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 librairies 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.
- 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*

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 $[\![0; n-1]\!]$ 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.

   On considère l'ordre lexicographique $\preccurlyeq_L$ sur l'ensemble des $n$uplets d'entiers, on sait que cette ordre est bien fondé. On note $(a_0, \dots, a_{n-1})$ (resp. $(a'_0, \dots, a'_{n-1})$) les éléments du tableau d'entiers avant (resp. après) une itération de la boucle tant que de l'algorithme. Cette itération échange a_i et a_j avec $i < j$ et $a_i > a_j$. Donc, $a_k = a'_k$ pour $k \in [\![0; n-1]\!], k \neq i$ et $k \neq j$, $a'_i = a_j$ et $a'_j = a_i$. Donc $a'_i < a_i$ et puisque $i < j$ c'est la première différence entre ces deux $n$uplets. Donc par défintion de l'ordre lexicographique, $(a'_0, \dots, a'_{n-1}) \preccurlyeq_L (a_0, \dots, a_{n-1})$ c'est donc une quantité strictement décroissante à chaque itération. Ce qui prouve la terminaison de l'algorithme.

2. On définit inductivement (et de façon non ambigüe) 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 $d \in D$.

   On note $P(d)$ la propriété $n_o(d) = n_f(d)$
   - Initialisation : on vérifie la propriété pour les éléments de X_0, comme $X_0 = \epsilon$ et $n_o(\epsilon) = 0$ et $n_f(\epsilon) = 0$. P est vraie pour tous les éléments de X_0.
   - Hérédité : on vérifie la conservation de la propriété par application de chacune des règles d'inférence. Si x et y sont deux mots de Dyck vérifiant P alors $n_o(\langle x \rangle y) = 1 + n_o(x) + n_o(y)$ et $n_f(\langle x \rangle y) = 1 + n_f(x) + n_f(y)$ et puisque x et y vérifient P on a $n_o(x) = n_f(x)$ et $n_o(y) = n_f(y)$ donc $n_o(\langle x \rangle y) = n_f(\langle x \rangle y)$

   Par induction structurelle on conclut que P est vraie pour tout $x \in D$.

❑ 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 $n > 0$.

```
1
2  void int tri(int t[], int n)
3  {
4
5  }
6
7  int *kmax(int [], int n, int k)
8  {
9      int *km = malloc(sizeof(int)*k)
10     tri(t,n);
11     for (int i=0;i<k;i++)
12     {
```

2. Donner en la justifiant brièvement la complexité de cette fonction.

> En notant n la taille du tableau, on effectue uniquement des opérations élémentaires dans une boucle de taille $n-1$ c'est donc un algorithme de complexité linéaire en la taille du tableau n.

3. Afin d'extraire les k premiers maximums, on propose d'extraire successivement les maximums à l'aide de la fontion écrite à la question 1 (pour extraire le kième 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).

> Pour extraire chacun des k maximums on parcourt un tableau d'au plus $n-1$ éléments et donc chaque extraction a une complexité en $O(n)$. L'algorithme a donc une complexité $O(kn)$.

■ **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`.

```
1  int *kmax(int [], int n, int k)
2  {
3      int *km = malloc(sizeof(int)*k)
4      tri(t,n);
5      for (int i=0;i<k;i++)
6      {
7          km[i] = t[i];
8      }
9      return km;
10 }
```

2. Quelle est la complexité de cette solution si la fonction de tri est le tri par sélection ?

> L'algorithme de tri par sélection a une complexité quadratique donc cette solution aura une complexité en $O(n^2)$

3. Quelle est la complexité de cette solution si la fonction de tri est le tri fusion ?

> L'algorithme de tri fusion a une complexité $O(n \log n)$ donc cette solution aura une complexité en $O(n \log n)$

■ **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.

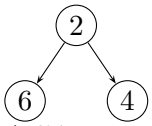
   Un tas est un arbre binaire complet et à chaque étape on remonte (ou on descend) d'un niveau dans cet arbre, la complexité de ces opérations est donc en $O(h)$ où h est la hauteur de l'arbre or l'arbre étant complet $O(h) = O(\log n)$ où n est la taille de l'arbre.

