- **Version avec mise à jour du 12/10/24. Si vous trouvez des bugs (voire des fautes de français...) dans cette correction qui ne sont pas corrigés dans la dernière version sur ecampus, le signaler à rosaz@lri.fr**

L3, U-Psud, Orsay — Algorithmique et Complexité

Feuille de TD N^o 2 : Listes-Piles

1 Listes-Piles : listes chaînées

Les opérations élémentaires sont :

- Tester si une liste est vide : Test `l == NULL`. À la rigueur, utilisation de la fonction `bool Estvide(liste l)`. Des écritures telles que `l == Vide`, `l est Vide`, `l == ∅`, `l == []` sont tolérées (pseudocode)
- Initialisation d'une liste à vide : Affectation `l = NULL ;` À la rigueur : `void Initvide(inout liste *L)`. Des écritures telles que `l = Vide ;`, `l = ∅ ;`, `l = [] ;` sont tolérées.
- Premier élément d'une liste : `(*l).valeur` ou `l->valeur`. À la rigueur : `élément premier(liste l)`
  La tentative d'accéder au premier élément d'une liste vide entraîne un plantage (Segmentation Fault).
- Ajout d'un élément en tête de liste. Pour cela, il y a une fonction et une procédure :
  • La fonction `liste ajoute(x, liste l)` rend un pointeur liste pointant vers un nouveau bloc contenant la valeur x et un champs `suite` copie de l. La liste l n'est pas modifiée, n'est pas recopiée, son espace mémoire est partagé avec celui du résultat de `ajoute`. Cf. code dans le cours dans lequel il y a un (et un seul) malloc.
  • La procédure `empile(x, inout *L)` modifie la liste en insérant x en tête. Elle équivaut à `l = ajoute(x,l) ;`
- Liste privée de son premier élément :
  • Accès à la suite de la liste : `(*l).suite` ou `l->suite`. À la rigueur : `liste suite(liste l)`
  La liste `l` n'est pas modifiée, elle n'est pas recopiée, son espace mémoire est partagé avec celui du résultat de `suite`. Il n'y a pas de désallocation mémoire (pas de free). C'est comme si on tournait la page d'un livre.
  • La procédure `depile(inout *L)` modifie la liste en enlevant son premier élément, lequel est désalloué, i.e. rendu à la mémoire. Cf. code dans le cours, qui inclut un free. C'est comme si on arrachait la page d'un livre.
  • "PointeurSuite" : Lorsque l'on appelle récursivement sur `l->suite` une procédure qui modifie les listes, on ne peut pas utiliser le passage par valeur. Il faudra donc faire un "PointeurSuite" et montrer les * et les &. Il faudra utiliser et comprendre des écritures telles que &(**L).suite ou &(*L)->suite.

  La tentative de faire `suite, depile` ou `PointeurSuite` sur une liste vide entraîne un plantage (Segmentation Fault).

Écrire les fonctionnalités suivantes :

1. **Doubleton (l)** qui rend vrai ssi l contient exactement deux éléments (éventuellement identiques)

   - (0) ce qui suit est pourri :
     bool singleton (l)
     rendre longueur(l) == 2
     L'annuaire de paris est-il un doubleton ? Pour y répondre, vous comptez le nombre de noms qui y figurent, vous comptez jusqu'à mettons 6548127 puis vous dites que 6548127 n'est pas 2. Bref, complexité en $\theta(n)$ plutôt qu'en $\theta(1)$. Ce genre de réponse ne vaut rien à l'examen. Il n'est pas explicitement demandé une bonne complexité, mais nous sommes en algo, il est donc implicite qu'il faut une bonne complexité, et cela devrait devenir implicite tout le temps (sauf à la rigueur si vous traitez des micro-données et que le code efficace est compliqué, pas la peine de coder le tri par tas pour trier un tableau de 10 éléments).

     (1)
     bool doubleton (l)
     si l == NULL
     alors rendre faux
     sinon
     si l->suite == NULL
     alors rendre faux
     sinon rendre l->suite->suite == NULL

     Remarques
     (1) Faire absolument le test si l == NULL, sinon l->suite va planter sur vide, et le code ne vaudra rien a l'exam. Puis le test sur l->suite sinon l->suite->suite va planter, et ce ne sera pas terrible pour l'exam.
     l->suite ne plantera pas grâce au test l == NULL

l->suite->suite ne plantera pas grâce au test l == NULL
l->suite->suite ne plantera pas grâce au test l->suite == NULL
(2) certains auront la maladresse de faire :

```
bool singleton (l)
        si l == NULL
        alors rendre faux
        sinon
        si l->suite == NULL
        alors rendre faux
        sinon
        si l->suite->suite == NULL
        alors rendre vrai
        sinon rendre faux
```

C'est une maladresse, (a==b) est une expression booléenne, qui peut être l'argument du rendre pour une fonction booléenne.
(3) On peut aussi utiliser le fait que les AND/OU sont paresseux

```
bool doubleton (l)
        si l == NULL ou l->suite == NULL
        alors rendre faux
        sinon rendre l->suite->suite == NULL
```

et même :

```
(4) bool singleton (l)
        rendre l ≠ NULL et l->suite ≠ NULL et l->suite->suite == NULL
```

2. **Affiche (l)** qui affiche les éléments dans l'ordre.

- Faire une première version, puis ajouter un retour chariot à la fin de l'affichage

En récursif : Si je veux afficher 5678 à l'endroits, je fais afficher 678 récursivement, je vois que pour finir le job, je dois afficher le 5 avant. Et si je veux un retour-chariot final, il suffit de le faire quand tout est fini, i.e. quand nous arrivons sur l'appel de fond de liste.

```
void affiche (L) :
        si L == NULL
        alors print retour-chariot
        sinon { print (L->valeur)
                affiche (L->suite) }
```

itératif, première version (un peu dangereuse) :

```
void affiche (L)
        tant que L ≠ NULL faire
                { print (L->valeur)
                L = L->suite }
        print retour-chariot
```

Cette première version est correcte mais un peu dangereuse : en effet, la procédure modifie son argument L jusqu'à le rendre NULL. Heureusement, l'argument est in, et donc c'est une copie qui est mise à NULL. Mais un jour, quelqu'un passera derrière (peut-être ce quelqu'un sera-t-il vous-même ... J'ai déjà vu un étudiant s'auto-piéger sur le projet dans un autre contexte (c'était le fait de ne pas mettre un else après un return)) et, par incompétence, ou pour adapter le programme à de nouvelles conditions, fera passer l'argument en inout. Ce jour-là, l'appel à affiche aura pour effet de bord de mettre l'argument à NULL (si on teste, on ne s'en rend compte que si on affiche la liste deux fois, ou si on l'utilise plus tard). Il vaut donc mieux faire ce qui suit :

itératif, deuxième version :

```
void affiche (L)
        P = L ;
        tant que P ≠ NULL faire
                { print (P->valeur)
                P = P->suite }
```

print retour-chariot

remarquez qu'il est plus difficile de faire une version non dangereuse en récursif.
On pourrait critiquer qu'il ne sert à rien de faire une copie de ce qui est déjà une copie. Critique défendable.
En pratique, au partiel, la démarche sécuritaire est acceptée et la démarche critique aussi.

3. **ehciffA (l)** qui affiche les éléments dans l'ordre inverse (dit aussi "ordre miroir")

   - En récursif, et sans le retour chariot : Si je veux afficher 5678 à l'envers (i.e. 8765), je fais afficher 678 récursivement à l'envers, ce qui affiche 876, je vois que pour finir le job, je dois afficher le 5 APRÈS. D'où le code :

     void ehciffA (L) :
     si L ≠ NULL
     alors { ehciffA (L->suite)
     print (L->valeur) }

     Et je veux mettre un retour chariot final ? Suffit-il de faire comme pour Affiche ?
     récursif :
     void ehciffA (L) :
     si L == NULL FAUX
     alors print retour-chariot
     sinon { ehciffA (L->suite)
     print (L->valeur) }

     NON, le retour-chariot est imprimé AVANT la liste. Imaginez que je demande au premier étudiant le boulot sur L qui va demander à son voisin avec une liste plus courte qui va demander à son voisin ... jusqu'à qqun qui va récupérer une liste vide et exécuter l'ordre sur la liste vide, puis se tournera vers l'avant dernier en disant "fini", l'avant dernier reprendra alors son calcul, puis dira "fini" au prédécesseur, ..., jusqu'au premier qui finira son boulot et me dira "fini". Les choses sont donc faites dans l'ordre suivant, le préfixe dans l'ordre, puis l'appel final, puis le suffixe dans l'ordre opposé.
     Pour le mettre au bon endroit, il faut une surfonction.

     void ffA (L) :
     si L ≠ NULL
     alors { ffA (L->suite)
     print (L->valeur) }

     void ehciffA (L)
     ffA (L)
     print retour-chariot

   Nous voyons que le code n'est pas récursif terminal, et il est compliqué de le faire (ou alors il faut tricher : une passe qui renverse la liste, affichage endroit, puis une passe qui remet la liste à l'endroit)

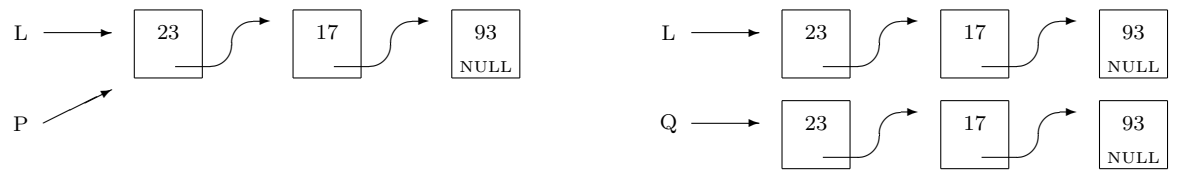
   Nous voyons de la même façon que le faire en itératif est problématique. Nous sommes obligé de faire un stockage annexe, de renverser la liste, ou de faire un algo quadratique (On précalcule la longueur ℓ puis de k = ℓ à 1 décroissant, on affiche le k^e élément).

   La théorie nous dit qu'il est possible de transformer tout récursif en itératif en manipulant une pile de ce qu'il reste à faire. Donc on empile "afficher le premier élément", puis "afficher le second", puis ... In fine, notre pile de ce qu'il reste à faire est la pile initiale à l'envers. On retombe sur les versions en plusieurs passes qui font des miroirs.

