# Récursif et itératif

Dans le *Don Quichotte* de Cervantès, on trouve cette anecdote : Sancho Pança est sur un pont et il a pour ordre de pendre toute personne qui passe sur le pont si et seulement si cette personne ment. Un homme passe et dit : « On va me pendre ». Que va faire Sancho Pança ? Si la personne dit la vérité, on va la pendre, mais Sancho Pança ne doit pas la pendre, et si elle ment, on ne va pas la pendre, mais Sancho Pança doit la pendre. C'est un dilemme où l'on tourne en rond indéfiniment. Cela s'appelle le paradoxe du menteur, dont la première référence connue remonte au philosophe Eubulide de l'Ecole de Mégare, il y a 2400 ans. Si je dis « je mens » et que je dis vrai, alors je mens, et si je dis « je mens » et que je mens, je dis vrai…

Là est le principe fondateur de la récursivité : une fonction récursive est une fonction qui se rappelle elle-même. Pour éviter le paradoxe du menteur, où la phrase s'auto-appellerait indéfiniment, on fait en sorte que les rappels imbriqués de la fonction en elle-même se fassent sur des objets à géométrie variable, le plus souvent décroissants, avec un test d'arrêt pour bloquer le processus. Cela peut sembler étrange. En fait les exemples les plus simples sont relatifs aux suites définies par récurrence.

## 1. Récursivité et suites

### 1.1. Exemple 1 : la factorielle

Par définition, factorielle de $n$, avec $n$ entier naturel, s'écrit $n!$ et vaut :

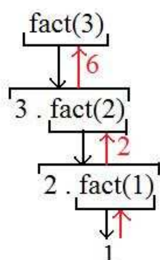
$n! = n\,(n-1)\,(n-2)\,\ldots\,3\times2\times1$ pour $n > 0$, et l'on prend $0! = 1$.

Considérons la suite de nombres $u_n$ tels que $u_n = n!$. Cette suite peut aussi être définie par la relation de récurrence $n! = n\,(n-1)!$ (ou encore $u_n = n\,u_{n-1}$) avec comme condition initiale $1! = 1$. Une telle définition donne immédiatement ce programme récursif :

```
main()
{ on se donne N ;
  factorielleN = fact(N) ;  afficher factorielleN ;
}

int fact(int n)
{ if (n==0 || n==1) return 1 ;  /* test d'arrêt : ce sont les conditions initiales
                                   de la définition de n !  */
  return n*fact(n – 1) ;         /* la fonction se rappelle sur n-1 : c'est la relation
                                   de récurrence dans la définition de n !  */
}
```

C'est simple, mais pourquoi ça marche ? Prenons l'exemple de $N = 3$.




On appelle *fact*(3). Que ce soit nous ou la machine, pourvu qu'on sache lire, le programme nous dit que c'est 3×*fact*(2). Mais on ne sait pas pour le moment ce qu'est *fact*(2). On met en attente le calcul de 3× *fact*(2). Le programme nous dit alors que *fact*(2) peut être remplacé par 2×*fact*(1). Là encore on met ce résultat en attente. Mais le programme dit que *fact*(1) vaut 1. Maintenant on remonte, et on calcule pas à pas ce qui était en attente, pour obtenir finalement 6 dans *factorielleN*.

Maintenant osons : utilisons une « fonction » qui n'a pas de variable :

```
int fact=1, i = 1 ; N= 6 ;  /* variables globales */

main()  { factorielle() ;  afficher fact ;}
void factorielle(void)
  { if (i<=N) {fact*=i ; i++ ;}
    factorielle() ;
  }
```

Dans les temps anciens, j'aurais considéré cette version récursive comme dégénérée et sacrilège, aujourd'hui je la trouve amusante, car on est quasiment arrivé à une méthode itérative. En effet modifions-la à peine :

```
main()
  { int fact=1, i = 1 ; N= 6 ;
    FFF :       /* FFF est un label, une sorte d'étiquette */
    if (i<=N) {fact*= i ; i++ ;}
    goto FFF ;                    /* nous voilà revenu au bon vieux style du Basic avec ses goto   */
    afficher fact
  }
```

