Informatique tronc commun

Programmes usuels sur les listes et les dictionnaires

Exercices à compléter — version enseignant

Ce document est à traiter en travail personnel.

Il est essentiel de savoir écrire de courts programmes sur les listes et dictionnaires.

Voici quelques petits exercices (à travailler à votre rythme).

Les **tris**, qui sont aussi des programmes usuels, ne sont pas repris ici : voir le cours de première année ou celui de **M. Vandaele** : https://pvdmaths.fr/contenuWeb/itc.php, mot de passe : xxxxxxxxxx.

Les opérations autorisées sur les listes et les dictionnaires, avec leurs complexités, sont rappelées ci-dessous (n et m désignent les longueurs des listes `L` et `M`, ou le nombre de clés du dictionnaire `d`).

Listes.

Opération	Coût	Remarque
création vide `[]`	$O(1)$	
création `[x] * n`	$O(n)$	les n cases référencent le même `x`
compréhension `[f(i) for i in range(n)]`	$O(n)$	si chaque `f(i)` coûte $O(1)$
longueur `len(L)`	$O(1)$	la taille est stockée
accès `L[i]`, modification `L[i] = x`	$O(1)$	calcul d'adresse
ajout en fin `L.append(x)`	$O(1)$ amorti	tableau dynamique
retrait en fin `L.pop()`	$O(1)$	
tranche `L[i:j]`	$O(j-i)$	copie des éléments
concaténation `L + M`	$O(n+m)$	nouvelle liste
copie `L.copy()`	$O(n)$	copie superficielle
parcours `for x in L`	$O(n)$	hors coût du corps de boucle

Attention. Le test d'appartenance `x in L` sur une **liste** est **interdit** dans ces exercices : il coûte $O(n)$ (c'est un parcours caché), les énoncés l'interdisent souvent, et il ne figure pas dans l'annexe du programme. On écrit la boucle.

Dictionnaires. Les clés doivent être hachables (non modifiables) : entiers, chaînes, tuples d'éléments hachables… mais pas listes.

Opération	Coût	Remarque
création vide `{}`	$O(1)$	
création `{c1: v1, c2: v2}`	$O(n)$	n couples clé-valeur
accès `d[k]`	$O(1)$ en moyenne	erreur si `k` n'est pas une clé
ajout ou modification `d[k] = v`	$O(1)$ en moyenne	amorti (redimensionnement)
appartenance `k in d`	$O(1)$ en moyenne	c'est tout l'intérêt du dictionnaire
nombre de clés `len(d)`	$O(1)$	
copie `d.copy()`	$O(n)$	copie superficielle
parcours des clés `for k in d`, ou `for k in d.keys()`	$O(n)$	hors coût du corps de boucle
parcours des couples `for k, v in d.items()`	$O(n)$	hors coût du corps de boucle

> Les coûts « en moyenne » sont ceux du hachage (voir le cours) ; ils supposent que le calcul du haché d'une clé coûte $O(1)$, ce qui est faux pour une longue chaîne ou un long tuple (coût proportionnel à la longueur). Les chaînes de caractères et les tuples, non modifiables, se comportent comme des listes pour `len`, l'accès `s[i]`, les tranches, la concaténation et le parcours.

Le type `set`, les méthodes `sort`, `index`, `count`, `insert`, `pop(i)`, l'instruction `del` et les fonctions `sorted`, `min`, `max`, `sum` ne sont pas au programme : on ne les utilise pas ci-dessous, on écrit les boucles. Pour représenter un ensemble, on utilise un dictionnaire dont toutes les valeurs valent `True`.

1 Compter des occurrences

Question. Écrire une fonction `occurrences(L)` prenant en argument une liste `L` d'éléments hachables et renvoyant le dictionnaire associant à chaque élément de `L` son nombre d'occurrences, en un seul parcours de `L`. Quelle est sa complexité ?

```python
def occurrences(L):
    d = {}
    for x in L:
        if x in d:
            d[x] = d[x] + 1
        else:
            d[x] = 1
    return d
```

$O(n)$ en moyenne : n tours de boucle, chacun fait un test `x in d` et une écriture `d[x] = …` en $O(1)$ en moyenne. Si les valeurs sont des entiers de $[\![0, m-1]\!]$, une liste `h = [0] * m` suffit (histogramme, $O(n+m)$ au pire).

