Informatique
UV21
Exercices corrigés sur les boucles

Exercice 1

Proposer un algorithme permettant de tester si une chaîne de caractères (contenue dans une variable s) est un palindrome. Le résultat (vrai/faux) sera stocké dans une variable booléenne b.
Un palindrome est une chaîne de caractères qui se lit de la même manière de gauche à droite et de droite à gauche. Par exemple, "kayak" est un palindrome mais "baobab" n'en est pas un.

Correction :

Analyse : on est donc amené à comparer les lettres d'un mot. Pour "baobab" par exemple, nous comparons la première et la 6eme (et dernière), puis la 2eme et la 5eme, puis la 3eme et la 4eme. Le mot est un palindrome si pour **toutes** les comparaisons on a la même lettre, ce qui n'est pas le cas pour "baobab".
Plus généralement, notons n la longueur (le nombre de lettres) du mot. Il s'agit donc de comparer les lettres de position 1 et n, puis 2 et n-1, puis 3 et n-2, etc. Autrement dit, il s'agit de comparer les lettres de position i et n+1-i pour i=1,2,3,…

Question : où faut-il s'arrêter ? Il suffit de s'arrêter « au milieu » du mot. Pour "baobab", qui contient 6 lettres, on s'arrête pour i=3 (comparaison des lettres 3 et 6+1-3=4). Pour "kayak" (de longueur impaire), on s'arrête pour i=2 (comparaison des lettres 2 et 4). Il est donc inutile de comparer quand i dépasse (strictement) n/2.

Solution 1 : boucle pour

Première variante (la plus naturelle) : on initialise b à vrai, et on le met à faux si l'on trouve une position où le test ne marche pas (le mot n'est pas un palindrome).

Deuxième variante (ça marche aussi…) : on compte le nombre de tests qui sont justes. Si tous les tests sont justes on met b à vrai, sinon b vaut faux. Il y a $\left\lfloor \frac{n}{2} \right\rfloor$ tests (iquo(n,2) en Maple) – attention à ne pas se tromper avec les cas n pair et n impair...

Variante 1		Variante 2

`b:=true:`
`n:=length(s):`
`for i from 1 to n/2 do`
`  if s[i]<>s[n+1-i] then`
`  b:=false:`
`  fi:`
`od:`
`print("La réponse est",b);`
		`c:=0:`
		`n:=length(s):`
		`for i from 1 to n/2 do`
		`  if s[i]=s[n+1-i] then`
		`  c:=c+1:`
		`  fi:`
		`od:`
		`if c=iquo(n,2) then`
		`  b:=true`
		`else b:=false:`
		`fi :`
		`print("La réponse est",b);`

Remarques :

- on peut faire une boucle de 1 à n (au lieu de n/2), cela marcherait tout aussi bien (en changeant le test en conséquence dans la variante 2). Simplement, on ferait des tests inutiles car on ferait chaque test 2 fois.
- Dès que l'on a trouvé un test faux, il est inutile de continuer (le mot n'est pas un palindrome). On pourrait rajouter une instruction `break:` après `b:=false:` (version 1). Cette instruction fait « sortir » de la boucle. Là encore, cela économise des tests inutiles.

Erreur classique à éviter :

Le programme suivant **ne fonctionne pas**.

```
b:=true:
n:=length(s):
for i from 1 to n/2 do
  if s[i]<>s[n+1-i] then
    b:=false:
  else b:=true:
  fi:
od:
print("La réponse est",b);
```

En effet, si le test est vrai, il ne faut pas remettre b à vrai ! Si b était déjà vrai, ça ne change rien, mais si b était faux, cela signifie qu'un test avait montré que le mot n'est pas un palindrome, b doit rester faux !
Exemple d'exécution avec "rosser". Avant la boucle, b vaut vrai. Quand i=1, on compare les lettres de position 1 et 6, le test est faux et b vaut vrai. Quand i=2, on compare "o" et "e" donc b devient faux. Quand i=3, on compare "s" et "s" donc b **redevient vrai** !

Solution 2 : boucle tant que

Il est également possible d'utiliser non pas une boucle pour mais une boucle tant que. Voici les deux variantes transformées en boucle tant que.