Pendant qu'on y est donnons la version itérative classique pour calculer *N* !, où l'on utilise une variable de cumul ici appelée *fact* :

```
fact=1 ;
for(i=1 ; i<=N ;i++) fact= i*fact ;
afficher fact ;
```

Cette méthode itérative part de 1 puis monte à 2, 3…, jusqu'à *N*, tandis que la méthode récursive classique part de *N* et descend jusqu'à 1 en mettant les calculs en attente, ceux-ci étant effectués à la remontée. Les deux méthodes, itérative et récursive, se valent en termes de performance. Choisissez celle que vous préférez !

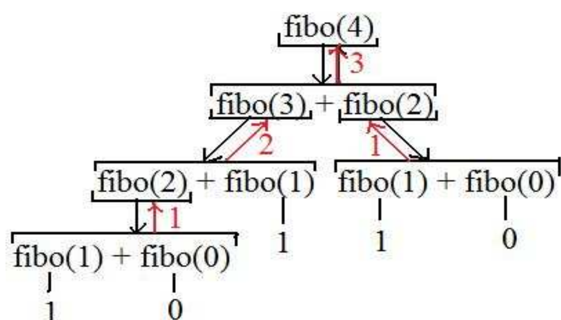
### 1.2. Exemple 2 : la suite de Fibonacci

On a déjà vu comment la programmer en itératif. Rappelons que cette suite obéit à la relation de récurrence $u_n = u_{n-1} + u_{n-2}$ ($n \geq 2$) avec au départ $u_0 = 0$ et $u_1 = 1$. Cela donne aussitôt le programme récursif pour avoir $u_N$, *N* étant donné:

```
main()
  { on se donne N ; uN = fibo(N) ; afficher uN ;
  }

int fibo(int n)
  { if (n==0 ) return 0 ;
    if (n==1) return 1 ;
    return fibo(n-1) + fibo(n-2) ;
   }
```

Faisons  marcher ce programme pour $N = 4$ :




On constate que les mêmes calculs sont répétés plusieurs fois, comme *fibo*(2) dans le cas présent pour $N = 4$). Un tel programme n'est pas performant. Il faudrait enregistrer les résultats intermédiaires pour éviter de les calculer plusieurs fois, ce que l'on fait en Lisp, mais en C autant passer en itératif.

Si la récursivité se réduisait au traitement des suites récurrentes, elle ne présenterait pas un intérêt majeur. En fait elle intervient dans de nombreux problèmes où elle s'impose comme la méthode la plus adaptée, pour ne pas dire la seule. Un exemple historique est celui des tours de Hanoi. Au début, il s'agissait d'un jeu inventé par Edouard Lucas, vers 1880. Près d'un siècle plus tard, à l'aube de l'informatique, c'est son traitement sur ordinateur qui a fait sensation, car avec un programme très simple on pouvait voir les mouvements multiples et compliqués des disques des tours se dérouler inexorablement sur un écran d'ordinateur.

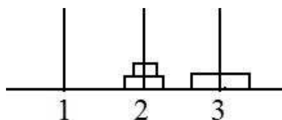
## 2. Tours de Hanoi

On a trois piquets numérotés 1, 2, 3. Au dépat, sur le piquet 1 sont enfilés $N$ disques de rayon décroissant vers le haut. On les numérotera 1, 2, 3,…, $N$ du plus petit au plus grand. Puis on va déplacer ces disques un par un en utilisant les trois piquets, de façon qu'on n'ait jamais un disque plus grand au-dessus d'un autre plus petit. L'objectif est d'arriver au placement de tous les disques sur le piquet 3, en un minimum de déplacements.

