Diviser pour régner

Les trois phases du paradigme "diviser pour régner" :
- Diviser : division du problème en un certain de nombres de sous-problèmes (semblables au problème initial mais de taille moindre, par exemple de taille moitié).
- Conquérir et Régner : résolution des sous-problèmes
  - par des appels récursifs,
  - directement lorsque le sous-problème est de taille "petite".
- Combiner : combinaison des solutions des sous-problèmes pour constituer la solution globale.

1 Puissances

On propose ci-dessous deux algorithmes pour calculer la puissance n d'un réel x (rédigé dans le langage Xcas) :

Xcas
```
puiss(x,n):={
si n==0 alors return 1;
sinon return x*puiss(x,n-1);
fsi;
}:;
```

Xcas
```
puiss2(x,n):={
local y;
si n==0 alors return 1;fsi;
si n==1 alors return x;fsi;
y:=puiss2(x,iquo(n,2));
si irem(n,2)==0
alors return y*y;
sinon return x*y*y;
fsi;
}:;
```

1. Quel est le nombre de multiplications effectuées pour calculer x^{100} avec chacun des deux algorithmes ?
2. n étant de la forme 2^p, quel est le nombre de multiplications pour le calcul de x^n pour chacun des deux algorithmes ?

Une résolution

1. Pour calculer x^{100} par le premier algorithme, on effectue 99 multiplications. Avec le second, on effectue 8 multiplications : $x^{100} = x^{50} \times x^{50}$, $x^{50} = x^{25} \times x^{25}$, $x^{25} = x \times x^{12} \times x^{12}$, $x^{12} = x^6 \times x^6$, $x^6 = x^3 \times x^3$, $x^3 = x \times x \times x$.

$$x^{100} = ((x(((x \times x^2)^2)^2)^2)^2)^2$$

2. Pour $n = 2^p$.
   Avec le premier algorithme, x^n demande $n-1$ multiplications.
   Avec le second algorithme, x^n demande p multiplications (si $M(p)$ est le nombre de multiplications pour x^{2^p}, on a $M(p) = M(p-1)+1$).

2 Recherche dans une liste

On se propose de comparer les deux algorithmes suivants (langage Xcas) qui ont pour objectif de chercher le rang d'un élément dans une liste dont les éléments sont supposés triés dans l'ordre strictement croissant.
Remarque 1 : les listes sont numérotées en xcas de 0 à dim(liste)-1.
Remarque 2 : pour alléger le code, on suppose que la valeur cherchée est effectivement dans la liste.

Xcas

```
sequentiel(liste,valeur):={
local rang;
rang:=0;
tantque liste[rang]<>valeur faire
rang:=rang+1;
ftantque;
retourne rang;
};;
```

Xcas

```
dichotomique(liste,valeur):={
local debut,fin,k,trouve;
trouve:=false;
debut:=0;fin:=dim(liste)-1;
repeter
k:=iquo(debut+fin,2);
si liste[k]==valeur alors
trouve:=true;
sinon
si liste[k]<valeur alors debut:=k+1
    sinon fin:=k-1;fsi;
fsi;
jusqu_a trouve;
retourne k;
};;
```

1. Combien, au plus, d'itérations de la boucle auront lieu lors d'une recherche pour chacun des deux programmes ? Pour simplifier on se placera dans le cas où le nombre d'éléments dans la liste est de la forme $n = 2^p$.
2. On suppose que $n = 2^{100}$. Admettons que la machine fasse un million de boucles en 1 seconde, quel temps (dans le pire des cas) sera utilisé pour chacun des deux programmes lors d'une recherche ?
3. Écrire (xcas) une version récursive de chacun des deux programmes (avec les mêmes hypothèses simplificatrices faites plus haut).

Une résolution

1. (a) Pour la recherche séquentielle, le pire des cas est la situation correspondant à une valeur se trouvant en fin de liste. n itérations de la boucle auront lieu.

   (b) Pour la recherche dichotomique. Notons v_n le nombre maximum d'itérations. Pour $n = 1$, on a $v_1 = 1$. Pour $n = 2$, on a $v_2 = 2$. Pour $n = 2^2$, on a $v_4 = 3 \ldots$

   Pour $n = 2^p$, les sous-listes contiennent au plus 2^{p-1} éléments. On a donc $v_{2^p} \leqslant 1 + v_{2^{p-1}}$ (croissance de la suite (v_n)).

