# Devoir surveillé d'informatique

**Consignes**

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

## ❑ Exercice 1 : *Questions de cours : terminaison*

On considère la fonction d'Ackerman $a : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ définie par par :

$$\begin{cases} a(0,m) &= m+1 \\ a(n,0) &= a(n-1,1) \text{ si } n > 0 \\ a(n,m) &= a(n-1,a(n,m-1)) \text{ si } n > 0 \text{ et } m > 0 \end{cases}$$

**Q1**– Calculer $a(1,2)$

**Q2**– Ecrire en OCaml une fonction `ack: int -> int -> int` qui prend en argument deux entiers positifs $n$ et $m$ et renvoie $a(n,m)$.

**Q3**– Prouver la terminaison de la fonction `ack` on précisera soigneusement le variant et la relation d'ordre bien fondée utilisée.

## ❑ Exercice 2 : *Questions de cours : preuve par induction structurelle*

**Q4**– Donner la définition inductive des arbres binaires sur un ensemble d'étiquettes $E$.

**Q5**– Ecrire en OCaml un type `arbrebin` représentant un arbre binaire en utilisant un type paramétré `'a` pour l'ensemble des étiquettes.

**Q6**– Rappeler la définition de la hauteur et de la taille d'un arbre binaire.

**Q7**– Prouver *par induction structurelle* que la taille d'un arbre binaire de hauteur $h$ est inférieure ou égale à $2^{h+1} - 1$.

## ❑ Exercice 3 : *Recherche des k premiers maximums d'une liste*

Les fonctions demandées dans cet exercice doivent être écrites en langage OCaml.

On s'intéresse au problème de la recherche des $k$ premiers maximums d'une liste de $n$ entiers. Dans toute la suite de l'exercice on supposera que la liste est *non vide* : $n > 0$ et qu'on extrait moins de maximums qu'il n'y a d'éléments dans la liste c'est à dire que $k \leqslant n$. On cherche donc à écrire une fonction `kmax : int list -> int -> int list` qui renvoie la liste des `k` premiers maximums de la liste donnée en argument. Par exemples :

— `kmax [2; 5; 1; 8; 3; 0; 4] 2` renvoie `[8, 5]`

— `kmax [7; 8; 8; 1; 6; 3; 2; 9] 3` renvoie `[9; 8; 8]`

— `kmax [1; 0; 1; 2; 4] 4` renvoie `[4; 2; 1; 1]`

Les trois parties sont indépendantes et dans chacune d'elle on propose un algorithme différent afin d'écrire la fonction `kmax`.

### ■ Partie I : Résolution par recherche successive des maximums

**Q8**– Ecrire une fonction `max_reste : int list -> int * int list` qui prend en argument une liste et renvoie le couple composé du maximum de cette liste et de cette liste privée d'un de ses maximums. Par exemple `max_reste [2; 6; 4; 6; 5]` renvoie `(6, [2; 4; 6; 5])`. On pourra procéder par correspondance de motif et traiter le cas de la liste vide par un `failwith`.

**Q9**– Donner en la justifiant brièvement la complexité de la fonction `max_reste`.

**Q10**– Ecrire une première version de la fonction `kmax` qui procède par extraction successive des `k` premiers maximums en utilisant la fonction `max_reste`. On procèdera de façon récursive sans utiliser les aspects impératifs de OCaml.

**Q11**– Quelle est la complexité de cette version de la fonction `kmax` en fonction du nombre $k$ de maximums à extraire et de la longueur $n$ de la liste ?

■ **Partie II** : Résolution par un tri

**Q12**– Ecrire une fonction `kpremiers int list -> int -> int list` qui prend en argument une liste `lst` et un entier `k` et renvoie la liste composée des `k` premiers éléments de `lst`. Par exemple `kpremiers [2; 7; 1; 8; 5] 3` renvoie la liste `[2; 7; 1]`. On procédera de façon récursive sans utiliser les apsects impératifs de OCaml.

**Q13**– On propose d'écrire la fonction `kmax` en triant la liste par ordre décroissant puis en prenant ses $k$ premiers éléments. En supposant que l'algorithme de tri utilisé a une complexité en $\mathcal{O}(n \log n)$, donner la complexité de ce nouvel algorithme (on ne demande *pas* de le programmer).