Prenons l'exemple de $N = 3$. Cela débute ainsi :

Il n'y a pas d'autres façons d'agir que ce que nous venons de faire. Au terme de ces trois premiers déplacements, on a fait passer deux disques du piquet 1 au piquet 2, en passant par l'intermédiaire du piquet 3.

Ensuite on déplace le disque le plus grand du piquet 1 au piquet 3.




Il reste enfin, en quelques déplacements, d'envoyer les deux disques du piquet 2 au piquet 3 en passant par l'intermédiaire du piquet 1.

Plus généralement pour déplacer $N$ disques, on commence par en déplacer $N$-1 du piquet 1 au piquet 2, puis on en déplace un (le plus grand) du piquet 1 au piquet 3, et enfin on déplace $N$-1 disques du piquet 2 au piquet 3.

Si l'on appelle $u(N)$ le nombre de déplacements, on a la relation de récurrence :

$u(N) = 2\,u(N-1) + 1$, avec au départ $u(1) = 1$. On en déduit la formule explicite $u(N) = 2^N - 1$. [1]

On en déduit aussi le programme récursif des tours de Hanoi, donnant les déplacements à effectuer à chaque fois. Pour cela on définit une fonction *hanoi*(*n*,*d*,*i*,*a*) ayant comme variables le nombre *n* de disques à déplacer, le piquet *d* d'où l'on part, le piquet *a* d'arrivée, et le piquet *i* intermédiaire. Le programme principal se contente d'appeler *hanoi*(*N*,1,2,3). En s'inspirant des trois étapes d'évolution du processus, la fonction s'écrit :

```
void hanoi (int n, int d, int i, int a)
{  if (n==1)  printf("\n aller de %d à  %d",d,a);
   else {  h(n-1, d, a,  i);
           h(1, d, i, a);
           h(n-1, i, d, a);
        }
}
```

L'exécution du programme pour $N = 3$ va donner : *aller de 1 à 3, aller de 1 à 2, aller de 3 à 2*, etc.

Il s'agit maintenant d'améliorer le programme en utilisant les numéros des disques déplacés. On veut avoir : *le disque 1 va de 1 à 3, le disque 2 va de 1 à 2, le disque 1 va de 3 à 2*, etc. Rappelons que les disques sont numérotés selon leur rayon croissant, de 1 à *N*. On rajoute comme variable la hauteur *H*[] des disques sur chacun des piquets. Pour le piquet *i*, on met dans la variable *p*[*i*][*altitude*] le numéro du disque dont le socle se trouve à une certaine *altitude* (l'*altitude* va de 0 à *N*-1). Le numéro du disque le plus haut sur un piquet *i* est *p*[*i*][*H*[*i*]-1], et c'est un tel disque qui est déplacé du piquet de départ au piquet d'arrivée. D'où le programme amélioré :

```
int p[4][N],H[4],numero ;  /* dans p[][] et H[], seules les cases 1,2,3 nous concernent */
main()
{  for(j=0;j<N;j++)  p[1][j]=N-j;                H[1]=N; H[2]=0 ; H[3]=0 ;
   for(i=2;i<=3;i++) for(j=0;j<N;j++) p[i][j]=0;  /* là il n'y a pas de disque, on met 0 */
   h(N,1,2,3);
}

void h(int n, int d, int i, int a)
{  if (n==1) { numero=p[d][H[d]-1];    H[d]- - ; p[a][H[a]]=numero;  H[a]++;
              printf("\n %3.d: le disque %d va de %d … %d",compteur,numero,d,a);
                }
   else {  h(n-1,d,a,i); h(1,d,i,a); h(n-1,i,d,a);}
}
```

---

[1] Si vous ne savez pas traiter une suite arithmético-géométrique, vous pouvez toujours faire :

$u(n) = 2\,u(n-1)+1$
$2 \quad u(n-1) = 2\,u(n-2)+1$
$2^2 \quad u(n-2) = 2\,u(n-3)+1$
………………
$2^{n-2} \quad u(2) = 2\,u(1)+1$

En multipliant chacune de ces égalités par des puissances successives de 2, puis en les additionnant membre à membre il se produit des simplifications en cascade et il reste