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` :

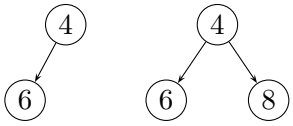
   — si `i < k` , le tas n'est pas plein et on y insère `t[i]`

   — 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 :

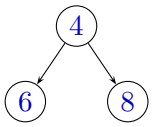
   

   A l'étape suivante, 8 étant plus grand que 2 (le minimum du tas), on extrait 2 du tas et on y insère 8 ce qui donne :

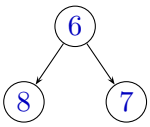
   

   a) Poursuivre le déroulement de cet algorithme en faisant figurer comme ci-dessus les étapes de l'évolution du tas.

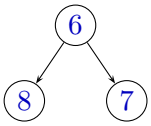
— On traite 3, comme $3 < 4$ pas de modification



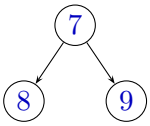
— On traite 7, comme $7 > 4$, on extrait 4 et on insère 7



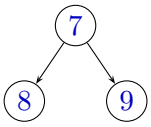
— On traite 1, comme $1 < 6$, pas de modification



— On traite 9, comme $9 > 6$, on extrait 6 et on insère 9



— On traite 5, comme $5 < 7$, pas de modification



b) Prouver que cet algorithme est correct (on pourra utiliser l'invariant valable pour tout `i>=k` : « le tas contient les `k` premiers maximums du sous tableau compris entre les indices `0` et `i-1` »).

On note $P(i)$ la propriété : « le tas contient les `k` premiers maximums du sous tableau compris entre les indices `0` et `i-1` » alors :

— $P(k)$ est vraie car avant d'entrer la boucle le tas contient la tranche de tableau comprise entre les indices 0 et $k-1$ qui sont bien les k premiers maximums de cette tranche.

— En supposant $P(i)$ vraie, montrons $P(i+1)$, l'algorithme compare $t[i+1]$ au minimum du tas qui par hypothèse de récurrence est le plus petit des k premiers maximums de la tranche $0 \dots i$. S'il est plus grand, cela signifie qu'il fait partie des k premiers maximum de la tranche $0 \dots i+1$ et le minimum est extrait et on insère $t[i+1]$. Sinon le tas n'est pas modifié dans les deux cas $P(i+1)$ est vérifiée.

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.

```c
int *kmax_heap(int t[], int n, int k)
{
    heap mh = make_heap(k);
    int *mk = malloc(sizeof(int) * k);
    int temp;
    for (int i = 0; i < k; i++)
    {
        insert_heap(t[i], &mh);
    }
    for (int i = k; i < n; i++)
    {
        if (t[i] > mh.tab[0])
        {
            temp = getmin(&mh);
            insert_heap(t[i], &mh);
        }
    }
    for (int i = 0; i < k; i++)
    {
        mk[i] = mh.tab[i];
    }
    return mk;
}
```

d) Donner en la justifiant la complexité de ce nouvel algorithme en fonction de k et n.

Les opérations d'insertion et d'extraction dans le tas sont toutes en $O(\log k)$ car le tas est de taille k. On effectue ces opérations au plus n fois (une fois pour chaque élément du tableau) et donc la complexité de ce nouvel algorithme est $O(n \log k)$

❑ 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.

```
let caillou_vide chemin_caillou =
  let n = Array.length chemin_caillou in
  let p = ref 0 in
  for i=0 to n-1 do
    if chemin_caillou.(i) = 0 then p := i;
  done;
  !p ;;
```

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`.

```
let echange chemin_caillou i j =
  let copie = Array.copy chemin_caillou in
  let temp = copie.(i) in
  copie.(i) <- copie.(j);
  copie.(j) <- temp;
  copie;;
```

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).

```
let randonneurG_avance chemin_caillou =
  let n = Array.length chemin_caillou in
  let ok = ref false in
  for i=0 to n-2 do
    if (chemin_caillou.(i)=1 && chemin_caillou.(i+1)=0) then ok := true;
  done;
  !ok;;
```

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.

```
let randonneurG_saute chemin_caillou =
    let n = Array.length chemin_caillou in
    let ok = ref false in
    for i=0 to n-3 do
      if (chemin_caillou.(i)=1 && chemin_caillou.(i+2)=0) then ok := true;
    done;
    !ok;;
```

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.

```
let mouvement_chemin chemin_caillou =
  let p = caillou_vide chemin_caillou in
  let mvts = ref [] in
  if (randonneurG_avance chemin_caillou) then (mvts := (echange chemin_caillou p
  ↪  (p-1))::!mvts);
  if (randonneurG_saute chemin_caillou) then (mvts := (echange chemin_caillou p
  ↪  (p-2))::!mvts);
  if (randonneurD_avance chemin_caillou) then (mvts := (echange chemin_caillou p
  ↪  (p+1))::!mvts);
  if (randonneurD_saute chemin_caillou) then (mvts := (echange chemin_caillou p
  ↪  (p+2))::!mvts);
  !mvts;;
```