> *« il faut penser à traiter à part le cas où le dictionnaire ne comporte pas encore d'entrée pour la valeur à ajouter »* (X-ENS 2018)

2 Maximum et position du maximum

Question. Écrire une fonction `max_argmax(L)` prenant en argument une liste non vide `L` de nombres et renvoyant le couple `(m, i)` formé du maximum `m` de `L` et du **premier** indice `i` où il est atteint. Quelle est sa complexité ?

```python
def max_argmax(L):
    i_max = 0
    for i in range(1, len(L)):
        if L[i] > L[i_max]:          # > strict : on garde le premier indice
            i_max = i
    return L[i_max], i_max
```

$O(n)$: un parcours, avec un travail en $O(1)$ par élément. Les sujets précisent souvent la règle de départage en cas d'égalité (premier ou dernier indice, plus petite abscisse…) : c'est le choix entre `>` et `>=`.

Question. Écrire une fonction `deux_max(L)` prenant en argument une liste `L` de nombres, de longueur au moins 2, et renvoyant le couple `(m1, m2)` de ses deux plus grandes valeurs, avec `m1 >= m2`, comptées avec multiplicité (`[5, 5, 3]` donne `(5, 5)`), en un seul parcours de `L`. Quelle est sa complexité ?

```python
def deux_max(L):
    if L[0] >= L[1]:
        m1, m2 = L[0], L[1]
    else:
        m1, m2 = L[1], L[0]
    for i in range(2, len(L)):       # invariant : m1 >= m2
        if L[i] > m1:
            m1, m2 = L[i], m1
        elif L[i] > m2:
            m2 = L[i]
    return m1, m2
```

$O(n)$: un seul parcours, au plus deux comparaisons par élément.

Question. Écrire une fonction `point_le_plus_bas(P)` prenant en argument une liste non vide `P` de couples `(x, y)` de nombres (les points d'un nuage) et renvoyant le point de plus petite ordonnée et, s'il y en a plusieurs, celui de plus petite abscisse. Quelle est sa complexité ?

```
def point_le_plus_bas(P):
    meilleur = P[0]
    for (x, y) in P:
        if y < meilleur[1] or (y == meilleur[1] and x < meilleur[0]):
            meilleur = (x, y)
    return meilleur
```

$O(n)$: un parcours de `P`. On compare d'abord les ordonnées, et les abscisses seulement en cas d'égalité.

« Parmi les erreurs les plus courantes, nous pouvons noter : la non-prise en compte de la directive de plus faible abscisse, dans le cas où plusieurs points ont l'ordonnée minimale ; un test simultané sur abscisse et ordonnée : un tel test simultané ne fonctionne en général pas ; calcul de l'ordonnée maximale et non minimale » (X-ENS 2015)

Question. Écrire une fonction `plus_frequent(L)` prenant en argument une liste non vide `L` d'éléments hachables et renvoyant un couple `(x, k)` formé d'un élément `x` le plus fréquent de `L` et de son nombre d'occurrences `k`. On pourra utiliser `occurrences`. Quelle est sa complexité ?

```
def plus_frequent(L):
    d = occurrences(L)
    meilleur = L[0]
    for x in d:
        if d[x] > d[meilleur]:
            meilleur = x
    return meilleur, d[meilleur]
```

$O(n)$ en moyenne : l'appel à `occurrences` en $O(n)$, puis un parcours des clés (au plus n) avec des accès en $O(1)$ en moyenne.

