Problème A – *Parties d'un ensemble fini*

1) Ici une version itérative semble naturelle, pas de difficulté particulière.

Je pars d'une liste vide A et je balaye l'ensemble des valeurs de l'indice k en ajoutant à A la valeur `E[k]` si et seulement si `p[k]` vaut 1. Justification banale.

```
def Partie(E,p):
    A=[]
    for k in range(len(E)):
        if p[k]==1: A.append(E[k])
    return A
```

2) a) L'exemple de l'énoncé était censé donner l'idée : ajouter 1 à un entier donné par son écriture en base 2 se fait en parcourant les chiffres de ladite écriture, à partir de celui de poids le plus faible, à la recherche du premier 0, que l'on remplace par un 1, tous les 1 rencontrés précédemment étant remplacés par des 0. Ce principe découle de l'identité

$$\sum_{j=0}^{k-1} 2^j = \frac{1-2^k}{1-2} = 2^k - 1$$

qui se traduit par l'écriture en base 2

$$\overline{\underbrace{11\ldots1}_{k \text{ fois } 1}}^2 + 1 \quad = \quad \overline{1\underbrace{00\ldots0}_{k \text{ fois } 0}}^2$$

```
def Ajoute1(p):
    k=0
    while p[k]==1:
        p[k]=0
        k+=1
    p[k]=1
    return p
```

J'applique le principe décrit ci-dessus, en remplaçant au fur et à mesure par des 0 les 1 des chiffres de poids croissants, jusqu'à trouver le premier 0 que je remplace par un 1. La boucle se termine bien, puisque si `p` ne contenait que des 1, c'est qu'il représenterait l'entier $j = 2^\ell - 1$, ce qui est exclu.

b) J'applique le principe décrit dans l'énoncé, qui justifie ce programme :

```
def EnsPartiesI(E):
    n=len(E)
    p=n*[0]
    L=[[]]
    for j in range(1,2**n):
        p=Ajoute1(p)
        L.append(Partie(E,p))
    return L
```

Ne pas oublier d'initialiser `p` !

On pourrait être tenté d'intervertir les deux instructions de la boucle et d'y passer 2^n fois, ce qui mettrait certes la partie vide sur le même plan que les autres, mais poserait problème au moment d'ajouter le dernier 1…

C'est pourquoi j'ai initialisé L avec déjà la partie vide : `[[]]` est une liste de longueur 1 dont l'unique élément est la liste vide !

3) Question subsidiaire moins détaillée… Voici une solution :

```
def EnsPartiesR(E):
    def Pres(k):
        if k==0:
            return [[]]
        else:
            L1=Pres(k-1)
            L=[]
            for t in L1:
                t.append(0)
                L.append(t.copy())
                t[-1]=1
                L.append(t)
            return L

    Lp=Pres(len(E))
    return [Partie(E,p) for p in Lp]
```

La fonction auxiliaire récursive `Pres` engendre récursivement tous les tableaux de présence possibles comme le suggérait l'énoncé, selon le principe suivant. `Pres(k)` renvoie la liste de tous les tableaux de présence à `k` éléments, suivant l'idée récursive "naturelle", qui justifie la preuve par récurrence :

- `Pres(0)` renvoie la liste `[[]]` dont le seul élément est la liste vide !
- Supposant que `Pres(k-1)` a renvoyé la liste prévue, notée `L1`, je construis la liste `L` des tableaux de présence souhaités en prenant chacun de ceux de `L1` augmenté d'un 0 (parties ne contenant pas le dernier élément), puis le même en remplaçant ce dernier 0 par un 1 (parties contenant le dernier élément).

Pour la fonction `EnsPartiesR`, il suffit alors de convertir les tableaux de présence en parties grâce à la fonction du **1)**.

Attention ! Il ne faut pas oublier de créer des copies des tableaux `t` constituant la liste `L1`, sinon on se retrouve avec des redondances dues au partage des données… Toutefois, pour la deuxième utilisation de `t`, la copie n'est plus nécessaire car `t` ne resservira pas.