$u(n)= 2^{n-1} + 1+2 + 2^2 + \ldots + 2^{n-2} = 2^{n-1} + 2^{n-1} -1 = 2^n -1$

Dans une version originelle des tours de Hanoi, il s'agissait de moines dans un temple qui devaient déplacer 64 tours en or, à raison d'un mouvement par jour. Quand ils auraient fini, ce serait la fin du monde. Cela demanderait donc $2^{64} - 1$ jours, soit un nombre comprenant une vingtaine de chiffres (pour le constater, utiliser le fait que $2^{10}$ vaut à peu près 1000), en gros plus d'un milliard de milliards d'années !

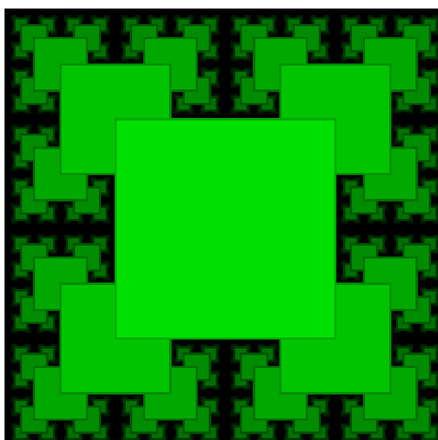
Il est enfin aisé de passer à la visualisation graphique. Chaque disque est dessiné sous forme d'un rectangle dont on s'est donné le rayon *R*[]. Au lieu du *printf*, il suffit d'effacer le rectangle de haut de pile sur le piquet de départ *d*, en le mettant couleur fond d'écran, puis de dessiner un nouveau rectangle en haut du piquet d'arrivée *a*. [2]

# 3. Récursivité et géométrie

De nombreuses formes, naturelles ou artificielles, ont tendance à se développer en se reproduisant à plus petite échelle de façon répétée. Nous allons donner deux exemples de tels grossissements par « bourgeonnements ».

## 3.1. Carrés grossissants

On part d'un carré. Puis en chaque coin de ce carré on dessine de nouveaux carrés de dimension moitié, et ayant ce coin comme centre, puis on recommence avec ces carrés, pour obtenir ce genre de forme :



Chaque carré est défini par son centre (*x*,*y*) et par la moitié de son côté, soit *c*. Dans le programme principal on se donne le centre (*xo*,*yo*) du carré central sur l'écran et la longueur *l* de son demi-côté. Puis on appelle la fonction carre (*xo*,*yo*,*l*). Cette fonction *carre*() a pour mission de dessiner le carré correspondant, puis elle se rappelle quatre fois pour avoir les nouveaux carrés aux quatre coins.

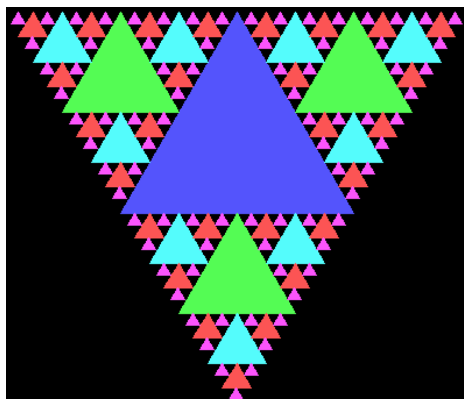
```
void carre(int x,int y, int c)
{  if (cote>0)
    { dessiner le carré de centre (x,y) et de demi-côté c, éventuellement colorier l'intérieur
      carre(x-c, y-c, c/2);
      carre(x-c, y+c, c/2);
      carre(x+c, y-c, c/2);
      carre(x+c, y+c, c/2);
     }
```