   Cet exemple montre l'intérêt du récursif, même sur des listes (ce sera bien plus flagrant sur les arbres)

4. **Copie (l)** qui rend une liste représentant la même liste mathématique que l, mais avec des blocs indépendants et neufs.

   - Si L est une liste, faire $P = L$ ne fait qu'une copie de pointeur, P pointe donc vers le premier bloc de la liste L. Par contre, si on fait $Q = copie(L)$, de nouveaux blocs sont créés, indépendants de ceux de L



• Récursif :

Si la liste n'est pas vide, je demande une copie de L->suite. J'obtiens donc un pointeur vers une liste neuve, copie de [17,93] sur l'exemple ci-dessus. Je fais un malloc, j'y copie de 23 et le pointeur que je viens de recevoir et je rends un pointeur vers le bloc créé. Tout ceci se résume par `rendre ajoute(L->valeur, copie(L->suite))`.

liste copie (L)
    si L == NULL
    alors rendre NULL
    sinon rendre ajoute(L->valeur, copie(L->suite))

Comment se passent les choses ? On a une liste $\rightarrow 23 \rightarrow 17 \rightarrow 93\ NULL$

L'appel 1 a pour argument un pointeur vers le bloc 23 $\rightarrow$, il lance un appel 2 qui pointe vers le bloc 17 $\rightarrow$ qui lance un appel 3 qui pointe vers le bloc 93 $NULL$ qui lance un appel 4 qui a un pointeur NULL.

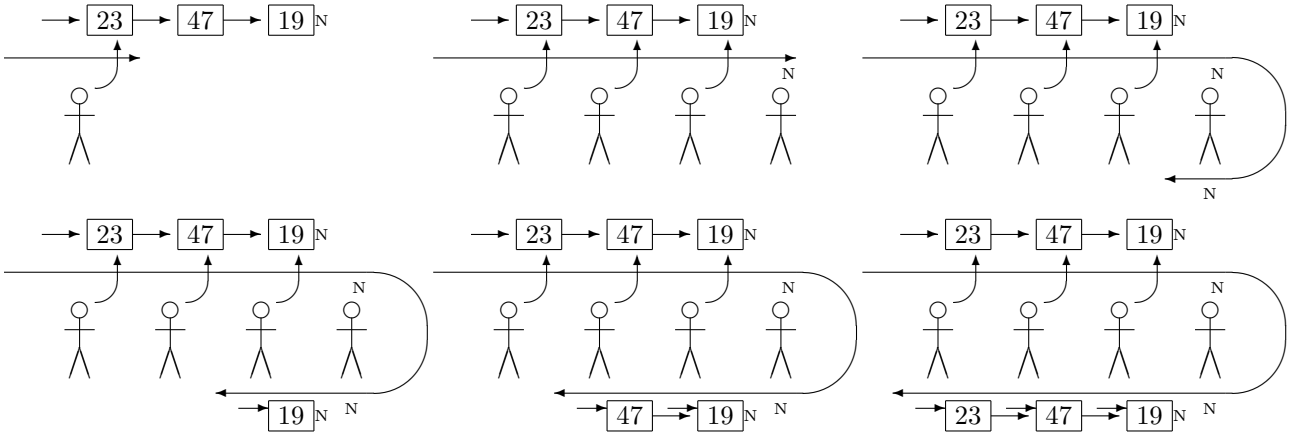
L'appel 4 rend NULL à l'appel 3.

L'appel 3 reprend la main, fait un ajoute, donc un malloc vers un nouveau bloc, y met comme valeur L->valeur pour son L local, soit 93, et comme `suite` met le NULL reçu de l'appel 4, et il rend le pointeur vers ce bloc à l'appel 2.

L'appel 2 reprend la main, fait un malloc vers un nouveau bloc, y met comme valeur 17, et comme `suite` le pointeur reçu de l'appel 3, et rend le pointeur vers ce bloc à l'appel 1.

L'appel 1 reprend la main ajoute le 23 et rend le pointeur vers son bloc à l'appelant.

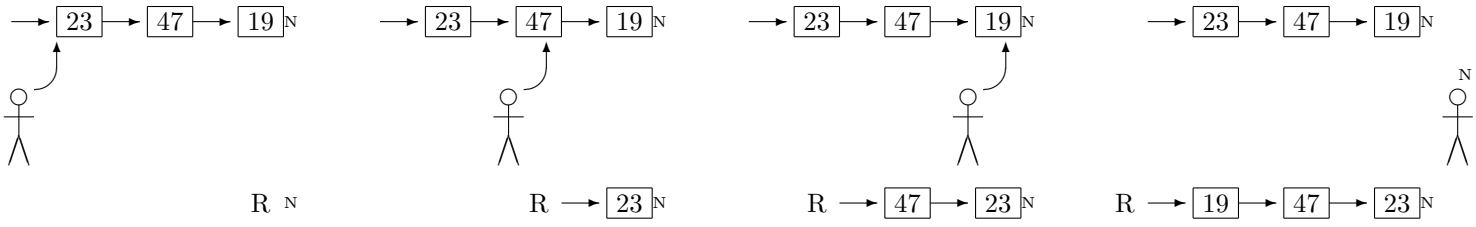
On voit que la liste se construit au retour



• Il peut être tentant de faire de l'itératif comme suit :

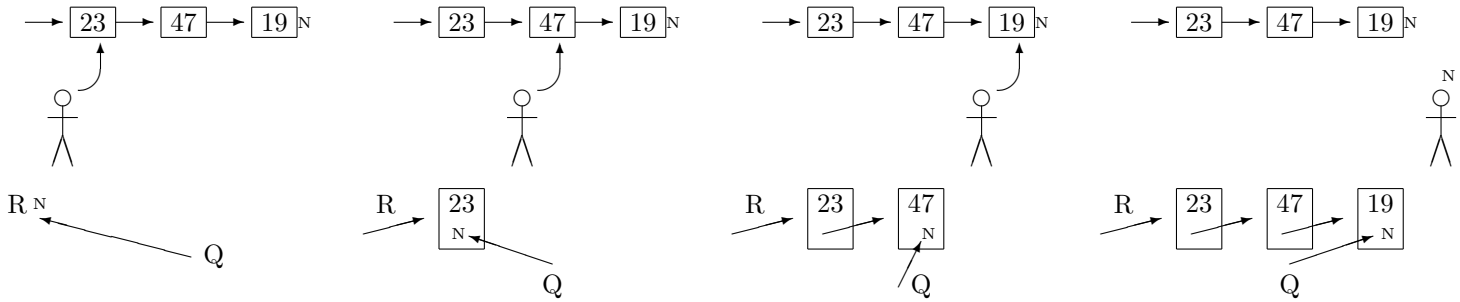
liste Copie(L)
    R = NULL
    P = L
    tant que P $\neq$ NULL
        empile(P->valeur, &R) CODE FAUX
        P = P->suite
    rendre R

mais c'est FAUX, car le résultat est à l'envers



• Certains pourraient être tentés de pallier le fait que la copie est à l'envers, en faisant une copie à l'envers de la copie à l'envers. C'est ridicule : Pour faire la photocopie du cours de votre camarade, faites-vous une photocopie en empilant les feuilles, d'où une copie à l'envers, puis une photocopie de la photocopie ? Bien sûr que non, ce serait du gaspillage, de temps et de papier. En programmation, c'est la même chose. Ceux qui font cela n'ont généralement pas le bon goût de rendre la liste intermédiaire à la liste. Quand on est sale, on est sale...

• Pour le faire en itératif, il faut mettre le résultat dans une liste R qui augmente petit à petit par le fond, et avoir un accès direct à la fin de R où l'on va progressivement empiler, via un pointeur Q qui pointe sur le champs `suite` du dernier bloc de R (ou sur R quand R sera encore NULL), bref sur le pointeur de valeur NULL qui va devoir accueillir le prochain élément. Q est de type pointeur vers liste, i.e. pointeur vers pointeur vers bloc.



liste Copie(L)
    R = NULL
    Q = & R
    P = L
    tant que P ≠ NULL
        empile(P->valeur, Q)
        P = P->suite
        Q = & (*Q)->suite // ou Q = & (**Q).suite
    rendre R

Remarquons que l'on n'empile pas sur Q, qui n'est pas une liste mais un pointeur vers liste. On empile sur *Q, le pointeur liste (faites bien la différence entre pointeur liste, i.e. pointeur vers bloc, et pointeur VERS liste, i.e. pointeur vers pointeur vers bloc), le dernier pointeur liste de la liste chaînée qui démarre en R. En C, nous ferons un passage par adresse, donc on appellera empile(..., &*Q) i.e. empile(...,Q), mais attention aux apparences, ce n'est pas un simple appel sur Q, mais bien un appel par adresse sur *Q. Quoique si, c'est un appel par valeur de Q, empile va effectivement travailler avec un pointeur VERS liste, i.e. pointeur de pointeur Q2 qui lui permettra d'accéder au pointeur liste où il faut empiler.

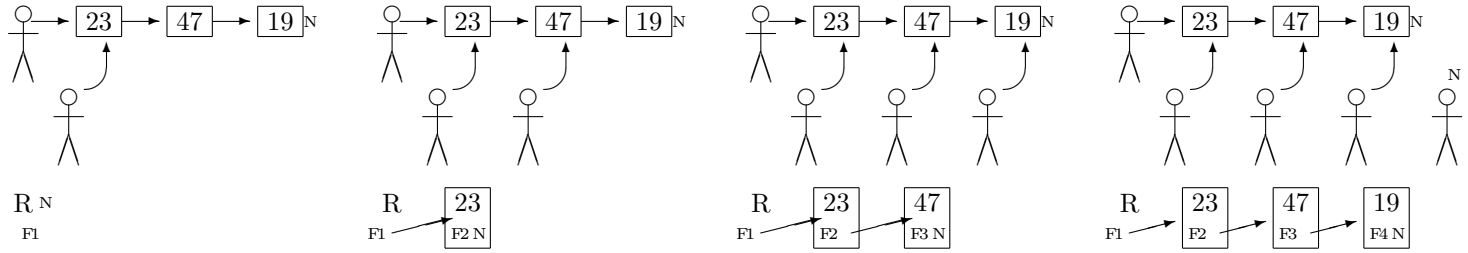
Expliquons "Q = & (**Q).suite". Q pointe sur un pointeur liste (soit R lui-même, soit un champs `suite` dans un bloc) et doit pointer sur le champs `suite` du bloc d'après. Donc on va chercher le champs `suite` du bloc d'après : On est sur Q qui pointe vers le pointeur liste, on fait Q*, on est sur R ou sur le champs `suite` dans le bloc, on fait **Q, on est sur le bloc qui suit, on fait (**Q).suite, on est sur le pointeur liste qui est le champs `suite` du bloc d'après. Q doit pointer là dessus, i.e. qu'il doit prendre comme valeur l'adresse du pointeur `suite` dans le bloc, d'où le & que l'on ajoute : & ((**Q).suite).

