

## Devoir surveillé d'informatique

### ⚠ Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml suivant l'exercice. Dans le cas du C, on suppose que les bibliothèques standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.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.
- Veuillez 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

1. On propose le principe suivant pour trier un tableau de  $n$  entiers  $(a_0, \dots, a_{n-1})$  : tant que le tableau n'est pas trié, on sélectionne deux indices  $i$  et  $j$  dans  $\llbracket 0; n-1 \rrbracket$  tels que  $i < j$  et  $a_i > a_j$  et on échange les éléments situés à ces indices. Montrer que cet algorithme termine, on précisera soigneusement le variant et la relation d'ordre bien fondée utilisée.
2. On définit inductivement (et de façon non ambiguë) l'ensemble  $D$  des mots de Dyck (ou parenthésages bien formés) par :  $X_0 = \{\epsilon\}$  et la règle d'inférence d'arité 2  $(x, y) \mapsto \langle x \rangle y$ . Les symboles «  $\langle$  », et «  $\rangle$  » représentent les parenthèses.  
Pour  $d \in D$ , on note  $n_o(d)$  (resp.  $n_f(d)$ ) le nombre de symboles  $\langle$  (resp.  $\rangle$ ) apparaissant dans  $d$ . Montrer par **induction structurelle** que  $n_o(d) = n_f(d)$  pour tout  $d \in D$ .

### □ Exercice 2 : Recherche des $k$ premiers maximums

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

On s'intéresse dans cet exercice à la recherche des  $k$  premiers maximums d'un tableau de  $n$  entiers  $t_0 \dots t_{n-1}$  avec  $k < n$ .

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

1. Ecrire une fonction de signature `int max(int t[], int n)` qui renvoie le premier maximum du tableau `t` de taille `n` en supposant  $n > 0$ .
2. Donner en la justifiant brièvement la complexité de cette fonction.
3. Afin d'extraire les  $k$  premiers maximums, on propose d'extraire successivement les maximums à l'aide de la fonction écrite à la question 1 (pour extraire le  $k$ ème maximum, on travaille sur le sous tableau privé des maximums précédents). Donner la complexité de cet algorithme en fonction de  $k$  et de  $n$  (on ne demande pas de programmer cet algorithme).

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

1. On propose de résoudre ce problème en triant le tableau  $t$  par ordre décroissant puis en prenant ses  $k$  premiers éléments. On suppose déjà écrite une fonction de tri de signature :  
`void int tri(int t[], int n)` qui tri en place et dans l'ordre décroissant le tableau `t` de taille `n` donné en argument. Ecrire une fonction de signature `int * kmax(int t[], int n, int k)` qui utilise cette fonction de tri et renvoie un tableau de taille `k` contenant les  $k$  premiers maximums de `t`.
2. Quelle est la complexité de cette solution si la fonction de tri est le tri par sélection ?
3. Quelle est la complexité de cette solution si la fonction de tri est le tri fusion ?

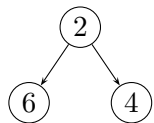
#### ■ Partie III : Résolution en utilisant un tas

Dans la suite on suppose que les tas de valeurs entières (type `int`), sont *déjà implémentés* par la structure de donnée `heap` et on suppose déjà écrites les fonctions suivantes permettant de manipuler cette structure de données :

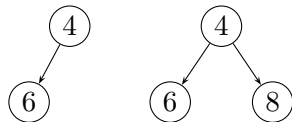
- `heap make_heap(int cap)` qui renvoie un tas binaire vide de capacité `cap`
- `int get_size(heap mh)` qui renvoie la taille du tas `mh`

- `bool insert_heap(int nv, heap* mh)` qui insère `nv` dans le tas `*mh`, dans le cas où l'insertion est impossible (tas plein), la fonction renvoie `false`.
  - `int get_min(heap mh)` qui renvoie la valeur minimale contenu dans le tas `mh` sans modifier le tas.
  - `int extract_min(heap * mh)` qui renvoie (en le supprimant du tas) le minimum du tas `*mh`.
1. Rappeler en les justifiant rapidement les complexités des opérations `insert_heap` et `extract_min` si on suppose que l'implémentation de la structure de tas est réalisée grâce à un tableau.
  2. Afin d'extraire les `k` premiers éléments d'un tableau `t` de taille `n`, on propose de créer un tas de taille `k`, puis on parcourt le tableau à l'aide d'un indice `i` :
    - si `i < k`, le tas n'est pas plein et on y insère `t[i]`
    - si `n > i >= k`, on compare `t[i]` avec le minimum du tas, s'il est plus grand on supprime le minimum du tas et on insère `t[i]` dans le tas.