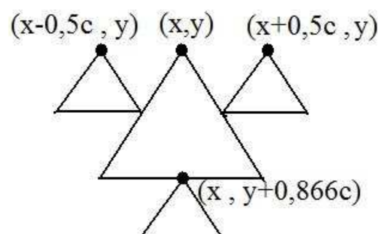
## 3.2. Bourgeonnements de triangles

On part d'un triangle équilatéral dessiné au centre de l'écran. Puis à partir des trois milieux de ses côtés on fait bourgeonner trois triangles équilatéraux de longueur moitié et de côtés toujours parallèles entre eux. Et l'on recommence … Pour pouvoir tracer un triangle équilatéral dans le cas présent, il

[2] On trouvera ce programme dans la rubrique : *Travaux complémentaires, algorithmes, 6- Tours de Hanoi.*

suffit de connaître les coordonnées (*x*,*y*) de sa pointe supérieure et la longueur *c* de ses côtés. La fonction *triangle*() se rappelle trois fois.



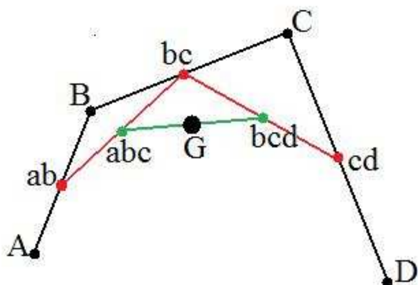



```
void triangle (int x,int y, int c)
{   if (c > 10)  /* on s'arrête quand le côté a 10 pixels de long ou moins */
      {  dessiner le triangle de pointe supérieure (x,y) et de côté de longueur c
         triangle(x-0.5*c , y , c/2);
         triangle ((x , y+0.866*c , c/2);
         triangle(x+0.5*c , y , c/2);
      }
}
```

### 3.3. Courbe de Bézier

On se donne des points de $A_0$ à $A_N$. La courbe de Bézier associée à ces points est une courbe qui passe par les deux points extrêmes et s'approche des autres points comme si ceux-ci jouaient le rôle d'aimants. Ce genre de courbe a été inventé au cours des années 1960 par Pierre Bézier dans les usines Renault et Paul de Casteljau chez Citroën pour le dessin des carrosseries de voitures. Nous allons ici tracer seulement la courbe de Bézier de quatre points dans un plan, et sans entrer dans la théorie.

On se donne 4 points *A*, *B*, *C*, *D* que l'on joint. Puis on prend le milieu *ab* du segment [*A B*], puis celui *bc* de [*B C*] et celui *cd* de [*C D*]. Ensuite on prend le milieu *abc* de [*ab bc*] et celui *bcd* de [*bc cd*]. Enfin on prend le milieu *G* de [*abc bcd*]. On vient d'obtenir un point de la courbe de Bézier, et même on montre que le segment [*abc bcd*] est tangent en *G* à cette courbe. Cette courbe passe par ailleurs par les points extrêmes *A* et *D* où elle a pour tangentes respectives [*AB*] et [*CD*]. Notons que la ligne brisée obtenue avec les points *A ab abc bcd cd D* est assez proche de ce va être la courbe finale, et beaucoup plus que ne l'était la ligne brisée initiale *A B C D*.




Ce que l'on a fait avec les quatre points *A B C D* en prenant les milieux précédents, on le recommence avec les quatre points *A ab abc G*, ainsi qu'avec les quatre points *G bcd cd D*. Et ainsi de suite, avec une fonction qui se rappelle deux fois à chaque coup. En quelques pas récursifs on obtient la courbe finale.

Pour le programme on se donne les coordonnées des quatre points *A B C D* sur l'écran, soit *xA*, *yA*, *xB*, *yB*, etc. Puis on appelle la fonction *bezier* (*int xA*, *int yA*, *int xB*, *int yB*, *int xC*, *int yC*, *int xD*, *int yD*, 3), le nombre 3 indiquant que la fonction va se rappeler trois fois. Il reste à programmer la fonction [3] :