Note, vu que `->`remplace (*)., on peut remplacer & (**Q).suite par & (*Q)`->`suite. L'étoile qui reste est celle qui est collée à Q, et qui fait passer de Q au pointeur R, ou au champs `suite` dans le bloc.

• Certaines personnes, qui ne savent pas gérer les pointeurs de pointeur, vont plutôt faire un pointeur liste vers le dernier bloc de R. C'est lourd et maladroit. Il faut gérer à part le premier bloc, et les manips sont plus lourdes avec des *.suite plein le code :

liste Copie(L)
    R = NULL
    si L ≠ NULL
    alors {
        empile(L->valeur, & R) CODE MOCHE
        Q = R
        P = L->suite
        tant que P ≠ NULL
            { empile(P->valeur, & Q->suite)
            P = P->suite
            Q = Q->suite }
        }
    rendre R

• Pour faire du récursif terminal, c'est la même idée qu'en itératif, on aura un pointeur R qui sera le résultat et un pointeur F en fin de la liste de R (celui qui est pointé par Q dans la version précédente). On aura une procédure récursive et une sur-fonction.




```
liste Copie(liste L)
        liste R = NULL
        Bis(L, & R)
        rendre R

void Bis(liste L,liste *Q)
        Si L ≠ NULL
        alors
                empile(L->valeur, Q)          // i.e. empile(..., & *Q)
                Bis(L->suite, & (*Q)->suite)
```

5. **LongueurK (L,K)** qui rend vrai ssi la longueur de L est K

- Attention, rendre (longueur(l) == K) est pourri pour cause de complexité. Si je fais LongueurK(Annuaire de Paris, 2), il est inutile de parcourir tout l'annuaire pour dire qu'il n'est pas de longueur 2.

```
bool LK(L,K)
        Si L == NULL
        alors rendre (K == 0)
        sinon
        si K == 0 // et L ≠ NULL
        alors rendre faux
        sinon rendre LK(L->suite, K-1)
```

remarques : on peut inverser l'ordre du test L et du test K

```
LK(L,K)
        Si K == 0
        alors rendre L == NULL
        sinon
        si L == NULL // et K non nul
        alors rendre faux
        sinon rendre LK(L->suite, K-1)
```

La version suivante est plus lourde (et fait plus de tests à l'exécution)

```
        si L == NULL et K == 0
        alors rendre vrai
        sinon
        si L == NULL ou K == 0
        alors rendre faux
        sinon ...
```

La suivate est lourde également (mais il vaut mieux faire lourd que faux....)

```
        si L == NULL et K == 0 alors ...
        sinon si L == NULL et K ≠ 0 alors ...
        sinon si L ≠ NULL et K == 0 alors ...
        sinon ...
```

Que se passe-t-il si on oublie de tester K == 0 et qu'on ne teste que L vide ? :

```
bool LK(L,K)
        Si L == NULL
        alors rendre (K == 0)
        sinon rendre LK(L->suite, K-1)
```

Et bien, ça marche, mais la complexité est médiocre, cf annuaire de paris et 2 : l'algo va parcourir tout l'annuaire, K va passer négatif, et a la fin, on constatera que K, très très négatif, n'est pas nul.

Au passage, si l'utilisateur lance avec un argument négatif, K dérive de plus en plus négatif et on parcourt la liste. Pour éviter cela : soit faire un test "si K <= 0 // et L non vide", ou mieux, faire une surfonction qui vérifie que K n'est pas strict négatif, de sorte que ce test n'est fait qu'une fois plutôt que une fois par appel.

6. **ComptageOccurrences** qui prend deux listes m et t de lettres, m étant interprété comme un mot, et t comme un texte, et rend le nombre d'apparitions du mot m dans le texte t. Exemple:
   $ComptageOccurrences([a,b,c,a,b],[b,a,b,c,a,b,b,b,a,a,b,c,a,b,c,a,b])$ rend 3.

   - On notera via l'exemple que si deux occurrences se chevauchent, elles sont comptées toutes les deux.

     On est sur un "diviser pour régner", qui fait non seulement un appel récursif mais aussi un appel à une fonction différente.

     Il y a deux listes en argument, on peut donc regarder quel appel récursif on va faire. Il y a a priori trois options : CA(m->suite, t->suite), CA(m->suite,t), CA(m, t->suite). C'est le troisième qui est pertinent, qui compte les occurrences de m dans t sauf celle éventuelle qui est au tout début. Il faudra donc prendre ce résultat récursif et ajouter 1 si m est préfixe de t, on va donc devoir écrire une nouvelle fonction annexe IntPrefixe(m,t) qui rendra 1 si m est préfixe de t et 0 sinon.

     Revenons à CA : l'appel récursif se fait sur t->suite, donc on fera un test "si t vide", mais pas besoin de faire un test sur "m vide"

```
int CA(m,t)
    si t == NULL
        alors si m == NULL alors rendre 1 sinon rendre 0
    sinon rendre IntPrefixe(m,t) + CA(m, t->suite)
```

     Écrivons à présent IntPrefixe. On voit d'abord que l'on appellera sur m->suite et t->suite, donc il faudra prégérer les cas m vide et t vide. Pour bien gérer ces cas de base et ne pas en oublier, on peut faire un tableau qui indique si m est préfixe de t en fonction de ce que l'on doit rendre en fonction des cas de base :

	t vide	t non vide
m vide	vrai	vrai
m non vide	faux	tval==mval ET Rec()

     Vide est préfixe de non vide, d'ailleurs, si on teste sur ab et abcd, on va rappeler sur b et bcd puis vide et cd, et c'est grâce à cette réponse que l'on va dire que ab est préfixe de abcd. De même, vide est préfixe de vide, et c'est grâce à cette réponse que l'on va dire que ab est préfixe de ab (test qui sera envoyé par CA si le mot apparaît tout à la fin du texte)

     On voit que si m est vide, la réponse ne dépend pas de t. Il est donc plus habile de commencer par le test m vide :

```
int IntPrefixe (m,t)
    Si m == NULL alors rendre 1
    sinon
    si t == NULL alors rendre 0
    sinon
    si m->valeur == t->valeur
    alors rendre IntPrefixe(m->suite, t->suite)
    sinon renre 0
```

     Note 1 : Ce n'est pas une question très pertinente pour un éditeur de texte (Avez-vous déjà fait contrôle F sur le mot vide ?) mais... : Combien de fois le mot vide est-il occurrence d'un texte t ? Réponse : $|t|+1$, par exemple, dans *abc*, il y est avant le a, entre le a et le b, entre le b et le c, et après le c. Observez comment le code ci-dessus donne cette réponse.

     Note 2, IntPrefixe est récursif terminal, mais pas CA. On peut avoir CA en RT par la technique habituelle de l'accumulateur.

```
int CA(m,t)
    r = 0
    Bis(m, t, & r)
    rendre r

void Bis(m, t, inout *r)
```

```
si t == NULL alors
        { si m == NULL alors (*r)++ }
sinon { *r = *r + IntPrefixe(m,t)
        Bis(m, t->suite, r) }
```

Et du coup, on peut aussi faire CA en itératif pur :

```
int CA(m,t) : int
    r = 0
    Pt = t
    tant que Pt ≠ NULL
        Pm = m
        Pt2 = Pt
        tant que Pm ≠ NULL et Pt2 ≠ NULL et Pm->valeur == Pt2->valeur
            Pm = Pm->suite
            Pt2 = Pt2->suite
        si Pm == NULL alors r++
        Pt = Pt->suite
    si m == NULL alors r++
    rendre r
```

Note 3 : les algos dans cette réponse sont de complexité $O(|m|*|t|)$. Il existe plein d'algos plus performants. On peut faire du $O(|m|+|t|*|A|)$ avec les automates (Faire l'automate déterministe minimal reconnaissant les occurrences de m, puis le passer par t. Le défi étant de construire l'automate en temps $O(|t|*|A|)$

7. **AjouteTrie(…)** qui prend pour arguments x et une liste l supposée triée et y insère x (le résultat étant trié, bien sûr). Faire deux versions: une version fonction (rend la nouvelle liste) et une version procédure (modifie l).

   - En fonction : (ne pas oublier de ré-ajouter L->valeur au résultat de l'appel trié)

```
liste inseretrie (x,L)
    si L == NULL
    alors rendre ajoute(x, NULL) // (ou ajoute(x,L) )
    sinon
    si x <= L->valeur
    alors rendre ajoute(x,L)
    sinon rendre ajoute(L->valeur, inseretrie(x, L->suite))
```

que je peux simplifier en

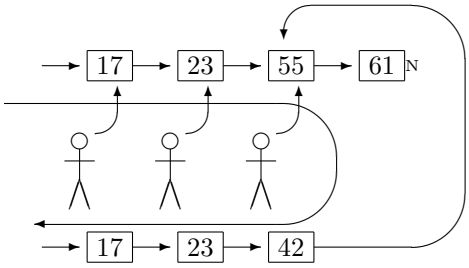
```
liste inseretrie (x,L)
    si L == NULL ou x <= L->valeur
    alors rendre ajoute(x,L)
    sinon rendre ajoute(L->valeur, inseretrie(x, L->suite))
```

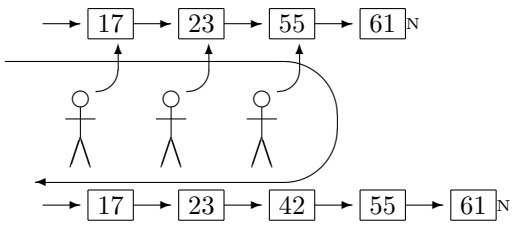
Note 1 : (dessin ci-dessous à gauche) si on regarde ce qui se passe pour la mémoire, on s'aperçoit que les blocs des valeurs plus petites que x sont dédoublés, ceux du résultat ne sont pas les mêmes que ceux de L. Par contre, sur la fin de la liste, les blocs de L et de R sont les mêmes. Bref, la liste R est une copie de L jusqu'à x, puis ce n'est plus une copie. Qu'en penser ?

