Algorithmique IV (UE-41) - TD 9.

TD 9. Tri par dénombrement, par répartition et tri lexicographique [1]

EXERCICE 1. **(1)** Calculez la dispersion $\Delta(L)$ de chacune des listes suivantes :

(a) $L := [1, 1, 2, 3, 1, 8, 1, 2, 1, 1]$,

(b) $L := [3, 1, 1, 2]$.

(2) Trouvez un encadrement des valeurs de la fonction de dispersion. La borne inférieure est-elle atteinte ? Même question avec la borne supérieure. Justifiez.

Solution. On rappelle la définition de la dispersion d'une liste L :

$$\Delta(L) := 1 - \frac{\#\{x_i \mid x_i \in L\}}{\max L - \min L + 1} \tag{1}$$

(1) (a) On a $\min L = 1$ et $\max L = 8$, l'étendue de L est donc égale à $8 - 1 + 1 = 8$. Le nombre de valeurs distinctes de L est égal à 4, on en déduit que $\Delta(L) = 1 - \frac{4}{8} = \frac{1}{2}$.
(b) On a $\min L = 1$ et $\max L = 3$, l'étendue de L est donc égale à $3-1+1 = 3$. Le nombre de valeurs distinctes de L_1 est égal à 3, on en déduit que $\Delta(L) = 1 - \frac{3}{3} = 0$.

(2) Si a et b sont deux entiers tels que $0 < a \leqslant b$ on a $0 < \frac{a}{b} \leqslant 1$ et par conséquent $0 \leqslant 1 - \frac{a}{b} < 1$, on en déduit que

$$0 \leqslant \Delta(L) < 1.$$

L'exemple (b) permet d'atteindre la borne inférieure et prouve donc que la dispersion minimale est nulle. Pour une liste de longueur n, l'étendue E peut être arbitrairement grande avec 2 valeurs distinctes dans la liste, minimisant ainsi le numérateur de la fraction, autrement dit $\Delta(L) = 1 - \frac{2}{E}$ qui tend vers 1 si $E \to +\infty$. Au sens mathématique, il n'existe donc pas de dispersion *maximale*, puisque la valeur 1 ne peut pas être atteinte, mais 1 est la borne supérieure pour la dispersion.

1. Version du 4 avril 2025, 19 : 23

EXERCICE 2. Démontrez que l'ensemble des entiers relatifs $\mathbb{Z}$ est équipotent à l'ensemble des entiers naturels $\mathbb{N}$. Indication : explicitez l'application $\iota : \mathbb{Z} \to \mathbb{N}$ qui associe les nombres négatifs aux entiers impairs et les nombres positifs aux entiers pairs et montrez qu'elle est bijective.

Solution. On considère l'application $\iota : \mathbb{Z} \to \mathbb{N}$ définie par :

$$\iota(n) = \begin{cases} -(2n+1) & \text{si } n < 0 \\ 2n & \text{si } n \geqslant 0 \end{cases} \tag{2}$$

(1) Montrons qu'elle est injective, i.e.

$$\forall (n, n') \in \mathbb{Z}^2 \quad \iota(n) = \iota(n') \Rightarrow n = n'. \tag{3}$$

Soit $(n, n') \in \mathbb{Z}^2$ tel que $\iota(n) = \iota(n')$. Si n et n' sont tous deux strictement négatifs, on a $-(2n+1) = -(2n'+1)$ et par conséquent $n = n'$, s'ils sont positifs on a $2n = 2n'$ et encore $n = n'$. Si on suppose qu'ils sont de signes opposés, alors $-2(n+1) = 2n'$ et donc $-2(n - n') = 1$, ce qui est contradictoire puisque n et n' sont des entiers relatifs. Autrement dit, si n et n' n'ont pas le même signe alors $\iota(n) = \iota(n')$ est faux et par conséquent l'implication (3) est vraie.