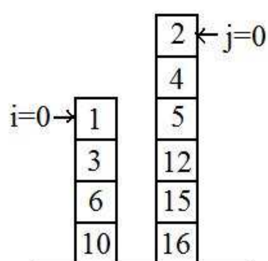
```
void bezier (int xa, int ya, int xb, int yb, int xc, int yc, int xd, int yd, int n)
{
 if (n==0) tracer les lignes joignant les points a b c d
 else
   { xab= (xa+xb)/2 ; yab=(ya+yb)/2 ; et de même pour bc puis cd
     xabc=(xab+xbc)/2 ; yabc=(yab+ybc)/2 ; et de même pour bcd
     xg=(xabc+xbcd)/2 ; yg=(yabc+ybcd)/2 ;
     bezier(xa,ya,xab,yab,xabc,yabc,xg,yg, n-1) ;
     bezier(xg,yg,xbcd,ybcd,xcd,ycd,xd,yd,n-1) ;
    }
}
```

# 4. Récursivité et tableaux

Au premier abord, la récursivité peut sembler allergique à l'usage de tableaux. Mais on a vu ci-dessus qu'elle pouvait s'accommoder de variables globales. Alors pourquoi pas des tableaux ? Deux programmes récursifs classiques utilisent ce genre de procédé. Il s'agit du tri par fusion, et de la transformée de Fourier rapide[4]. Nous allons prendre le premier exemple, celui du tri par fusion. Cela se fait en deux temps. D'abord on va apprendre à fusionner deux listes déjà triées, puis on utilisera cela pour le tri par fusion.

## 4.1. Fusion de deux listes déjà triées

A partir de deux listes déjà triées, on va constituer une seule liste triée englobant tous les éléments des deux listes. Cela permet de « grossir » le tri, sans pour autant le créer. Par exemple la fusion des deux listes triées 1 3 6 10 et 2 4 5 12 15 16 va donner la liste triée 1 2 3 4 5 6 10 12 15 16. Les deux listes de nombres sont placées dans deux tableaux *a*[*N*1] et *b*[*N*2] de longueurs *N*1 et ***N***2 (4 et 6 dans l'exemple précédent). Le résultat de la fusion sera placé dans un tableau *c*[*N*1+*N*2] vide au départ. Dessinons les deux tableaux verticalement :



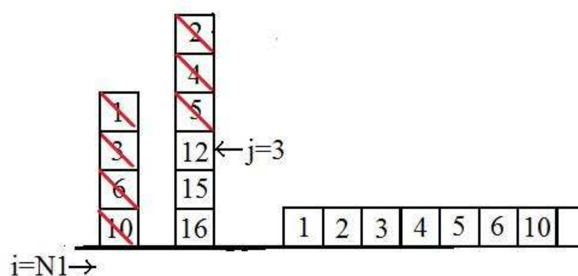

On commence par comparer les deux têtes (en *i*=0 et *j*=0). Le nombre le plus petit, 1, est envoyé dans la case 0 du tableau *c*, et l'on descend d'un cran (*i* passe à 1) dans le tableau concerné. Tout se passe comme si on avait coupé la tête de *a*[].Puis on compare les contenus des cases *i*=1 et *j*=0, et le nombre le plus petit est envoyé dans *c*, il s'agit de *b*[0]=2 et *j* passe à son tour à *j*=1. Et l'on continue, en comparant les têtes (en positions *i* et *j*) et en les coupant.

A un certain moment, une des deux listes est vide, c'est-à-dire que *i* vaut *N*1 ou bien *j* vaut *N*2. Dans le cas présent on est arrivé à cette situation :

---

[3] Pour aller plus loin, voir *courbe de Bézier en mouvement*, dans *Travaux complémentaires, algorithmes.*

[4] La transformée de Fourier est traitée dans *Travaux complémentaires, algorithmes, chap. 11 et 12.*




On a déjà ce début du programme de la fusion :

```
i=0 ; j=0 ; k=0 ;
while ( ! (i== N1 || j==N2)  /* tant qu'on n'a pas l'un ou l'autre tableau vide */
if (a[i]<b[j]) c[k++]=a[i++]
else c[k++]=b[j++] ;
```