Problème B – *Transformée de Fourier rapide*

1) *Quelques utilitaires*

a) Je commence par remplacer chaque élément de P par l'arrondi de sa partie réelle, puis j'élimine les zéros en partant de la fin et en m'arrêtant à la première valeur non nulle (le coefficient dominant), ou lorsque la liste est vide (cas du polynôme nul, sur lequel l'énoncé a jeté un voile pudique puisqu'il s'agit de calculer des produits de polynômes…).

```
def arrondi(P):
    P=[round(z.real) for z in P]
    while P!=[] and P[-1]==0: P.pop()
    return P
```

b) Sans commentaires… Je stocke $2k\pi/n$ pour économiser quelques calculs.

```
def u(k,n):
    t=2*k*pi/n
    return complex(cos(t),sin(t))
```

c) Pas de difficulté particulière, la double instruction de la boucle ne posera pas de problème car t est de longueur paire par hypothèse !

```
def separe(t):
    t0,t1=[],[]
    i=0
    while i<len(t):
        t0.append(t[i]);i+=1
        t1.append(t[i]);i+=1
    return t0,t1
```

d) Ici, attention à la consigne concernant la complexité ! Il ne faut pas recalculer une puissance de w à chaque itération... Or je dois remplacer, pour $1 \leq k \leq n-1$, a_k par $a_k \cdot w^k$ (a_0 est inchangé !) ; c'est pourquoi j'initialise une variable c à la valeur w et — pour k allant de 1 à $n-1$ — je multiplie a_k par c, puis je multiplie c par w. Ainsi, c contient bien w^k, à l'entrée de la boucle pour la valeur k, ce qui justifie ce programme, qui effectue $2(n-1)$ multiplications.

```
def z_wz(P,w):
    c=w
    for i in range(1,len(P)):
        P[i]*=c
        c*=w
    return P
```

2) *Calcul de $Y = P\langle U\rangle$, P étant donné*

a) Il suffit d'écrire, en séparant les termes de rang pair des termes de rang impair :

$$P(z) = \sum_{k=0}^{2m-1} a_k \cdot z^k = \sum_{j=0}^{m-1} a_{2j} \cdot z^{2j} + \sum_{j=0}^{m-1} a_{2j+1} \cdot z^{2j+1} = \sum_{j=0}^{m-1} a_{2j} \cdot \left(z^2\right)^j + z \cdot \sum_{j=0}^{m-1} a_{2j+1} \cdot \left(z^2\right)^j,$$

ainsi :

$$\boxed{P(z) = P_0\left(z^2\right) + z \cdot P_1\left(z^2\right) \quad \text{avec} \quad P_0(X) = \sum_{j=0}^{m-1} a_{2j} \cdot X^j \quad \text{et} \quad P_1(X) = \sum_{j=0}^{m-1} a_{2j+1} \cdot X^j.}$$

De plus, je remarque (habilement) que, pour $0 \leq k \leq n-1$,

$$P_0\left(u_{k,n}^2\right) = P_0\left(e^{2i\cdot 2k\pi/n}\right) = P_0\left(e^{2ik\pi/m}\right) = P_0\left(u_{j,m}\right)$$

en posant $\begin{cases} j = k & \text{si } 0 \leq k \leq m-1 \\ j = k-m & \text{si } m \leq k \leq n-1 \text{ (puisque } e^{2im\pi/m} = 1 \text{ !)} \end{cases}$.

