. Ceci étant, ce genre de programme est à bannir dans la mesure du possible ou prouver de manière adaptée.

Un exemple de variant pour le programme suivant est la taille de la liste manipulée. C'est évidemment un entier positif, et cet entier décroît strictement à chaque tour de boucle car la méthode `pop` retire le dernier élément de la liste sur laquelle elle agit. Ainsi, la boucle n'est pas infinie et par suite la fonction termine.

```
def transfert(l, ll):
    while l != []:
        ll.append(l.pop())
```

Exercice 5

Pour convertir un entier en binaire, on procède par divisions euclidiennes successives par 2 (autrement dit des tests de parité). Cela permet d'écrire les bits en commençant par le poids faible, c'est-à-dire qu'on écrit le nombre en binaire de droite à gauche.

La conversion dans l'autre sens se fait par une simple évaluation de fonction polynomiale, en assimilant les bits à des coefficients (poids le plus faible = degré zéro) et en prenant 2 pour valeur à laquelle la fonction associée est évaluée.

Exercice 6

Dans la mesure où il n'y a pas d'arc de poids strictement négatif, l'algorithme de Dijkstra peut être mis en œuvre. Il est plus rapide et facile à faire tourner à la main que les autres (qui ne sont d'ailleurs pas à maîtriser en s'en tenant au programme officiel).

On commence par créer une liste des distances optimales depuis le sommet 0 vers tous les sommets. En l'absence de traitement initial, ces distances sont considérées comme infinies sauf celle au sommet 0 lui-même qui est à distance nulle.

On utilise aussi une structure où les sommets en attente de traitement sont stockés, où seul 0 figure initialement, et une structure où les sommets traités sont mémorisés, ce qui n'est initialement le cas d'aucun d'entre eux.

Par la suite, tant que la structure avec les sommets en attente est non vide, on réalise les étapes suivantes.

Première étape : identifier dans la structure d'attente un sommet minimisant la distance à 0, n'importe lequel en cas d'égalité, et le retirer de la structure d'attente, tout en le marquant comme traité au sens de la deuxième structure.

Deuxième étape : considérer tous les voisins du sommet en cours de traitement et étudier leur distance optimale actuelle. Si elle est infinie, cela va changer car on a identifié un chemin qui passe par le sommet en cours de traitement (lui-même acté comme accessible par sa mise en attente préalable), la distance associée est la distance au sommet traité plus le poids de l'arc permettant de considérer les deux sommets comme voisins. Le voisin est ajouté dans la structure d'attente. Si la distance optimale actuelle n'était pas infinie mais plus grande que la distance optimale au sommet en cours de traitement plus le poids de l'arc, on procède à une mise à jour au vu de l'optimisation, et sinon on ne fait rien pour ce voisin.

Une fois la structure avec les sommets en attente vide, la liste des distances optimales calculées correspond effectivement à la liste des distances optimales dans le graphe, avec la possibilité d'ajouter une information annexe pour tous sommets permettant de déduire un chemin minimisant la distance en question.

Pour le graphe proposé, le sommet 0 sera traité en premier, permettant de découvrir les sommets 1, 2 et 3. Ensuite, c'est le sommet 2, minimisant la distance, qui sera traité, et sa distance (valant 2) sera alors garantie comme minimale, permettant de voir que la distance entre 0 et 3 est finalement inférieure ou égale à 9. En attendant, c'est le sommet 1 qui est traité après 2, permettant de découvrir le sommet 5 à distance 11 de 0 a priori, mais surtout d'améliorer encore la distance entre 0 et 3, pour obtenir 7 comme valeur définitive. En effet, c'est bien le sommet 3 qui minimise la distance à 0 parmi les deux sommets en attente, et il permet de découvrir le sommet 4, à distance 8 de 0, qui sera traité dans la foulée et permettra de découvrir le sommet 6, à la même distance du sommet 0, qui lui-même améliorera la distance entre le sommet 0 et le sommet 5 pour une valeur finale de 9, puisque le traitement du sommet 5 n'aura aucun effet.