INFORMATIQUE TRONC COMMUN
DEVOIR SURVEILLÉ 2
Corrigé

1 Tri d'entiers en complément à deux

1. Donner l'intervalle représentable en complément à deux sur 11 bits.
   Corrigé
   L'intervalle représentable est $[\![-2^{10}, 2^{10}-1]\!]$, c'est-à-dire $[\![-1024, 1023]\!]$.
2. Donner la représentation en complément à deux sur 8 bits de -37.
   Corrigé
   1101 1011
3. On souhaite programmer un tri sur les représentations d'entiers en complément à deux.

   (a) La première étape est de pouvoir comparer deux représentations. Écrire une fonction `inferieur` prenant en argument deux listes de bits de même longueur, et renvoyant `True` si le nombre représenté par la première est inférieur ou égal à celui de la seconde, et `False` sinon.
   Corrigé

```
def inferieur(L1, L2):
    if L1[0] == L2[0]:
        n = len(L1)
        for i in range(1,n):
            if L1[i] < L2[i]:
                return True
            elif L1[i] > L2[i]:
                return False
        return True
    else:
        return L1[0] > L2[0]
```

   (b) En déduire une fonction `tri_complement` prenant en argument une liste de listes de bits, toutes de même longueur, et triant la liste dans l'ordre croissant des nombres représentés. On précisera quel algorithme de tri est utilisé.
   Corrigé
   On donne plusieurs solutions (une seule était demandée) :

```
#Tri sélection
def indice_minimum(L, i):
    imin = i
    for k in range(i+1, len(L)):
        if inferieur(L[k], L[imin]):
            imin = k
    return imin

def tri_complement(L):
    n = len(L)
    for i in range(0,n-1):
        imin = indice_minimum(L,i)
        L[i], L[imin] = L[imin], L[i]

# Tri par insertion
```

```
def tri_complement(L):
    n = len(L)
    for i in range(1,n):
        j = i
        while j>0 and inferieur(L[j], L[j-1]):
            L[j-1], L[j] = L[j], L[j-1]
            j = j-1

# Tri par partition-fusion
def fusion(L1, L2):
    i1 = 0
    i2 = 0
    n1 = len(L1)
    n2 = len(L2)
    L = []
    while i1 < n1 and i2 < n2:
        if inferieur(L1[i1], L2[i2]):
            L.append(L1[i1])
            i1 = i1+1
        else:
            L.append(L2[i2])
            i2 = i2+1
    return L+L1[i1:]+L2[i2:]

def tri_complement(L):
    n = len(L)
    if n <=1:
        return L
    else:
       m = n//2
       L1 = tri_complement(L[0:m])
       L2 = tri_complement(L[m:n])
    return fusion(L1,L2)

# Tri rapide
def repartition(L):
    G = []
    D = []
    n = len(L)
    pivot = L[n-1]
    for i in range(len(L) - 1):
        x = L[i]
        if inferieur(x, pivot):
            G.append(x)
        else:
            D.append(x)
    return G, pivot, D

def tri_complement(L):
    if len(L) <= 1:
        return L
    G, pivot, D = repartition(L)
    return tri_complement(G) + [pivot] + tri_complement(D)
```

(c) Donner la complexité de la fonction précédente dans le pire cas, en fonction du nombre n de listes et du nombre commun k de bits.

Corrigé

Le coût dans le pire cas de la fonction `inferieur` est en $O(k)$, lorsqu'il faut parcourir l'ensemble des deux listes. En combinant ce coût de chaque comparaison à la complexité des algorithmes de tris vue en cours, on en déduit les complexité suivantes :

- Tri par sélection : $O(kn^2)$
- Tri par insertion : $O(kn^2)$
- Tri fusion : $O(k(n\log n))$
- Tri rapide : $O(kn^2)$

2 Voyageur de commerce

Le voyageur de commerce est un problème classique de l'informatique : Étant données n villes et les distances les séparant, il s'agit de trouver le chemin le plus court passant par chaque ville, et revenant à la ville de départ.
Formellement, on se donne en entrée un graphe orienté, pondéré à poids positifs et complet (chaque couple de sommets est relié par un arc), dont les sommets forment l'ensemble $[\![0, n-1]\!]$, avec $n > 1$. On appelle **cycle hamiltonien** un chemin commençant par 0, passant par chaque sommet exactement une fois, auquel on ajoute une seconde occurrence de 0 comme sommet final. On attend alors en sortie le cycle hamiltonien de poids minimal.

2.1 Représentation du graphe

On choisit de représenter un tel graphe par une matrice d'adjacence, le coefficient d'indices i, j contenant le poids de l'arc du sommet i au sommet j.

1. Une représentation par listes d'adjacences aurait-elle permis de réduire l'espace mémoire utilisé ? Justifier.

   Corrigé

   L'espace mémoire utilisé par la matrice d'adjacence est en $O(n^2)$. Puisque le graphe à représenter est complet, chaque liste d'adjacence serait de longueur $n-1$), pour un espace mémoire total encore en $O(n^2)$. Une représentation par listes d'adjacence n'aurait donc ici pas réduit l'espace mémoire.