3 Valeurs distinctes, doublons

Question. Écrire une fonction `distincts(L)` prenant en argument une liste `L` d'éléments hachables et renvoyant la liste des valeurs distinctes de `L`, dans l'ordre de première apparition. Quelle est sa complexité ? La comparer avec celle d'une version qui, pour chaque élément, parcourt la liste résultat pour savoir s'il y figure déjà.

```
def distincts(L):
    vus = {}                    # dictionnaire de True : un ensemble
    res = []
    for x in L:
        if x not in vus:
            vus[x] = True
            res.append(x)
    return res
```

$O(n)$ en moyenne : chaque tour fait un test et au plus une insertion en $O(1)$ en moyenne, et un `append` en $O(1)$ amorti. En parcourant la liste résultat à chaque tour, chaque recherche coûte $O(k)$, où k est le nombre de valeurs distinctes : $O(nk)$, donc $O(n^2)$ au pire.

> *« la meilleure approche n'était pas d'enlever des éléments de la table à l'aide de del ou de pop(i) […] (del et pop(i) de Python sont en O(len(L))), mais de reconstruire une table exempte de doublons »* (X-ENS 2018)

Question. Écrire une fonction `a_un_doublon(L)` prenant en argument une liste `L` d'éléments hachables et renvoyant `True` si une valeur apparaît au moins deux fois dans `L`, `False` sinon. Quelle est sa complexité ?

```
def a_un_doublon(L):
    vus = {}
    for x in L:
        if x in vus:
            return True
        vus[x] = True
    return False
```

$O(n)$ en moyenne : au plus n tours, chacun en $O(1)$ en moyenne, contre $O(n^2)$ pour la double boucle qui compare toutes les paires.

Question. Écrire une fonction `paire_de_somme(L, s)` prenant en argument une liste `L` d'entiers et un entier `s`, et renvoyant `True` s'il existe deux indices $i < j$ tels que `L[i] + L[j] == s`, `False` sinon. Quelle est sa complexité ? La comparer avec celle d'une double boucle.

```
def paire_de_somme(L, s):
    vus = {}                    # valeurs déjà parcourues
    for x in L:
        if s - x in vus:        # x = L[j] complète une valeur déjà vue
            return True
        vus[x] = True
    return False
```

$O(n)$ en moyenne : un parcours, un test et une insertion en $O(1)$ en moyenne par élément. La double boucle sur les paires $i < j$ est en $O(n^2)$. On teste avant d'insérer, pour ne pas associer `x` à lui-même (cas `s == 2 * x`).

Question. Une marche sur la grille $\mathbb{Z}^2$ est donnée par la liste `chemin` des positions visitées successivement, chaque position étant une liste `[x, y]` d'entiers (comme dans CCMP 2021). Écrire une fonction `est_auto_evitante(chemin)` prenant en argument une telle liste et renvoyant `True` si la marche ne repasse jamais par une même position, `False` sinon. Justifier que sa complexité est $O(n)$ en moyenne, où n est la longueur de `chemin`.

```
def est_auto_evitante(chemin):
    vus = {}
    for p in chemin:
        cle = (p[0], p[1])          # tuple : hachable
        if cle in vus:
            return False
        vus[cle] = True
    return True
```

n tours de boucle, chacun en $O(1)$ en moyenne (construction d'un couple, test, insertion) : $O(n)$ en moyenne. En parcourant une liste des positions déjà vues à chaque tour, on obtiendrait $O(n^2)$; le sujet impose une version par tri, en $O(n \log n)$.

4 Adressage direct : tableau de booléens

Quand les valeurs sont des entiers d'un petit intervalle $[\![0, m-1]\!]$, une liste de m booléens indexée par ces entiers remplace avantageusement le dictionnaire de `True` : coût $O(1)$ au pire par opération (et non plus en moyenne), mais $O(m)$ pour la création.

