TD : Exercices sur les algorithmes

Exercice 1 : Exercice

On considère un tableau $t = (t_0, \dots, t_{n-1})$ d'entiers. On dit que t présente un *maximum local* en $i \in [\![0, n[\![$ lorsque si t_i est supérieur ou égal à son éventuel voisin de gauche et son éventuel voisin de droite. Par exemple, t_0 est un maximum local si et seulement si $t_0 \geqslant t_1$.

1. Montrer que tout tableau non vide possède au moins un maximum local.
2. Écrire un algorithme naif
3. Proposer un algorithme efficace pour déterminer un maximum local d'un tableau, et donner sa complexité.
4. Implémenter cet algorithme en OCaml, sous la forme d'une fonction prenant en argument un tableau supposé non vide et renvoyant l'indice d'un maximum local (ou pic) de ce tableau.

```
indice_pic : 'a array -> int
```

5. **Difficile.** On considère maintenant un tableau bidimensionnel de taille $n \times n$. Une case du tableau possède (au plus) quatre voisins (au-dessus, en dessous, à gauche et à droite) et l'on cherche toujours un maximum local, c'est-à-dire une case dont le contenu est supérieur ou égal à celui de tous ses voisins. Trouver un algorithme permettant de résoudre ce problème en temps $O(n)$.

Exercice 2 : Partitions d'un entier

Une *partition* d'un entier naturel n est une décomposition de n comme somme d'entiers strictement positifs. Deux partitions ne différant que par l'ordre des termes sont considérées comme égales, et l'on convient que la somme vide est l'unique partition de 0. On note $p(n)$ le nombre de partitions de n. On a par exemple $p(5) = 7$:

— $5 = 5$;
— $5 = 4 + 1$;
— $5 = 3 + 2$;
— $5 = 3 + 1 + 1$;
— $5 = 2 + 2 + 1$;
— $5 = 2 + 1 + 1 + 1$;
— $5 = 1 + 1 + 1 + 1 + 1$.

Combien vaut $p(1000) \pmod{2^{64}}$?
— Le lecteur perspicace remarquera que le nombre de partitions de n est exactement le nombre de découpes distinctes d'une barre de Toblérone de taille n.
— On pourra s'intéresser à $s(n, k)$, nombre de partitions de n ne faisant intervenir que des entiers inférieurs ou égaux à k.

Exercice 3 : Hanoi

Dans cet exercice, on s'intéresse au problème des tours de Hanoi lorsqu'on dispose que 4 tiges et de n disques. On note t_n le nombre minimal de mouvements nécessaires pour résoudre le problème.

1. Montrer que si $k \in [\![0, n]\!]$, alors $t_n \leqslant 2t_{n-k} + 2^k - 1$. En déduire que
$$\forall p \in \mathbb{N}, \quad t_{p(p+1)/2} \leqslant 2t_{p(p-1)/2} + 2^p - 1.$$
2. Justifier la croissance de la suite (t_n) et en déduire l'existence d'un algorithme résolvant le problème à 4 tiges avec un coût en
$$O\left(\sqrt{n} 2^{\sqrt{2n}}\right).$$

Exercice 4 : Problème du lâcher d'œufs

On considère un immeuble de N étages (numérotés de 1 à N), dans lequel on souhaite mener à bien une expérience scientifique capitale. On dispose de p œufs, rigoureusement identiques, et l'on veut déterminer le plus petit entier k tel que la chute depuis l'étage k d'un œuf se termine par l'explosion de ce dernier. On pourra faire les hypothèses suivantes :

— si un œuf se brise après une chute de l'étage k, alors il se serait également brisé depuis une chute depuis un étage $k' \geqslant k$;
— inversement, si un œuf survit à une chute de l'étage k, il aurait survécu à une chute d'un étage $k' \leqslant k$;
— de plus, si un œuf survit à une chute, il n'en garde aucune séquelle : il est possible de l'utiliser dans les expériences suivantes sans que cela n'en affecte le résultat.

Le but de l'exercice est de calculer efficacement le nombre minimal de lâchers d'œufs qu'il faut effectuer dans le pire cas pour déterminer k_{min} (l'étage à partir duquel les œufs se brisent). On notera $\varphi(N, p)$ ce nombre minimal de lancers, et l'on posera $k_{min} = N + 1$ si l'œuf ne se casse pas quand il est lâché du dernier étage.

1. Déterminer $\varphi(N, 1)$.
2. Supposons que l'on dispose de $p > 1$ œufs, et que l'on décide de lâcher le premier œuf depuis un étage k vérifiant $2 \leqslant k < N - 1$.
   (a) Si l'œuf se casse, combien de lancers nous faudra-t-il dans le pire des cas pour déterminer k_{min} ?
   (b) Même question si l'œuf ne casse pas.
3. Comment choisir les valeurs de $\varphi(0, p)$ et $\varphi(n, 0)$ pour que cette relation reste valable dans les cas $p = 1$, $k = 1$ et $k = N$?
4. En déduire une relation de récurrence permettant de calculer $\varphi(N, p)$.
5. Écrire un programme C permettant de calculer $\varphi(N, p)$ par la méthode récursive « naïve ». Ce programme prendra N et p comme arguments en ligne de commande.
   — Il sera plus pratique de coder les valeurs de $\varphi(N, p)$ par des nombres flottants (type `double` par exemple) pour pouvoir utiliser la valeur `INFINITY`.
   — `INFINITY` est définie dans `math.h`, qu'il faudra inclure. Il faudra également ajouter l'option `-lm` à la fin de la ligne de commande de compilation.
6. Écrire une nouvelle version de φ utilisant la programmation dynamique ascendante pour accélérer le calcul.
7. Déterminer la complexité en temps et en espace de cet algorithme.
8. Que peut-on dire du sens de variation de $k \mapsto \varphi(k - 1, p - 1)$ et de $k \mapsto \varphi(N - k, p)$?
9. En notant $f : k \mapsto \max(\varphi(k - 1, p - 1), \varphi(N - k, p))$, en déduire qu'un minimum local de f est forcément un minimum global.
10. Proposer alors une amélioration du calcul de φ permettant d'obtenir une complexité temporelle en $\mathrm{O}(pN \log N)$.
11. On définit $\psi(t, p)$ comme la plus grande valeur de N pour laquelle il est possible de déterminer k_{min} en au plus t essais si l'on dispose de p œufs. Montrer que $f(t, p) = \sum_{i=0}^{p} \binom{t}{i}$ et en déduire un algorithme résolvant le problème en temps $\mathrm{O}(p \log N)$.

Exercice 5 : Tri fusion d'un tableau

1. Écrire une fonction `merge` ayant la spécification et le prototype suivants :

```
void merge(int arr[], int mid, int len, int buffer[]);
```

Préconditions
— `arr` et `buffer` sont de longueur (au moins) `len` ;
— $0 \leqslant$ `mid` $<$ `len` ;
— `arr[0]` $\leqslant$ `arr[1]` $\leqslant \ldots \leqslant$ `arr[mid - 1]` et `arr[mid]` $\leqslant \ldots \leqslant$ `arr[len - 1]`.

Postcondition Après l'appel, `arr[0]` $\leqslant \ldots \leqslant$ `arr[mid]` $\leqslant \ldots \leqslant$ `arr[len - 1]`

2. En déduire une fonction `merge_sort` triant un tableau à l'aide de l'algorithme du tri fusion. On précisera sa complexité en temps et en espace.

```
void merge_sort(int arr[], int len);
```