Option 1 : les listes sont faites pour être modifiées, et alors c'est très bizarre, car si je modifie la liste L, la modification se répercutera sur R si elle est en fin de liste et ne se répercutera pas si elle est en début de liste. Il faudrait ici que tous les blocs soient copiés, y compris ceux de la fin. (dessin ci-dessous à droite)

Si l'on veut cela, il suffit de remplacer "rendre ajoute(x,L)" par "rendre ajoute(x, copie(L))"

Option 2 : On ne modifiera pas les listes, on ne fait que la lecture et de la création, dans ce cas, on ne copie que le nécéssaire pour économiser du temps et de la mémoire, et on laisse comme ça (dessin ci-dessous à gauche). C'est le point de vue de Caml (ce qui explique par exemple que append(L1, L2) va copier L1 mais pas L2, et être de complexité $\theta(|L1|)$)





Note 2 : x <= L->valeur ou x < L->valeur ?
Dans le premier cas, si x existe déjà, le nouveau est inséré avant les autres. Dans le deuxième cas, c'est après. Vu que la spécification n'impose rien, je choisis la première qui est plus rapide.

En procédure :

```
void inseretrie (x, inout * L)
    si *L == NULL ou x <= (*L)->valeur
    alors empiler(x,L)
    sinon inseretrie(x, & (*L)->suite)
```

Apparition du fameux **pointeursuite**. Cf page 23 et fin du poly.
Mettons que je doive insérer x entre le premier et le deuxième bloc. Il faut donc créer un nouveau bloc B, mettre x dans le champs élément de B, faire pointer le champs `suite` de B vers le 2e bloc de L, et faire pointer le champs `suite` du 1er bloc de L vers B.
C'est ce dernier point qui est problématique si j'utilise L->suite: en effet, le deuxième l'appel, celui qui effectuera le empile, reçoit comme argument le champs `suite` du 1er bloc de L, mais comme il travaille par valeur, il ne reçoit pas la variable "bloc.suite", mais n'en reçoit que la valeur (l'adresse du 2e bloc) qu'il mettra dans une variable locale qui ne sera donc qu'une COPIE du champs `suite` du premier bloc. L'opération empile sera effectuée sur cette copie, et donc l'adresse du bloc B ne sera pas mise dans le champs `suite` du 1er bloc de L, mais dans la copie qu'est cette variable locale. Bref, le bloc B sera bien crée avec un champs élément contenant x et un champs `suite` contenant un pointeur vers le 2e bloc, mais le champs `suite` du 1er bloc continuera de pointer vers le "vieux" 2e bloc.
Il faudra mettre en place qqch pour que cet appel ne travaille pas avec une copie. C'est ce qu'indique le "pointeursuite"

Note : Que penser de :

```
void inseretrie (x, inout * L)
    si *L == NULL ou x <= (*L)->valeur
    alors empiler(x,L)
    sinon
        s = (*L)->valeur
        depile(L)
        inseretrie(x,L))
        empile(s,L)
```

Si je vous demande d'insérer une assiette neuve entre les la 2e et 3e assiette d'une pile, vous prenez en photo la première assiette que vous jetez, vous prenez en photo la deuxième assiette que vous jetez, vous posez l'assiette neuve sur la pile, puis vous demandez à un potier de vous faire un assiette identique à celle sur la photo 2 et vous posez l'assiette sur la pile, puis vous faites de même avec la photo 1...
Non mais groumpf non quoi !
Et ne me dites pas que ça marche. Si un jour vous reprenez votre code et que vous ajoutez un champs aux blocs, et bien ce champs ne sera pas pris en compte par votre manip et le programme sera bugé. Par ailleurs, si des pointeurs ailleurs dans la programme pointaient vers les blocs de votre liste, et bien maintenant ils pointent vers des fantômes.

Je peux le faire aussi en itératif (donc je pourrais aussi le faire en récursif terminal). Allez, cette fois-ci, je ne fais pas de copie P de l'argument local L.

```
void inseretrie (x, inout * L)
    tant que (*L) ≠ NULL et x > (*L)->valeur
        L = & (*L)->suite
    empile(x,L)
```

Et les dégueux vous diront qu'ils n'aiment pas les pointeurs de pointeurs et les `& (*L)->suite`, et qu'on peut le faire avec un pointeur en arrière, et ils feront donc la chose suivante :

```
void inseretrie (x, inout * L)
    si *L == NULL ou x <= (*L)->valeur
```

alors empile(x,L)
sinon { P = L
tant que P->suite ≠ NULL et x > P->suite ->valeur
P = P->suite
empile(x, & P->suite) }

8. **ElimineEnPremiereOccurrence** qui prend en argument une liste l, un élément x et modifie l en y éliminant la première occurrence de x. (Si x n'apparaît pas dans l, la procédure ne fait rien.)

- void EPO (x, inout *L)
Si *L == NULL alors ne rien faire
sinon
si x == (*L)->valeur
alors depile(L)
sinon EPO(x, & (*L)->suite)

9. **ElimineToutesOccurrences** qui prend en argument une liste l, un élément x et modifie l en y éliminant toutes les occurrence de x.

- void ETO (x, inout * L)
Si *L == NULL alors ne rien faire
sinon
si x == (*L)->valeur
alors
{ depile(L)
ETO(x,L) }
sinon ETO(x, & (*L)->suite)

ATTENTION AU PIÈGE : Apres le depile, il faut appeler sur L et non sur (pointeur)suite(L) car vous venez de dépiler, l'ancienne suite(L) est devenue L elle-même. Si on appelle sur `suite`, on ignore un x sur 2 quand il y a des x successifs, et ça plante si on depile un x qui est le dernier élément de la liste.
On pourrait faire un appel commun à ETO(& (*L)->suite) dans le then et le else en le faisant avant l'appel récursif, mais dans ce cas, ce n'est plus terminal.
void ETO (x, inout * L)
Si *L == NULL alors ne rien faire
sinon { ETO(x, & (*L)->suite)
si x == (*L)->valeur
alors depile(L) }

10. **ElimineDerniereOccurrence** qui prend en argument une liste l, un élément x et modifie l en y éliminant la dernière occurrence de x. (Si x n'apparaît pas dans l, la procédure ne fait rien.) Ne faire qu'une seule passe.

- L'interdiction de faire plusieurs passes vous empêche de faire des trucs du genre : je retourne la liste (une passe), puis j'enlève la première occurrence (2e passe) puis je retourne la liste (3e passe), ou bien je compte le nombre k d'occurrences de x dans la liste (une passe) puis j'enlève le k^{eme} x.

Option 1 : on procède comme pour les questions précédentes.
Quand on arrive sur un x, il est impossible de savoir s'il faut l'enlever, puisque la fin de la liste n'a pas été parcourue et que l'on ne sait pas s'il y a des x plus loin. On va donc lancer récursivement d'abord pour aller au fond de la liste, et on va profiter de cet appel pour demander si on a vu un x plus loin. Nous allons donc utiliser une sous-procédure avec une variable "vu" en type de catégorie "out", les appels ayant pour mission d'y écrire pour dire si x a été vu en fin de liste.
void EDO (x, inout *L)
int vu
Bis(x, L, & vu)

void Bis(in x, inout * L, out * vu)
si *L == NULL
alors *vu = faux
sinon { EDO(x, & (*L)->suite, vu)
si x == (*L)->valeur et non *vu

```
                alors { depile(L)
                        *vu = vrai } }
```

Ceci n'est pas du tout récursif terminal. On peut noter qu'il n'est pas nécessaire de faire attendre un appel intermédiaire s'il ne voit pas un x. Ce qui pousse à faire :

```
void EDO (x, inout * L)
int vu
        Bis(x, L, & vu)

void Bis(in x, inout * L, out * vu)
        si *L == NULL
        alors *vu = faux
        sinon si x == (*L)->valeur
                alors { EDO(x, & (*L)->suite, vu)
                        si non *vu
                        alors { depile(L)
                                *vu = vrai } }
                sinon EDO(x, & (*L)->suite, vu)
```

Du coup, la hauteur de pile sera en nombre de x plutôt qu'en longueur de L

Option 2 : On parcourt la liste en mémorisant le dernier endroit où l'on a vu un x et arrivé au bout, on dépile à cet endroit-là.

En itératif, on va utiliser un pointeur M vers liste (donc un pointeur de pointeur de bloc) qui va pointer vers le dernier pointeur liste connu qui pointe vers un bloc contenant un x. Si aucun x n'a été vu, M sera le pointeur NULL. On va parcourir la liste avec un pointeur de pointeur.

```
void EDO (x, inout * L)
        P = L
        M = NULL
        tant que *P ≠ NULL
                { { si (*P)->valeur == x
                    alors M = P } // copie de pointeur de pointeur, qui pointent donc vers le même pointeur
                P = & (*P)->suite }
        Si M ≠ NULL alors depile(*M)
```

et version récursive terminale, sur le même principe que l'itératif :

```
void EDO(x, inout * L)
        Q = NULL
        Bis(x, L, & Q)

void Bis(x, inout * L, inout * M)
        si *L est NULL
        alors { si *M ≠ NULL alors depile(*M) }
        sinon si x == (*L)->valeur
                alors Bis(x, & (*L)->suite, L)
                sinon Bis(x, & (*L)->suite, M)
```

On ne peut pas faire "depile(M)" dans la surfonction car P est un pointeur de pointeur seulement pour mémoriser un emplacement de pointeur, il ne pointera pas sur autre chose. On ne rappelle pas sur un P modifié, on rappelle soit sur P inchangé, soit sur un nouveau "P", à savoir L. Le dernier appel ne voit pas un P modifié mais un autre P. Et P n'est inout qu'à cause du depile(*P). La variable M de la surfonction vaut toujours NULL à la fin du calcul.