(2) Montrons qu'elle est surjective. Considérons un entier naturel m. S'il est pair, disons $m = 2k$, alors $k \geqslant 0$ et $\iota(k) = m$, il admet donc un antécédent k. S'il est impair disons $m = 2k+1$ avec $k \geqslant 0$ alors $-(k+1) < 0$ et $\iota(-(k+1)) = m$, il admet là encore un antécédent ce qui prouve que ι est surjective. Finalement ι est bijective, par conséquent $\mathbb{N} \approx \mathbb{Z}$.

NB. Une telle application permet indirectement d'indexer une liste L avec des index négatifs en composant L avec l'application $\iota : L[\iota(n)]$ où $n \in \mathbb{Z}$. On peut également indexer des structures énumérées à k dimensions avec une structure énumérée à une seule dimension. Par exemple en dimension 2 avec $\mathbb{N} \times \mathbb{N}$ en rangeant le couple (i, j) à la i-ème ligne et la j-ème colonne d'une table et en numérotant les cases de cette table suivant les diagonales $D_0 := \{(0, 0)\}$, $D_1 := \{(1, 0), (0, 1)\}$, $D_2 := \{(2, 0), (1, 1), (0, 2)\}$, etc.

EXERCICE 3. L'algorithme de comparaison lexicographique est rappelé en Algo. 1.

(1) Faites une preuve d'arrêt.

(2) Démontrez qu'à la sortie de la boucle, on a

$$\forall j \in [\![1, i-1]\!] \quad u_j = v_j.$$

```
ALGORITHME COMPARAISON-LEXICO(u,v):{-1,0,+1}
DONNEES
 · u,v: mots sur un alphabet
VARIABLES
 · i: entier
DEBUT
 · i ← 1
 · TQ ((i <= |u|) ET (i <= |v|) ET (u[i] = v[i])) FAIRE
 ·  · i ← i + 1
 · FTQ
(0) · SI ((i > |u|) ET (i > |v|)) ALORS
 ·  · RENVOYER 0
(-) · SINON SI ((i > |u|) OU ((i <= |v|) ET (u[i] < v[i]))) ALORS
 ·  · RENVOYER -1
(+) · SINON
 ·  · RENVOYER +1
 · FSI
FIN
```

Algo. 1. Comparaison pour l'ordre lexicographique.

(3) Faites la preuve de correction partielle. Indications : construisez la table de vérité des 4 propositions $i \leqslant |u|$, $i \leqslant |v|$, $u[i] < v[i]$ et $u[i] > v[i]$ et associez les différentes interprétations de ces quatre propositions aux trois conditions de l'instruction conditionnelle `SI/ALORS/...` qui déterminent les valeurs de retour 0, -1 et $+1$ de l'algorithme.

Solution. **(1)** La séquence de terme initial 1 formée par les valeurs successives de la variable i est strictement croissante et ne peut être majorée, elle dépassera donc nécessairement la valeur $\min\{|u|, |v|\}$, ce qui invalidera la condition d'entrée dans la boucle et arrêtera l'algorithme.

(2) On le montre à l'aide d'une récurrence finie sur le prédicat $P(i)$ suivant : avant la i-ème entrée dans la boucle, on a

$$\forall j \in [\![1, i-1]\!] \quad u_j = v_j. \tag{4}$$

Initialisation. On rappelle que $\forall x \in X\ P(x)$ signifie $\forall x\ (x \in X) \Rightarrow P(x)$, donc si $X = \varnothing$ alors $x \in X$ est faux et donc $(x \in X) \Rightarrow P(x)$ est toujours vrai. Ainsi $P(1)$ est vrai.