Par exemple, si on veut extraire les 3 premiers maximums du tableau `[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 :



- a) Poursuivre le déroulement de cet algorithme en faisant figurer comme ci-dessus les étapes de l'évolution du tas.
- b) Prouver que cet algorithme est correct (on pourra utiliser l'invariant valable pour tout  $i \geq k$  : « le tas contient les `k` premiers maximums du sous tableau compris entre les indices 0 et  $i-1$  »).
- c) Ecrire une fonction `int* kmax_heap(int t[], int n, int k)` qui extrait les `k` premiers maximum du tableau `t` de taille `n` et les renvoie dans un tableau de taille `k` en utilisant cet algorithme. Afin de manipuler le tas, on utilisera les fonctions déjà disponibles sur la structure de tas et dont les signatures sont données en début de partie.
- d) Donner en la justifiant la complexité de ce nouvel algorithme en fonction de `k` et `n`.

### □ Exercice 3 : Traversée de rivière

CCINP 2023

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

Dans une vallée des Alpes, un passage à gué fait de cailloux permet de traverser la rivière. Deux groupes de randonneurs arrivent simultanément sur les berges gauche et droite de cette rivière et veulent la traverser. Le chemin étant très étroit, une seule personne peut se trouver sur chaque caillou de ce chemin. Un randonneur sur la berge de gauche peut avancer d'un caillou (vers la droite sur figure ci-dessous) et sauter par dessus le randonneur devant lui (un caillou à droite) si le caillou où il atterrit est libre. De même, chaque randonneur de la berge de droite peut avancer d'un caillou (vers la gauche sur la figure ci-dessous) et sauter par dessus le randonneur devant lui, dans la mesure où le caillou sur lequel il atterrit est libre. Une fois engagés, les randonneurs ne peuvent pas faire marche arrière. De plus, pour simplifier, on suppose qu'une fois tous les randonneurs sur le chemin, il ne reste qu'un caillou de libre.



Le chemin de cailloux est défini par un tableau d'entiers :

```
type chemin_caillou = int array
```

Dans ce tableau, un randonneur venant de la berge de gauche est représenté par un 1, un randonneur issu de la berge de droite par un 2, et un caillou libre par un 0. Avec cette représentation, la figure ci-dessus correspond au tableau `[|1; 1; 1; 0; 2; 2|]`. A partir de cette position, les mouvements possibles sont :

- déplacement du premier randonneur de gauche : `[|1; 1; 0; 1; 2; 2|]`,
  - saut du second randonneur de gauche par dessus le premier : `[|1; 0; 1; 1; 2; 2|]`
  - déplacement du premier randonneur de droite : `[|1; 1; 1; 2; 0; 2|]`
  - saut du second randonneur de droite par dessus le premier : `[|1; 1; 1; 2; 2; 0|]`
1. Écrire une fonction de signature `caillou_vide : chemin_caillou -> int` qui détermine la position du caillou inoccupé.
  2. Écrire une fonction de signature `echange : chemin_caillou -> int -> int -> chemin_caillou` qui permute les valeurs codées sur deux cailloux et renvoie le tableau correspondant à la nouvelle position. Le tableau d'entiers initial représentant le chemin n'est pas modifié. On pourra utiliser ici la fonction `copy` du module `Array`.
  3. Écrire une fonction de signature `randonneurG_avance : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse avancer (vers la droite).
  4. Écrire une fonction de signature `randonneurG_saute : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse sauter (vers la droite) au-dessus d'un randonneur.  
On supposera dans la suite les fonctions de signature `randonneurD_avance : chemin_caillou -> bool` et `randonneurD_saute : chemin_caillou -> bool` écrites de manière similaire pour les randonneurs venant de la berge de droite.
  5. Écrire une fonction de signature `mouvement_chemin : chemin_caillou -> chemin_caillou list` qui, en fonction de l'état du chemin, calcule la liste possible des états suivants après les opérations (si elles sont permises) :
    - i) déplacement d'un randonneur venant de la berge de gauche,
    - ii) déplacement d'un randonneur venant de la berge de droite,
    - iii) saut d'un randonneur venant de la berge de gauche,
    - iv) saut d'un randonneur venant de la berge de droite.
  6. Écrire une fonction `init_chemin int -> int -> chemin_caillou` qui prend en argument deux entiers `ng` et `nd` et renvoie le tableau correspondant à la situation initiale de `ng` randonneurs venant de la gauche et `nd` randonneurs venant de la droite séparé par un caillou vide. Par exemple, `init_chemin 3 2` renvoie le tableau `[| 1; 1; 1; 0; 2; 2 |]`.
  7. Écrire une fonction de signature `passage : int -> int -> chemin_caillou list`, utilisant la question précédente, telle que l'appel `passage ng nd` résout le problème du passage de `ng` randonneurs venant de la berge de gauche et `nd` randonneurs venant de la berge de droite. Par exemple, `passage 3 2` renvoie une liste d'états permettant de passer de `[|1; 1; 1; 0; 2; 2|]` à `[| 2; 2; 0; 1; 1; 1 |]`.