Et en réalité, ce n'est pas tout à fait la version itérative, car si x n'a pas été vu, M est un pointeur NULL dans la version itérative, mais pointe vers un pointeur NULL dans la version ci-dessus. On peut faire une version récursive, un poil bizarre, qui fait comme dans l'itératif. C'est M et non *M qui est NULL si x n'a pas été vu. Et Du coup, il n'y a plus besoin d'une variable M au niveau de la surfonction. Ce qui donne :

```
void EDO(x, inout * L)
        Bis(x, L, NULL) // C'est bizarre et atypique, mais oui c'est le pointeur de pointeur qui est NULL
                        // M est NULL tant que aucun x n'a été trouvé
                        // sinon M pointe sur le pointeur pointant vers le dernier bloc connu contenant x
void Bis(x, inout * L, inout * P)
        si *L est NULL
```

```
        alors { si P ≠ NULL alors depile(*P) }
        sinon si x == (*L)->valeur
                alors Bis(x, & (*L)->suite, L)
                sinon Bis(x, & (*L)->suite, P)
```

Note : Et si, pour savoir si un x que l'on trouve est le dernier, on lance EstDans(x,L) ? :

```
void EDO(x, inout * L)
        si L ≠ NULL
        alors si x == (*L)->valeur
                alors si EstDans(x, (*L)->suite)
                        alors EDO(x, & (*L)->suite)
                        sinon depile(L)
                sinon EDO(& (*L)->suite)
```

```
bool EstDans(x,L)
        si L == NULL alors rendre faux
        sinon si x == L->valeur alors rendre vrai
        sinon rendre EstDans(x, L->suite)
```

Ça ne répond pas à la question puisque l'on utilise des parcours secondaires (les "EstDans"), mais par curiosité, quelle est la complexité ?

À première vue c'est quadratique, car on a un parcours principal qui lance un parcours secondaire linéaire à chaque appel (Cf la complexité de "TousDifferents" vue en cours)

Et en fait non, à cause de deux phénomènes : EstDans n'est lancé que quand on trouve un x, et EstDans s'arrête au premier x trouvé. Du coup, chaque appel de EstDans part d'une occurrence d'un x et parcourt la liste jusqu'au x suivant. Sa complexité est la somme des distances entre les deux x.

La complexité totale des parcours EstDans est donc la somme des distances entre deux x (ou entre le dernier x et la fin), qui se majore par longueur de L (il ne manque que la distance du début au premier x)

Et donc cet algo est linéaire !!!

PS : Si vous proposez un tel algo en examen, que vous voyez qu'il est linéaire alors que ça ne saute pas aux yeux, expliquez pourquoi il est linéaire, sinon je supposerai que vous ne l'avez pas vu et vous n'aurez pas les points pour un algo linéaire.

11. **DernierePosition(l,x)** rend la position de x dans la liste L (par exemple 4 si $x = 8$ et $l = [4, 3, 1, 8, 9]$). Si x n'apparaît pas dans la liste, la fonction rendra 0. Si x apparaît plusieurs fois, la fonction rendra la position de la dernière occurrence. Écrire une version récursive sans sous-fonction, une itérative et une récursive terminale.

- Récursif sans sous-fonction, qui est pour une fois plus difficile que les versions récursives terminales ou itératives :

  On observe que si x est en DP 8 dans la suite de L, alors il est en DP 9 dans L, ce qui pousse à écrire :

  ```
  int DP (x,L)
          si L == NULL alors rendre 0
          sinon rendre DP(x, L->suite) +1
  ```

  mais c'est FAUX, ça ne lit même pas x et en fait, ça rend la longueur de la liste. Pourquoi ? Parce que la règle DP (x,L) = DP(x, L->suite)+1 n'est valable que si x est dans suite(L). Avant d'incrémenter, il faut savoir si x est dans la liste. Il serait donc logique de faire une sous-fonctionnalité avec un argument out qui dit si x a été vu. Mais l'énoncé l'interdit. On peut se passer de "vu" parce qu'en fait le résultat de l'appel récursif permet de savoir, en effet, cela revient à regarder si DP(L->suite) est non nul.

  Le fait que x == L->valeur a aussi une influence, on remplit donc un petit tableau pour s'y retrouver :

	DP == 0	DP ≠ 0
x == prem	1	Rec+1
x ≠ prem	0	Rec+1

  D'où la version :

  ```
  int DP (x,L)
          si L == NULL alors rendre 0
          sinon si DP(x, L->suite) ≠ 0
                  alors rendre DP(x, L->suite)+1
                  sinon si L->valeur == x
                          alors rendre 1
                          sinon rendre 0
  ```

Cette fois, c'est juste mais c'est POURRI : en effet, l'appel sur L lance deux appels sur suite qui lanceront 2 appels chacun sur suite de suite, ce qui fera 4 appels au lieu d'un, chacun faisant 2 appels sur suite de suite de suite, soit 8 appels au total au lieu d'un. Bref on aura 2^L appels sur NULL, la complexité est exponentielle. On pallie le problème en ne calculant qu'une fois le résultat et en le stockant dans une variable.

```
int DP (x,L)
    si L == NULL alors rendre 0
    sinon { y = DP(x, L->suite)
        si y ≠ 0
        alors rendre y+1
        sinon si L->valeur == x
            alors rendre 1
            sinon rendre 0 }
```

Une curiosité : Quand DP=0 et x=prem, on rend 1 qui se trouve être à nouveau DP+1.

	DP == 0	DP ≠ 0
x == prem	1 (=Rec+1 !)	Rec+1
x ≠ prem	0	Rec+1

Donc le code ci-dessous est correct, bien qu'á la limite de l'absurde :

```
int DP (x,L)
    si L == NULL alors rendre 0
    sinon { y = DP(x, L->suite)
        si x ≠ L->valeur ET y == 0
        alors rendre 0
        sinon rendre y+1 }
```

Itératif

```
int DP(x,L)
    p = L
    cpt = 0
    posX = 0
    tant que p ≠ NULL
        cpt++
        si L->valeur == x alors posX = cpt
        p = p->suite
    rendre posX
```

Atention, si on initialise cpt à 0, il faut l'initialiser avant le test. Ou alors on le met après le test mais on initialise à 1

```
int DP(x,L)
    p = L
    cpt = 1
    posX = 0
    tant que p ≠ NULL
        si L->valeur == x alors posX = cpt
        cpt++
        p = p->suite
    rendre posX
```

Récursif terminal, directement inspiré de l'itératif (faire attention à l'ajustement à plus ou moins un près des compteurs), avec une sous-procédure :

```
int DP(x,L)
    posX = 0
    Bis(x, L, 1, & posX)
    rendre posX
void Bis(x, L, cpt, inout * posX)
    si L ≠ NULL alors // pas de sinon
    { { si L->valeur == x alors *posX = cpt }
```

Bis(x, L->suite, cpt+1, posX) }

Le même avec une sous-fonction :

```
int DP(x,L)
        rendre Bis(x, L, 1, 0)
int Bis(x, L, cpt, posX)
        si L == NULL
        rendre posX
        sinon si L->valeur == x
                alors rendre Bis(x, L->suite, cpt+1, cpt+1)
                sinon rendre Bis(x, L->suite, cpt+1, posX)
```

ou bien

```
int DP(x,L)
        rendre Bis(x, L, 1, 1)
int Bis(x, L, cpt, posX)
...
                alors rendre Bis(x, L->suite, cpt+1, cpt)
...
```

12. **Pif(l)** qui rend le nombre d'éléments qui sont égaux à ((leur position depuis le début) fois (leur position depuis la fin)). Exemple, Pif([2,8,7,0,5]) rendra 2 (cf. le 8 (=2*4) et le 5 (=5*1)). Ne faire qu'une seule passe.

- 8 est en position 2 et retro-position 4, et 8=2*4, il nous intéresse. 5 est en position 5 et retro-position 1, et 5=5*1, il nous intéresse. 7 est en position 3 et retro-position 3, et 7≠3*3, il ne nous intéresse pas. De même, 2 et 0 ne nous intéressent pas. Il y a deux élément intéressants, donc on rend 2.

• • La première chose à comprendre est qu'il faut une sous-procédure pour manipuler tous les arguments supplémentaires à gérer.
La seconde est de les énumérer et de donner leur statut :

• La liste L, en "in".

• Les positions depuis le début et depuis la fin. Celle depuis le début se propage depuis la tête vers le fond, elle est donc "in". Celle depuis le fond se propage depuis le fond vers la tête, elle donc "out".

• Il faut enfin un cpt qui donnera le résultat. On peut se demander s'il sera "inout" ou "out". Nous pouvons dire "inout" par défaut. Il s'avérera que nous pouvons le gérer en "out".

• Pif2(L, in posG, out posD, out cpt).

• • Puis il faut les gérer :

• Les arguments IN sont initialisés dans la surfonction et gérés AVANT l'appel récursif (posG+1 est sur la même ligne que l'appel récursif Pif2, mais à l'exécution, PosG+1 sera évalué avant de lancer l'appel récursif).

• Les arguments OUT sont initialisés au bout de la liste et sont gérés APRES l'appel récursif. Les variables des arguments out sont déclarés dans la surfonction et envoyés non initialisés à la sous-fonction.

• Les arguments INOUT : Cela dépend ... Il est très fréquent qu'ils soient initialisés dans la surfonction. Ils sont quoiqu'il en soit déclarés dans la surfonction.

• • Pour le cpt, il sera incrémenté quand nous trouverons un élément égal à posG*posD. Nous connaîtrons posG rapidement, avant l'appel récursif, mais nous devrons attendre après l'appel récursif pour connaître posD. Nous devrons donc attendre après l'appel récursif pour gérer cpt, en faisant cpt++. Du coup, il n'est pas utile d'initialiser cpt très en avance. Nous pouvons l'initialiser en fin de liste et en faire un argument "out". Nous pourrions le gérer en inout en l'initialisant dans la surfonction au lieu de la faire en fond de liste, ce qui reviendrait à l'initialiser très en avance par rapport à la version avec compteur out.

• Une fois que tout est compris, le code est finalement facile...

```
int Pif(L)
        int posD, cpt
        Pif2(L, 1, & posD, & cpt)
        rendre cpt

void Pif2(L, in posG, out * posD, out * cpt)
        si L == NULL
        alors {*cpt = 0
```

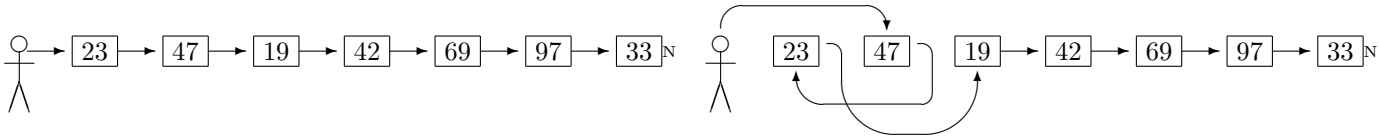
*posD = 0 }
sinon { Pif2(L->suite, posG+1, posD, cpt)
(*posD) ++
si L->valeur == posG*(*posD) alors (*cpt) ++ }

Note :

Attention à bien gérer posG et posD pour que les premier et dernier élément soient respectivement en position 1 et rétroposition 1.

