Première partie – Savoir coder avec la notion d'ensemble.

EXERCICE 1 – LES OPPOSÉS S'ATTIRENT

Écrire une fonction qui étant donné un tableau d'entiers détermine en renvoyant un booléen s'il existe deux éléments x et y dans le tableau tels que $x = -y$.

La complexité de cette fonction doit être linéaire en la taille du tableau.

Correction. Voici l'algorithme :

```
Fonction Opposes_dans(tab : tableau)
        deja_vu = ensemble vide (ou table de hachage)
        Pour i allant de 0 à longueur(tab) - 1
        Faire x = tab[i]
                Si (-x est dans deja_vu)
                Alors   Renvoyer VRAI
                Sinon   Ajouter x dans deja_vu
        Renvoyer FAUX
```

Tester si un élément appartient bien dans un ensemble se fait en temps constant grâce à une table de hachage.

Donc la complexité de l'algorithme est bien en $O(n)$.

EXERCICE 2 – SUITE DE VAN ECK

La suite de Van Eck (v_n) se définit comme suit. Elle commence par $v_0 = v_1 = 0$.

Pour $i \geq 1$, le terme v_{i+1} se calcule en se demandant où v_i est apparu pour la dernière fois dans la liste des termes précédents :

- Si v_i est la première occurrence de cette valeur dans la suite, alors on définit $v_{i+1} = 0$.
- Si v_i est apparu pour la dernière fois en position j avec $j < i$ (on a donc $v_i = v_j$), alors on définit v_{i+1} comme étant égal à la différence entre ces deux positions, à savoir $i - j$.

Pour clarifier, calculons les termes suivants :

- **Calcul de v_2.** Le terme précédent $v_1 = 0$, est déjà apparu en position 0 : $v_0 = 0$. Donc $v_2 = 1 - 0 = 1$.
- **Calcul de v_3.** Le terme précédent est $v_2 = 1$. C'est la première fois que 1 apparaît dans la liste. Donc $v_3 = 0$.
- **Calcul de v_4.** Le terme précédent, $v_3 = 0$, est apparu pour la dernière fois deux positions auparavant, donc $v_4 = 2$.

On obtient ainsi comme premiers termes :

$$0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0, 5, 0, 2, 6, \ldots$$

Écrire une fonction `Van_Eck` qui étant donné un entier n renvoie le n-ième terme de la suite de Van-Eck. La complexité de cette fonction doit être en $O(n)$.

Correction.

```
Fonction Van_Eck(n : entier)
        derniere_apparition = dictionnaire vide
        derniere_apparition[0] = 0
        dernier_nombre = 0
        Pour i allant de 2 à n
        Faire   Si (dernier_nombre est dans dictionnaire)
                Alors   j = derniere_apparition[dernier_nombre]
                        derniere_apparition[dernier_nombre] = i - 1
                        dernier_nombre = i - 1 - j
                Sinon   derniere_appartition[dernier_nombre] = i - 1
                        dernier_nombre = 0
        Renvoyer dernier_nombre
```

Attention au $i-1$ dans le code qui correspond à un changement d'indice par rapport à l'énoncé. Ici dans la boucle, on cherche à calculer v_i à partir de la dernière apparition de v_{i-1}.
Pas de difficulté par contre pour la complexité, qui est bien en $O(n)$.

EXERCICE 3 – ALGORITHME MYSTÈRE (CT 2021)

Q1. Que renvoie la fonction `mystere` suivante ?

```
Fonction mystere(tab : tableau d'entiers)
        n = taille de tab
        E = ensemble vide
        Pour i allant de 0 à n-1
        Faire   Pour j allant de 0 à i-1
                Faire   Si (tab[i] == tab[j]) et (tab[i] n'appartient pas à E)
                        Alors   Ajouter tab[i] à E
                        FinSi
                FinFaire
        FinFaire
        Renvoyer E
```

Correction. La fonction renvoie l'ensemble des éléments du tableau qui y apparaissent plus qu'une fois.