De même pour P_1, d'où finalement, en remarquant que $u_{m+j,n} = e^{2i(m+j)\pi/n} = e^{i\pi} \cdot u_{j,n} = -u_{j,n}$:

$$\boxed{\forall j \in [\![0, m-1]\!] \quad \begin{cases} P\left(u_{j,n}\right) = P_0\left(u_{j,m}\right) + u_{j,n} \cdot P_1\left(u_{j,m}\right) \\ P\left(u_{m+j,n}\right) = P_0\left(u_{j,m}\right) - u_{j,n} \cdot P_1\left(u_{j,m}\right) \end{cases}}.$$

b) Le résultat précédent permet de "diviser pour régner" :

```
def PtoY(P):
    if len(P)==1:
        return [P[0]]
    else:
        n=len(P);m=n//2
        P0,P1=separe(P)
        Y0=PtoY(P0);Y1=PtoY(P1)
        Y=n*[0]
        for j in range(m):
            tmp=u(j,n)*Y1[j]
            Y[j]=Y0[j]+tmp
            Y[m+j]=Y0[j]-tmp
        return Y
```

La terminaison est assurée par la décroissance stricte de n au fur et à mesure des appels récursifs et par l'arrêt pour $n = 1$;

L'exactitude du résultat découle des questions précédentes et d'une récurrence sur l'entier p tel que $n = 2^p$. Soit, pour $p \in \mathbb{N}$, $\mathcal{P}_p$ le prédicat suivant : “Pour tout polynôme P de degré $n-1$, où $n = 2^p$, l'appel `PtoY(P)` remplit $Y_0, \ldots, Y_{n-1}$ avec $P(u_{0,n}), \ldots, P(u_{n-1,n})$”.

* $\mathcal{P}_0$ est bien vérifié (P coïncidant dans ce cas avec la constante $a_0 = P(0)$!).
* Je suppose $p \in \mathbb{N}$ tel que $\mathcal{P}_p$ soit vrai et je considère P de degré $n-1$, où $n = 2^{p+1}$; alors `separe(P,n,P0,P1)` remplit les tableaux `P0`, `P1` représentant les polynômes P_0, P_1 du **a)**, puis les deux appels à `PtoY` remplissent `Y0`, `Y1` selon l'hypothèse de récurrence ; enfin la boucle `for` remplit convenablement `Y`, en vertu du **a)** *in fine*.

Quant au nombre $C(n)$ de multiplications effectuées lors de l'appel `PtoY(P)`, il vérifie la relation de récurrence :

$$C(n) = 2C(n/2) + n/2,$$

d'où, grâce au “rappel” de l'énoncé :

$$\boxed{C(n) = O(n \log_2 n).}$$

3) *Calcul de P, $Y = P\langle U\rangle$ étant donné*

a) Soit P^* le polynôme défini par

$$\forall z \in \mathbb{C} \quad P^*(z) = \frac{1+z^m}{2} \cdot P_0(z) + \frac{1-z^m}{2} \cdot P_1\left(e^{-2i\pi/n} \cdot z\right).$$

Je montre que P^* vérifie $P^*\langle U\rangle = Y$; soit $k \in [\![0, n-1]\!]$:

* si k est pair, $k = 2j$ avec $j \in [\![0, m-1]\!]$; alors
$$u_{k,n} = e^{2ij\pi/m} = u_{j,m} \quad \text{et} \quad u_{k,n}^m = u_{j,m}^m = 1,$$
d'où
$$P^*(u_{2j,n}) = P_0(u_{j,m}) = y_{2j} \text{ par définition de } P_0.$$
* si k est impair, $k = 2j+1$ avec $j \in [\![0, m-1]\!]$; alors
$$e^{-2i\pi/n} \cdot u_{k,n} = u_{j,m} \quad \textit{et} \quad u_{k,n}^m = \left(u_{j,m} \cdot e^{i\pi/m}\right)^m = -1,$$
d'où
$$P^*(u_{2j+1,n}) = P_1(u_{j,m}) = y_{2j+1} \text{ par définition de } P_1.$$

Ainsi, j'ai bien $P^*\langle U\rangle = Y$, or P^* est de degré au plus égal à $2m-1 = n-1$, d'où, par unicité du polynôme vérifiant ces deux conditions, $P^* = P$, autrement dit :

$$\boxed{\forall z \in \mathbb{C} \quad P(z) = \frac{1+z^m}{2} \cdot P_0(z) + \frac{1-z^m}{2} \cdot P_1\left(e^{-2i\pi/n} \cdot z\right).}$$