Question. Écrire une fonction `plus_petit_absent(L)` prenant en argument une liste `L` de n entiers naturels (quelconques, éventuellement très grands) et renvoyant le plus petit entier naturel qui n'apparaît pas dans `L`. On utilisera une liste de booléens de longueur $n+1$. Quelle est sa complexité ?

```python
def plus_petit_absent(L):
    n = len(L)
    present = [False] * (n + 1)       # present[k] : k est dans L
    for x in L:
        if x <= n:                    # les valeurs > n sont inutiles
            present[x] = True
    k = 0
    while present[k]:
        k = k + 1
    return k
```

Les $n+1$ entiers $0, \ldots, n$ ne peuvent pas tous figurer parmi les n éléments de `L` : le résultat est au plus n, et la boucle `while` s'arrête avant de sortir de la liste. $O(n)$ au pire : création en $O(n)$, parcours de `L`, puis au plus $n+1$ tours. C'est par exemple la « plus petite couleur non utilisée par les voisins » dans une coloration gloutonne de graphe.

5 Suites récurrentes : détecter un cycle

Soit f une fonction d'un ensemble fini E dans lui-même et $u_0 \in E$. La suite définie par $u_{n+1} = f(u_n)$ prend un nombre fini de valeurs, donc finit par boucler : il existe un plus petit indice μ tel que u_μ réapparaisse plus loin, et un plus petit $\lambda \geq 1$ tel que $u_{\mu+\lambda} = u_\mu$; la suite est alors périodique de période λ à partir du rang μ. Par exemple, pour un générateur pseudo-aléatoire $u_{n+1} = (a u_n + c) \,\%\, m$, une période λ courte est rédhibitoire.

Question. Écrire une fonction `cycle(f, u0)` prenant en argument une fonction `f` (de E dans E, les éléments de E étant hachables) et un élément `u0` de E, et renvoyant le couple `(mu, lam)` défini ci-dessus. Quelle est sa complexité, en supposant que chaque appel à `f` coûte $O(1)$?

```python
def cycle(f, u0):
    indice = {}                   # terme déjà vu -> son indice
    u = u0
    n = 0
    while u not in indice:
        indice[u] = n
        u = f(u)
        n = n + 1
    return indice[u], n - indice[u]     # u = u_n = u_mu
```

La boucle s'arrête au premier n tel que u_n a déjà été vu, c'est-à-dire $n = \mu + \lambda$, et $u_n = u_\mu$. On fait donc $\mu + \lambda$ tours, chacun en $O(1)$ en moyenne : $O(\mu + \lambda)$ en temps (en moyenne) et en mémoire. Ici le dictionnaire associe à chaque terme son indice : un simple dictionnaire de `True` dirait qu'il y a un cycle, pas où il commence.

6 Intersection, listes disjointes

Question. Écrire une fonction `intersection(L1, L2)` prenant en argument deux listes `L1` et `L2` (de longueurs n_1 et n_2) d'éléments hachables et renvoyant la liste des éléments distincts de `L1` qui sont aussi dans `L2`. Justifier que sa complexité est $O(n_1 + n_2)$ en moyenne.

```python
def intersection(L1, L2):
    dans_L2 = {}
    for x in L2:
        dans_L2[x] = True
    deja = {}
    res = []
    for x in L1:
        if x in dans_L2 and x not in deja:
            deja[x] = True
            res.append(x)
    return res
```

Un parcours de `L2` puis un parcours de `L1`, avec des opérations en $O(1)$ en moyenne à chaque tour : $O(n_1 + n_2)$ en moyenne (contre $O(n_1 n_2)$ si l'on parcourt `L2` pour chaque élément de `L1`).

Le même schéma (on range l'une des listes dans un dictionnaire, puis on parcourt l'autre) donne la différence de deux listes, ou le test « les deux listes n'ont aucun élément commun » (X-ENS 2023).

> *« Certains candidats construisent séparément le dictionnaire des caractères contenus dans texte1 et celui des caractères contenus dans texte2. C'est inutile. »* (X-ENS 2023)

7 Index et regroupement

Question. Écrire une fonction `positions(L)` prenant en argument une liste `L` d'éléments hachables et renvoyant le dictionnaire associant à chaque valeur de `L` la liste croissante des indices où elle apparaît. Quelle est sa complexité ?

```python
def positions(L):
    d = {}
    for i in range(len(L)):
        if L[i] in d:
            d[L[i]].append(i)
        else:
            d[L[i]] = [i]
    return d
```

$O(n)$ en moyenne : n tours, chacun avec un test, un accès et un `append` en $O(1)$ (en moyenne ou amorti). C'est un dictionnaire de listes : chaque valeur est une **nouvelle** liste `[i]` (et non une liste partagée).