2. On souhaite pouvoir former une telle matrice d'adjacence à partir d'une liste de couples correspondant aux coordonnées spatiales des villes dans un plan.

   (a) Écrire une commande pour importer la bibliothèque `math`. On pourra utiliser la fonction `sqrt` de cette bibliothèque dans la suite.

   Corrigé

   ```
   from math import *
   ```

   (b) Écrire une fonction `distance` prenant en argument deux couples de flottants, et renvoyant la distance euclidienne entre eux.

   Par exemple, `distance((1.5, 2), (-1.5, 6))` doit renvoyer `5.0`

   Corrigé

   ```
   def distance(u,v):
       xu, yu = u
       xv, yv = v
       return sqrt((xu - xv)**2 + (yu - yv)**2)
   ```

   (c) Écrire une fonction `construire_graphe` prenant en argument une liste de couples de flottants `L`, et renvoyant la matrice d'adjacence correspondante. La case de coordonnées i, j de la matrice doit ainsi contenir la distance entre les villes d'indices i et j dans `L`.

   Corrigé

   ```
   def construire_graphe(L):
       n = len(L)
   ```

```
    M = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            M[i][j] = distance(L[i], L[j])
    return M
```

ou plus concisément :

```
def construire_graphe(L):
    return [[distance(u,v) for v in L] for u in L]
```

3. Dans le cas général, les graphes que l'on considère pour ce problème sont orientés, au sens où le poids entre deux sommets n'est pas forcément le même dans un sens ou dans l'autre.

   (a) Écrire une fonction `est_oriente` testant si un graphe est orienté.

   Corrigé

```
def est_oriente(G):
    n = len(G)
    for i in range(n):
        for j in range(i):
            if G[i][j] != G[j][i]:
                return False
    return True
```

   (b) Pourquoi est-ce un problème de tester l'égalité entre flottants ? Proposer une modification de la fonction précédente évitant ce problème.

   Corrigé

   Les calculs sur les flottants sont soumis à des approximations, qui peuvent fausser les programmes se basant sur des tests d'égalité entre flottants.

   Pour éviter ce problème, on peut par exemple prendre en argument un seuil d'erreur `epsilon` en entrée, et considérer que deux flottants sont égaux lorsque la valeur absolue de leur différence est inférieure à ce seuil :

```
def est_oriente(G, epsilon):
    n = len(G)
    for i in range(n):
        for j in range(i):
            if abs(G[i][j] - G[j][i]) > epsilon:
                return False
    return True
```

2.2 Approche par force brute

Une approche permettant de garantir que la solution optimale a été trouvée est de tester tous les cycles hamiltoniens, et de rechercher celui réalisant le poids minimal.

1. Écrire une fonction `poids_chemin` prenant en argument une matrice d'adjacence et un chemin, et renvoyant le poids du chemin dans le graphe représenté.

   Par exemple, si

```
G1 = [
    [0,  1,  1,  50, 50],
    [1,  0,  1,  50, 50],
    [1,  1,  0,  50, 50],
    [50, 50, 50, 0,  1],
    [50, 50, 50, 1,  0]
]
```

   et `L1 = [0, 3, 1, 2, 4, 0]`, `poids_chemin(G1,L1)` doit renvoyer 201.

   Corrigé

```
def poids_chemin(G, chemin):
    n = len(chemin)
    r = 0
    for i in range(n-1):
        u = chemin[i]
        v = chemin[i+1]
        r += G[u][v]
    return r
```

2. Donner sans justifier un cycle hamiltonien de poids minimal pour le graphe `G1`.

   Corrigé

   `[0, 1, 2, 3, 4, 0]` est un cycle hamiltonien de poids minimal égal à 103.

3. Écrire une fonction `cycle_hamiltonien` prenant en argument un graphe `G` sous forme de matrice d'adjacence et un chemin `L` sous forme de liste, et testant si `L` est un cycle hamiltonien de `G`. On demande une complexité linéaire en le nombre de sommets n. Cette fonction ne sera pas nécessaire pour la suite.

   Corrigé

   ```
   def cycle_hamiltonien(G, L):
       n = len(G)
       if len(L) != n+1 or L[0] != 0 or L[n] != 0:
           return False
       est_dans_L = [False]*n
       for x in L:
           est_dans_L[x] = True
       for b in est_dans_L:
           if not b:
               return False
       return True
   ```

   Les deux boucles, et donc la complexité totale, sont bien dans le pire cas en $O(n)$ comme demandé.