13. **Swap (…)**, qui intervertit les deux premiers blocs de l. Si l vaut [23,47,19,42,69,97,33] avant l'appel , elle vaut [47,23,19,42,69,97,33] après.

- Note : il faut faire des déplacements de pointeurs.



Ne pas faire des malloc (via empile) et des desallocation (via depile) :
x = (*L)->valeur ; depile(L) ; y = (*L)->valeur ; depile(L) ; empile(x, L) ; empile(y, L) ;
Pour intervertir les deux premières pages d'une pile de feuilles, votre protocole est-il ? : prendre en photo 1 le sommet de pile, brûler la feuille du sommet de pile, prendre en photo 2 le sommet de pile, brûler la feuille du sommet de pile, faire une impression de la photo 1 que vous mettez sur la pile, faire une impression de la photo 2 que vous mettez sur la pile.
Non ? Alors ne faites pas cela avec des structures chaînées.

Ne pas changer non plus les champs valeur des blocs :
tmp = L->valeur, L->valeur = L->suite->valeur, L->suite->valeur = tmp.
Pour intervertir les deux premières pages d'une pile de feuilles, votre protocole est-il ? : recopier la première feuille, celle du sommet, sur une feuille annexe, gommer la première feuille et recopier la seconde sur la première, gommer la seconde et recopier la feuille annexe sur la seconde feuille.
Non ? Alors ne faites pas cela avec des structures chaînées.

Remarquez que si quelqu'un, vous ou autrui, réutilise votre code en ajoutant un champs aux blocs, les deux méthodes qui précédent perdent les informations supplémentaires ou les déplacent. Il y aura aussi des soucis si des pointeurs extérieurs pointent sur les blocs de cette liste.

On doit donc intervertir trois pointeurs de façon circulaire:
*L = (*L)->suite,
(*L)->suite = (*L)->suite->suite,
(*L)->suite->suite = *L
Pour intervertir 2 valeurs a,b, on passe par un tmp et on fait tmp = a ; a = b ; b = tmp ; Notez que l'on peut aussi faire la variante tmp = b ; b = a ; a = tmp.
Pour intervertir de façon circulaire 3 valeurs a,b,c, on utilise le même principe : tmp = a ; a = b ; b = c ; c = tmp ; Et l'on peut faire les variantes qui commencent par tmp = b ; ou par tmp = c ; Nous noterons que dans ce genre de séquence, la variable que l'on lit dans une ligne est celle dans laquelle on écrit la ligne suivante, c'est normal, on a sauvé son ancien contenu, à présent, on peut y écrire.

void Swap (liste *L)
si *L ≠ NULL et (*L) ->suite ≠ NULL
alors
tmp = (*L)->suite
(*L)->suite = tmp->suite
tmp->suite = *L
*L = tmp

Dans la version ci-dessus, nous aurions pu écrire (*L)->suite = (*L)->suite->suite au lieu de tmp->suite. Mais pas
(*L)->suite = (*L)->suite->suite
(*L)->suite->suite = *L // FAUX

En effet, la variable lue sur une ligne est la variable écrite la ligne suivante, mais elle n'est pas forcément accessible par les mêmes expressions. Ici (*L)->suite->suite ne désigne pas la même chose avant et après la première ligne car (*L)->suite change.

```
void Swap (liste *L)
si *L ≠ NULL et (*L)->suite ≠ NULL
alors
        tmp = *L
        *L = (*L)->suite
        tmp->suite = (*L)->suite
        (*L)->suite = tmp
```

```
void Swap (liste *L)
si *L ≠ NULL et (*L)->suite ≠ NULL
alors
        tmp = (*L)->suite->suite
        (*L)->suite->suite = *L
        *L = (*L)->suite
        (*L)->suite->suite = tmp
```

14. Que calcule la fonction Truc ? :

```
Pof (inout * P, in l, out * Resultat)
si l == NULL alors *Resultat = vrai
else    Pof(P, l->suite, Resultat)
        si *P->valeur ≠ l->valeur alors *Resultat = faux
        *P = (*P)->suite
```

```
bool Truc(l)
bool resultat ;
liste p = l ;
Pof (inout & p, in l, out & resultat) ;
rendre resultat ;
```

- Truc rend vrai ssi la liste est un palindrome.

  La surfonction Truc crèe un pointeur p qui est sur le premier élément de L, ce sera une variable globale aux appels Pof, passée en inout. De même, résultat est une variable globale passée en out, ces deux variables sont déclarées par Truc.

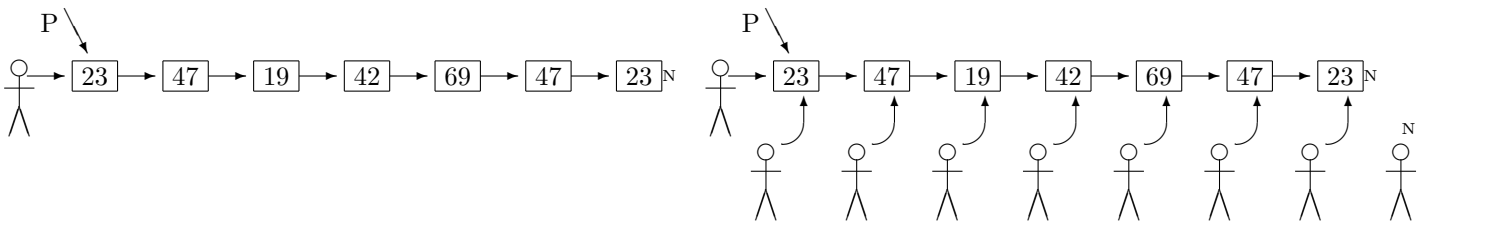
  Puis Truc lance la sous-procédure récursive Pof. On observe que, quand son argument est non vide, Pof commence par appeler récursivement. Donc les appels récursifs s'enchaînent sans que rien ne se passe jusqu'à ce que le fond soit atteint.

  Le dernier appel, dont le pointeur local L est NULL, initialise résultat à vrai avant de repasser la main à l'appel qui pointe sur le dernier bloc du L global.

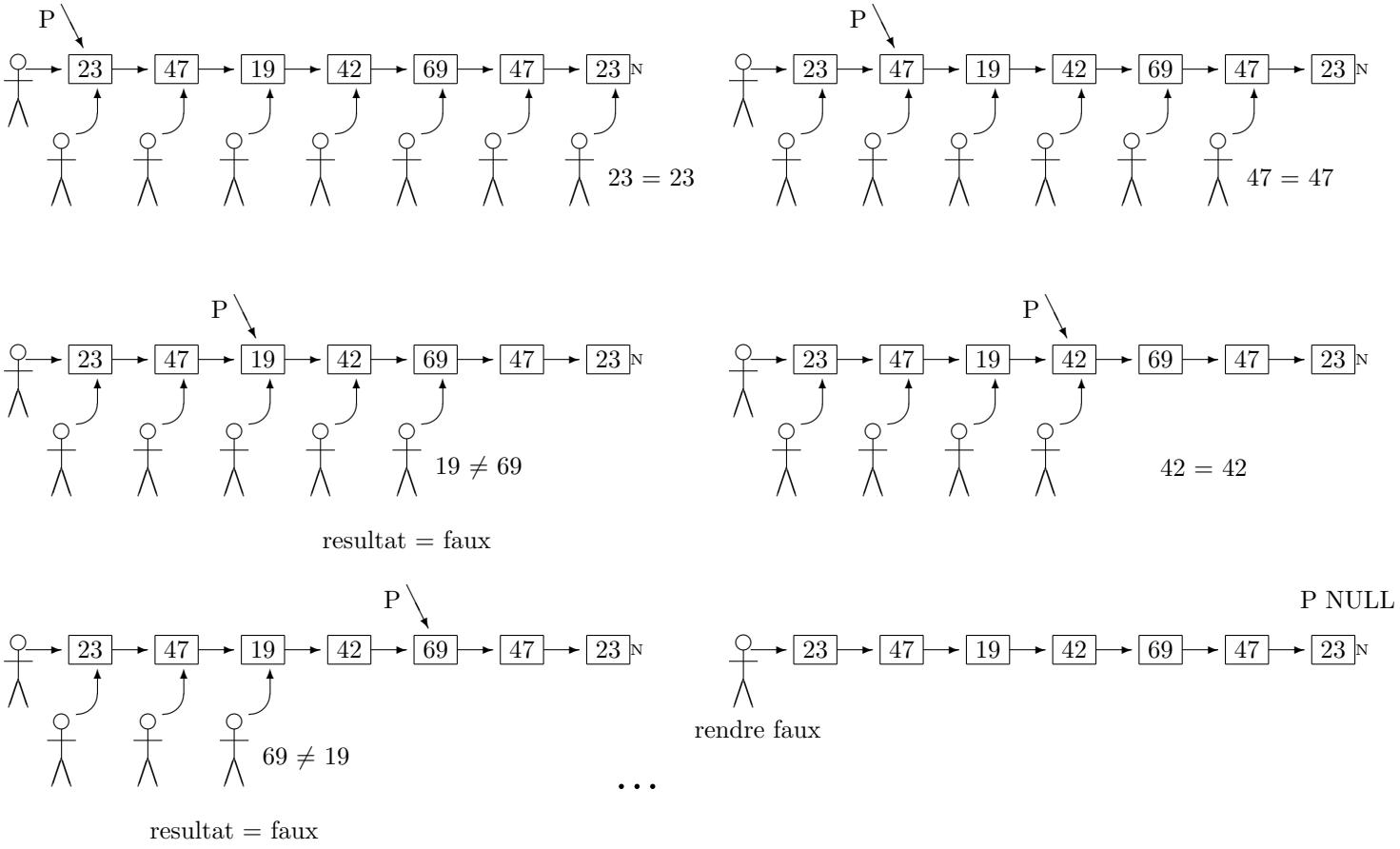
  Celui-ci vérifie si P->valeur, i.e. le premier du L global car P n'a pas bougé, est égal au premier de son L local, i.e. le dernier élément du L global. Il fait avancer P qui pointe à présent sur le deuxième élément du L global, puis il rend la main à l'appel qui pointe sur l'avant-dernier bloc du L global.