> *« il était plus judicieux d'itérer sur les indices que sur les éléments »* (X-ENS 2018)

Question. Écrire une fonction `inverse(d)` prenant en argument un dictionnaire `d` dont les valeurs sont hachables et renvoyant le dictionnaire associant à chaque valeur `v` de `d` la liste des clés `k` telles que `d[k] == v`. Quelle est sa complexité ?

```
def inverse(d):
    inv = {}
    for k in d:
        v = d[k]
        if v in inv:
            inv[v].append(k)
        else:
            inv[v] = [k]
    return inv
```

$O(n)$ en moyenne, où n est le nombre de clés de `d` : un parcours des clés, avec des opérations en $O(1)$ en moyenne.

Question. Deux mots sont **anagrammes** s'ils sont formés des mêmes lettres avec les mêmes multiplicités (« chien », « niche », « chine »). Écrire une fonction `anagrammes(mots)` prenant en argument une liste `mots` de chaînes de lettres minuscules non accentuées et renvoyant la liste des groupes de mots anagrammes les uns des autres (chaque groupe étant une liste de mots ; un mot sans autre anagramme forme un groupe à lui seul, et un mot répété figure autant de fois dans son groupe). On associera à chaque mot une clé : la chaîne de ses lettres rangées dans l'ordre alphabétique, construite sans trier, en comptant ses lettres dans une liste de 26 entiers. Quelle est sa complexité ?

```
def anagrammes(mots):
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    rang = {}                          # lettre -> son rang (0 à 25)
    for i in range(26):
        rang[alphabet[i]] = i
    groupes = {}                       # clé -> liste de mots
    for m in mots:
        compte = [0] * 26
        for c in m:
            compte[rang[c]] = compte[rang[c]] + 1
        cle = ""                       # lettres de m dans l'ordre alphabétique
        for i in range(26):
            for j in range(compte[i]):
                cle = cle + alphabet[i]
        if cle in groupes:
            groupes[cle].append(m)
        else:
            groupes[cle] = [m]
    res = []
    for cle in groupes:
        res.append(groupes[cle])
    return res
```

Deux mots sont anagrammes si et seulement s'ils ont la même clé (une liste de 26 entiers ne conviendrait pas : elle n'est pas hachable). Pour un mot de longueur ℓ : 26 opérations, donc $O(1)$, pour créer `compte`, $O(\ell)$ pour compter, $O(\ell^2 + 26)$ pour construire la clé (chaque `cle = cle + …` recopie `cle`, de longueur au plus ℓ), $O(\ell)$ pour la hacher, puis $O(1)$ en moyenne pour le test et l'insertion : $O(\ell^2 + 26)$ en moyenne

par mot. Pour k mots de longueur totale N et de longueur au plus $\ell_{\max}$: $O(N\ell_{\max} + 26k)$ en moyenne, linéaire en $N + k$ pour des mots courts.

Un tel index permet aussi de faire une **jointure** (au sens des bases de données) sans comparer toutes les paires de lignes : c'est l'objet de la partie III du sujet X-ENS 2018.

