Séance 9 : Animations et algorithmes

L1 – Université Côte d'Azur

Exercice 1 — Algorithme d'Euclide (⋆)

1. On cherche à calculer le PGCD de deux nombres entier. Par exemple, si $n = 21$ et $m = 15$, leur PGCD vaut 3. En effet 3 est un diviseur en commun à $21 = 3 \times 7$ et $15 = 3 \times 5$ et c'est même le plus grand possible.
   Pour calculer ce PGCD, on fait un raisonnement par cas
   - on se ramène au cas dans lequel $n \geq m$ quitte à échanger n et m
   - si n et m sont égaux, c'est fini ; ils sont égaux à leur pgcd.
   - sinon, le PGCD de n et m est égal au PGCD de $n - m$ et m

   Écrivez une fonction *récursive* `affiche_et_calcule_pgcd_naïf(n,m)` qui affiche les étapes du calcul du PGCD et renvoie le résultat. Par exemple,

```
>>> d=affiche_et_calcule_pgcd_naïf(21,15)
calcule le pgcd de n=21 et m=15
calcule le pgcd de n=6 et m=15
calcule le pgcd de n=15 et m=6
calcule le pgcd de n=9 et m=6
calcule le pgcd de n=3 et m=6
calcule le pgcd de n=6 et m=3
calcule le pgcd de n=3 et m=3
>>> print('Le PGCD de 21 et 15 vaut',d)
Le PGCD de 21 et 15 vaut 3
```

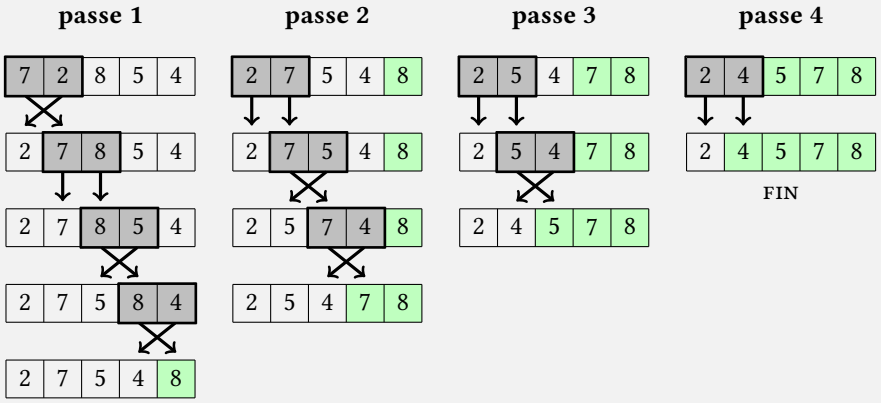
2. L'agorithme précedent n'est pas efficace. Par exemple si $n = 100$ et $m = 12$, il va falloir faire de nombreuses soustractions du nombre 12 : $100 \to 88 \to 76 \to 64 \to \cdots \to 4$. On peut calculer par avance le nombre de soustraction à faire (ici 8) en faisant la division euclidienne de 100 par 12 ; ce qui nous donne 8 reste 4.
   On peut donc accélérer l'algorithme en passant non pas de n à $n - m$ mais de n à $n - qm$, où q est le quotient de la division euclidienne de n par m. C'est le fameux algorithme d'Euclide.
   Écrivez une fonction `affiche_et_calcule_pgcd(n,m)` pour effectuer le calcul dans sa version accélérée. On fera attention au cas d'arrêt.

```
>>> d=affiche_et_calcule_pgcd(21,15)
calcule le pgcd de n=21 et m=15
calcule le pgcd de n=15 et m=6
calcule le pgcd de n=6 et m=3
calcule le pgcd de n=3 et m=0
>>> print('Le PGCD de 21 et 15 vaut',d)
Le PGCD de 21 et 15 vaut 3
```

3. Écrivez une fonction `affiche_et_calcule_pgcd_while(n,m)` qui utilise une boucle `while` plutôt que des appels récursifs.