  Ce dernier vérifie si P->valeur, i.e. le second du L global car P vient d'avancer d'un élément, est égal au premier de son L local, i.e. l'avant-dernier élément du L global. Il fait avancer P qui pointe à présent sur le troisième élément du L global, puis il rend la main à l'appel qui pointe sur l'avant-avant-dernier bloc du L global.

  Ce dernier compare le troisième et l'avant-avant-dernier éléments, puis fait avancer P ... etc.



resultat = vrai



15. **fusion (l1,l2)** qui rend l'union triée des listes $l1$, $l2$ en argument, supposées triées.

- La fusion va commencer par le plus petit des premiers de L1 et de L2, mettons que ce soit L1, puis il y aura ... la fusion de (la suite de L1) et de L2, donc ce sera ajoute(l1->valeur, fusion(l1->suite,l2)). Reste à faire le cas où cela commence par le premier de L2, puis les cas de base.

```
liste fusion (l1,l2)
        si l1 == NULL alors rendre l2
        sinon
        si l2 == NULL alors rendre l1
        sinon
        si l1->valeur < l2->valeur
        alors rendre ajoute (l1->valeur, fusion(l1->suite, l2))
        sinon rendre ajoute (l2->valeur, fusion(l1, l2->suite))
```

Cf le phénomène de "ajoutetrie". Quand on fait "rendre l1" / "rendre l2", on obtient un espace mémoire partagé entre la fin de L1 ou L2, et la fin du résultat. Si on veut un espace indépendant, on fera :

```
liste fusion (l1,l2)
        si l1 == NULL alors rendre copie(l2)
        sinon
        si l2 == NULL alors rendre copie(l1)
        sinon ...
```

Pour faire de l'itératif ou du récursif terminal, il faut utiliser la même technique que pour copie.

```
liste fusion (l1,l2)
        R = NULL
        Q = & R
        P1 = L1, P2 = L2
        tant que P1 ≠ NULL et P2 ≠ NULL
                { { si P1->valeur ≤ P2->valeur
                alors { empile(P1->valeur, Q)
                        P1 = P1->suite }
                sinon { empile(P2->valeur, Q)
```

```
                P2 = P2->suite }
            }
        Q = & ((*Q)->suite)
        }
    rendre R
```

16. **Fmiroir** et **Pmiroir** qui prennent une liste en argument. La première est une fonction qui rend une copie miroir. La seconde est une procédure qui transforme l en son miroir. Miroir de *algorithme*, c'est *emhtirogla*.

- La version suivante est quadratique.
  Par ailleurs, elle fait plein de fuite mémoire car tout le résultat est recopié à chaque appel.

```
liste miroir (l)
    si l == NULL
    alors rendre NULL
    sinon ajouteenqueue(l->valeur, miroir(l->suite))

liste ajouteenqueue (x,l)
    si l == NULL
    alors rendre ajoute(x, NULL)
    sinon rendre ajoute(l->valeur, ajouteenqueue(x, l->suite))
```

Pour enlever la fuite mémoire :

```
liste miroir (l)
    si l == NULL
    alors rendre NULL
    sinon { R = miroir(l->suite)
        ajouteenqueue(l->valeur, & R)
        rendre R }
void ajouteenqueue (x, inout *l)
    si *l == NULL
    alors empile(x,l)
    sinon ajouteenqueue(x, & (*l)->suite)
```

comment faire linéaire ?: comment fait-on pour inverser une pile d'assiettes ? on déplace les assiettes une par une de la pile à une autre. Pour programmer, on fait pareil, on voit alors que l'on a temporairement deux piles, une pile à vider et une à remplir. On peut noter que c'était la version bugée de Copie parce que à l'envers. Or ici, on veut justement une copie à l'envers.

en itératif

```
liste miroir (l)
    p = l
    l2 = NULL
    tant que p ≠ NULL
        { empile(p->valeur, & l2)
        p = p->suite }
    rendre l2
```

et en récursif (terminal) avec sous-procédure

```
liste miroir (l)
    l2 = NULL
    miroirBis(l, & l2)
    rendre l2

void miroirBis (l, inout * l2)
    si l ≠ NULL
    alors { empile(l->valeur, l2)
        miroirBis (l1->suite, l2) }
```

ou avec sous-fonction :

```
liste slave (l1,l2)
    si l1 == NULL
```

```
        alors rendre l2
        sinon rendre slave(l1->suite, ajoute(l1->valeur, l2))

liste miroir (l)
        rendre slave (l, NULL)
```

PROCEDURE Pmiroir

Il n'est pas question de faire des malloc (en utilisant ajoute ou empile) ou des free (en utilisant depile). Pour renverser une pile de feuilles, la technique n'est pas de faire des phocopies en remplissant les poubelles jaunes d'intermédiaires inutiles.
Il faut uniquement déplacer les pointeurs

En itératif :

```
void Pmiroir(inout * L)
        L2 = NULL
        tant que *L ≠ NULL
                tmp = (*L)->suite
                (*L)->suite = L2
                L2 = *L
                *L = tmp
        *L = L2
```

En fait, on effectue un échange de trois valeurs a b c, ce qui se fait par tmp = a, a = b, b = c, c = tmp. avec ici pour a b c : L, (*L)->suite et L2. On peut commencer aussi par tmp = b, ou tmp = c, ce qui donne les variantes :

```
                tmp = *L
                *L = (*L)->suite // ou tmp->suite
                tmp->suite = L2
                L2 = tmp
```

et

```
                tmp = L2
                L2 = *L
                *L = (*L)->suite // ou L2->suite
                L2->suite = tmp
```

récursif

```
void Pmiroir(inout *L)
        L2 = NULL
        PM(L, & L2)
        *L = L2

void PM(inout *L, inout *L2)
        si *L ≠ NULL
        alors { tmp = (*L)->suite
                (*L)->suite = *L2
                *L2 = *L
                *L = tmp
                PM(L,L2) }
```

17. Une liste $l = (x_1, ..., x_n)$ est un interclassement de l_1 et de l_2 ssi on peut partitionner $[1..n]$ en deux parties I et J telle que l_1 soit la sous-liste des éléments de L dont les indices sont dans I, l_2 celle des éléments dont les indices sont dans J. Par exemple, la liste des interclassements de $[1, 2, 3]$ et de $[4, 5]$ est

$[[1, 2, 3, 4, 5], [1, 2, 4, 5, 3], [1, 4, 5, 2, 3], [4, 5, 1, 2, 3], [1, 2, 4, 3, 5], [1, 4, 2, 5, 3], [4, 1, 5, 2, 3], [1, 4, 2, 3, 5], [4, 1, 2, 5, 3], [4, 1, 2, 3, 5]]$

Écrire le pseudo-code de **Interclassements** qui prend en argument l_1 et l_2 et rend la liste des interclassements de l_1 et de l_2 (Si l_1 et l_2 ont des éléments en commun, il est autorisé et normal d'obtenir plusieurs fois une même liste. Par exemple, $[1, 2, 2, 3]$ devrait apparaître deux fois dans `Interclassements` de $[1, 2]$ et de $[2, 3]$)

- La bonne façon de voir les choses est de dire que (sauf cas de base) les interclassements se partitionnent en deux : ceux qui commencent par le premier de L1 et ceux qui commencent par le premier de L2.

```
fonction listeinterclassements (l1,l2) : liste de liste d'entiers
        si l1 == NULL
        alors rendre [l2]
        sinon          si l2 == NULL
        alors rendre [l1]
        sinon rendre concatene(
                            AETTL(premier(l1), listeinterclassements(suite(l1), l2)),
                            AETTL(premier(l2), listeinterclassements(l1, suite(l2))
                                )
```

AETTL et concatene ont été faits pour "permutation", dont voici le pseudo-code :

```
fonction permutation (n) : liste de listes d'entiers
            rend la liste de toutes les permutations de [1..n]
si n == 0
alors rendre [[]]
sinon rendre ATPTL (n, permutation(n-1))

fonction ATPTL(n, liste de liste d'entiers ll) : liste de listes d'entier
                        /* Ajoute n a Toutes les Positions de Toutes de les Listes de ll */
si ll == []
alors rendre []
sinon rendre concatene ( ATP(n, premier(ll)), ATPTL(n, suite(ll)) )

fonction concatene (l1, l2)
si l1 == []
alors rendre l2
sinon rendre ajoute(premier(l1), concatene(suite(l1), l2))

fonction ATP(n, liste d'entiers l) : liste de liste d'entiers
                        /* Ajoute n a Toutes les Positions de l */
si l == []
alors rendre [[n]]
sinon rendre ajoute(ajoute(n,l),
                        AETTL(premier(l), ATP(n, suite(l)) )

fonction AETTL(x,liste de liste d'entiers ll) : liste de listes d'entiers
                        /* Ajoute x en tete de Toutes les Listes de ll */
si ll == []
alors rendre []
sinon rendre ajoute(ajoute(x, premier(ll)),
                        AETTL(x, suite(ll)))
```

Feuille de TD N^o 3 : Listes chaînées et pointeurs

2 & et *

Comparez $*(\&n)$ et n. Comparez $\&(*p)$ et p.

- Pour le premier : n est une variable de type truc, &n est un pointeur vers un truc, ce n'est pas une variable, sa valeur est l'adresse de n. Puis * de ce pointeur est la variable pointée par le pointeur, i.e. la variable dont l'adresse est la valeur du pointeur. Bref *(&n), c'est la variable n.

  Pour le second : p est manifestement un pointeur, *p est la variable pointée par p, et &(*p) a pour valeur la valeur de p. Mais il y a deux différences entre &(*p) et p. La première est que p est une variable mais pas &(*p) (alors que *(&n) et n sont la même variable). Deuxième différence, pour faire &(*p), il faut que la variable *p existe et que l'on y ait suffisamment accès pour la considérer et prendre son adresse, si ce n'est pas la cas, &(*p) peut générer une erreur. Par exemple, si p est NULL, alors &(*p) génère un Segmentation Fault mais pas p.