Question. Deux tables sont données par des listes de tuples `T1` (de longueur n_1) et `T2` (de longueur n_2). Écrire une fonction `jointure(T1, T2, a, b)` prenant en argument ces deux listes et deux indices de colonnes `a` et `b`, et renvoyant la liste des couples `(l1, l2)` de lignes de `T1` et de `T2` telles que `l1[a] == l2[b]`. Quelle est sa complexité ? La comparer avec celle d'une double boucle.

```
def jointure(T1, T2, a, b):
    index2 = {}                          # valeur de la colonne b -> lignes de T2
    for l2 in T2:
        if l2[b] in index2:
            index2[l2[b]].append(l2)
        else:
            index2[l2[b]] = [l2]
    res = []
    for l1 in T1:
        if l1[a] in index2:
            for l2 in index2[l1[a]]:
                res.append((l1, l2))
    return res
```

Avec l'index : $O(n_1 + n_2 + r)$ en moyenne, où r est le nombre de couples du résultat : on ne parcourt que les lignes de `T2` qui ont la bonne valeur. Double boucle : $O(n_1 n_2)$.

« Il était intéressant de noter que la longueur des listes renvoyées par le dictionnaire est en général bien plus petite que la longueur de la table, apportant donc un gain appréciable. » (X-ENS 2018)

8 Graphes : de la liste d'arêtes au dictionnaire d'adjacence

Un graphe non orienté, simple et sans boucle, est donné par la liste `sommets` de ses n sommets (hachables) et la liste `aretes` de ses a arêtes (couples de sommets).

Question. Écrire une fonction `adjacence(sommets, aretes)` prenant en argument ces deux listes et renvoyant le dictionnaire associant à chaque sommet la liste de ses voisins (le graphe étant simple et sans boucle, chaque arête `(s, t)` vérifie `s != t` et n'apparaît qu'une fois, dans un seul sens). Quelle est sa complexité en fonction de n et a ?

```
def adjacence(sommets, aretes):
    g = {}
    for s in sommets:
        g[s] = []
    for (s, t) in aretes:
        g[s].append(t)
        g[t].append(s)              # graphe non orienté
    return g
```

$O(n + a)$ en moyenne : un parcours des sommets, puis un des arêtes, avec des opérations en $O(1)$ (en moyenne ou amorti).

Question. Écrire une fonction `degres(g)` prenant en argument le dictionnaire d'adjacence `g` d'un graphe simple, sans boucle, et renvoyant le dictionnaire associant à chaque sommet son degré. Quelle est sa complexité ?

```python
def degres(g):
    deg = {}
    for s in g:
        deg[s] = len(g[s])
    return deg
```

$O(n)$ en moyenne : `len` est en $O(1)$.

Question. Écrire une fonction `voisins(s, sommets, aretes)` prenant en argument un sommet `s` et les listes `sommets` et `aretes` d'un graphe simple, sans boucle, et renvoyant la liste des voisins de `s`, sans construire le dictionnaire d'adjacence. Un élève parcourt la liste `sommets` et, pour chaque sommet `t`, parcourt la liste `aretes` pour savoir si `(s, t)` ou `(t, s)` en fait partie : quelle est la complexité de sa méthode ? Et celle de la vôtre ?

```python
def voisins(s, sommets, aretes):
    res = []
    for (u, v) in aretes:
        if u == s:
            res.append(v)
        elif v == s:
            res.append(u)
    return res
```

Chez l'élève, pour chacun des n sommets, un parcours des a arêtes : $O(na)$. Ici, un seul parcours des arêtes : $O(a)$ (la liste `sommets` ne sert pas). Et si l'on doit répondre pour beaucoup de sommets, on construit une fois pour toutes le dictionnaire d'adjacence, en $O(n+a)$.

« la première consiste à faire une boucle sur les n individus, et appeler la fonction précédente sontAmis() : l'inconvénient de cette approche est qu'elle est de complexité O(n × m) […] la seconde consiste à faire une boucle sur les m liens […] : cette approche est effectivement de complexité O(m), comme demandé. » (X-ENS 2016)

9 Agréger par catégorie

Question. Écrire une fonction `moyenne_par_categorie(couples)` prenant en argument une liste `couples` de couples `(c, v)`, formés d'une catégorie `c` (hachable) et d'une valeur numérique `v`, et renvoyant le dictionnaire associant à chaque catégorie la moyenne des valeurs correspondantes (c'est l'équivalent d'un `GROUP BY` avec `AVG` en SQL). Quelle est sa complexité ?

```
def moyenne_par_categorie(couples):
    somme, effectif = {}, {}
    for (c, v) in couples:
        if c in somme:
            somme[c] = somme[c] + v
            effectif[c] = effectif[c] + 1
        else:
            somme[c] = v
            effectif[c] = 1
    moyenne = {}
    for c in somme:
        moyenne[c] = somme[c] / effectif[c]
    return moyenne
```

$O(n)$ en moyenne : un parcours des n couples, puis un parcours des catégories (au plus n), avec des opérations en $O(1)$ en moyenne.