   On en déduit $v_{2^p} \leqslant p + v_1$ c'est à dire $v_n \leqslant 1 + \log_2(n)$.

   Pour n quelconque. On a : $n \leqslant 2^p$ où l'on a posé $p = \lceil \log_2(n) \rceil$. On a donc $v_n \leqslant v_{2^p} \leqslant 1 + p$. Soit $v_n \leqslant 1 + \lceil \log_2(n) \rceil$.
2. Pour la recherche par dichotomie, la recherche est quasi instantanée (une centaine de boucles). Pour la recherche séquentielle, on doit faire 2^{100} boucles soit environ $4 \cdot 10^{16}$ années.

3.

Xcas
```
sequentiel_rec(liste,valeur,k):={
si liste[k]==valeur
alors retourne k;
sinon
retourne sequentiel_rec(liste,
    valeur,k+1);
fsi;
};;
```

Xcas
```
dich_rec(liste,valeur,debut,fin):={
local k;
k:=iquo(debut+fin,2);
si liste[k]==valeur
alors
retourne k;
sinon
si liste[k]<valeur
alors
retourne dich_rec(liste,valeur,k+1,
    fin);
sinon
retourne dich_rec(liste,valeur,
    debut,k-1);
fsi;
fsi;
};;
```

Appels :

A:=[2,3,6,9,12];sequentiel_rec(A,9,0);

dich_rec(A,9,0,dim(A)−1)

3 La suite de Fibonacci

On rappelle la définition de la suite de Fibonacci :

$$F_0 = F_1 = 1 \quad \text{et} \quad \forall n \geqslant 1, F_{n+1} = F_n + F_{n-1}$$

1. Établir :
$$\forall n \geqslant 1, \forall p \geqslant 1, F_{n+p} = F_n F_p + F_{n-1} F_{p-1}$$
2. On pose $T_n := (F_{n-1}, F_n)$ pour $n \geqslant 1$. Exprimer T_{2n} et T_{2n+1} en fonction de F_n et F_{n-1}.
3. En déduire un programme fibo de calcul de F_n du type "diviser pour régner". Donner une estimation du nombre d'appels à fibo pour calculer F_n.

Une résolution

1. Récurrence sur p. Amorce assurée par $F_0 = F_1 = 1$. Pour l'hérédité :
$$\begin{aligned} F_{n+p+1} &= F_{n+p} + F_{n+p-1} \\ &= \left(F_n F_p + F_{n-1} F_{p-1}\right) + \left(F_n F_{p-1} + F_{n-1} F_{p-2}\right) \\ &= F_n \left(F_p + F_{p-1}\right) + F_{n-1} \left(F_{p-1} + F_{p-2}\right) \\ &= F_n F_{p+1} + F_{n-1} F_p \end{aligned}$$

2. Avec la relation précédente : $\boxed{F_{2n} = F_n^2 + F_{n-1}^2}$

   $F_{2n-1} = F_{n+n-1} = F_nF_{n-1} + F_{n-1}F_{n-2} = F_{n-1}(F_n + F_{n-2})$ soit $\boxed{F_{2n-1} = F_{n-1}(F_n + F_n - F_{n-1})}$.

   Et $F_{2n+1} = F_{n+n+1} = F_nF_{n+1} + F_{n-1}F_n$ soit $\boxed{F_{2n+1} = F_n(F_n + F_{n-1} + F_{n-1})}$.

   $T_{2n} = (F_{2n-1}, F_{2n}) = (F_{n-1}(2F_n - F_{n-1}), F_n^2 + F_{n-1}^2)$,

   $T_{2n+1} = (F_n^2 + F_{n-1}^2, F_n(F_n + 2F_{n-1}))$.

3. Algorithme :

Xcas

```
// calcul du couple (F_{n-1},F_n)
fibo(n):={
local T,u,v;
si n==0 alors return (0,1);fsi;
si n==1 alors return (1,1);fsi;
T:=fibo(iquo(n,2));
u:=T[0];v:=T[1];
si irem(n,2)==0 alors
return (u*(2*v-u),u^2+v^2);
sinon
return (u^2+v^2,v*(v+2*u));
fsi;
}:;
```

Scilab

```
function y=fibo(n)
  if n==0 then
    y=[0,1]
  else
    if n==1 then
      y=[1,1]
    else
      t=fibo(quotient(n,2));u=t(1);v=t(2);
      if reste(n,2)==0 then
        y=[u*(2*v-u),u*u+v*v]
      else
        y=[u*u+v*v,v*(v+2*u)]
      end
    end
end
endfunction
```

Notons A_n le nombre d'appels à fibo pour le calcul de F_n (ou de T_n).