Q2. Quelle est la complexité asymptotique des lignes suivantes (qu'on retrouve dans la fonction `mystere`) ? Justifiez.

```
Si (tab[i] == tab[j]) et (tab[i] n'appartient pas à E)
        Alors   Ajouter tab[i] à E
FinSi
```

Correction. La complexité ne dépend de la taille du tableau et est donc en $O(1)$. En effet, accéder à un élément donné du tableau se fait en temps constant, l'appartenance dans un ensemble se fait en temps constant, ajouter un élément dans un ensemble se fait en temps constant.

Q3. Quelle est la complexité asymptotique de la fonction `mystere` ? Justifiez.

Correction. La boucle dont la variable est `j` s'effectue en temps $O(i)$. Donc le temps total de la boucle se fait en temps

$$O(1) + O(2) + \cdots + O(n-1) = O(n(n-1)/2) = O(n^2).$$

EXERCICE 4 – INTERSECTION DE DEUX LISTES (CT 2021 DEUXIÈME SESSION)

Q1. Écrire une fonction `intersection` qui prend en paramètres deux tableaux d'entiers `tab1` et `tab2` et qui renvoie le nombre d'éléments qui sont à la fois dans `tab1` et `tab2`.

Par exemple `intersection([15,43,11,65],[11,86,15,2,15,6])` renvoie `2` car deux seuls nombres se trouvent à la fois dans les deux tableaux : `11` et `15`. Attention, les doublons ne comptent que pour une fois.

Bien sûr, votre fonction doit être la plus efficace que possible !

Correction.

```
Fonction intesection(tab1 : tableau d'entiers, tab2 : tableau d'entiers)

        E = ensemble vide

        Pour i allant de 0 à taille(tab1) - 1
        Faire   Si tab1[i] n'appartient pas à E
                Alors   Ajouter tab1[i] à E

        cpt = 0

        Pour k allant de 0 à taille(tab2) - 1
        Faire   Si tab2[k] appartient à E
                Alors   cpt = cpt + 1
                        Supprimer tab2[k] de E

        Renvoyer cpt
```

Q2. Quelle est la complexité asymptotique de votre fonction `intersection` ? Justifiez.

Correction. Toutes les opérations de la fonction `intersection` se font en temps constant (notamment parce que les opérations d'appartenance, d'ajout, de suppression, se font en temps constant). Donc la complexité de la fonction est en $O(longueur(tab1) + longueur(tab2))$.

EXERCICE 5 (CHALLENGE) – PLUS LONGUE SOUS-CHAÎNE SANS RÉPÉTITION D'UN CARACTÈRE

On considère le problème suivant : on souhaite écrire une fonction qui prend en paramètre une chaîne de caractères `ch` et qui renvoie la longueur de la plus longue sous-suite de `ch` qui ne contient pas deux fois le même caractère.

Par exemple, si `ch` vaut `"babcbbdb"`, la fonction doit renvoyer `3` car `"abc"` est une sous-chaîne sans répétition. Il n'y a pas de sous-chaînes de longueur ≥ 4 qui ne contient pas deux fois la même lettre. (Par exemple, `"abcb"` contient deux fois "b" et `"abcd"` n'est pas considéré comme une sous-chaîne.) Autre exemple, sur l'entrée `"zzzzz"`, la fonction doit renvoyer `1`.

Q1. Ecrire de manière naïve une fonction qui teste si la sous-chaîne de `ch` commençant à l'indice `i` et finissant à l'indice `j` contient deux fois le même caractère.

Correction.

```
Fonction sans_repetition(ch,i,j)
        Pour k allant de i à j-1
        Faire   Pour l allant de k+1 à j
                Faire   Si ch[k] == ch[l]
                        Alors Renvoyer FAUX
        Renvoyer VRAI
```

Q2. En utilisant la fonction précédente, en déduire un algorithme naïf qui répond au problème.

Correction.

```
Fonction longueur_du_plus_grand_sans_repetition(ch)
        res = 0
        Pour i allant de 0 à taille de ch - 1
        Faire   Pour j allant de i à taille de ch - 1
                Faire   Si sans_repetition(ch,i,j)
                        Alors res = max(res,j-i+1)
        Renvoyer res
```

Q3. Quelle est la complexité de l'algorithme naïf ?

Correction. On a quatre boucles imbriquées, donc la complexité est en $O(n^4)$.