■ **Partie III** : Résolution en utilisant un tas

Dans la suite on suppose que la structure de données de tas d'entiers (type `int`), est *déjà implémentée* par un type `tas` sur lequel on dispose des fonctions suivantes :

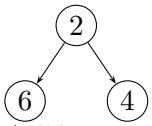
- `cree_tas : int -> tas` qui prend en argument un entier `cap` et renvoie un tas binaire vide de capacité maximale `cap`.
- `donne_taille : tas -> int` qui prend en argument un tas et renvoie sa taille (le nombre d'éléments actuellement stocké dans le tas).
- `insere : int -> tas -> unit` qui insère une nouvelle valeur dans le tas. Cette fonction échoue lorsque le tas est plein.
- `donne_min : tas -> int` qui renvoie la valeur minimale contenu dans le tas *sans modifier le tas*. Cette fonction échoue lorsque le tas est vide.
- `extrait_min : tas -> int` qui renvoie, *en le supprimant du tas* le minimum du tas. Cette fonction échoue lorsque le tas est vide.

**Q14**– Rappeler en les justifiant, les complexités des opérations `insere` et `extrait_min` en fonction de la taille du tas notée `k` si on suppose que l'implémentation de la structure de tas est réalisée grâce à un tableau.

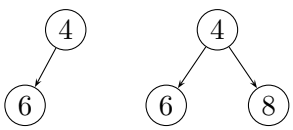
Afin d'extraire les `k` premiers éléments d'une liste de taille `n`, on propose créer un tas de taille `k` puis de parcourir récursivement la liste, pour chaque élément rencontré :

- si le tas n'est pas plein on y insère l'élément.
- sinon, on compare l'élément avec le minimum du tas, s'il est plus grand on extrait le minimum du tas et on insère l'élément dans le tas.

Par exemple, si on veut extraire les 3 premiers maximums de la liste `[4; 6; 2; 8; 3; 7; 1; 9; 5]`, après l'insertion des trois premiers éléments, le tas est :




A l'étape suivante, 8 étant plus grand que 2 (le minimum du tas), on extrait 2 du tas et on y insère 8 ce qui donne :




**Q15**– Poursuivre le déroulement de cet algorithme en faisant figurer comme ci-dessus les étapes de l'évolution du tas.

**Q16**– Donner une implémentation de la fonction `kmax` utilisant ce nouvel algorithme. On rappelle qu'on pourra utiliser les fonctions de manipulation de la structure de `tas` données en début de partie. Comme précédemment, on procédera par récurrence sans utiliser les aspects impératifs de OCaml.

**Q17**– Donner en la justifiant la complexité de ce nouvel algorithme en fonction de $k$ et $n$.

❑ **Exercice 4** : *Saut de valeur maximale* 🎓CAPES NSI 2023

Les fonctions demandées dans cet exercice sont à écrire en langage C.

Dans un tableau de flottants (type `double` du langage C) `tab` de taille `n`, on appelle *saut* un couple $(i, j)$ avec $0 \leq i \leq j < $ `n` et la *valeur* d'un saut est la valeur `tab[j]-tab[i]`. Le but de l'exercice est de rechercher la valeur maximale d'un saut dans un tableau. Par exemple, dans le tableau `{ 2.0, 0.2, 3.0, 5.3, 2.0}`, la valeur maximale d'un saut est `5.3 - 0.2 = 5.1`, cette valeur est obtenue en considérant le saut `(1,3)`.

■ **Partie I** : Questions préliminaires et résolution naïve

**Q18**– Ecrire une fonction de signature `int valeur(int tab[], int i, int j, int n)` qui prend en argument un tableau `tab` de taile `n` ainsi que deux indices `i` et `j` et renvoie la valeur du saut `(i,j)`. On vérifiera les préconditions sur `i` et `j` à l'aide d'instructions `assert`. Par exemple si le tableau `tab` est `{2.0; 0.2; 3.0; 5.3; 2.0}` alors `valeur(tab, 2, 0, 5)` renvoie `1.0` (car `tab[2]-tab[0] = 1.0`).

**Q19**– Donner un exemple de tableau avec exactement deux sauts de valeur maximale et préciser ces sauts.

**Q20**– À l'aide d'un contre-exemple, montrer qu'on ne peut pas se contenter de chercher le minimum et le maximum d'un tableau pour trouver un saut de valeur maximale.

