ALGORITHMES CLASSIQUES

Partie A - Écritures binaires des entiers et codages des parties

1) Écritures binaires des entiers naturels

a) (♣) Écrire une fonction `binaire2(n,p)` qui étant donnés deux entiers naturels n et p vérifiant $0 \leq n < 2^p$ renvoie la p-liste $[x_0, ..., x_{p-1}]$ telle que $n = \sum_{i=0}^{p-1} x_i 2^i$, avec $\forall i,\ x_i \in \{0, 1\}$.

b) Écrire une fonction `binaire(n)` qui étant donné un entier naturel n renvoie la liste $[x_0, ..., x_{p-1}]$ telle que $n = \sum_{i=0}^{p-1} x_i 2^i$, avec $\forall i,\ x_i \in \{0, 1\}$ et $x_{p-1} = 1$ lorsque n n'est pas nul, et qui renvoie la liste vide [ ] si $n = 0$.

2) Codage des parties de $E = \{0, 1, ..., p-1\}$

On peut coder une partie A de E par la liste (croissante) de ses éléments. On peut aussi coder une partie par sa fonction caractéristique, c'est-à-dire la liste $[x_0, ..., x_{p-1}]$ définie par $x_i = 1$ si $i \in A$, et 0 sinon.

On peut alors associer à A l'entier naturel $n(A) = \sum_{i=0}^{p-1} x_i 2^i$. On obtient ainsi une bijection de $\mathcal{P}(E)$ sur $[\![0, 2^p - 1]\!]$.

a) (♣) Écrire une fonction `partie(n,p)` qui étant donné un entier $n < 2^p$ renvoie la partie A de E telle que $n = n(A)$.

Par exemple, `partie(13,3)` renvoie $[0, 2, 3]$ car $5 = 1.2^0 + 0.2^1 + 1.2^2 + 1.2^3$.

b) Écrire une fonction `enumere(p)` qui renvoie la liste des parties de E (sous forme d'une liste de listes).

Par exemple, `enumere(2)` renvoie `[[],[0],[0,1],[1]]`.

3) Algorithme d'exponentiation rapide

Étant données deux matrices A et $B \in \mathcal{M}_p(\mathbb{R})$, la fonction `dot(A,B)` renvoie la matrice AB.

`zeros((p,p))` renvoie la matrice carrée nulle d'ordre p, et `eye(p)` renvoie la matrice I_p.

a) (♣) Écrire une fonction `power(A,n)` qui étant donnés une matrice carrée A et un entier n renvoie A^n selon l'algorithme d'exponentiation rapide : il s'agit de l'algorithme récursif basé sur le principe suivant : $A^n = (A^m)^2 A^r$ pour $n = 2m + r$, avec $r \in \{0, 1\}$.

Autrement dit, on effectue $A^n = (A^m)^2$ si $n = 2m$ est pair, et $A^n = (A^m)^2 A$ si $n = 2m + 1$ est impair.

La complexité est en $O(\log n)$ opérations.

b) (*plus difficile*) Ecrire une variante itérative en commençant par calculer la décomposition de n en base 2 : en effet, on obtient ainsi la liste L des restes r qui permettent d'effectuer ensuite le choix des opérations (en dépilant la pile obtenue L).

Remarque : Il s'agit en fin de compte de "dérécursifier" la fonction récursive définie au a).

B. Ordre lexicographique

4) a) (♣) On définit l'ordre lexicographique sur $\mathbb{N}^p$ par : $(x_0, x_1, ..., x_{p-1}) \preceq (y_0, y_1, ..., y_{p-1})$ ssi :

- ou bien $\exists\, k \in \{0, 1, ..., p-1\}$, $(\forall j \in [\![0, k-1]\!],\ x_j = y_j)$ et $x_k < y_k$
- ou bien $(x_0, x_1, ..., x_{p-1}) = (y_0, y_1, ..., y_{p-1})$.

Définir une fonction `ordre(X,Y)` qui étant donnés deux p-listes entières renvoie `True` ssi $X \preceq Y$.

b) (♣) On définit plus généralement l'ordre lexicographique sur les uplets d'entiers (ou autres) par :

$$(x_0, x_1, ..., x_{p-1}) \preceq (y_0, y_1, ..., y_{q-1}) \text{ ssi } \begin{cases} p = 0 \\ \text{ou } x_0 < y_0 \\ \text{ou } x_0 = y_0 \text{ et } (x_1, ..., x_{p-1}) \preceq (y_1, ..., y_{q-1}) \end{cases}$$

Définir une fonction `ordre(X,Y)` qui renvoie `True` ssi $X \preceq Y$.

