TP MAPLE N°5 : une technique de programmation : la récursivité

1 Présentation et premier exemple

La récursivité est une méthode de programmation qui pourrait avoir la morale suivante : on résout quelque chose de compliqué en fonction d'une version un peu moins compliquée de la même chose jusqu'à arriver à quelque chose de très facile, puis on revient en arrière pour résoudre le problème initial.

Un algorithme récursif fait appel à lui-même dans le corps de sa propre définition, il est constitué de deux éléments remarquables :
- **le cas de base** (correspond à la version facile du problème)
- **la formule de propagation ou d'hérédité** (permet de passer à une version plus simple)

Par exemple l'algorithme d'Euclide pour calculer le pgcd de deux entiers peut s'écrire de manière récursive :
- le cas de base est lorsque $b = 0$, dans ce cas $\text{pgcd}(a, 0) = a$.
- la formule de propagation est $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par b. En effet, le problème «trouver $\text{pgcd}(b, r)$» est bien «une version plus simple» du problème «trouver $\text{pgcd}(a, b)$» puisque $0 \leqslant r < b$.

Nous appelons `eucliderec` notre algorithme d'Euclide écrit en récursif.

algorithme : `eucliderec`
Entrées : $a, b \in \mathbb{N}$
Résultat : $\text{pgcd}(a, b)$
  Si $b = 0$ alors
    Renvoyer a
  Sinon
    Renvoyer `eucliderec(b,r)`
où r est le reste de la divison de a par b.

```
> eucliderec:=proc(a::nonnegint,b::nonnegint)
>    if b=0 then
>       RETURN(a);
>    else
>       RETURN(eucliderec(b,irem(a,b)));
>    fi;
> end:
```

Examinons la trace de `eucliderec(18459,3809)` que nous livre Maple :

```
> trace(eucliderec);
> eucliderec(18459,3809);
enter eucliderec, args = 18459, 3809
enter eucliderec, args = 3809, 3223
enter eucliderec, args = 3223, 586
enter eucliderec, args = 586, 293
enter eucliderec, args = 293, 0
exit eucliderec (now in eucliderec) = 293\}
exit eucliderec (now in eucliderec) = 293\}
exit eucliderec (now in eucliderec) = 293\}
exit eucliderec (now in eucliderec) = 293\}
exit eucliderec (now at top level) = 293\}
```

293

La fonction `eucliderec` est appelée une première fois avec les arguments 18459 et 3809.

Ensuite, par la formule de propagation elle appelle `eucliderec` avec les arguments 3809 et 3223. La fonction `eucliderec` est ainsi successivement appelée avec les arguments 3223 et 586, 3223 et 586, 586 et 293 et enfin 293 et 0. C'est le cas de base et donc le dernier appel.

On remarque que le nombre de calculs effectués par l'algorithme est le même que celui que l'on effectuerait à la main. Nous verrons dans la section suites récurrentes que ce ne sera pas toujours le cas.

L'algorithme d'Euclide possède aussi une version itérative (qu'on opposera à récursive) à l'aide d'une boucle while (tant que).

algorithme : `euclide`
Entrées : $a, b \in \mathbb{N}$
Résultat : $\text{pgcd}(a, b)$
Variables : u, v, r
  $u \leftarrow a$
  $v \leftarrow b$
  Tant que $v \neq 0$, Faire
    $r \leftarrow$ le reste de la divison de u par v
    $u \leftarrow v$
    $v \leftarrow r$
  Fin du Tant que
  Renvoyer u

```
> euclide:=proc(a::nonnegint,b::nonnegint)
>    local u,v,r;
>    u:=a;
>    v:=b;
>    while v<>0 do
>       r:=irem(u,v);
>       u:=v;
>       v:=r;
>    od;
>    RETURN(u);
> end:
```

On aperçoit ici un des avantages de la programmation récursive : le code est plus simple à écrire mais aussi à lire. Nous n'avons pas eu à recourir à l'affectation et donc aux variables, contrairement à la version itérative. De plus dans notre exemple, `euclideрec` effectue le même nombre de divisions que `euclide`. Les deux algorithmes ont la même complexité. On va voir que ce n'est pas toujours le cas.

2 Le cas des suites récurrentes

2.1 Exemple de la somme des entiers de 1 à n

On pose pour $n \geqslant 1$, $S_n = 1 + 2 + \cdots + n$. La suite (S_n) est définie par la relation de récurrence $S_n = S_{n-1} + n$ et la condition initiale $S_1 = 1$. Voici un algorithme récursif nommé `S` de paramètre n et renvoyant la valeur S_n.

```
> S:=proc(n::posint)
>    if n=1 then
>       RETURN(1);
>    else
>       RETURN(S(n-1)+n);
>    fi;
> end:
```

Le cas de base est la condition initiale et la formule de propagation est la relation de récurrence.