Q4. Ecrire un algorithme qui marche et qui est de complexité $O(n^2)$. Indication : Faites varier une variable f de 1 à la dernière position du tableau, et essayez de décrire la sous-chaîne de longueur maximale sans répétition qui termine en f à partir de la sous-chaîne de longueur maximale sans répétition qui termine en $f-1$.

Correction.

```
Fonction longueur_du_plus_grand_sans_repetition(ch)
        longueur_max = 1
        depart = 0
        Pour fin allant de 1 à longueur(ch) - 1
        Faire
                j = depart

                Tant que (ch[j] != ch[fin])
                Faire j = j + 1

                Si (j == fin)
                Alors longueur_max = max(longueur_max,fin-depart+1)
                Sinon depart = j+1

        Renvoyer longueur_max
```

Il y a deux boucles imbriquées, donc sa complexité est en $O(n^2)$.

Pourquoi ça marche ? Il faut un peu s'accrocher et l'exécuter sur un exemple.

En gros, à chaque étape, on veut trouver la plus longue sous-chaîne sans répétition qui termine en la position `fin`. La variable `depart` est choisie de telle sorte que la lettre à la position `depart - 1` se trouve forcément entre `depart` et `fin`. Par conséquent, il n'existe pas de sous-chaîne commençant avant `depart` et finissant exactement à `fin` sans répétition (à cause de cette lettre à la position `depart - 1`. De plus, on s'assure qu'entre `depart` et `fin-1`, il n'y a jamais répétition d'un même caractère.
Ainsi à chaque début de boucle", on cherche à savoir si on peut ajouter la lettre à la position `fin` au sous-mot qui est entre `depart` et `fin-1` (qu'on sait sans répétition). Pour cela, il suffit juste de savoir si la lettre de "fin" est dans ce sous-mot là. De plus, si cette lettre y est, on sait qu'elle y sera au plus une fois (vu que le mot est sans répétition).

La partie où on initialise `j` à `depart` et où on l'augmente sert précisément ce but-là : chercher cette lettre.

Si elle n'y est pas, alors `j` vaudra `fin`, et on a un potentiel mot plus long entre `depart` et `fin`. Il faudra alors comparer `longueur_max` à la longueur de ce mot (à savoir `fin-depart+1`).

Si elle y est, alors il faudra changer `depart` pour le prochain passage de la boucle : la lettre en position `j` se trouve également en position `fin`. On change alors `depart` en `j+1`. Entre `j+1` et `fin`, on sait de plus qu'il n'y a pas de répétition car à part la dernière lettre, il s'agit d'un sous-mot du sous-mot entre `depart` et `fin-1` qu'on savait sans répétition. L'invariant sur la variable `depart` est bien conservé.
Cette explication est bien sûr non exigible pour un étudiant en L3.

Q5. Améliorer la question 1 de sorte à que ce soit un peu moins naïf, en utilisant une table de hachage. (Si vous aviez déjà utilisé une table de hachage, bravo !)

Correction.

```
Fonction sans_repetition(ch,i,j)
        deja_vu = ensemble vide
        Pour k allant de i à j
        Faire   Si ch[k] est dans deja_vu
                Alors Renvoyer FAUX
                Sinon Ajouter ch[k] à deja_vu
        Renvoyer VRAI
```

Q6. Quelle est du coup maintenant la complexité de l'algorithme de la question 2 ?

Correction. En recopiant tel quel l'algorithme de la question 2, on voit qu'on a maintenant 3 boucles imbriquées. La complexité est donc en $O(n^3)$.

Q7. Cramez les derniers neurones qui vous restent en essayant de trouver un algorithme qui est linéaire en n ! Si vous avez réussi à faire la question 4 de l'exercice 3 sans table de hachage, il s'agit d'améliorer cet algorithme avec une table de hachage.