Attention : si dans la boucle *while* précédente vous écrivez *while( i !=N1 || j !=N2)* c'est complètement faux. Ce qui serait juste est *while (i !=N2 && j !=N2).*

Après avoir coupé les têtes, il reste une deuxième étape : vider le tableau où il reste des éléments, dans l'exemple il s'agit du tableau *b*[] avec les nombres restants 12, 15, 16. En général il convient de se demander d'abord quel est le tableau qui est vidé :

```
if (i==N1)  while(j<N2)  c[k++]=b[j++] ;
else while (i<N1)  c[k++]=a[i++] ;
```

## 4.2. Tri par fusion

Commençons par la méthode itérative. On part d'une liste de *N* nombres, en supposant que *N* est une puissance de 2 (si tel n'est pas le cas, on peut toujours s'arranger en rajoutant des éléments factices). Puis on fait des fusions en prenant des morceaux de plus en plus grands. On commence par diviser le tableau par blocs de deux éléments, ces blocs étant formés chacun de deux sous-blocs d'un élément. On fusionne ces deux sous-blocs. Chaque bloc est alors trié. Puis on recommence avec des blocs deux fois plus grands. Comme dans l'exemple ci-dessous avec *N*=8 :

4 5 7 3 9 4 1 8 — *n* = 1. On coupe le tableau en blocs de 2*n* = 2 nombres et l'on fait dans chaque bloc une fusion à partir des sous-blocs de longueur *n*=1.

4 5 3 7 4 9 1 8 — *n* = 2. On coupe le tableau en blocs de 2*n* = 4 nombres et l'on fait la fusion à partir des sous-blocs de longueur *n*=2.

3 4 5 7 1 4 8 9 — *n* = 3. On coupe le tableau en blocs de 2*n* = 8, et l'on y fait la fusion à partir des sous-blocs de longueur *n*=4.

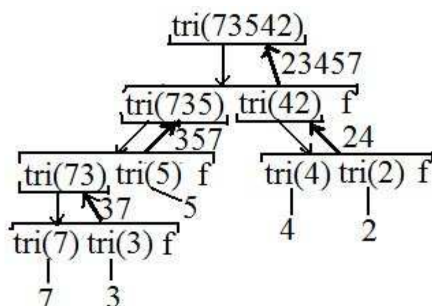
1 3 4 4 5 7 8 9 — *n* = 4. Fin

Pour le programme on a déjà la boucle principale :
*n=1 ;   while (n<N) { ….. ;  n=2*n ;}*

Pour chaque valeur de *n*, on prend un indice *g* marquant chaque début de bloc, d'où la boucle : *for(g=0 ; g<N ; g+=2*n)*. Dans chaque bloc, les indices *i* et *j* des débuts des deux sous-blocs sont *i=g* et *j=g+n*. Le programme s'ensuit, en y intégrant celui de la fusion :

```
remplir le tableau a[N] de la case 0 à N-1 (avec des nombres au hasard et N=8192 par exemple)
n=1 ;
while (n<N)
{
for(g=0 ;g<N ; g+=2*n)
  { i=g ; j=g+n ; k=0 ;
    while (i – g < n && j – g – n < n)
    if (ai]<a[j]) c[k++]=a[i++] :
    else c[k++]=a[j++] ;
    if (i – g == n) while (j – g – b < n)  c[k++]=a[j++]
    else while (i – g <n)  c[k++]= a[i++] ;
    for(i=g ; i < g+ 2*n ; i++)  a[i]=c[i-g] ;
  }
n=2*n ;
}
afficher le tableau a[] : il est trié.
```

Passons maintenant à la méthode récursive, qui fonctionne à l'envers de la précédente. Trier la liste, c'est trier chacune des deux moitiés, puis les fusionner. La fonction de tri se rappelle deux fois sur des listes de longueur moitié. A force de couper en deux, on finit toujours par tomber sur une liste de longueur 1, et il n'y a plus rien à faire dans ce cas : c'est le test d'arrêt. Voici un exemple (où *f* désigne la fusion) :