4. Pour énumérer tous les cycles hamiltoniens, on va définir une fonction **récursive** `enumerer_cycles` prenant en arguments :
   - `G` : la matrice d'adjacence du graphe considéré ;
   - `debut` : la liste des sommets formant le début des cycles à énumérer ;
   - `dispo` : une liste de booléens telle que `dispo[u]` vaut `True` si et seulement si u n'est pas dans `debut` ;
   - `resultat` : une liste dont le premier élément est le poids minimal, et le second élément est le cycle hamiltonien réalisant le poids minimal, parmi les cycles déjà énumérés.

   L'exécution de cette fonction doit énumérer tous les cycles commençant par la liste `debut`, et modifier le tableau `resultat` par effet de bord en conséquence.

   Le cas de base de la fonction est lorsque `debut` contient tous les sommets. Il suffit alors d'ajouter 0 à la fin de la liste `debut` pour obtenir le nouveau cycle hamiltonien à considérer, et de modifier `resultat` si nécessaire.

   Le cas récursif consiste, pour chaque sommet `u` disponible, à l'ajouter à `debut`, à marquer `u` comme non disponible, à énumérer récursivement tous les cycles commençant ainsi, puis à supprimer `u` de `debut` et à le marquer disponible à nouveau.

   Écrire en Python cette fonction `enumerer_cycles`. On pourra utiliser l'instruction `L.pop()` pour supprimer le dernier élément d'une liste `L`.

   Corrigé

   ```
   def enumerer_cycles(G, debut, dispo, resultat):
       n = len(G)
       if len(debut) == n:
           poids_min, chemin_min = resultat
   ```

```
            poids = poids_chemin(G, debut+[0])
            if poids < poids_min:
                resultat[0] = poids
                resultat[1] = debut+[0]
        else:
            for u in range(1,n):
                if dispo[u]:
                    dispo[u] = False
                    debut.append(u)
                    enumerer_cycles(G,debut, dispo, resultat)
                    debut.pop()
                    dispo[u] = True
```

5. En déduire une fonction `voyageur_force_brute` prenant en argument une matrice d'adjacence `G` et renvoyant le cycle hamiltonien réalisant le poids minimal, avec le poids associé. On pourra utiliser `float('inf')` pour obtenir une valeur infinie.

   Corrigé

```
def voyageur_force_brute(G):
    resultat = [float('inf'), None]
    n = len(G)
    enumerer_cycles(G,[0], [False]+[True]*(n-1), resultat)
    return resultat
```

6. Combien existe-t-il de cycles hamiltoniens ? La fonction `voyageur_force_brute` est-elle de complexité polynomiale en le nombre n de villes ?

   Corrigé

   Le premier sommet d'un cycle hamiltonien est forcément 0. Il y a ensuite $n-1$ choix pour le second sommet, puis $n-2$ pour le troisième, et ainsi de suite. Il existe donc $(n-1)!$ cycles hamiltoniens.

   La fonction `voyageur_force_brute` va énumérer chacun de ces cycles hamiltonien, sa complexité est donc supérieure à $O((n-1)!)$, donc à $O(2^n)$. La complexité de cette fonction ne peut donc pas être polynomiale.

2.3 Algorithme glouton

Afin d'obtenir un temps de calcul plus raisonnable quand le nombre de villes devient grand, on considère à présent l'algorithme glouton suivant : à chaque étape, on sélectionne, parmi les sommets qui n'ont pas encore été ajoutés au chemin, le sommet le plus proche du dernier sommet sélectionné. Quand tous les sommets ont été sélectionnés, on ajoute 0 comme dernier sommet pour obtenir un cycle hamiltonien.

1. Donner un exemple montrant que cet algorithme ne donne pas toujours la solution optimale.

   Corrigé

   On considère les villes dont les positions planaires sont :

   `liste_points = [(0,0), (1,0), (-2, 0), (10, 0)]`

   Cet algorithme glouton va alors donner le cycle hamiltonien `[0,1,2,3,0]`, de poids $1+3+12+10=26$, alors que le cycle hamiltonien `[0,2,1,3,0]` a un poids de $2+3+9+10=24$.

2. Écrire la fonction `voyageur_plus_proche_voisin` implémentant cet algorithme.

   Corrigé

```
def voyageur_plus_proche_voisin(g):
    sommet_actuel = 0
    chemin = [0]
    n = len(g)
    marque = [False]*n
    marque[0] = True
```

```
    for _ in range(1,n):
        poids_min = float('inf')
        sommet_suivant = None
        for u in range(n):
            if not marque[u] and g[sommet_actuel][u] < poids_min:
                poids_min = g[sommet_actuel][u]
                sommet_suivant = u
        chemin.append(sommet_suivant)
        marque[sommet_suivant] = True
        sommet_actuel = sommet_suivant
    chemin.append(0)
    return poids_chemin(g, chemin), chemin
```

3. Donner la complexité de la fonction précédente, en fonction du nombre n de villes.

   Corrigé

   On a deux boucles `for` imbriquées contenant des instructions en $O(1)$, la complexité totale est donc en $O(n^2)$.