$A_0 = 1, A_1 = 1$, $A_2 = 2$, $A_n = 1 + A_{\lfloor \frac{n}{2} \rfloor}$.

On a donc un nombre d'appels qui vaut approximativement $1 + \log_2(n)$.

4 Quick sort

4.1 Partition

On considère l'algorithme suivant (langage xcas) dans lequel T est une liste :

Xcas

```
partition(T,deb,fin,clef):={
local d,f,tmp;
d:=deb;f:=fin;
repeter
        tantque d<=fin faire
        si T[d]<=clef alors d:=d+1 sinon break;fsi;
        ftantque;
        tantque f>=deb faire
        si T[f]>clef alors f:=f-1 sinon break;fsi;
        ftantque;
        si d<f alors
        tmp=<T[d];T[d]=<T[f];T[f]=<tmp;   // on échange T[d] et T[f]
        d:=d+1;f:=f-1;
        fsi;
jusqu_a d>f;
retourne f;
};;
```

1. Que renvoie partition ([5,10,12,3,7,4,2,6,9,8,1,13],0,11,7) ?
2. Justifier que l'algorithme se termine.
3. Justifier qu'après application de l'algorithme, tous les éléments de T[deb] à $T[f]$ sont inférieurs ou égaux à clef et tous les éléments de $T[f+1]$ à T[fin] sont supérieurs strictement à clef. Pour cela, démontrer par récurrence l' "invariant" : « A la fin de l'itération k de la boucle répéter, tous les éléments T[deb] à $T[d-1]$ sont inférieurs ou égaux à clef et tous les éléments de $T[f+1]$ à T[fin] sont supérieurs strictement à clef. »
4. Quel est le nombre de comparaisons avec la clef dans le pire des cas (dans les deux boucles tant que) ? le nombre d'échanges dans le pire des cas (dans le « si $d < f$ ») ?

Une résolution

1. 6.

   Détail : clef= 7, deb= 0, fin= 11.

   (a) Étape 1 :

indices	$d=0$	1	2	3	4	5	6	7	8	9	10	$f=11$
valeurs	5	10	12	3	7	4	2	6	9	8	1	13

   (b) Étape 2 :

indices	0	d=1	2	3	4	5	6	7	8	9	f=10	11
valeurs	5	10	12	3	7	4	2	6	9	8	1	13

indices	0	1	d= 2	3	4	5	6	7	8	f=9	10	11
valeurs	5	1	12	3	7	4	2	6	9	8	10	13

(c) Étape 3

indices	0	1	d= 2	3	4	5	6	f=7	8	9	10	11
valeurs	5	1	12	3	7	4	2	6	9	8	10	13

indices	0	1	2	d=3	4	5	f=6	7	8	9	10	11
valeurs	5	1	6	3	7	4	2	12	9	8	10	13

(d) Étape 4

indices	0	1	2	3	4	5	f=6	d=7	8	9	10	11
valeurs	5	1	6	3	7	4	2	12	9	8	10	13

(e) Étape 5 : return 6.

A la fin de l'algorithme, les valeurs du tableau de T[deb] à $T[f]$ sont $\leqslant$clef et les valeurs de $T[f+1]$ à T[fin] sont > clef.

2. Tant que $d < f$, d est incrémenté. d est donc incrémenté au moins une fois à chaque itération de la boucle « repeter » (et f n'augmente pas) donc la condition $d > f$ aura nécessairement lieu.

   Pour la première boucle « tantque » interne : si elle n'est pas arrêtée par le break, alors d est incrémenté à chaque tour et finira par dépasser fin (fin reste fixe). Idem pour la boucle « tantque » concernant f.

3. On démontre ("invariant") : « A la fin de l'itération k de la boucle répéter, tous les éléments T[deb] à $T[d-1]$ sont inférieurs ou égaux à clef et tous les éléments de $T[f+1]$ à T[fin] sont supérieurs strictement à clef. »

   (a) La propriété est vraie à la fin de la première itération :

      i. La première boucle tant que parcourt les éléments du tableau :

         A. si la boucle s'arrête parce que d = fin+1, c'est qu'on a $T[j] \leqslant$ clef pour tous les indices j, f sera alors inchangé dans la suite du programme qui s'arrête et renvoie f = fin, la propriété est dans ce cas vérifiée.