Correction.

```
Fonction longueur_du_plus_grand_sans_repetition(ch)
        longueur_max = 1
        lettres_dans_sous_chaine_max = ensemble contenant ch[0]
        debut = 0
        Pour fin allant de 1 à longueur(ch) - 1
        Faire
                Si (ch[fin] est dans lettres_dans_sous_chaine_maximale)
                Alors
                        j = debut
                        Tant que (ch[j] != ch[fin])
                        Faire   Retirer ch[fin] de lettres_dans_sous_chaine_maximale
                        debut = j + 1
                Sinon
                        longueur_max = max(longueur_max,fin-depart+1)
                        Ajouter ch[fin] à lettres_dans_sous_chaine_maximale


        Renvoyer longueur_max
```

Pas le temps d'expliquer ! Dans les grandes lignes, ça imite l'algorithme de la question 4, mais qui retient toutes les lettres qui sont dans la sous-chaîne maximale précédente !
Pour la complexité, on a bien du $O(n)$. La variable j ne prend jamais deux fois la même valeur, donc la boucle en j n'entraîne pas une complexité quadratique.

Deuxième partie – Table de hachage

Cette partie peut être vue à la fois comme un TD sur feuille (en pseudo-code), ou un TP sur machine (en `C`). Il s'agit d'implémenter "à la main" le type abstrait "ensemble" (`set` en Python) via une table de hachage. Si vous souhaitez le faire sur machine, il est recommandé de l'écrire en `C`.

EXERCICE 6 – IMPLÉMENTATION DU TYPE "ENSEMBLE D'ENTIERS"

On définira deux structures :

```
Structure Noeud {
        valeur : entier
        suivant : pointeur vers Noeud
}

ListeChainee = pointeur vers Noeud
```

et

```
Structure Ensemble {
        table : tableau de ListeChainee
        taille : entier
}
```

En C, cela s'écrit :

```c
struct Noeud{
        int valeur;
        struct Noeud* suivant;
};

typedef struct Noeud* ListeChainee;


struct Ensemble{
        ListeChainee* table;
        int taille;
};
```

Petite liste des opérations qu'il sera possible de faire en pseudo-code:

- Allouer de la mémoire pour un ensemble
- Allouer de la mémoire pour un noeud de liste chaînée
- Allouer un tableau de taille donnée
- Accéder et modifier à un membre d'une structure. Par exemple :
  - `Pour i allant de 0 à E.taille - 1`
  - `E.table[3] = pointeur nul`
  - `L = L.suivant`
  - ...

Q1. Ecrire une fonction `creeEnsemble` qui prend en paramètre un entier[1] `n` et qui renvoie un élément de structure Ensemble de taille `n`.

[1] Pourquoi impose-t-on ici une taille pour notre table de hachage, alors qu'en python ou en Java, quand on crée un ensemble, on ne précise jamais la taille ? Et bien, `python` ou `java` calcule automatiquement la taille de la table de hachage de sorte que chaque opération élémentaire associée à la notion d'ensemble se fasse en temps constant. Plus précisément, si le nombre d'éléments dans la table de hachage devient trop grand ou trop petit, on alloue une nouvelle table de hachage de taille plus adéquate, puis on recopie les valeurs de l'ancienne table de hachage vers la nouvelle. J'arrête les détails : cela reste un peu trop technique pour que ce soit réellement demandé dans le cadre de cet exercice.

Correction. En pseudo-code :

```
Fonction creeEnsemble(n : entier){
        Allouer un nouvel ensemble E
        E.table = tableau de taille n remplis de vecteurs nuls
        E.taille = n
        Renvoyer E
}
```

En C :

```
struct Ensemble creeEnsemble(int n){

        struct Ensemble E;

        E.table = malloc(sizeof(ListeChainee)*n);
        for (int i = 0; i < n; i++) E.table[i] = NULL;
        /* Pour les deux dernières lignes, un "calloc"
        aurait pu très bien marcher aussi */

        E.taille = n;

        return E;
}
```

Q2. Ecrire une fonction `affiche` qui prend en paramètre un ensemble E et qui affiche tous les éléments dans E.

