TD 8 Algorithmes récursifs

I Programmation impérative/fonctionnelle

Jusqu'à maintenant, nous avons surtout utilisé des algorithmes *itératifs* ou *impératifs* employant des *boucles* et des *affectations*.
Ces algorithmes ont plusieurs défauts :
- ils s'éloignent des mathématiques en utilisant l'*affectation* comme base de programmation. Une même variable peut prendre des valeurs tout à fait différentes selon le moment où on l'évalue ;
- la mémoire est constamment modifiée par le programme : le programmeur doit à chaque instant connaître son état pour savoir ce que représentent les variables affectées ;
- on est obligé de créer des variables temporaires pour que certaines zones de la mémoire restent dans un certain état : cela est source de nombreux bogues dus à une mauvaise maîtrise de la chronologie des affectations.

Les langages fonctionnels bannissent totalement les affectations. Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée. Les effets négatifs listés précédemment disparaissent donc. Les fonctions peuvent être testées une par une car elle ne changent pas selon le moment où elles sont appelées dans le programme. On parle de *transparence référentielle*. Au contraire, une procédure avec *effets de bord* ne dépend pas uniquement des arguments entrés et donc peut changer de comportement au cours du programme.
Par exemple, définissons une fonction `plus` dépendant d'une entrée `k` :

```
n:=1:

plus:=proc(k)
global n;
n:=n+k;
end:
```

Alors

```
plus(3)
```

renvoie 4 et si on demande à nouveau

```
plus(3)
```

on obtient 7. Ainsi `plus` n'est pas une fonction au sens mathématique car 3 peut avoir des images distinctes !

La récursivité est le mode habituel de programmation avec de tels langages : il s'agit de fonctions qui s'appellent elles-mêmes. Pendant longtemps, la qualité des ordinateurs et des langages fonctionnels n'était pas suffisante et limitaient les domaines d'application de ces langages.
MAPLE n'est pas un langage fonctionnel mais permet de créer des fonctions récursives :

```
suivant:=proc(n)
if n=0 then 1
else 1+suivant(n-1)
fi
end:
```

Ainsi,

```
suivant(700)
```

renvoie bien 701 mais pour 7000 on obtient *Error, (in suivant) too many levels of recursion.*
Et oui, MAPLE ne traite pas efficacement les récursions, comme beaucoup d'autres langages, c'est pourquoi pendant longtemps seuls les langages impératifs ont prévalus.
Utilisons par exemple le langage fonctionnel CAML étudié par vos camarades de MP ayant choisi l'option informatique.

```
# let rec suivant(n)=
  if n=0 then 1
  else 1+suivant(n-1);;
```

Alors par exemple :

```
# suivant(36000);;
- : int = 36001
```

Mais

```
# suivant(3600000);;
Stack overflow during evaluation (looping recursion?).
```

Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile* où sont accumulés les appels récursifs afin de ne pas l'encombrer. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.
Ici, ce n'est pas le cas car l'appel récursif `suivant(n-1)` est enrobé dans la fonction $x \mapsto 1 + x$.
On peut y remédier en introduisant une fonction intermédiaire qui sera récursive totale :

```
# let rec suivant_rec(n,resultat)=
        if n=0 then resultat
        else suivant_rec(n-1,resultat+1)
```

puis on appelle cette fonction en prenant comme résultat de départ 1, le suivant de 0 :

```
# let suivant(n)=
  suivant_rec(n,1);;
```

Alors :

```
# suivant(360000000);;
- : int = 360000001
```

MAPLE n'étant pas un langage fonctionnel, la différence est moins visible...

```
suivant_rec:=proc(n,resultat)
  if n=0 then resultat
         else suivant_rec(n-1,resultat+1)
  fi
end:
```

```
suivant_total:=proc(n)
  suivant_rec(n,1)
end:
```

II Écrire des algorithmes récursifs avec MAPLE

Même si nous n'avons pas affaire à un langage traitant la récursion efficacement, nous pouvons malgré tout réécrire la plupart des algorithmes vus cette année sous forme récursive.

a. Factorielle

Donnez une version récursive du calcul de la factorielle d'un entier puis imaginez un algorithme récursif total.

b. Partie entière

Je ne vous apprend rien en vous rappelant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor = \lfloor x + 1 \rfloor - 1$.
Imaginez alors un programme récursif déterminant la partie entière d'un nombre réel positif puis d'un nombre réel quelconque.

c. Méthode d'Euler pour la résolution des EDO

Vous vous souvenez de la méthode d'Euler pour résoudre une EDO du type $y' = f(y, x)$.
Que pensez-vous du programme :

```
euler_rec:=proc(f,x,xmax,y,h,liste_points)
if x>=xmax then liste_points
else euler_rec(f,x+h,xmax,y+f(y,x)*h,h,[op(liste_points),[x,y]])
fi
end:
```

Comment l'utiliser pour représenter graphiquement une approximation de la solution de $y' = f(y, x)$ avec $y(x_0) = y_0$?

d. Spirale d'Archimède

Imaginez une procédure récursive pour tracer une spirale d'Archimède comme dans le TD n°2.
On pourra l'appeler `spirale_rec :=proc(n,z,liste_points)`.
On l'utilisera ensuite avec :

```
spirale:=proc(n)
  plot([spirale_rec(n,-I,[])])
end:
```

e. Dichotomie, méthode de Newton

Ces méthodes sont par essence récursives. Rien de plus naturel que d'écrire l'algorithme récursif correspondant.
On ne s'encombrera pas de compteur mais on affichera directement le résultat.
Normalement, en entrant :

```
dicho_rec(x->x^2-2,1.0,2.0,0.00000001);
```

on obtient `1.414213559`
et en entrant

```
newton_rec(x->x^2-2,1.0,10.0^(-50));
```

on obtient `1.41421356237309504880168872420969807856967187537 69`
Il faudra faire attention à `evalf`

f. Développements limités

Il s'agit de trouver une bonne relation de récurrence. Petit rappel MAPLE : la *fonction* dérivée d'ordre n de f s'obtient en entrant `(D@@n)(f)`.
Alors, en entrant :

```
DL(x->tan(x),15);
```

vous devez obtenir :

```
929569           21844           1382         62        17         2       1
--------- x^15 + ------- x^13 + ------ x^11+ ---- x^9 + --- x^7 + -- x^5 + - x^3 + x
638512875        6081075         155925       2835      315        15      3
```

g. Intégration : méthode des rectangles

Vous vous souvenez : pour obtenir l'approximation de l'intégrale d'une fonction f entre a et b on peut faire la somme des aires de rectangles particuliers. Imaginez un algorithme récursif permettant de calculer l'approximation d'une intégrale par cette méthode.
Par exemple :

```
  4*rectangles(x->1/(1+x^2),0,1,0.0001);
```

donne `3.141892663`