         B. sinon cette boucle tant que s'arrête avec la première valeur de d telle que $T[d] >$ clef.

      ii. La seconde boucle tant que parcourt les éléments du tableau :

         A. si la boucle s'arrête parce que f = deb -1, c'est qu'on a $T[j] >$ clef pour tous les indices j, d a alors été inchangé dans la boucle précédente du programme qui s'arrête et renvoie $f =$ deb -1, la propriété est dans ce cas vérifiée.

         B. sinon cette boucle tant que s'arrête avec la première valeur de f telle que $T[f] \leqslant$ clef.

      iii. A cette étape (début du « si $d < f$ »), on a soit $d > f$, soit $d < f$ ($d = f$ est impossible puisque $T[d] >$ clef et $T[f] \leqslant$ clef). Si on a $d > f$, l'algorithme se termine et la propriété est vérifiée pour le tableau. Si $d < f$, on échange les valeurs de $T[d]$ et $T[f]$ et on a donc T[deb], ..., $T[d]$ inférieurs ou égaux à clef et $T[f]$, ..., T[fin] supérieurs strictement à la clef.

      On incrémente alors d et décrémente f et on a bien à la fin de la boucle : tous les éléments T[deb] à $T[d-1]$ sont inférieurs ou égaux à clef et tous les éléments de $T[f+1]$ à T[fin] sont supérieurs strictement à clef.

   (b) On passe de l'étape k à l'étape $k+1$ par les mêmes arguments.

   (c) Comme l'algorithme s'arrête avec $d > f$, la propriété annoncée pour le tableau en sortie est vérifiée.

4. Les comparaisons $T[d] \leqslant$ clef et $T[f] >$ clef s'arrêtent lorsque d dépasse f. On a un nombre de comparaisons en O(fin-deb).
   Pour les échanges, le "pire" arrive lorsque tous les tests des tant que ne sont jamais satisfaits et qu'on échange à chaque coup. Soit approximativement (fin-deb)/2 échanges.
   On peut donc estimer que la complexité de partition est en O(n).

4.2 Quick Sort

On considère l'algorithme suivant (écrit en langage xcas) :

Xcas

```
tri_rap(T,deb,fin):={
local d,f,clef;
si deb<fin alors
        d:=deb;f:=fin;clef=<T[deb];
        repeter
                tantque d<=fin faire
                si T[d]<=clef alors d:=d+1 sinon break;fsi;
                ftantque;
                tantque f>=deb faire
                si T[f]>clef alors f:=f-1 sinon break;fsi;
                ftantque;
                si d<f alors
                tmp=<T[d];T[d]=<T[f];T[f]=<tmp;
                d:=d+1;f:=f-1;
                fsi;
        jusqu_a d>f;
T[deb]=<T[f];T[f]=<clef; //échange de T[deb] et T[f]
T=<tri_rap(T,deb,f-1);T=<tri_rap(T,f+1,fin);
fsi;
return T;
};;
```

1. Quel est l'effet de tri_rap sur $T := [2,8,6,12,5]$ par exemple ?
2. Démontrer, par récurrence sur la taille $n = \text{fin} - \text{deb} + 1$ du tableau, l'effet de tri_rap sur une liste T.
3. Lorsque le tableau de taille n donné en entrée est déjà trié, estimer la complexité.

Une résolution

1. T:=[2,8,6,12,5]:; tri_rap(T,0,dim(T)-1)) donne un tableau trié dans l'ordre croissant : [2,5,6,8,12].
2. L'algorithme proposé se résume ainsi :

```
Tri_Rap(T,deb,fin)
si deb<fin alors
        f:=partition(T,deb,fin,T[deb])
```

```
        échanger T[deb] et T[f]
        Tri_Rap(T,deb,f-1)
        Tri_Rap(T,f+1,fin)
finSi
```

Démontrons maintenant (récurrence sur la taille du tableau $n = \text{fin} - \text{deb} + 1$) la validité de l'algorithme.

(a) Lorsque $n \leqslant 1$, c'est à dire fin$\leqslant$ deb, l'algorithme ne modifie pas le tableau et celui-ci, comportant au plus un élément, est trié.