Correction. En pseudo-code :

```
Fonction affiche(E : ensemble){
        Pour i allant de 0 jusqu'à E.taille - 1
        Faire
                L = E.table[i]
                Tant que (L n'est pas le pointeur nul)
                Faire   Afficher( L -> valeur )
                        L = L -> suivant
}
```

En C :

```
void affiche(struct Ensemble E){

        for(int i = 0; i < E.taille ; i++){

                ListeChainee L = E.table[i];
                while( L != NULL ){
                        printf("%d ", L->valeur );
                        L = L->suivant;
                }

        }
}
```

Q3. On va considérer la fonction de hachage :

$$h : x \mapsto (15073 \times x) \text{ modulo (taille de la table de hachage)}$$

Autrement dit, quand on va insérer un entier x dans l'ensemble, on va allouer un nouveau noeud contenant x et mettre l'adresse mémoire de ce noeud à la position $h(x)$ dans la table de hachage

associée.

Écrire une fonction `ajout` qui prend en paramètre un ensemble `E` et un entier `x`, et qui rajoute dans `E` l'entier x. (On rajoutera une occurrence de x même si x est dans E)

Correction. En pseudo-code :

```
Fonction ajout(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille
        L = pointeur vers un nouveau noeud
        L.valeur = x
        L.suivant = E.table[hx]
        E.table[hx] = L
}
```

En C :

```
void ajout(struct Ensemble E, int x){

        int hx = (15073*x) % E.taille;
        ListeChainee L = malloc(sizeof(struct Noeud));
        L -> valeur = x;
        L -> suivant = E.table[hx];
        E.table[hx] = L;

}
```

Note : Pour être plus propre, il aurait mieux valu écrire la fonction de hachage en dehors de la fonction `ajout`, voire même de l'incorporer dans la structure `Ensemble`.

Q4. Écrire une fonction `appartient` qui prend en paramètre un ensemble `E` et un entier `x`, et qui renvoie `VRAI` si `x` appartient dans `E` ; `FAUX` sinon.

Correction. En pseudo-code :

```
Fonction appartient(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille

        L = E.table[hx]
        Tant que (L n'est pas nul)
        Faire Si L.valeur == x
                Alors Renvoyer VRAI
                SINON L = L->suivant

        Renvoyer L
}
```

En C :

```
bool appartient(struct Ensemble E, int x){
        int hx = (15073*x) % E.taille;
        ListeChainee L = E.table[hx];

        while(L != NULL){
                if (L->valeur == x) return true;
                L = L->suivant;
        }

        return false;
}
```

Q5. Écrire une fonction `supprime` qui prend en paramètre un ensemble `E` et un entier `x` (on supposera que E contient bien x), et qui supprime de E une occurrence de x.

Correction. En pseudo-code :

```
Fonction enleveOccurrence(L : ListeChainee, x : entier) {
        Si (L est nul) Alors Renvoyer Pointeur Nul
        /* Pas nécessaire si on suppose que L contient x */

        Si (L.valeur == x)
        Alors Renvoyer L.suivant

        L.suivant = enleveOccurrence(L.suivant,x)
        Renvoyer L
}

Fonction supprime(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille
        E.table[hx]     = enleveOccurrence(E.table[hx],x)
}
```

En C :

```
ListeChainee enleveOccurrence(ListeChainee L, int x){
        if (L == NULL) return NULL; //pas nécessaire si on suppose que x est dans L
        if (L->valeur == x) {
                ListeChainee tmp = L->suivant;
                free(L);
                return tmp;
        }
        L->suivant = enleveOccurrence(L->suivant,x);
        return L;
}

void supprime(struct Ensemble E, int x){
        int hx = (15073*x) % E.taille;
        E.table[hx] = enleveOccurrence(E.table[hx],x);
}
```

Note : Il est possible de faire une version itérative, mais elle est plus délicate car il faut a priori séparer le cas où E.table[$h(x)$] pointe directement vers un noeud avec x comme valeur et le cas contraire. Voir cette correction en C pour la version itérative :

```
void supprime2(struct Ensemble E, int x){

        int hx = (15073*x) % E.taille;

        ListeChainee l = E.table[hx];

        if (l->valeur == x){
                E.table[hx] = l->suivant;
                free(l);
        }
        else{

                while(l->suivant->valeur != x){
                        l = l->suivant;
                }

                ListeChainee aSupprimer = l->suivant;
                l->suivant = l->suivant->suivant;
                free(aSupprimer);

        }
}
```