L'algorithme de tri à bulles

Le tri à bulles est un algorithme de tri qui repose sur l'idée que tant que deux éléments adjacents ne sont pas dans le bon ordre, on doit les échanger pour se rapprocher d'une liste triée. Il s'agit donc d'un tri *en place* (on ne crée pas une nouvelle liste), où les seuls échanges se font entre des valeurs adjacentes. Plus précisément, on effectue $n-1$ passes (où n est la longueur de la liste à trier) et dans chaque passe on parcourt les couples de valeurs adjacentes en partant du début de la liste et on les échange si elles ne sont pas dans le bon ordre.



On peut donc voir le tri à bulle comme une variante du tri par sélection, où on commence par positionner la valeur la plus grande à sa position correcte, puis on répète le procédé sur la sous-liste qui s'arrête juste avant ce dernier élément. Notons enfin que si durant une passe on n'a fait aucun échange, la liste est triée, et on peut donc omettre les passes restantes.

Exercice 2 — Tri à bulles (⋆⋆)

1. Écrivez une fonction `compare_et_échange(L,i)` qui prend en argument une liste d'entiers `L` de longueur $n \geq 2$ et un entier `i`$\leq n-2$ et qui échange les entiers aux indices `i` et `i+1` si ils ne sont pas dans le bon ordre. Par exemple, on aura

```
>>> L = [3,4,5,1]
>>> compare_et_échange(L,0)  # -> ne change rien
>>> compare_et_échange(L,2)  # -> échange 5 et 1
>>> L
[3, 4, 1, 5]
```

2. Écrivez une fonction `passe_tri_bulles(L,k)` qui prend en argument une liste d'entiers `L` de longueur $n \geq 2$ et qui effectue la passe k de l'algorithme du tri à bulle.
3. En déduire une fonction `tri_bulles(L)` qui trie en place la liste `L` par l'algorithme de tri à bulles.
4. Optionnel : permettez à votre fonction de terminer avant la passe $n-1$ si durant une passe aucun échange n'est effectué.

Exercice 3 — Tri par comptage (⋆⋆)

Pour trier une liste de nombres entiers positifs ou nuls, on peut compter le nombre de fois que chaque nombre apparait dans la liste, puis créer une nouvelle liste qui contient chacun de ces nombres autant de fois que nécessaire. On obtient un algorithme de tri en temps linéaire en la taille de la liste, (mais aussi linéaire en la valeur du plus grand nombre).

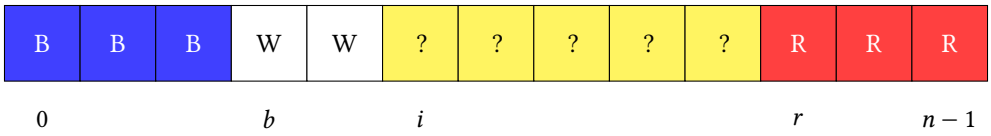
1. Écrivez une fonction `liste_nombre_occurrences(L)` qui prend en argument une liste d'entiers positifs `L` et qui renvoie une liste N de longueur `max(L)+1` telle que `N[i]` soit le nombre de fois où i apparaît dans `L`. Par exemple, `liste_nombre_occurrences([0,1,0,3,5])` renvoie `[2,1,0,1,0,1]` (**2** occurences de 0, **1** occurence de 1, **0** occurence de 2, **1** occurence de 3, **0** occurence de 4 et **1** occurence de 5). On obtient bien une liste de longueur 6 (=5+1).

2. Écrivez une fonction `crée_liste_triée(N)` qui prend en argument une liste `N` d'entiers positifs ou nuls et qui renvoie la liste `Lt` composée de `len(N)` blocs (possiblement de longueur nulle), le i-ème bloc étant de longueur `N[i]`. Par exemple, `crée_liste_triée([2,1,0,1,0,1])` renvoie `[0,0,1,3,5]`.
3. (bonus) Cherchez une solution qui n'utilise pas de concaténation de liste ni la méthode append, mais uniquement une allocation de liste remplie de 0 (`L = [0 for i in range(n)]`) et des mutations de contenu (`L[i] = ...`)
4. En déduire une fonction `tri_comptage(L)` qui prend en argument une liste d'entiers positifs ou nuls et qui renvoie une nouvelle liste de même contenu, mais triée.