Variante 1		Variante 2

`b:=true:`
`n:=length(s):`
`i:=1:`
`while i<=n/2 do`
`  if s[i]<>s[n+1-i] then`
`  b:=false:`
`  fi:`
`  i:=i+1:`
`od:`
`print("La réponse est",b);`
		`c:=0:`
		`n:=length(s):`
		`i:=1:`
		`while i<=n/2 do`
		`  if s[i]=s[n+1-i] then`
		`  c:=c+1:`
		`  fi:`
		`  i:=i+1:`
		`od:`
		`if c=iquo(n,2) then`
		`  b:=true`
		`else b:=false:`
		`fi :`
		`print("La réponse est",b);`

Remarques :

- Ne pas oublier d'initialiser i (`i:=1:`) et de le mettre à jour à chaque passage dans la boucle (`i:=i+1:`).
- Ne pas mettre la mise à jour (`i:=i+1:`) dans le test (entre `b:=false:` et `fi:`). Il faut mettre à jour à chaque passage.
- On pourrait remplacer le test `i<=n/2 par (i<=n/2 and b)`. Cela permettrait de sortir de la boucle quand b est faux et, comme précédemment, d'éviter des tests inutiles.

Autre solution :

On peut aussi réécrire le mot à l'envers, puis comparer le mot obtenu avec le mot initial. Par exemple, si le mot est "baobab", l'écriture inversée est "baboab". On va alors tester les lettres des mots "baobab" et "baboab" une par une, et répondre faux si une lettre au moins diffère, vrai si toutes les lettres sont les mêmes.

```
s2:="":
for i from 1 to length(s) do
  s2:=cat(s[i],s2):
od:
b:=true:
n:=length(s):
for i from 1 to n do
  if s[i]<>s2[i] then
    b:=false:
  fi:
od:
print("La réponse est",b);
```

La première boucle consiste à créer l'image inversée du mot s (variable s2).
Comme précédemment, on pourrait faire en réalité faire une boucle uniquement de 1 à n/2, cela éviterait des tests inutiles.

Exercice 2

On considère des dépenses effectuées dans différents postes budgétaires. A chaque dépense est associée le nom du fournisseur, sous forme de liste de deux éléments (nom du fournisseur et montant de la dépense), par exemple [``Dupont", 877]. Nous considérons alors la liste de ces dépenses, donc une liste de listes LL. Un même fournisseur pourra apparaître plusieurs fois.

Par exemple: [["Dupont", 87],["Ets Moulin", 233],["Martin", 877],["Durand", 82],["Dupont", 4269],["Dupont", 321],["Martin", 921]]

Donner un algorithme permettant de construire la liste récapitulative contenant les dépenses totales faites auprès de chaque fournisseur, et de l'afficher.

Dans l'exemple, cela doit donner:[["Dupont", 4677],["Ets Moulin", 233],["Martin", 1798],["Durand", 82]]

Correction :

Dans la liste des dépenses, on ne connaît pas ni l'ordre des fournisseurs, ni le nombre de fois ou chacun d'eux apparaît. L'objectif étant de « résumer » la liste (sommer les dépenses pour chaque fournisseur), il faudra prendre en compte chaque sous-liste donc la structure algorithmique à utiliser est une boucle.

Il existe plusieurs structures pour faire des boucles. La question à se poser pour choisir concerne la connaissance ou non du nombre de cycles de la boucle. Soit ce nombre est connu et il faudra mettre en œuvre une boucle *pour* ou ce nombre n'est pas connu et il faudra mettre en œuvre une boucle *répéter* ou *tant que*. Dans notre cas, puisqu'on doit prendre en compte chaque sous liste, on connaît le nombre d'itérations, il s'agira d'une boucle pour. On peut donc pour le moment résumer la solution à :

```
> for i from 1 to nops(LL) do
  # partie relative au traitement du ieme fournisseur de la
  # liste LL
  end do;
```

Remarque :

1) le premier élément d'une liste est à la position 1 et le dernier à la position nops(liste).
2) La variable i utilisée dans la boucle ne doit pas être modifiée dans la partie relative au traitement d'un fournisseur. La valeur de cette variable est gérée par la boucle et vous assure d'accéder à tous les fournisseurs.