J'en déduis les relations entre coefficients : si je suppose

$$P_0(z) = \sum_{j=0}^{m-1} b_j \cdot z^j \quad \text{et} \quad P_1(z) = \sum_{j=0}^{m-1} c_j \cdot z^j,$$

j'ai, en posant $w = e^{-2i\pi/n}$

$$\boxed{\forall z \in \mathbb{C} \quad P(z) = \sum_{j=0}^{m-1} \frac{b_j + w^j \cdot c_j}{2} \cdot z^j + \sum_{j=0}^{m-1} \frac{b_j - w^j \cdot c_j}{2} \cdot z^{m+j}.}$$

b) Là encore, divisons pour régner :

```
def YtoP(Y):
    if len(Y)==1:
        return [Y[0]]
    else:
        n=len(Y);m=n//2
        Y0,Y1=separe(Y)
        P0=YtoP(Y0);P1=YtoP(Y1)
        P1=z_wz(P1,u(-1,n))
        P=n*[0]
        for j in range(m):
            P[j]=0.5*(P0[j]+P1[j])
            P[m+j]=0.5*(P0[j]-P1[j])
        return P
```

Les justifications sont similaires à celles de la question **2)b)**...

J'ai utilisé la fonction `z_wz` pour remplacer une fois pour toutes les c_j par les $w^j \cdot c_j$.

4) Pour appliquer ce qui précède au calcul du produit $C = A \cdot B$ de deux polynômes A, B de degrés d_A, d_B, l'idée est — avec les notations précédentes — de calculer $C\langle U\rangle$ et d'en déduire C à l'aide de la fonction `YtoP`. Or le calcul des composantes de $C\langle U\rangle$ est banal, à partir de celles de $A\langle U\rangle$ et $B\langle U\rangle$, puisque

$$\forall (k,n) \in \mathbb{N}^2 \quad C(u_{k,n}) = A(u_{k,n}) \cdot B(u_{k,n})\,.$$

Il faut bien sûr commencer par choisir n : la plus petite puissance de 2 strictement supérieure à $d_C = d_A + d_B$ est le bon choix ! Il n'y a ensuite plus qu'à calculer $A\langle U\rangle$ et $B\langle U\rangle$ et à conclure comme indiqué ci-dessus :

```
def mult(A,B):
    dA=len(A)-1;dB=len(B)-1;d=dA+dB
    n=1
    while n<=d: n*=2
    for i in range(n-len(A)): A.append(0)
    for i in range(n-len(B)): B.append(0)
    YA=PtoY(A);YB=PtoY(B)
    return YtoP([YA[i]*YB[i] for i in range(n)])[:d+1]
```

La tranche `[:d+1]` permet de ne conserver que les coefficients *a priori* significatifs (de 0 à d).

D'après les questions précédentes, le nombre de multiplications effectuées est un $O(n \log_2 n)$, mais

$$n/2 \le d_C < n \quad \text{donc} \quad d_C < n \le 2d_C \quad \text{et} \quad \log_2 d_C < \log_2 n \le 1 + \log_2 d_C\;;$$

il en résulte que $n \log_2 n$ est un $O(dc \log_2 dc)$ et, finalement :

$$\boxed{\text{Le nombre de multiplications nécessaires est un } O(d_C \log_2 d_C).}$$

Noter que, de même, comme l'annonçait le préambule, cet algorithme calcule le produit de deux polynômes de degré n au prix d'un nombre de multiplications en $O(n \log_2 n)$ (tout au moins des valeurs approchées des coefficients dudit produit).

"Un bon algorithme est comme un couteau tranchant — il fait exactement ce qu'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant."

(Cormen, Leiserson & Rivest)