Hérédité. Supposons que $P(i)$ soit vrai. La proposition (4) est donc satisfaite. Si la condition d'entrée dans la boucle est testée pour la $(i+1)$-ème fois, c'est qu'elle a été satisfaite la i-ème fois ce qui supposait que $u_i = v_i$, donc

$$\big((\forall j \in [\![1, i-1]\!]\ u_j = v_j) \wedge (u_i = v_i)\big) \ \equiv\ (\forall j \in [\![1, i]\!]\ u_j = v_j)$$

et donc $P(i+1)$ est vrai. On conclut que si le test de boucle est réalisé pour une valeur de i arbitraire, nécessairement $\forall j \in [\![1, i-1]\!]\ u_j = v_j$.

Quand on sort de la boucle, $P(i)$ est vrai.

(3) Pour montrer que l'algorithme est correct, il faut s'assurer que :

(i) l'on n'accède jamais à une cellule inexistante de u ou v ;

(ii) l'algorithme renvoie toujours la bonne valeur.

(i) Nous faisons l'hypothèse que dans notre pseudo-langage algorithmique, l'évaluation d'une forme disjonctive (resp. conjonctive) se fait de gauche à droite, et s'arrête dès que l'une des clauses est satisfaite (resp. n'est pas satisfaite). Ainsi, la condition `(u[i] = v[i])` de la boucle n'est évaluée que si $i \leqslant |u|$ et $i \leqslant |v|$, tout comme la condition `(u[i] < v[i])` en `(-)`. Par conséquent, on n'accède jamais à une cellule inexistante.

(ii) Il reste à montrer que l'algorithme renvoie la valeur :

0 si et seulement si $u = v$ (cf. `(0)`)

-1 si et seulement si u est un préfixe de v ou que u précède v (cf. `(-)`)

1 si et seulement si v est un préfixe de u ou que v précède u (cf. `(+)`)

La valeur renvoyée par l'algorithme est déterminée par les valeurs de vérité des formules propositionnelles impliquées dans la structure conditionnelle `SI/ALORS/SINON`. Il nous faut donc analyser ce qu'elles signifient sur les mots u et v.

Comme ces conditions ne sont évaluées qu'après la sortie de la boucle, on sait d'une part que la proposition (4) et la négation de la condition d'entrée dans la boucle sont satisfaites :

$$\forall j \in [\![1, i-1]\!]\ u_j = v_j \tag{E}$$

$$(i > |u|) \vee (i > |u|) \vee \underbrace{(u_i < v_i) \vee (u_i > v_i)}_{\equiv (u_i \neq v_i)}. \tag{P}$$

Dressons la table de vérité de (P) en fonction des 4 conditions qui la composent (cf. table 1). Sur les 16 interprétations possibles, on peut éliminer les 4 où les propositions $(u_i < v_i)$ et $(v_i < u_i)$ sont vraies ce qui est impossible simultanément.D'autre part, comme les propositions impliquant u_i ou v_i ne sont évaluées que si $(i \leqslant |u|) \wedge \neg(i \leqslant |v|)$, autrement dit quand les propositions des deux premières colonnes sont fausses, on peut regrouper les 3 interprétations commençant par (F, V), puis les trois commençant par (V, F) et les trois commençant par (V, V) (les deux dernières colonnes sont grisées). La première interprétation est celle qui permet d'entrer dans la boucle avec $\neg P$, on ne s'intéresse qu'à celles qui sont vraies pour les conditions qui suivent sa sortie.

		$i > \|u\|$	$i > \|v\|$	$u_i < v_i$	$u_i > v_i$	P
1		F	F	F	F	F
2	(+)	F	F	F	V	V
3	(-)	F	F	V	F	V
4	(+)	F	V			V
5	(-)	V	F			V
6	(0)	V	V			V

Table 1. Table de vérité de la proposition P.

Dans tous les cas étudiés ci-dessous, la proposition E est satisfaite.