Remarque culturelle : Exemples d'utilisations : Comparaison des chaînes de caractères (à partir de l'ordre alphabétique `'a'<'b'<'c'<...` ; comparaison de deux entiers naturels (en machine : on compare en fait leurs écritures en base 2 pour l'ordre lexicographique défini par $0 < 1$).

Partie C - Listes et tableaux

5) Recherche d'un élément

a) Écrire une fonction `is(x,L)` qui renvoie `True` si x apparaît dans `L`, et `False` sinon.

Mieux : Renvoyer le plus petit indice x tel que $L[i] = x$, et on renvoie n (longueur de L) sinon.

b) (♣) Écrire une fonction `minimum(L)` qui renvoie la valeur maximale de L (liste non vide à valeurs entières).

(♣) *Mieux* : Définir `indiceMin(L)` qui renvoie un indice i tel que $L[i] = \min L$.
Si le maximum apparaît plusieus fois, la fonction renvoie le plus petit indice i.

c) Écrire une procédure `premier(L)` qui par *un* échange, met en première position le plus petit élément.

6) Liste miroir

(♣) Écrire une fonction `inverse(L)`qui en temps linéaire inverse l'ordre des éléments

7) Cas des listes triées (*important*)

Les listes `L` considérées ici sont supposées triées par ordre croissant.
Cette structure permet notamment de coder les ensembles (on trie les éléments par ordre strictement croissant).

a) (♣) Écrire une fonction `is(x,L)` qui en $O(\log n)$ opérations teste si x apparaît dans E..

Indication : On procèdera **par dichotomie**.
On pourra utiliser une fonction auxiliaire `aux(x,L,i,j)` qui teste si $x \in \{L[k], i \leq k < j\}$.
Deux versions possibles : itérative et récursive.

b) (♣) Écrire une fonction `fusion(L,M)` qui en $O(n+m)$ opérations renvoie la liste $L \cup M$.

On pourra proposer deux variantes, une avec une boucle `while`, l'autre avec une boucle `for`.

Remarque : On peut de même calculer en temps linéaire $L \cap M$ et $L \Delta M = (L \cup M) \setminus (L \cap M)$.

8) Suppression des doublons

a) Écrire une fonction `ensemble(L)` qui étant donnée une liste L renvoie la liste obtenue en supprimant les doublons, en utilisant le principe suivant : on sélectionne pour chaque valeur apparaissant dans L uniquement sa première occurrence.

Par exemple, si $L = [5, 2, 1, 2, 5, 4, 2]$, `ensemble(L)` renvoie $[5, 2, 1, 4]$.

Préciser la complexité de la fonction proposée.

b) (♣) On suppose que L est une liste triée d'entiers naturels. Écrire une fonction `reduit(L)` qui supprime les doublons. Par exemple, `reduit([1,1,2,2,3]) renvoie [1,2,3]`.

Remarque : On peut se ramener à cette situation en $O(n \log n)$ opérations si on connâit un ordre sur l'ensemble des éléments de L.

c) (♣) En utilisant un **dictionnaire**, écrire une nouvelle fonction `ensemble(L)` de complexité linéaire amortie.

9) Files de priorité (implémentation naïve)

On considère un ensemble d'éléments E où chaque élément x est affectée d'une valeur $v(x) \in \mathbb{N}$.
Une file de priorité est une structure dynamique munie des opérations suivantes :

- récupérer l'élément x tel que $v(x)$ est maximum en temps $O(1)$
- ajouter un élément
- supprimer un élément
- modifier la priorité $v(x)$ d'un élément x.

La structure naïve choisie ici consiste à stocker les couples $(x, v(x))$ dans une liste (pile) L où les éléments sont classés **par ordre croissant selon la valeur de** $v(x)$.

Par exemple, `L = [("b",2),("d",4),("a",10)]`

a) On suppose connue la fonction `element(x,L)` qui renvoie la position de l'élément x si L contient un élément $e = (x, v)$, et renvoie -1 sinon.

En procédant par dichotomie, on peut obtenir une complexité en $O(\log n)$, où n est la longueur de L.

Écrire les fonctions (ou procédures) suivantes :

- `elementMax(L)` qui renvoie l'élément x maximisant $k(x)$ dans L
- `supprime(L)` qui supprime dans L l'élément maximisant $k(x)$ et renvoie x
- `ajoute(e,L)` qui étant donné un couple $e = (x, k)$ ajoute cet élément à L (sauf si il y est déjà)
- `modif(e,L)` qui étant donné un couple $e = (x, h)$ modifie la valeur de $k(x)$ en prenant h comme nouvelle valeur
- `creer(E)` qui étant donnée une liste de n couples $e = (x, k)$ où les valeurs de x sont distinctes, renvoie une file dont les éléments sont les éléments de E.

b) (♣) Préciser les complexités des fonctions précédentes.