Examinons la trace de `S(4)`. La procédure `S` est d'abord appelée 4 fois :

`S(4)=S(3)+4, S(3)=S(2)+3, S(2)=S(1)+2, S(1)=1`.

puis on remonte en faisant quatre additions,

`S(2)=1+2=3,S(3)=3+3=6,S(4)=6+4=10`.

Il y a donc eu sept opérations (six si l'on néglige S(1)=1).

Voici maintenant la version itérative nommée `Siter`.

```
> Siter:=proc(n::posint)
> local somme,k;
> somme:=1;
> for k from 2 to n do
>    somme:=somme+k;
> od;
> RETURN(somme);
> end:
```

Examinons la trace de `Siter(4)`. Sont calculés successivement :

`somme :=1` ; puis `somme :=1+2=3` ; puis `somme :=3+3` ; enfin `somme :=6+4=10` ;

Il y a donc eu quatre opérations (trois si l'on néglige la première affectation `somme :=1`). Cela correspond à ceux que l'on fait à la main $1 + 2 = 3$ puis $3 + 3 = 6$ et enfin $6 + 4 = 10$.

La version récursive est donc ici moins performante que la version itérative (double d'opérations), mais elle reste facile à écrire et à lire.

2.2 La suite de Fibonacci

La suite de Fibonacci (F_n) est la suite définie par $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$ et $F_0 = 0$ et $F_1 = 1$.

2.2.1 Version récursive

La suite de Fibonacci (F_n) est un cas particulier de suite récurrente. On peut écrire un programme récursif nommé `fiborec` qui renvoie la valeur du n-ième terme F_n. En effet :
- les conditions initiales $F_0 = 0$ et $F_1 = 1$ constituent le cas de base
- la formule de récurrence $F_n = F_{n-1} + F_{n-2}$ constitue la formule de propagation.

algorithme : `fiborec`
Entrées : $n \in \mathbb{N}$
Résultat : F_n
  Si $n = 0$ ou $n = 1$ alors
    Renvoyer n
  Sinon
    Renvoyer `fiborec(n-1)+fiborec(n-2)`

```
> fiborec:=proc(n::nonnegint)
>    if n=0 or n=1 then
>       RETURN(n);
>    else
>       RETURN(fiborec(n-1)+fiborec(n-2));
>    fi;
> end:
```

Testons cet algorithme avec $n = 5, 10, 15, 20, 25$. Essayons maintenant avec 30 puis 40. Il faut l'avouer, ça a l'air de prendre beaucoup de temps. Ce n'est vraiment pas efficace. Essayons de comprendre ce qui se passe et donc ce que calcule l'ordinateur.

On voit que F_2 a été calculé deux fois, on a fait 4 additions et on a appelé 9 fois la fonction `fiborec`.

Nos remarques sont confirmées par la propre fonction trace de Maple :

```
> trace(fiborec);
> fiborec(4);
```

```
 enter fiborec, args = 4

 enter fiborec, args = 3

 enter fiborec, args = 2

enter fiborec, args = 1

 exit fiborec (now in fibo) = 1\}

 enter fiborec, args = 0

 exit fiborec (now in fiborec) = 0\}

 exit fiborec (now in fiborec) = 1\}

 enter fiborec, args = 1

 exit fiborec (now in fiborec) = 1\}

 exit fiborec (now in fiborec) = 2\}

 enter fiborec, args = 2

 enter fiborec, args = 1

 exit fiborec (now in fiborec) = 1\}

 enter fiborec, args = 0

 exit fiborec (now in fiborec) = 0\}

 exit fiborec (now in fiborec) = 1\}

 exit fiborec (now at top level) = 3\}
```


Si on trace un arbre pour `fiborec(5)`, on voit que l'on fait 7 additions et que la fonction `fiborec` a été appelée 15 fois. Beaucoup de calculs sont inutilement répétés. L'algorithme évalue deux fois `fiborec(3)` et trois fois `fiborec(2)`. Ce processus se termine lors de l'évaluation de `fiborec(0)` ou de `fiborec(1)` qui constituent le cas de base.

Alors faut-il jeter à la poubelle `fiborec` ? Pas forcément. On peut considérablement l'améliorer en lui demandant de garder en mémoire les calculs intermédiaires. Ceci est possible avec Maple avec l'instruction `option remember ;` que l'on insère à la deuxième ligne du programme. Cette fois ci, ça marche mieux et semble aussi rapide qu'avec `fibo`. On ne peut toutefois oublier le défaut majeur de notre programme, il nécessite beaucoup de mémoire, sa «complexité spatiale» est mauvaise.

