

## TD n°1

Dans les exercices suivants, on s'intéresse à des ensembles de  $n$  entiers tous distincts rangés dans un tableau  $T[1], \dots, T[n]$ . Les algorithmes considérés ici effectuent des **affectations** et leur seul critère de décision est la **comparaison** de deux éléments ( $=$  et  $<$ ). En aucun cas ils ne peuvent effectuer des opérations arithmétiques, comme l'addition ou la multiplication.

### 1 Maximum de $n$ entiers

**Question 1.1** Écrire un algorithme (naïf !) qui calcule le maximum de  $n$  entiers. Quelle en est la complexité (en nombre de comparaisons effectuées, en nombre d'affectations effectuées, dans le pire des cas, le meilleur) ?

**Solution:** *On donne ici un algorithme naïf pour le calcul du maximum de  $n$  entiers :*

*début*

*$max \leftarrow T[1]$*

*pour  $i$  de 2 à  $n$  faire*

*si  $T[i] > max$  alors  $max \leftarrow T[i]$*

*fin*

*Retourner  $max$*

*fin*

*Complexité en nombre de comparaisons : quel que soit le cas, on effectue  $n - 1$  comparaisons dans l'algorithme, chaque élément après le premier étant comparé une fois.*

*Complexité en nombre d'affectations :*

*- dans le pire des cas :  $n$  (si le tableau est ordonné de manière croissante),*

*- dans le meilleur des cas : 1 (si le maximum est en première position),*

□

**Question 1.2** Quelle est la complexité en moyenne en nombre d'affectations effectuées, en nombre de comparaisons effectuées ?

*Indications pour le calcul des affectations en moyenne :* soit  $P_{n,k}$  le nombre de permutations  $\sigma$  de  $\{1, \dots, n\}$  telles que sur la donnée  $T[1] = \sigma(1), \dots, T[n] = \sigma(n)$  l'algorithme effectue  $k$  affectations. Donner une relation de récurrence pour  $P_{n,k}$ . Soit  $G_n(z) = \sum P_{n,k} z^k$ . Montrer que  $G_n(z) = z(z+1) \cdots (z+n-1)$ . En déduire le résultat.

**Solution:** *Complexité en nombre de comparaisons en moyenne : quel que soit le cas, on effectue  $n - 1$  comparaisons, donc la complexité est la même que dans le pire et le meilleur cas.*

*Complexité en nombre d'affectations en moyenne : pour le calcul de la complexité en moyenne, il va falloir faire appel à un calcul plus poussé :*

**Rappel :** *complexité en moyenne d'un algorithme  $A$  de données de taille  $n$  :*

$$moy_A(n) = \sum_{n \text{ données de taille } m} p(d) \text{coût}_A(d)$$

- où  $p(d)$  est la probabilité que  $d$  soit une entrée de l'algorithme  $A$ .
- les données sont les permutations de  $[1, \dots, n]$  stockées dans  $T$ . Les permutations étant totalement aléatoires et n'étant régies par aucune loi mathématique, on peut les supposer équiprobables.
  - le coût d'un algorithme  $A(T)$  est son nombre d'affectation. Ainsi, pour chaque cas de  $P_{n,k}$ , l'algorithme effectue  $k$  affectations. On obtient donc ainsi que le coût d'un algorithme  $A(T)$  est de  $kP_{n,k}$ .
  - d'où :

$$\text{moy}_A(n) = \frac{1}{n!} \sum_{T \text{ permutations de } [1, \dots, n]} \text{cout}_A(T) = \frac{1}{n!} \sum_{k=1}^n kP_{n,k}$$

- trouvons  $P_{n,k}$  :
  - supposons le max en  $T[n]$  : pour avoir  $k$  affectations en tout, il faut :  $P_{n-1, k-1}$
  - sinon, les affectations sont réalisées avant et on a alors  $n-1$  permutations possibles :  $P_{n-1, k}$
  - Donc  $P_{n,k} = P_{n-1, k-1} + (n-1)P_{n-1, k}$
- cas limites :
  - $P_{n,0} = 0$
  - si  $k > n$  :  $P_{n,k} = 0$
  - $\forall n \in \mathbb{N} : P_{n,n} = 1$
- Utilisons les séries génératrices :  $G_n(z) = \sum_{k=1}^n P_{n,k} z^k$

$$\begin{aligned} G_{n+1}(z) &= \sum_{k=1}^{n+1} P_{n+1,k} z^k \\ &= \sum_{k=1}^{n+1} (P_{n,k-1} + nP_{n,k}) z^k \\ &= \sum_{k=1}^{n+1} z P_{n,k-1} z^{k-1} + n \sum_{k=1}^{n+1} P_{n,k} z^k \\ &= z \sum_{k=0}^n P_{n,k} z^k + n \sum_{k=1}^{n+1} P_{n,k} z^k \end{aligned}$$