10) Partitions par pivot

On se donne un tableau A de longueur $n \geq 1$ et un élément x (appelé pivot) appartenant à A, par exemple $x = A[0]$.

(♣) Écrire une procédure-fonction `pivot(A)` qui étant donné un tableau A :
- modifie les positions des éléments de A de sorte à placer les éléments de valeur $\leq x$ avant x et ceux $> x$ après x
- renvoie l'indice i de A tel que $A[i] = x$ (l'indice i désignant donc la position finale du pivot).

Par exemple, si $A = [\mathbf{5}, 4, 3, 1, 6, 2, 5, 4, 7]$, alors `pivot(A)` transforme A en $[4, 3, 1, 2, 5, 4, \mathbf{5}, 6, 7,]$ et renvoie 6, la position du pivot $x = 5$ dans le tableai final.

On utilisera un algorithme de complexité linéaire en $O(n)$. On proposera deux (ou trois) méthodes :

- on utilise un tableau auxiliaire B : on lit les éléments de A et on les place à gauche ou à droite dans un nouveau tableau B selon qu'ils sont inférieurs ou supérieurs au pivot.

- implémentation sur place (= sans tableau auxiliaire) : on procède par des échanges en utilisant deux curseurs i et j qui partent des deux extrémités du tableau. L'algorithme s'arrête lorsque $i \geq j$.

(*variante* : on parcourt le tableau par un indice i croissant : chaque fois qu'on rencontre un élément strictement supérieur au pivot, on l'échange avec un élément situé à la fin du tableau, où la fin du tableau est gérée par un autre indice j, qui diminue de 1 à chaque échange. L'algorithme s'arrête lorsque $i \geq j$).

Remarque culturelle : La complexité est $O(n \log n)$ en moyenne et $O(n^2)$ dans le cas le pire, et elle peut être améliorée dans les pires cas par un choix judicieux des pivots.

11) Algorithmes de sélections

Il s'agit de déterminer le p-ième plus petit élément dans un tableau d'entiers de longueur n.
On veut éviter de trier tout le tableau car on souhaite une complexité moyenne en $O(n)$.

a) On considère une méthode "naïve" consistant à chercher tous les p plus petits éléments :
Pour tout $i \in \{0, 1, 2, ..., p-1\}$, on met successivement en position i le $(i+1)$-ième plus petit élément.
Écrire une fonction `select(A,p)` de complexité $O(np)$ renvoyant le p-ième plus petit élément.

b) (♣) On se propose d'utiliser les partitions par pivot : écrire une fonction auxiliaire récursive `pivot_aux(A,p,i,j)` qui étant donné $p \leq j - i$, renvoie le p-ième plus petit élément du sous-tableau $A[i : j]$.

Le principe est le suivant : On choisit comme pivot $x = A[i]$, on applique une partition au sous-tableau relativement à ce pivot. Enfin, en notant k la position finale du pivot x : si $k - i + 1 = p$, on renvoie le pivot x ; si $k - i + 1 > p$, on se ramène à déterminer le p-ième plus petit élément du sous-tableau $A[i : k]$; si $k - i + 1 < p$, on se ramène à déterminer le $(p - (k - i + 1))$-ième plus petit élément du sous-tableau $A[k + 1 : j]$.

On suppose connue une procédure-fonction `pivot(A,i,j)` qui étant donnés $0 \leq i < j \leq n$, modifie le sous-tableau $A[i : j]$ en plaçant le pivot $A[i]$ en bonne position, les éléments inférieirs avant et les suppérieurs après, et renvoie la nouvelle position du pivot.

Remarque culturelle : La complexité est donnée par $c(n) = \max(c(k), c(n-1-k)) + O(n)$.
Si le pivot se trouvait être le médian, on aurait $c(n) = c(\lfloor \frac{1}{2} n \rfloor) + O(n)$.
D'où une complexité en $O(n + \frac{1}{2}n + \frac{1}{4}n + ...) = O(n)$.
En moyenne, la valeur moyenne de $\max(k, n-k)$ est $\frac{3}{4}n$.
On peut donc estimer la complexité moyenne à $\frac{3}{4}n + (\frac{3}{4})^2 n + ... = O(n)$.
Mais dans le pire cas, la complexité vérifie $c(n) = c(n-1) + O(1)$, donc $c(n) = O(n^2)$.

Une variante de l'algorithme consistant à choisir à chaque étape judicieusement le pivot permet d'obtenir une complexité en $O(n)$ même dans le pire cas.