6. Ecrire 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 |]`.

```
let rec init_chemin ng v1 nd v2 =
  let chemin = Array.make (ng+nd+1) 0 in
  for i = 0 to ng-1 do
    chemin.(i) <-v1 ;
  done;
  for i= ng+1 to ng+nd do
    chemin.(i) <-v2;
  done;
  chemin;;
```

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 |]`.

```
let rec backtrack etat cible =
  if etat = cible then true else
    (try
      let poss = (Array.of_list (mouvement_chemin etat)) in
      for i=0 to Array.length poss -1  do
        if (backtrack poss.(i) cible)  then
          (mvts := poss.(i)::(!mvts);
          raise Exit
          )
      done;
      false
    with Exit -> true
    );;
```

❑ **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 $0 \leq i \leq j < \texttt{n}$ 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. Ecrire 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`).

```
let valeur t i j = t.(j) -. t.(i);;
```

2. Donner un exemple de tableau avec exactement deux sauts de valeur maximale et préciser ces sauts.

> La liste `[2, 6, 1, 5]` possède deux sauts de valeurs maximale : `(0,1)` et `(2,3)` (ces deux sauts ont une valeur de 4)

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.

> Dans la liste `[2, 6, 1, 5]` le minimum est à l'indice 2 (c'est 1) et le maximum à l'indice 1 (c'est 3) et comme le minimum est après le maximum ce n'est pas le saut maximal.

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 $0 \leq i \leq j < \texttt{n}$.

```
let saut_max_naif t =
  let n =  Array.length t in
  let vmax = ref 0. in
  for i=0 to n-1 do
    for j=i to n-1 do
      if (valeur t i j) > !vmax then vmax := (valeur t i j);
    done;
  done;
  !vmax;;
```

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 $j_g < p$
— (i_d, j_d) un saut de valeur maximale lorsque $id \geqslant p$
— (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.

> Un saut de valeur maximal (i, j) du tableau `t` est tel que $i \leqslant j$ et donc on est forcément dans l'une des trois situations suivantes :
>
> — $i \leqslant j < p$ et donc c'est un saut de valeur maximale du sous tableau constitué des éléments d'indice $0 \ldots p-1$
>
> — $p \leqslant i \leqslant j$ et donc c'est un saut de valeur maximale du sous tableau constitué des éléments d'indice $p \ldots n-1$
>
> — $i < p < j$
>
> Ce qui correspond bien aux trois situations décrites 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`).

   On raisonne par l'absurde, si tel n'était pas le cas, on aurait une valeur d'indice q dans la moitié gauche strictement inférieur à $t[i_m]$ et donc le saut (q, j_m) aurait une valeur supérieure au saut (i_m, j_m) ce qui contredit que (i_m, j_m) est un saut de valeur maximale.

7. Ecrire 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.

```
let rec saut_max_aux t a b =
  if b-a=1 then (a,a,a,a) else
  (let n = (a+b)/2 in
    let ig, jg, ming, maxg = saut_max_aux t a n in
    let id, jd, mind, maxd = saut_max_aux t n b in
    let sautg = valeur t ig jg in
    let sautd = valeur t id jd in
    let sautgd = valeur t ming maxd in
    let mint = ref 0 in
    let maxt = ref 0 in
    if t.(ming)<t.(mind) then mint :=ming else mint:=mind;
    if t.(maxg)>t.(maxd) then maxt :=maxg else maxt:=maxd;
    if (sautg > sautd && sautg > sautgd) then (ig, jg, !mint, !maxt) else
      if (sautg > sautgd) then (id, jd, !mint, !maxt) else (ming, maxd, !mint,
      ↪  !maxt));;
```

8. En déduire une fonction de `saut_max_dpr` qui renvoie un saut de valeur maximale du tableau donné en argument.

```
let saut_max_dpr t =
  let i,j, m, n = saut_max_aux t 0 (Array.length t) in
  valeur t i j;;
```

9. Donner (en les justifiant) les complexités des deux méthodes (naïves et diviser pour régner).

   La méthode naïve a une complexité quadratique car on a deux boucles de taille n imbriquées