Le tableau contenant la liste va être coupé en morceaux, et le tri a lieu dans ces morceaux. Comment savoir quel est le morceau à trier ? Il suffit de gérer les indices *g* et *d* des deux cases extrêmes de ce morceau. Si *g* est inférieur à *d*, le tri entre *g* et *d* consiste à trier les deux moitiés puis à les fusionner. Si *g* est égal à *d*, le morceau n'a qu'un élément, et il n'y a rien à faire. Le programme s'en déduit. Remarquons que pour couper le morceau entre *g* et *d*, on calcule le milieu *m*, et les morceaux moitiés sont l'une entre g et *m*, et l'autre entre *m*+1 et *d*.

La fusion se fait entre ces deux morceaux moitiés à l'intérieur du tableau *a*[], d'où la nécessité d'adapter le programme de fusion précédent qui se faisait sur deux listes notées différemment. On utilise toujours un tableau auxiliaire *c*[] pour placer les éléments triés, mais une fois cela fait, on remet ce tableau *c*[] dans le morceau de *a*[] concerné : ce morceau est désormais trié.

```
int a[N] ;   /* le tableau est déclaré comme variable globale */
main()  {  remplir et afficher le tableau a[] ;
            tri(0,N-1) ;
            afficher le tableau a[] : il est trié
          }
void tri(int g, int d)
{ if (g<d)   { m=(g+d)/2 ;  tri(g,m) ; tri(m+1,d) ;  /* tri des deux moitiés */
               i=g ; j=m+1 ; k=0 ;   /* fusion */
              while (i<m+1 && j<d+1) if(a[i]<a[j]) c[k++]=a[i++] ;
                                     else c[k++]=a[j++] ;
              if (i==m+1) while (j<d+1) c[k++]=a[j++] ;
```

```
        else while(i<m+1) c[k++]=a[i++] ;
       for(i=g ; i<=d ; i++) a[i]=c[i-g] ;   /* on remet c[] dans le morceau de a */
      }
}
```

## 4.3. Performance du tri par fusion

Pour trier $N$ données, ce qui demande un temps $T(N)$, il convient de trier deux fois $N/2$ données, d'où un temps de 2 $T(N/2)$, puis à fusionner les deux listes. La fusion a l'avantage de se faire en un seul parcours, comme on l'a vu, soit un temps proportionnel à $N$. D'où la relation de récurrence

$T(N) = 2\,T(N/2) + N$, avec en conditions initiales $T(1) = 1$ : c'est le temps élémentaire pour trier une donnée.

Pour simplifier, on prend $N = 2^P$ afin que les coupes se fassent toutes en deux parties égales. Signalons que $N = 2^P$ signifie que $P = \log N$ par définition du logarithme, ici en base 2.

$$
\begin{array}{ll}
 & T(2^P) = 2T(2^{P-1}) + 2^P \\
2 & T(2^{P-1}) = 2T(2^{P-2}) + 2^{P-1} \\
2^2 & T(2^{P-2}) = 2T(2^{P-3}) + 2^{P-3} \\
 & \ldots\ldots \\
2^{P-1} & T(2) = 2T(1) + 2
\end{array}
$$

En multipliant la deuxième ligne par 2, la troisième ligne par $2^2=4$,…, et la dernière par $2^{P-1}$, puis en additionnant les lignes membre à membre, il se produit des simplifications en cascade, et il reste :

$T(2^P)= 2^P\,T(1) + (2^P + 2^P + 2^P +\ldots+ 2^P) = 2^P + P2^P= (P+1)2^P \approx P\,2^P = N \log N$

Pour $N$ grand, ce tri est en $N \log N$, et comme $\log N$ a tendance à devenir infiniment petit par rapport à $N$, ce tri devient très supérieur aux tris en $N^2$ que l'on a vu dans le chapitre précédent.