La boucle étant définie, le second problème concerne le traitement de la $i^{éme}$ dépense. Le problème est qu'il faut pour la $i^{ème}$ dépense prendre en compte le résultat du traitement des i-1 premières dépenses (pour savoir notamment s'il y a déjà eu des dépenses relatives au fournisseur de la $i^{ème}$ dépense). Pour pouvoir le faire, il faut avoir enregistré le résultat des traitements précédents. La seule solution est d'enregistrer les résultats précédents dans une nouvelle liste que nous appellerons *resultat*.

Sur l'exemple, la liste résultat vaudra successivement :
- i=1 : [["Dupont", 87]]
- i=2 : [["Dupont", 87],["Ets Moulin", 233]]
- i=3 : [["Dupont", 87],["Ets Moulin", 233],["Martin", 877]]
- i=4 : [["Dupont", 87],["Ets Moulin", 233],["Martin", 877],["Durand", 82]]
- i=5 : [["Dupont", 4356],["Ets Moulin", 233],["Martin", 877], ["Durand", 82]]
- i=6 : [["Dupont", 4356],["Ets Moulin", 233],["Martin", 1798], ["Durand", 82]]
- i=7 : [["Dupont", 4677],["Ets Moulin", 233],["Martin", 1798],["Durand", 82]]

Erreur classique : La liste résultat doit être initialisée. La difficulté est de savoir quand cette initialisation doit être faîte.
Le code ci-dessous est faux :

```
>     for i from 1 to nops(LL) do
      resultat = [] ;
      # partie relative au traitement du ieme fournisseur de la
      # liste LL prenant en compte les résultats des i-1
      # fournisseurs enregistrés dans resultat.
      end do;
```

Quel que soit le traitement dans la partie non encore décrite, cette solution ne peut être que fausse. A toutes les itérations, la variable résultat est initialisée à NULL effaçant le résultat enregistré au cycle précédent. Pour vous en convaincre, il suffit d'afficher la valeur de résultat juste avant *end do*.

```
>     for i from 1 to nops(LL) do
      resultat = [] ;
      # Traitement illustratif :
      resultat := op(resultat), LL[i] ;
      print(resultat);
      end do:
```

Dans cet exemple, les éléments de *LL* devraient être ajoutés les uns après les autres mais le mauvais placement de l'initialisation provoque l'effacement des valeurs enregistrées.
La solution est donc de n'initialiser *resultat* qu'une seule fois, avant la boucle. L'algorithme correct est donc :

```
>     resultat = [];
      for i from 1 to nops(LL) do
      # partie relative au traitement du ieme fournisseur de la
      # liste LL prenant en compte les résultats des i-1
      # fournisseurs enregistrés dans resultat.
      end do;
```

A présent, il faut traiter chaque nouvelle dépense en fonction du contenu de la liste résultat. Nous avons 2 cas : soit le fournisseur a déjà été pris en compte et est donc enregistré dans résultat (cas i=5,6,7 dans l'exemple), soit c'est un nouveau fournisseur du point de vue de *resultat* (cas i=1,2,3,4 dans l'exemple).

A présent il faut donc savoir si le fournisseur est déjà enregistré dans résultat ou pas. La seule solution est de parcourir la liste résultat et de regarder élément par élément si le nom du fournisseur correspond à celui qui est recherché. De nouveau, il faut prévoir une boucle et donc s'interroger sur sa nature. Connaît-on le nombre de cycles (d'itérations) ?

La réponse est non car le nombre de cycles dépend du succès de la recherche d'un fournisseur : 1) le fournisseur recherché est nouveau alors on regardera tous les éléments de la liste, le nombre de cycles est donc de *nops(resultat)* ; 2) le fournisseur n'est pas nouveau et on doit arrêter la boucle dés qu'on l'a trouvé. Comme on ne sait pas a priori quand on s'arrêtera on choisit une boucle *tant que* :

```
resultat = [] ;
for i from 1 to nops(LL) do
  while <test> do
     # recherche du fournisseur
  end do
# partie relative au traitement du ieme fournisseur
# sachant s'il est connu ou pas.

end do;
```

Remarque : <notation> signifie que c'est une partie de la déclaration qui reste à remplir.