#### □ Exercice 4 : Saut de valeur maximale

🎓 CAPES NSI 2023

Les fonctions demandées dans cet exercice sont à écrire en OCaml.

Dans un tableau de flottants (type `float` de OCaml) `t` 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 `t.(j) - t.(i)`. Le but de l'exercice est de rechercher un saut de valeur maximale dans un tableau. Par exemple, dans le tableau `[| 2.0; 0.2; 3.0; 5.3; 2.0 |]`, un tel saut est  $(1, 3)$  (car 0.2 et 5.3 sont aux indices 1 et 3 respectivement) et la valeur maximale d'un saut est donc  $5.3 - 0.2 = 5.1$ .

1. Écrire une fonction `valeur float array -> int -> int -> float` qui prend en argument un tableau ainsi que deux indices `i` et `j` et renvoie la valeur du saut  $(i, j)$ . Par exemple sur le tableau `[|2.0; 0.2; 3.0; 5.3; 2.0|]` avec les indices 0 et 2 cette fonction renvoie 1.0 (car `t.(2) - t.(0) = 1.0`).
2. Donner un exemple de tableau avec exactement deux sauts de valeur maximale et préciser ces sauts.

3. 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.
4. Écrire une fonction `saut_max_naif` qui renvoie un saut de valeur maximale dans un tableau de taille `n` en testant tous les couples  $(i, j)$  tels que  $0 \leq i \leq j < n$ .

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  $\lfloor \cdot \rfloor$  désigne la partie entière). On souhaite calculer :

- $(i_g, j_g)$  un saut de valeur maximale lorsque  $j_g < p$
- $(i_d, j_d)$  un saut de valeur maximale lorsque  $i_d \geq p$
- $(i_m, j_m)$  un saut de valeur maximal lorsque  $i_m < p < j_m$

5. Justifier qu'un saut de valeur maximale du tableau `t` est nécessairement un des trois ci-dessus.
6. 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`).
7. Écrire une fonction `saut_max_aux` qui prend en argument un tableau ainsi que deux entiers `a` et `b` (avec  $a < b$ ) et renvoie un quadruplet constitué des deux indices d'un saut de valeur maximale dans `t` entre les deux indices `a` (inclus) et `b` (exclu) ainsi que les indices d'un minimum et d'un maximum de `t` entre ces deux mêmes indices. Par exemple `saut_max_aux [| 2.0; 5.0; 3.0; 4.0; 6.0; 1.0|] 2 6` doit renvoyer `(2, 4, 5, 4)` car entre les indices 2 (inclus) et 6 (exclu), le saut de valeur maximale est `(2,4)` (de valeur 3.0), le minimum est 1 et le maximum est 6.

*Cette fonction doit être récursive et utiliser la méthode diviser pour régner.*

8. En déduire une fonction de `saut_max_dpr` qui renvoie un saut de valeur maximale du tableau donné en argument.
9. Donner (en les justifiant) les complexités des deux méthodes (naïves et diviser pour régner).