2.2.2 Version itérative

Donnons une version itérative du calcul de F_n

algorithme : `fibo`
Entrées : $n \in \mathbb{N}$
Résultat : F_n
Variables : u, v, s, k
  Si $n = 0$ ou $n = 1$ alors
    Renvoyer n
  Fin du si
  $u \leftarrow 0, v \leftarrow 1$
  Pour k de 1 à $n-1$, faire
    $s \leftarrow u + v$
    $u \leftarrow v$
    $v \leftarrow s$
  Fin du pour
  Renvoyer s

```
> fibo:=proc(n::nonnegint)
> local u,v,s,k;
> if n=0 or n=1 then RETURN(n);
> fi;
> u:=0;v:=1;
> for k from 0 to n-2 do
> s:=u+v;
> u:=v;
> v:=s;
> od;
> RETURN(s);
> end:
```

Remarquons que les variables a et b changent de valeur à chaque itération, on y stocke les deux valeurs précédentes de F_n, leur somme est stockée dans la troisième variable s.

L'algorithme `fiborec` est inconstestablement plus facile à écrire que l'algorithme itératif `fibo`.

2.2.3 Comparaison de la complexité des algorithmes récursif et itératif `fibo` et `fiboiter`

1. Pour calculer F_n avec l'algorithme `fibo`, on effectue $n-1$ additions. On ne parlera pas des affectations que nous négligerons (2 initiales et 3 affectations pour chaque itération donc au total $2 + 3(n-1)$). On négligera aussi les comparaisons que l'on a à faire à chaque itération pour savoir si la variable k est comprise entre 2 et n.
   On dit que la complexité «temporelle» de `fibo` est linéaire car en $O(n)$.
2. Pour l'algorithme récursif `fiborec` c'est plus compliqué.
   Notons a_n le nombre d'appels à la fonction `fiborec` et s_n le nombre d'additions nécessaires pour calculer F_n. On a alors
   $$a_0 = a_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ a_n = 1 + a_{n-1} + a_{n-2}.$$
   En effet, l'appel initial de `fiborec(n)` entraîne l'appel de `fiborec(n-1)` et de `fiborec(n-2)`. De même
   $$s_0 = s_1 = 0 \quad \text{et} \quad \forall n \geqslant 2,\ s_n = 1 + s_{n-1} + s_{n-2}.$$
   En effet comme `fiborec(n)=fiborec(n-1)+fiborec(n-2)`, une addition est nécessaire plus celles pour calculer `fiborec(n-1)` et `fiborec(n-2)`.
   On voit donc en particulier, que pour $n \geqslant 2,\ s_n > s_{n-1}+s_{n-2}$ et comme $s_2 = F_2 = 1$ et $s_3 = F_3 = 3$, on a pour $n \geqslant 2, s_n > F_n$ et donc s_n devient énorme lorsque n grandit. En effet, on a déjà vu dans la première section que F_n était équivalent au voisinage de $+\infty$ à $\dfrac{\phi^n}{\sqrt{5}}$. Aucun ordinateur, aussi rapide soit-il, ne peut par cet algorithme récursif calculer F_{50} qui vaut 12586269025.
   On dit que la complexité «temporelle» de `fiborec` est exponentielle car en $O(\phi^n)$ avec $\phi > 1$.
   Voici un tableau indiquant le temps de calcul nécessaire à un ordinateur pour exécuter des algorithmes de complexités différentes. On suppose que notre ordinateur exécute une opération élémentaire en 100 nanosecondes (10^{-7} s). On lit en abscisse la complexité de l'algorithme et en ordonnée la taille de l'entrée.

	$\log_2 n$	n	$n.\log_2 n$	n^2	n^3	2^n
10^2	0.66 μ s	10 μ s	66 μ s	1 ms	0.1s	4×10^{15} ans
10^3	1 μ s	100 μ s	1 ms	0.1s	100 s	
10^4	1.3 μ s	1 ms	13 ms	10 s	1 jour	
10^5	1.6 μ s	10 ms	0.1 ms	16 min	3 ans	
10^6	2 μ s	100 ms	2 s	1 jour	3100 ans	
10^7	2.3 μ s	1 s	23 s	115 jours	3 ans	

Pour finir, remarquons que la fonction `rsolve` de Maple nous donne une formule explicite pour les suites (a_n) et (s_n).

```
> rsolve({a(n)=1+a(n-1)+a(n-2),a(0)=1,a(1)=1},a);
```

$$-\frac{4}{5}\,\frac{\sqrt{5}\,(-\frac{2}{-\sqrt{5}+1})^n}{-\sqrt{5}+1}+\frac{4}{5}\,\frac{\sqrt{5}\,(-\frac{2}{\sqrt{5}+1})^n}{\sqrt{5}+1}-1$$

```
> rsolve({s(n)=1+s(n-1)+s(n-2),s(0)=0,s(1)=0},s);
```

$$-1+\frac{2}{5}\,\frac{\sqrt{5}\,(\frac{2}{\sqrt{5}-1})^n}{\sqrt{5}-1}+\frac{2}{5}\,\frac{\sqrt{5}\,(-\frac{2}{\sqrt{5}+1})^n}{\sqrt{5}+1}$$

3 Exercices

Exercice 1 (Factorielle) Écrire deux algorithmes l'un récursif et l'autre itératif, permettant de calculer la factorielle d'un entier naturel n.