La nouvelle boucle doit nous permettre de savoir si c'est un nouveau fournisseur ou pas. Sorti de cette boucle on devra modifier la liste *resultat* en conséquence. La boucle *while* doit prendre fin dés qu'on a trouvé le fournisseur ou qu'on a atteint la fin de la liste *resultat,* ce dernier cas correspondant à un échec de la recherche (c'est un nouveau fournisseur).
Comment écrire le test :
((<test concernant l'échec de la recherche>) and (<test concernant le nom du fournisseur>))
La boucle continue tant que le test est vrai.

<test concernant l'échec de la recherche> : il y a succès tant qu'on teste des éléments dans *résultat* dont la position est inférieur à la taille de *résultat*. Par conséquent, il faut une variable (que nous appellerons *position*) pour connaître la position de l'élément auquel on accède.
<test concernant le nom du fournisseur> : il y a succès si le nom recherché (premier élément du $i^{ème}$ élément de LL) est différent du nom du fournisseur courant (premier élément de chaque position^{ème} élément).
Donc on obtient :

((position <= nops(resultat)) and (LL[j,1] <> resultat[position][1]))

Remarque : les notations [j,1] et [j][1] sont équivalentes.

Une variable ne peut pas être utilisée avant d'avoir eu une valeur. Dans notre test, *j* est connue mais pas *position*. Il faut donc ajouter au programme une commande pour donner à *position* sa valeur initiale et une autre pour modifier sa valeur afin de tester un autre fournisseur. Nous obtenons donc le programme :

```
> resultat := [LL[1]];
  for i from 1 to nops(LL) do
  position := 1;
  while ( (position <= nops(resultat))
          and
          (LL[i][1] <> resultat[position,1])) do
               position := position + 1;
  end do;
# partie relative au traitement du ieme fournisseur
# sachant s'il est connu ou pas.
end do;
```

Donc on incrémente la valeur de position pour tester chaque fournisseur, jusqu'à ce qu'il soit trouvé ou que l'on ai testé tous les fournisseurs de *résultats*.
Sorti de la boucle on doit modifier *résultat* en fonction du succès de la recherche. Il y a donc deux questions à se poser : 1) comment sait on si la recherche a abouti ou pas ? 2) comment modifie-t-on *résultat* selon le succès ?
Pour la première question, il faut se demander dans quel état, quelles sont les valeurs de mes variables en fonction du succès de la boucle. La seule information qu'on l'on ait vient de la valeur de la variable *position* qui dépend du succès de la boucle. Soit la recherche a abouti (un fournisseur a été trouvé), soit elle a échoué. Dans le premier cas, la valeur de *position* n'est pas connue mais on peut déduire que sa valeur est inférieure à la taille de la liste *résultat*. Cette information est suffisante adapter son comportement.

```
> resultat := [];
  for i from 1 to nops(LL) do
  position := 1;
  while ( (position <= nops(resultat))
          and
          (LL[i][1] <> resultat[position,1])) do
               position := position + 1;
  end do;
  if (position <= nops(resultat)) then
     # le fournisseur existe il faut modifier l'information
     # enregistrée dans resultat
  else
     # le fournisseur est nouveau, il faut l'ajouter à
     #résultat.
  end if;
end do;
```

Enfin il faut modifier *résultat*. Si le fournisseur est connu on ajoute le montant à celui déjà enregistré

resultat[position][2] := resultat[position,2] + LL[i,2] ;

S'il est inconnu, il faut l'ajouter à la liste.

resultat := [op(resultat),LL[i]];

L'algorithme final est donc :

```
> resultat := [];
  for i from 1 to nops(LL) do
   position := 1;
   while ( (position <= nops(resultat))
           and
           (LL[i][1] <> resultat[position,1])) do
                position := position + 1;
   end do;
   if (position <= nops(resultat)) then
     resultat[position,2] := resultat[position,2] + LL[i,2] ;
   else
     resultat := [op(resultat),LL[i]];
   end if;
  end do;
```