3 Echange

Que font les programmes suivants ?

```
void echange_1 (int x, int y)
{
int tmp ;
    tmp = x ;
    x = y ;
    y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_1(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_2 (int *x, int *y)
{
int *tmp ;
     tmp = x ;
     x = y ;
     y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_2(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_2 (int *x, int *y)
{
int *tmp ;
     tmp = x ;
     x = y ;
     y = tmp ;
 }

void main()
{
int a = 2 ;
int b = 3 ;
echange_2(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_3 (int *x, int *y)
{
int tmp ;
    tmp = *x ;
    *x = *y ;
    *y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_3(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_3 (int *x, int *y)
{
int tmp ;
     tmp = *x ;
     *x = *y ;
     *y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_3(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_4 (int *x, int *y)
{
int *tmp ;
     *tmp = *x ;
     *x = *y ;
     *y = *tmp ;
 }

void main()
{
int a = 2 ;
int b = 3 ;
echange_4(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

- On rappelle qu'en C, tout appelant envoie les **valeurs** de ses arguments. L'appelé les récupère et les copie dans des variables locales.

  L'appel par adresse d'une variable est en réalité un passage par valeur d'un pointeur vers la variable. La valeur de ce pointeur, i.e. l'adresse de la variable pointée, est copiée dans une variable POINTEUR chez l'appelé, l'appelé a donc un pointeur COPIE du pointeur de l'appelant, mais qui pointe vers la même variable.

  La procédure "main" va demander au gestionnaire de la mémoire pour a et b. Mettons qu'il obtienne les mémoires 007 et 008. La procédure "echange" fera de même pour x, y, et tmp, mettons qu'il obtienne les mémoires 017, 018, et 019.

  (1) 2 3 : c'est un passage par valeur, x et y sont des copies de a et b, et c'est x et y que l'on échange.

  Le main envoie les valeurs de ses arguments, 2 et 3, qui sont récupérées par echange_1 qui les copie dans x et y.

  Le registre 017 contient la valeur 2 et le registre 018 contient la valeur 3.

  Puis echange_1 fait un échange de x et de y, i.e. du contenu des mémoires 017 et de 018. Et a et b, ie le contenu des mémoires 007 et 008, sont inchangées. Ce sont des copies de a et de b qui ont été échangées.

  (2) 2 3 : x et y sont bien des pointeurs vers a et b, mais on échange x et y au lieu de *x, i.e. a et *y i.e. b. À présent, x,y pointent vers b et a respectivement, mais a et b n'ont pas changé.

  Le main envoie les valeurs de ses arguments, mais comme ce sont des pointeurs vers a et b, in fine, ce sont les adresses de a et b qui sont envoyées, la valeur d'un pointeur étant l'adresse de la variable pointée. Le main envoie donc 007 et 008.

  echange_2 les copie dans x et y, i.e. que le registre 017 contient la valeur 007 et le registre 018 contient la valeur 008, donc x et y pointent vers a et b.

  Puis echange_2 fait un échange de x et de y, donc le registre 017 prend la valeur 008 et le registre 018 prend la valeur 007. À présent, c'est x qui pointe vers b et y qui pointe vers a. Mais le contenu de a et de b n'a pas bougé. On a intervertit les POINTEURS au lieu des variables pointées par les pointeurs.

(3) 2 3 : C'est la version précédente, sauf que le programmeur a oublié de mettre les "&" dans les appels (bug très classique).

À la compilation, un problème de typage va être découvert car echange_2 déclare des arguments pointeurs et le main envoie des entiers.

Dans un langage propre sur le typage, ça ne compile pas. Mais C dit que les pointeurs étant des entiers, il peut le faire quand même. Le compilateur se contente de faire un warning.

À l'exécution, le main envoie les entiers 2 et 3, echange_2 reçoit ces valeurs et les interprète comme des adresses qu'il copie dans x et y. Le registre 017 contient la valeur 002 et le registre 018 contient la valeur 003.

x et y sont des pointeurs vers les registres 002 et 003. Ces derniers existent-ils ? Sont-ils accessibles ? On ne sait pas trop.

Puis echange_2 fait un échange de x et de y, donc le registre 017 prend la valeur 003 et le registre 018 prend la valeur 002.

Rien n'a planté, même si les registres 002 et 003 n'existent pas, car à aucun moment, on n'a cherché à accéder à ces registres.

(4) 3 2 : la bonne méthode

Le main envoie les valeurs de ses arguments, 007 et 008.

echange_3 les copie dans x et y, i.e. que le registre 017 contient la valeur 007 et le registre 018 contient la valeur 008, donc x et y pointent vers a et b.

Puis echange_3 fait un échange de *x et de *y, or *x, c'est la variable pointée par x, donc c'est a, et la variable *y, c'est b.

tmp = *x. L'adresse de tmp est évaluée, c'est 019. La valeur de *x est évaluée, x est le registre 017, *x est le registre dont l'adresse est la valeur du registre x, i.e. du registre 017. Le registre 017 contient la valeur 007, donc *x est la variable du registre 007. La valeur de *x est donc la valeur du registre 007, donc 2. Donc 2 est écrit dans le registre 019.

*x = *y. L'adresse de *x est évaluée, c'est 007. La valeur de *y est évaluée, c'est 3. Donc 3 est écrit dans le registre 007.

*y = tmp. 2 est écrit dans le registre 008.

Tout s'est bien passé

(5) 2 3, ou plantage Segmentation Fault, ou Bus error

C'est le 4 avec le bug "oubli de mettre les & au moment de l'appel".

C fait un warning conflit de type *int et int, mais compile quand même.

Le main envoie les entiers 2 et 3. echange_3 reçoit ces valeurs, les interprète comme des pointeurs qu'il met en x et y.

Le registre 017 contient 002 et le registre 018 contient 003.

Donc x et y pointent vers les registres 002 et 003 dont on ne sait pas s'ils existent et s'ils sont accessibles.

Échange tente alors d'échanger *x et *y i.e. les contenus des mémoires 002 et 003.

tmp = *x : possible uniquement si le registre 002 existe, et est autorisé en lecture. Si c'est possible, le contenu de ce registre 002, par exemple 7852, est écrit dans tmp, le registre 019.

*x = *y : possible uniquement si le registre 002 est autorisé en écriture, et si le registre 003 existe et est autorisé en lecture. Si c'est possible, le contenu du registre 003, par exemple 3654 est écrit dans le registre 002, et écrase l'ancienne valeur, 7852.

*y = tmp : possible uniquement si le registre 003 est autorisé en écriture. Si c'est possible, le contenu de tmp, 7852 est écrit dans le registre 003 et écrase l'ancienne valeur 3654.

Bref, soit les variables n'existent pas (Segmentaion Fault) soit elles ne sont pas accessibles en lecture (elle appartient a mon voisin ...) ou pas accessible en écriture (si elle est au voisin, heureusement que le systême m'empêche d'y écrire...) soit tout est OK et en fait c'est le pire, car on modifie le contenu de mémoires qui ont un usage par ailleurs (Imaginez que ce soient des registres mémoires utilisés pour le code de démarrage de votre ordinateur ... En vrai, de tels espaces mémoire ne seront sans doute pas accessibles en lecture et certainement pas en écriture).

De toutes façons, a et b ne sont pas échangés. Peu de chances qu'un utilisateur garde un tel code.

(6) 3 2, ou plantage Segmentation Fault, ou Bus error

C'est la quatrième version, la bonne, sauf que tmp est un *int au lieu d'un int.

Le hic, c'est que la déclaration "int *tmp" va réserver de la mémoire pour un pointeur vers un entier. Ce pointeur tmp n'est pas initialisé et contient n'importe quoi, par exemple 1568. *tmp est donc supposé être le registre 1568, on ne sait pas si ce registre existe et s'il est accessible.

*tmp = *x. On évalue l'adresse de *tmp. tmp étant un pointeur en registre 019 de valeur 1568. *tmp est le registre 1568. L'affectation n'est possible que si le registre 1568 existe et est accessible en écriture. Si c'est possible, la valeur de *x, 2 est écrite dans le registre 1568 et écrase l'ancienne valeur de 1568, par exemple 42.

*x = *y. *x aka a aka le registre 007 reçoit la valeur de *y aka b, i.e. 3, bref "a = 3".

*y = *tmp : possible uniquement si *tmp est autorisé en lecture (il est possible qu'il soit autorisé en écriture mais pas en lecture, même si c'est une configuration peu courante). Si c'est possible, 2 est écrit dans la variable b.

Situation un peu effrayante : soit ça plante, soit a et b sont échangés ce qui laisse croire que tout va bien, sauf que le registre 1568 a été corrompu. Heureusement, en pratique, ce code va plutôt planter, donc ne pas vous inciter à le garder.

4

Que pensez-vous des codes suivants ?

```
void FactBis (int n, out int *R)
si n == 0 alors *R = 1 ;
sinon { *R = (*R)*n ;
        FactBis(n-1, R) ; }
```

```
int Factorielle1 (int n)
int r ; int * p = & r ;
FactBis(n,p) ;
rendre r ;
```

```
int Factorielle2 (int n)
int * p = (int*) malloc(sizeof(int)) ;
FactBis(n,p) ;
rendre *p ;
```

- Pour commencer, FactBis est bugé. *R est supposé être out, il est effectivement initialisé sur le dernier appel, mais est modifié avant les appels récursifs. Du coup, il est modifié avant d'être initialisé. La valeur par défaut qui se trouve dans l'espace mémoire au moment de la déclaration est modifiée, puis *R prend 1 puis c'est fini. Et donc Factorielle va rendre 1.

  Deux patchs possibles : Soit on garde *R en out, mais dans ce cas, il faut faire *R = (*R)*n après l'appel récursif (et ce n'est plus terminal, mais terminal et variable "out" ne sont pas compatibles). Soit on passe *R en inout et dans ce cas, il faut l'initialiser dans la surfonction.

  Factorielle1 est maladroit, il est inutile de créer une variable p pour passer par adresse, il suffit de faire FactBis(n, & r) ;

  Factorielle2 est plus grave : un espace mémoire est créé par un malloc, mais n'est jamais rendu à la mémoire, il y a donc fuite mémoire. Il pourrait être rendu par un free, mais c'est lourd. Je ne peux pas mettre le free après le `rendre` car le code après le `rendre` n'est pas exécuté. Je ne peux pas le faire avant car si je fais un free(p), la valeur de *p n'a aucune garantie d'être restée pour le `rendre`. Seule solution : tmp = *p ; free(p) ; rendre tmp ; Bonjour la lourdeur.

  Bref, faire

```
void FactBis (int n, inout int *R)
si n ≠ 0
alors { *R = (*R)*n ;
        FactBis(n-1,R) ; }
```

```
int Factorielle1 (int n)
int r = 1 ;
FactBis(n,p) ;
rendre r ;
```