(b) Supposons que pour tout tableau tel que fin−deb+1 $< n$ l'algorithme trie le tableau. Considérons un tableau T tel que fin−deb+1 $= n$. partition(T,deb,fin, T[deb]) sépare le tableau en deux sous-tableaux tels que T[deb], ..., $T[f]$ contiennent tous des éléments $\leqslant T[deb]$ et $T[f+1]$, ..., T[fin] contiennent tous des éléments $> T$[deb]. On échange ensuite T[deb] et $T[f]$. L'élément d'indice f est donc maintenant à sa place et les appels récursifs trient (hypothèse de récurrence) les sous-tableaux T[deb$\ldots f-1$] et $T[f+1\ldots$fin].

3. Lorsque le tableau T[deb$\ldots$fin] de taille n est donné trié, partition(T,deb,fin,T[deb]) renvoie f =deb et ne change rien au tableau. Tri_Rap s'applique alors sur un sous-tableau gauche vide (cet appel se termine immédiatement) et un sous tableau droit de taille $n-1$. Comme l'appel à partition est en O(n), la complexité de Tri_Rap vérifie alors :

$$C(n) = C(n-1) + O(n) \text{ pour } n > 1$$

Avec la partie 1 , on a pour partition un α telle que la "complexité" est $\leqslant \alpha n$ pour tout n. On a donc : $C(n) \leqslant C(n-1) + \alpha n$ d'où $C(n) \leqslant C(0) + \alpha(1+2+\cdots+n)$ soit $C(n) \leqslant C(0) + \frac{1}{2}\alpha n(n+1)$.
$C(n)$ est donc un $O(n^2)$.

5 Karatsuba

1. On considère l'algorithme suivant (donné en langage xcas) :

Xcas

```
gaga(P,Q):={
local dp,dq,R,x,k,j;
dp:=dim(P)-1;dq:=dim(Q)-1;
R=<makelist(x->0,0,dp+dq);
pour k de 0 jusque dp faire
pour j de 0 jusque dq faire
R[k+j]=<R[k+j]+P[k]*Q[j];
fpour;fpour;
retourne R;
}:;
```

(a) Que renvoie l'appel gaga ([3,4,5],[4,7,9,10]) ?

(b) Combien de multiplications et additions lors de l'exécution de ce programme ?

2. Montrer comment multiplier les deux polynômes $P(X) = a_1X + a_0$ et $Q(X) = b_1X + b_0$ en utilisant seulement trois multiplications « élémentaires » où l'une des multiplications est $(a_0+a_1)\times(b_0+b_1)$.
3. En déduire un algorithme du type "diviser pour régner" permettant de multiplier deux polynômes de degré au plus $n-1$. Soit P le polynôme $P(X) = \sum_{j=0}^{n-1} a_j X^j$, on peut diviser ainsi :

$$P(X) = \left(a_0 + a_1X + \cdots + a_{k-1}X^{k-1}\right) + X^k\left(a_k + a_{k+1}X + \cdots + a_{n-1}X^{\ell-1}\right)$$

où l'on a posé $k = \lfloor \frac{n}{2} \rfloor$ et $\ell = n-k = \lceil \frac{n}{2} \rceil$.

4. Écrire le détail pour le produit de $P(x) = a_0 + a_1x + a_2x^2$ et $Q(x) = b_0 + b_1x + b_2x^2$.
5. Évaluer le nombre de multiplications et le nombre d'additions (en se limitant au cas où n est une puissance de 2 pour simplifier).

Une résolution

1. (a) On obtient la liste [12,37,75,101,85,50] c'est à dire la liste des coefficients du produit expand ((3+4*x+5*x^2)*(4+7*x+9*x^2+10*x^3)) (= 50*x^5+85*x^4+101*x^3+75*x^2+37*x+12).