Vérifier qu'il fonctionne avec 0 et qu'il n'y a pas de problème majeur si n est négatif.

Exercice 2 (Exponentiation rapide) C'est une méthode très efficace pour calculer les puissances n-ièmes d'un réel a (ou plus généralement d'un élément dans un groupe multiplicatif, par exemple une matrice carrée). Elle repose sur le principe suivant :

$$\begin{cases} a^0 &= 1 & \\ a^n &= \left(a^2\right)^{\frac{n}{2}} & \text{, si } n \text{ est pair et } n>0 \\ a^n &= a\times\left(a^2\right)^{\frac{n-1}{2}} & \text{, si } n \text{ est impair} \end{cases}$$

On décompose en fait l'entier n en base 2.

1. Soit a un réel. Appliquer l'algorithme d'exponentiation rapide pour calculer a^{16} puis a^{14}.Indiquer dans les deux exemples le nombre de multiplications effectuées puis comparer avec la méthode «naïve».
2. Écrire une version récursive de l'algorithme d'exponentiation rapide, on le notera `exporap`.
3. Écrire une version itérative de l'algorithme d'exponentiation rapide, on pourra utiliser des variables nommées `produit, exposant et alpha`.
4. Écrire un algorithme `exponaif` qui calcule naïvement le produit a^n. Comparer avec l'algorithme d'exponentiation rapide.
5. *Pour aller plus loin* : comment utiliser l'exponentiation rapide avec des matrices pour calculer le nombre de Fibonacci F_n ? On pose $X_n=\begin{pmatrix}F_n\\F_{n+1}\end{pmatrix}$. Déterminer une matrice $A\in M_2(\mathbb{R})$ telle que $X_{n+1}=AX_n$. Conclure.

Exercice 3 (Maximum d'une liste)

1. Construire une fonction $\max_2$ qui renvoie le maximum de deux réels.
2. Écrire en français un algorithme récursif `maximum` donnant le maximum d'une liste de nombres réels quelconques. On pourra utiliser le fait que $\max(a,b,c)=\max(a,\max(b,c))$.
3. Taduire votre algorithme en langage «Maple». On pourra utiliser `subsop` qui supprime un élément d'une liste et démarrer par `maximum :=proc(l : :list)`.

Exercice 4 (Somme des chiffres au carré) On veut écrire un algorithme récursif `f` de paramètre n un entier strictement positif renvoyant la somme de ses chiffres au carré.

Par exemple `f(45)` vaut 2^2+3^2.

1. Soit n un entier strictement positif et $\overline{a_1a_2\ldots a_k}$ son écriture décimale. Exprimer $f(n)$ en fonction de $f(\overline{a_1a_2\ldots a_{k-1}})$ et de a_k.
2. Écrire l'algorithme.
3. Tester l'algorithme sur pas mal d'entiers et observer les «orbites» de ces nombres sous l'action de la fonction `f`.

Exercice 5 (Calcul de pgcd par l'algorithme des différences)

1. Soit a et b des entiers. Démontrer que $\text{pgcd}(a,b) = \text{pgcd}(a-b,b)$.
2. Écrire en langage français, un algorithme récursif permettant de calculer le pgcd en utilisant la propriété précédente.

Exercice 6 (Coefficients de Bezout) Soit a et b deux entiers et d leur pgcd. On a démontré en cours qu'en appliquant l'algorithme de Bezout, on obtient deux entiers u et v tels que $d = au + bv$. Si r est le reste de la division euclidienne de a par b, on sait que $d = pgcd(a,b) = pgcd(b,r)$. il existe donc des entiers u' et v' tels que $d = bu' + rv'$.

1. Exprimer u et v en fonction de u' et v'. C'est une formule de propagation.
2. En déduire un algorithme récursif renvoyant les coefficients de Bezout de deux entiers a et b. On pourra vérifier avec la fonction `igcdex` de Maple.

Exercice 7 (Solution d'une équation par dichotomie) Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a,b]$ où la fonction semble changer de signe. On note ensuite m le milieu du segment $[a,b]$. On évalue le signe de $f(m)$. Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence. Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue. Donner une procédure récursive prenant comme arguments une fonction f, les bornes de l'intervalle d'étude a et b et une précision de calcul eps.

Exercice 8 (Tri) Écrire une procédure prenant en argument une liste de réels et renvoyant cette liste rangée dans l'ordre croissant.