10 Polynômes creux

Un polynôme creux (avec peu de coefficients non nuls, comme $X^{1000} + 3X^2 - 1$) se représente par le dictionnaire degré → coefficient non nul : `{1000: 1, 2: 3, 0: -1}`. On note $|p|$ le nombre de clés de `p`.

Question. Écrire une fonction `somme_poly(p, q)` prenant en argument deux polynômes creux `p` et `q` et renvoyant le polynôme creux $p + q$, sans modifier `p` ni `q`, et sans garder de coefficient nul. Quelle est sa complexité ?

```
def somme_poly(p, q):
    s = {}
    for k in p:
        if k in q:
            c = p[k] + q[k]
        else:
            c = p[k]
        if c != 0:
            s[k] = c
    for k in q:
        if k not in p:
            s[k] = q[k]
    return s
```

$O(|p| + |q|)$ en moyenne, indépendamment des degrés (une liste de coefficients coûterait $O(\max(\deg p, \deg q))$).

Question. Écrire une fonction `produit_poly(p, q)` prenant en argument deux polynômes creux `p` et `q` et renvoyant le polynôme creux pq, sans coefficient nul. Quelle est sa complexité ?

```
def produit_poly(p, q):
    r = {}
    for i in p:
        for j in q:
            if i + j in r:
                r[i + j] = r[i + j] + p[i] * q[j]
            else:
                r[i + j] = p[i] * q[j]
    res = {}                    # on retire les coefficients nuls
    for k in r:
        if r[k] != 0:
            res[k] = r[k]
    return res
```

$O(|p|\,|q|)$ en moyenne : $|p|\,|q|$ tours en $O(1)$ en moyenne, puis un parcours de `r`, qui a au plus $|p|\,|q|$ clés. Des termes peuvent se compenser, comme dans $(X+1)(X-1) = X^2 - 1$: d'où la seconde boucle. Avec des listes de coefficients, le produit naïf coûte $O((\deg p + 1)(\deg q + 1))$.

11 Décoder avec une table de correspondance

Question. Un code préfixe est donné par un dictionnaire `code` qui associe à chaque caractère une chaîne de `'0'` et de `'1'` (par exemple `{'a': '0', 'b': '10', 'c': '11'}`), aucun mot de code n'étant le début d'un autre. Écrire une fonction `decoder(bits, code)` prenant en argument une chaîne de bits `bits`, supposée valide (c'est une suite de mots de code), et un tel dictionnaire `code`, et renvoyant la chaîne décodée. Quelle est sa complexité ?

```
def decoder(bits, code):
    table = {}
    for c in code:
        table[code[c]] = c        # dictionnaire inverse
    res = ""
    tampon = ""
    for b in bits:
        tampon = tampon + b
        if tampon in table:
            res = res + table[tampon]
            tampon = ""
    return res
```

Le code préfixe garantit qu'on peut décoder dès qu'un mot est reconnu. Pour N bits et des mots de code de longueur au plus ℓ : chaque tour construit `tampon` et le hache en $O(\ell)$, puis le test `tampon in table` coûte $O(1)$ en moyenne ; d'où $O(N\ell)$ en moyenne, plus l'inversion de `code`. En toute rigueur, `res = res + …` recopie `res` à chaque fois (coût quadratique au pire, comme `L = L + [x]`) ; si une liste de caractères suffit, `res = []` et `res.append(table[tampon])` évitent ce problème.