Or  $P_{n,0} = 0$  et  $P_{n,n+1} = 0$  d'où :

$$G_{n+1}(z) = z \sum_{k=0}^n P_{n,k} z^k + n \sum_{k=1}^n P_{n,k} z^k$$

d'où :

$$G_{n+1}(z) = (z + n)G_n(z)$$

Or  $G_1(z) = P_{1,1}z = z$  d'où  $G_n(z) = z(z+1)\dots(z+n-1)$

- de plus :

$$\begin{aligned}G'_n(z) &= \sum_{k=1}^n kP_{n,k}z^{k-1} \\d'où : G'_n(1) &= \sum_{k=1}^n kP_{n,k} \\de même : G_n(1) &= \sum_{k=1}^n P_{n,k} = n!\end{aligned}$$

- On obtient donc :

$$\begin{aligned}\text{moy}_A(n) &= \frac{G'_n(1)}{G_n(1)} \\&= [\ln G(z)]'(1) \\&= \left[ \sum_{i=0}^{n-1} \ln(z+i) \right]'(1) \\&= 1 + \frac{1}{2} + \dots + \frac{1}{n} \\&= H_n \text{ (la suite harmonique à l'ordre } n\text{)}\end{aligned}$$

$$\text{moy}_A(n) = O(\ln n)$$

□

**Question 1.3** L'algorithme que vous avez proposé est-il optimal pour la complexité en comparaisons dans le pire des cas ?

**Solution:** Une comparaison est un match entre deux entiers, le gagnant étant le plus grand. Si le max est connu, c'est que tout les autres entiers ont perdu au moins une fois un match contre lui, ce qui demande en tout  $n - 1$  comparaisons. Notre algorithme en faisant toujours  $n - 1$ , il est optimal pour la complexité en nombre de comparaisons. □

## 2 Plus petit et plus grand

Dans l'exercice suivant, on ne s'intéresse plus qu'à la **complexité dans le pire des cas et en nombre de comparaisons** des algorithmes.

**Question 2.1** On s'intéresse maintenant au calcul (simultané) du maximum et du minimum de  $n$  entiers. Donner un algorithme naïf et sa complexité.

**Solution:** On donne ici un algorithme naïf pour le calcul du maximum et du minimum de  $n$  entiers :

**début**

$max \leftarrow T[1]$

$i_{max} \leftarrow 1$

$min \leftarrow T[1]$

**pour**  $i$  de 2 à  $n$  **faire**

**si**  $T[i] > max$  **alors**  $max \leftarrow T[i]$ ;  $i_{max} \leftarrow i$

**fin**

**pour**  $i$  de 2 à  $n$  **faire**

**si**  $i \neq i_{max}$  **et**  $T[i] < min$  **alors**  $min \leftarrow T[i]$

**fin**

Retourner  $(max, min)$

**fin**

Complexité en nombre de comparaisons : On effectue  $n - 1$  comparaisons pour trouver le maximum, et  $n - 2$  pour trouver le minimum si le maximum n'est pas en  $T[1]$ , sinon on effectue tout de même  $n - 1$  comparaisons. On a donc une complexité en  $2n - 2$ .  $\square$

**Question 2.2** Une idée pour améliorer l'algorithme est de regrouper *par paires* les éléments à comparer, de manière à diminuer ensuite le nombre de comparaisons à effectuer. Décrire un algorithme fonctionnant selon ce principe et analyser sa complexité.

**Solution:** On regroupe maintenant les éléments par paire pour ensuite effectuer les opérations de comparaisons.

L'algorithme suivant permet de résoudre le problème posé :

**début**

**pour**  $i$  de 1 à  $\lfloor \frac{n}{2} \rfloor$  **faire**

**si**  $T[2i - 1] > T[2i]$  **alors** échange  $T[2i - 1]$  et  $T[2i]$

**fin**

$max \leftarrow T[2]$

**pour**  $i$  de 2 à  $\lfloor \frac{n}{2} \rfloor$  **faire**

**si**  $T[2i] > max$  **alors**  $max \leftarrow T[2i]$

**fin**

$min \leftarrow T[1]$

**pour**  $i$  de 2 à  $\lfloor \frac{n}{2} \rfloor$  **faire**

**si**  $T[2i - 1] < min$  **alors**  $min \leftarrow T[2i - 1]$

**fin**

**si**  $n$  impair **alors**

**si**  $T[n] > max$  **alors**  $max \leftarrow T[n]$

**sinon si**  $T[n] < min$  **alors**  $min \leftarrow T[n]$

**fin**

Complexité associée :

- si  $n$  est pair, on va avoir  $\frac{n}{2}$  comparaisons pour faire les paires,  $\frac{n}{2} - 1$  comparaisons pour

trouver le maximum et autant pour le minimum. Dans ce cas là, la complexité vaut donc :

$$\frac{3n}{2} - 2$$

- si  $n$  est impair, le nombre de comparaisons est inférieur ou égal à  $3\lfloor \frac{n}{2} \rfloor$ , donc inférieur ou égal à  $\lceil \frac{3n}{2} \rceil - 2$ .