Exercice 4 — Boucle `while` : le drapeau hollandais (⋆⋆)

Le problème du drapeau hollandais est un classique des exercices de programmation promu par Edgser W. Dijkstra, un des « pères fondateurs » de l'informatique en tant que discipline scientifique, qui a laissé son nom dans des domaines aussi variés que l'algorithmique et les systèmes d'exploitations.

Le problème est le suivant : on dispose d'une liste `L` contenant trois types de valeurs : `'B'` (bleu), `'W'` (blanc), et `'R'` (rouge). Le but est d'écrire une fonction qui trie la liste `L` *en place*[1] de sorte que les couleurs apparaissent dans l'ordre du drapeau hollandais (`'B'`<`'W'`<`'R'`). L'algorithme consiste à diviser la liste en quatre zones consécutives, voir dessin ci-dessous, et à réduire à chaque étape la zone « jaune » des valeurs qui ne sont pas encore triées.



Plus précisément, votre fonction utilisera en une boucle `while` et manipulera les variables i, b et r. À chaque tour de boucle, il faudra réaliser l'une des actions ci-dessous, selon la couleur de la valeur de `L[i]` :

— si `L[i]` est bleu, on échange L[i] avec L[b], on avance b et on avance i
— si `L[i]` est rouge, on échange L[i] avec le dernier jaune, et on recule r
— si `L[i]` est blanc, on avance i

1. Comment sont initialisées les variables b, i, et r ? Quelle est la condition de sortie de la boucle ?
2. Écrivez la fonction `tri_hollandais(L)` en lui demandant d'afficher les étapes intermédiaires.

Exercice 5 — Tri fusion (⋆⋆)

L'algorithme de tri fusion est un algorithme classique de tri de liste par récurrence. Il est basé sur une approche « diviser pour régner » et de complexité asymptotique dans le pire cas $O(n \cdot \log(n))$. L'algorithme peut être décrit ainsi :

— soit L la liste à trier.
— si L a moins de 2 éléments, elle est triée
— sinon on note L_1 et L_2 les deux demi-listes obtenues en coupant L en son milieu, avec L_1 contenant l'élément de position médiane si L est de longueur impaire. Par exemple, si $L = [5, 1, 3, 4, 2]$, $L_1 = [5, 1, 3]$ et $L_2 = [4, 2]$
— on trie L_1 et L_2 récursivement (par exemple, $L_1 = [1, 3, 5]$ et $L_2 = [2, 4]$).
— on fusionne L_1 et L_2 (par exemple, $[1, 2, 3, 4, 5]$).

1. Écrivez une fonction `fusion(L1,L2)` qui prend en arguments deux listes triées L_1 et L_2 et qui renvoie une nouvelle liste contenant les contenus de L_1 et L_2 fusionnés.
2. En déduire une fonction définie par récurrence `tri_fusion(L)` qui renvoie une nouvelle liste contenant le résultat du tri de L.

1. On pourrait envisager un tri par comptage pour ce problème, mais ce ne serait pas un tri en place (le comptage requiert de créer une nouvelle liste). L'originalité de l'algorithme présenté dans cet exercice est de faire un tri en place de même complexité que le tri par comptage, i.e. linéaire en la taille de la liste, le nombre de valeurs différentes dans la liste étant considéré comme une constante (ici 3).

Exercice 6 — Exponentiation rapide (⋆ ⋆ ⋆)

On considère la fonction suivante pour le calcul de x puissance n

```
def puissance(x,n) :
    acc = 1
    for i in range(n) :
        acc = acc * x
    return acc
```

1. Combien de multiplications sont nécessaires pour calculer x^n avec cette méthode ?
2. Écrivez une fonction qui calcule x^n en $O(\log(n))$ multiplications.

Exercice 7 — Dichotomie (⋆⋆)

On considère la fonction suivante :

```
def recherche_dichotomique_erronée(x,L) :
    g = 0
    d = len(L) - 1
    while g != d :
        m = (g + d) // 2
        if L[m] == x :
            return m
        elif L[m] < x :
            g = m
        else :
           d = m
    if L[g] == x :
        return g
    return -1
```

1. Quelles sont les valeurs successives de `g`, `d` et `m` si l'on exécute `recherche_dichotomique_erronée(7 ,[i for i in range(2,21,2)])` ? Quel est le problème ?
2. Corrigez la fonction ci-dessus.