**Q21**– Écrire une fonction `sautmax_naif` qui renvoie un saut de valeur maximale dans un tableau de taille `n` en testant tous les couples $(i, j)$ tels que $0 \leq i \leq j <$ `n`.

**Q22**– Quelle est la complexité de la fonction `sautmax_naif` en fonction de la taille $n$ du tableau ?

■ **Partie II** : Résolution avec une méthode diviser pour régner

On propose maintenant d'utiliser une méthode diviser pour régner afin de calculer la valeur maximale d'un saut. On note $n$ la taille du tableau `t` et $p = \lfloor \frac{n}{2} \rfloor$ (où $\lfloor\ \rfloor$ désigne la partie entière). On souhaite calculer :

— $(i_g, j_g)$ un saut de valeur maximale lorsque $j_g < p$ (c'est à dire un saut maximal dans la moitié gauche)

— $(i_d, j_d)$ un saut de valeur maximale lorsque $id \geqslant p$ (c'est à dire un saut maximal dans la moitié gauche)

— $(i_m, j_m)$ un saut de valeur maximal lorsque $i_m < p < j_m$ (c'est à dire un saut maximal dont le premier indice est dans la moitié gauche et le second dans la moitié droite)

**Q23**– Justifier qu'un saut de valeur maximale du tableau `t` est nécessairement un des trois ci-dessus. On pourra faire un schéma pour illustrer le raisonnement.

**Q24**– Justifier que $i_m$ est nécessairement l'indice d'une valeur minimale dans la moitié gauche de `t` (on admettra que de même $j_m$ est nécessairement l'indice d'une valeur maximale dans la moitié droite de `t`).

**Q25**– Ecrire une fonction de signature `int min(int tab[], int a, int b)` qui prend en argument un tableau `tab`, ainsi que deux entiers `a` et `b` (avec `a<=b`) et renvoie l'indice d'un minimum de `tab` entre les deux indices `a` (inclus) et `b` (exclu).
On supposera dans la suite *déjà écrite* une fonction qui `max` qui prend les mêmes arguments et renvoie l'indice d'un maximum du sous tableau `{tab[a],...,tab[b-1]}`

**Q26**– Ecrire une fonction de signature `int sautmax_dpr(int tab[], int a, int b)` qui prend en argument un tableau `tab` ainsi que deux entiers `a` et `b` (avec `a<=b`) et renvoie la valeur d'un saut maximale dans `tab` entre les deux indices `a` (inclus) et `b` (exclu).*Cette fonction doit être récursive et utiliser la méthode diviser pour régner.* On pourra supposer déjà écrite une fonction `max3` qui renvoie le maximum des trois `double` donnés en argument.

**Q27**– Déterminer la complexité de la fonction `sautmax_dpr`.

■ **Partie III** : Résolution par programmation dynamique

On cherche maintenant à résoudre ce problème par programmation dynamique, et on adopte les notations suivantes :

— $t$ est un tableau de taille $n$ contenant des flottants

— $t[i]$ est l'élément d'indice $i$ $(0 \leqslant i < n)$ de $t$,

— pour $0 < k \leqslant n$, $t_k$ est le sous tableau $t[0], \ldots, t[k-1]$

— $m_k$ est l'indice d'un minimum de $t_k$ pour $0 < k < n$.

— $(i_k, j_k)$ est un saut de valeur maximale dans $t_k$ pour $0 < k < n$.

**Q28**– Donner les valeurs de $i_1$, $j_1$ et $m_1$

**Q29**– Donner la relation de récurrence liant $m_{k+1}$, $m_k$ et $t[k+1]$.

**Q30**– Justifier que la relation suivante est correcte :

$$(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) \text{ si } t[k] - t[m_k] < t[j_k] - t[i_k] \\ (m_k, k) \text{ sinon} \end{cases}$$

**Q31**– Ecrire une fonction de signature `int sautmax_dyn(int tab[], int n)` qui prend en argument un tableau et sa taille et renvoie la valeur maximale d'un saut de ce tableau. On procèdera *de façon ascendante* en utilisant les relations de récurrences de la question précédente et en calculant successivement les valeurs maximales de saut dans les sous tableaux $t_1$ puis $t_2$, ... jusqu'à $t_n$.

**Q32**– Déterminer la complexité de la fonction `sautmax_dyn`