Car on a  $3\lfloor \frac{n}{2} \rfloor = \lfloor \frac{n}{2} \rfloor + 2 \times \lfloor \frac{n}{2} \rfloor = (\lfloor \frac{n}{2} \rfloor - 1) + (n - 1) = \lceil \frac{3n}{2} \rceil - 2$ .

□

**Question 2.3** Montrons l'optimalité d'un tel algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer. Nous utiliserons la méthode de l'*adversaire*.

Soit  $A$  un algorithme qui trouve le maximum et le minimum. Pour une donnée fixée, au cours du déroulement de l'algorithme, on appelle *novice* (N) un élément qui n'a jamais subi de comparaisons, *gagnant* (G) un élément qui a été comparé au moins une fois et a toujours été supérieur aux éléments auxquels il a été comparé, *perdant* (P) un élément qui a été comparé au moins une fois et a toujours été inférieur aux éléments auxquels il a été comparé, et *moyens* (M) les autres. Le nombre de ces éléments est représenté par un quadruplet d'entiers  $(i, j, k, l)$  qui vérifient bien sûr  $i + j + k + l = n$ .

Donner la valeur de ce quadruplet au début et à la fin de l'algorithme. Exhiber une stratégie pour l'adversaire, de sorte à maximiser la durée de l'exécution de l'algorithme. En déduire une borne inférieure sur le nombre de tests à effectuer.

**Solution:** Cette méthode va être utilisée pour fournir une borne inférieure sur le nombre de comparaisons à effectuer et ainsi montrer l'optimalité de l'algorithme précédent.

**Définitions :**

- *Novice* : élément qui n'a jamais été comparé
- *Gagnant* : élément qui a gagné tout ses matchs
- *Perdant* : élément qui a perdu tout ses matchs
- *Moyen* : élément qui a gagné et perdu

Soit  $A$  notre algorithme. On note  $(i, j, k, l)$  respectivement le nombre de novices, gagnants, perdants et moyens (avec  $i + j + k + l = n$ ).

Au début de l'algorithme, on a  $(i, j, k, l) = (n, 0, 0, 0)$ . A la fin,  $(i, j, k, l) = (0, 1, 1, n - 2)$ .

Le but de l'adversaire est de maximiser la durée de l'algorithme tout en conservant une stratégie cohérente. On peut alors donner l'ensemble des opérations résumées dans le tableau ??

/ représente les cas où l'entier reste inchangé. "au choix" : arbitrairement sauf si les réponses précédentes forcent le résultat, par exemple  $L[2] < L[5]$  puis  $L[5] < L[4]$  alors  $L[2] < L[4]$ .

Cherchons à donner une borne inférieure sur le nombre de tests à effectuer.

Il faut que chaque novice soit comparé au moins une fois. La comparaison la plus efficace est alors  $N : N$  car elle diminue le nombre de novice de deux, et il en faut au moins  $\lceil \frac{n}{2} \rceil$  pour atteindre le bon nombre de novices.

De plus, il faut à la fin que l'on ait  $n - 2$  moyens, ce qui nécessite au moins  $n - 2$  autres comparaisons.

D'où au minimum

$$n - 2 + \lceil \frac{n}{2} \rceil = \lceil \frac{3n}{2} \rceil - 2 \text{ comparaisons.}$$

comparaison	choix	i	j	k	l
$N : N$	au choix	$i - 2$	$j + 1$	$k + 1$	/
$N : G$	$G > N$	$i - 1$	/	$k + 1$	/
$N : P$	$P < N$	$i - 1$	$j + 1$	/	/
$N : M$	$M > N$	$i - 1$	/	$k + 1$	/
	$N > M$	$i - 1$	$j + 1$	/	/
$G : G$	au choix	/	$j + 1$	/	$l + 1$
$G : P$	$G > P$	/	/	/	/
$G : M$	$G > M$	/	/	/	/
$P : P$	au choix	/	/	$k - 1$	$l + 1$
$P : M$	$P < M$	/	/	/	/
$M : M$	au choix	/	/	/	/

*La borne inférieure sur le nombre de tests à effectuer est égale à la complexité de l'algorithme précédent. On montre ainsi que ce dernier est optimal.* □