   (b) Avec une liste P de longueur p (soit un polynôme de degré $p-1$) et une liste Q de longueur q (polynôme de degré $q-1$), le nombre de multiplications est $p \times q$, ainsi que le nombre d'additions. Soit de l'ordre de $2n^2$ opérations pour le produit de deux polynômes, à comparer avec l'algorithme proposé ensuite pour les grandes valeurs de n.
2. $$P(X)Q(X) = a_0b_0 + ((a_0+a_1)\times(b_0+b_1) - a_0b_0 - a_1b_1)\,X + a_1b_1X^2$$
3. On divise :

$$P(X) = \left(a_0 + a_1X + \cdots + a_{k-1}X^{k-1}\right) + X^k\left(a_k + a_{k+1}X + \cdots + a_{n-1}X^{\ell-1}\right) = P_0 + X^kP_1$$

et

$$Q(X) = \left(b_0 + b_1X + \cdots + b_{k-1}X^{k-1}\right) + X^k\left(b_k + b_{k+1}X + \cdots + b_{n-1}X^{\ell-1}\right) = Q_0 + X^kQ_1$$

Le produit :

$$R(X) = \left(P_0 + X^kP_1\right)\times\left(Q_0 + X^kQ_1\right)$$

s'écrit :

$$R(X) = P_0Q_0 + X^k\left((P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1\right) + X^{2k}P_1Q_1$$

Xcas

```
karat(P,Q):=
{local k,l,j,n,P0,Q0,P1,Q1,P01,Q01,R0,R1,R2,R;
n:=max(dim(P),dim(Q));

// on met P et Q au même degré :
P=<P+makelist(x->0,1,n);
Q=<Q+makelist(x->0,1,n);

// cas de base :
si dim(P)==1 et dim(Q)==1 alors return [P[0]*Q[0]];fsi;

// on divise :
k:=iquo(n,2);l:=n-k;
P0=<mid(P,0,k);Q0=<mid(Q,0,k);
P1=<mid(P,k,n-k);Q1=<mid(Q,k,n-k);
P01=<P0+P1;Q01=<Q0+Q1;
R0=<karat(P0,Q0);
R1=<karat(P01,Q01);
R2=<karat(P1,Q1);

// on assemble :
R=<makelist(x->0,0,2*n-2);
pour j de 0 jusque 2*k-2 faire R[j]=<R0[j];fpour;
pour j de 0 jusque 2*l-2 faire R[j+2*k]=<R2[j];fpour;
pour j de 0 jusque 2*k-2 faire R[j+k]=<R[j+k]+R1[j]-R0[j]-R2[j];fpour;
pour j de 2*k-1 jusque 2*l-2 faire R[j+k]=<R[j+k]+R1[j]-R2[j];fpour;
return R;
}:;
```

en plaçant des zéros pour clarifier (?) le rôle des boucles finales dans la reconstitution :

Xcas

```
karat(P,Q):=
{local k,l,j,n,P0,Q0,P1,Q1,P01,Q01,R0,R1,R2,R3,R,x;

// on met P et Q au "même degré" :
n:=max(dim(P),dim(Q));
P=<P+makelist(x->0,1,n);
Q=<Q+makelist(x->0,1,n);

// cas de "base" :
si dim(P)==1 et dim(Q)==1 alors return [P[0]*Q[0]];fsi;

// on divise :
k:=iquo(n,2);l:=n-k;
P0=<mid(P,0,k);Q0=<mid(Q,0,k);
P1=<mid(P,k,n-k);Q1=<mid(Q,k,n-k);
P01=<P0+P1;Q01=<Q0+Q1;

// sous-cas :
R0=<karat(P0,Q0);
R1=<karat(P01,Q01);
R2=<karat(P1,Q1);

// assemblage :
R3=<R1-R0-R2;
R3=<augment(makelist(x->0,0,k-1),R3);
R2=<augment(makelist(x->0,0,2*k-1),R2);
R=<R0;
R=<R+R2;
R=<R+R3;
return R;
};;
```

4. Illustration de la version 2 ci-dessus avec $P := [a_0, a_1, a_2]$ et $Q := [b_0, b_1, b_2]$.
   karat(P,Q)
   (a) divise :
   $P0 := [a_0]$, $Q0 := [b_0]$,
   $P1 := [a_1, a_2]$, $Q1 := [b_1, b_2]$,
   $P01 := [a_0 + a_1, a_2]$, $Q01 := [b_0 + b_1, b_2]$.
   (b) Appels récursifs :
      i. karat(P0,Q0) renvoie $R0 := [a_0 b_0]$.
      ii. karat(P01,Q01) divise :
      $P0 := [a_0 + a_1]$, $Q0 := [b_0 + b_1]$,
      $P1 := [a_2]$, $Q1 := [b_2]$,
      $P01 := [a_0 + a_1 + a_2]$, $Q01 := [b_0 + b_1 + b_2]$.