- `(0)` Dans le `SI`, seule l'interprétation 6 est possible, ce qui entraîne l'égalité des deux mots u et v. On renvoie bien 0 si et seulement si $u = v$.
- `(-)` Dans le `SINON SI`, on a d'une part $i \leqslant |u|$ ou $i \leqslant |v|$ et deux interprétations possibles : l'interprétation 3 correspond au cas où $i \leqslant |u|$ et $i \leqslant |v|$ et $u_i < v_i$, le mot u précède donc v dans l'ordre lexicographique. On renvoie donc la valeur -1 si et seulement si $u < v$; l'interprétation 5 correspond au cas où $i > |u|$ puisque nécessairement $i \leqslant |v|$ et le mot u est donc un préfixe de v.
- `(+)` Dans le `SINON`, les deux interprétations 2 et 4 restantes correspondent respectivement au cas où v précède u dans l'ordre lexicographique et au cas où v est un préfixe de u. On renvoie donc la valeur 1 si et seulement si $v < u$.

EXERCICE 4. **(1)** Modifiez l'algorithme TriRépartition pour écrire un algorithme `Partition(@L)` qui partitionne une liste L de mots en listes L_k des mots de longueur k et renvoie la table P de ces listes. À l'issue du partitionnement, la liste L est vide. On utilisera librement l'algorithme `SupTete(@L)` qui renvoie la valeur à la tête de la liste L et supprime cette cellule, et l'algorithme `InsQueue(@L,x)` qui insère une liste atomique d'une seule cellule contenant x à la queue de la liste L.

(2) Calculez la complexité de l'algorithme qui réalise cette partition. On supposera que l'algorithme `InsQueue(@L,x)` est en $\Theta(1)$.

Solution. **(1)** La première étape de l'algorithme consiste à calculer la longueur du mot le plus long pour l'allocation des "casiers" de répartition P. Notons que l'algorithme élimine un à un les mots dans la liste d'appel L au fur et à mesure de leur insertion dans les casiers de P. À la sortie de l'algorithme, la liste L est donc vide.

```
ALGORITHME Partition(@L):liste
DONNEES
 · L: liste de mots
VARIABLES
 · P: tableau de listes
 · mot: mot
DEBUT
 · lmax ← LongueurMax(L)
 · Allouer(P, lmax, [])
 · TQ (L != []) FAIRE
 ·  · mot ← SupTete(L)
 ·  · InsQueue(P[|mot|],mot)
 · FTQ
 · RENVOYER P
FIN
```

Algo. 2. Partition par les longueurs de mots.

Notons $n := |L|$ la taille de la liste L. La recherche de la longueur du mot le plus long est en $\Theta(n)$ si l'on admet que le calcul de la longueur d'un

mot est en $\Theta(1)$, ce qui n'est pas toujours pertinent selon le langage utilisé (cf. la fonction `strlen()` du C par exemple). L'allocation des casiers de répartition a un coût de $\Theta(l_{\max})$.

(2) Si l'on utilise une structure *ad hoc*, l'insertion en bout de liste peut se faire en $\Theta(1)$ même avec une liste simplement chaînée, il suffit que la structure intègre un pointeur vers le dernier élément de la liste et soit mis à jour pour les différents opérateurs de liste. L'extraction/répartition des n éléments de la liste L se fait donc en $\Theta(n)$. La complexité globale est donc en $\Theta(n) + \Theta(l_{\max}) = \Theta(n)$ si la longueur maximale des mots est faible devant le nombre n de mots de la liste L.

EXERCICE 5. Dans le tri lexicographique, pourquoi avant le tri par répartition sur la k-ème lettre, faut-il concaténer la liste L_k des mots de longueur k *avant* la liste L des autres mots ?

Solution. Si un mot $u = u_1u_2 \dots u_k$ de L_k est préfixe d'un mot $v = u_1u_2 \dots u_k \dots$ de L, alors il sera rajouté à la fin de la liste associée au symbole u_k *avant* le mot v. La concaténation des listes dans l'ordre alphabétique assure que u précèdera v et pas le contraire conformément à l'ordre lexicographique.