Appels récursifs :

A. karat(P0,Q0) renvoie $[(a_0+a_1)\times(b_0+b_1)]$.

B. karat(P01,Q01) renvoie $[(a_0+a_1+a_2)\times(b_0+b_1+b_2)]$.

C. karat(P1,Q1) renvoie $[a_2\times b_2]$

et assemblage :

$R3 := [(a_0+a_1+a_2)\times(b_0+b_1+b_2)] - [(a_0+a_1)\times(b_0+b_1)] - [a_2\times b_2]$ soit $R3 = [a_0b_2 + a_1b_2 + a_2b_0 + a_2b_1]$ puis $R3 := [0, a_0b_2 + a_1b_2 + a_2b_0 + a_2b_1]$ et $R2 := [0,0,a_2\times b_2]$ puis $R := [(a_0+a_1)\times(b_0+b_1), a_0b_2 + a_1b_2 + a_2b_0 + a_2b_1, a_2\times b_2]$

iii. karat(P1,Q1) divise :

$P0 := [a_1]$, $Q0 := [b_1]$,
$P1 := [a_2]$, $Q1 := [b_2]$,
$P01 := [a_1+a_2]$, $Q01 := [b_1+b_2]$.

Appels récursifs :

A. karat(P0,Q0) renvoie $[a_1\times b_1]$.

B. karat(P01,Q01) renvoie $[(a_1+a_2)\times(b_1+b_2)]$.

C. karat(P1,Q1) renvoie $[a_2\times b_2]$

et assemblage :

$R3 := [(a_1+a_2)\times(b_1+b_2)] - [a_1\times b_1] - [a_2\times b_2]$ soit $R3 = [a_1b_2+a_2b_1]$ puis $R3 := [0, a_1b_2 + a_2b_1]$ et $R2 := [0,0,a_2\times b_2]$ puis $R := [a_1\times b_1, a_1b_2+a_2b_1, a_2b_2]$

(c) assemble :

$R3 := R1 - R0 - R2 = [a_0b_1 + a_1b_0, a_0b_2 + a_2b_0, 0]$,
puis $R3 := [0, a_0b_1 + a_1b_0, a_0b_2 + a_2b_0, 0]$
et $R2 := [0,0,a_1b_1, a_1b_2 + a_2b_1, a_2b_2]$.
puis $R := R0 + R3 = [a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_2b_0, 0]$
et enfin $R := R + R2 = [a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_2b_0 + a_1b_1, a_1b_2 + a_2b_1, a_2b_2]$

5. Soit $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ et $Q(x) = b_0 + b_1x + b_2x^2 + b_3x^3$.
On détermine PQ par :

$$
\begin{aligned}
P(x)Q(x) &= (a_0+a_1x)\times(b_2+b_3x) && \text{trois produits}\\
&\quad + x^4[(a_2+a_3x)\times(b_2+b_3x)] && \text{trois produits}\\
&\quad + x^2\times[((a_0+a_1x)+(a_2+a_3x))\times((b_0+b_1x)+(b_2+b_3x)) && \text{trois produits}\\
&\quad - (a_0+a_1x)\times(b_2+b_3x) - (a_2+a_3x)\times(b_2+b_3x)]
\end{aligned}
$$

Pour deux polynômes de degré $n-1$, $n = 2^p$.
On note $M(p)$ le nombre de multiplications. On a $M(0) = 1$ et $M(p) = 3M(p-1)$. D'où $M(p) = 3^p$, soit un nombre de multiplications de $2^{p\times\log_2(3)} = n^{\log_2(3)}$.
On note $A(p)$ le nombre d'additions. $A(0) = 0$,

et (en s'appuyant sur le programme 1 xcas) : $A(p) = 3A(p-1) + 2 \times 2^{p-1} + 3 \times (2^p - 1)$
$(3A(p-1)$: appels récursifs ; $2 \times 2^{p-1}$: construction de $P01$, $Q01$; $3 \times (2^p - 1)$: assemblage).
Soit $A(p) = \frac{13}{2} \times 3^p - 8 \times 2^p + \frac{3}{2}$.
soit $\frac{13}{2} \times n^{\log_2(3)} - 8n + \frac{3}{2}$.
Nombre d'opérations en $\Theta(n^{\log_2(3)})$.