Devoir en temps limité n°3 – 3h

Calculatrices interdites

On veillera à présenter très clairement sa copie : il faut rédiger les réponses et encadrer les résultats. Pour le code, il doit être indenté, on ne commence pas une fonction en bas de page et on utilise de la couleur pour les commentaires.

Le code doit être commenté dès qu'il dépasse les 5 lignes.

Les fonctions en C et en Ocaml doivent avoir le type précisé. Il est donc recommandé d'utiliser des fonctions auxiliaires en Ocaml.

1 Questions proches du cours

1. Rappeler les primitives qui caractérisent la structure de pile.
2. Rappeler une manière d'implémenter une file vue dans le cours. Quelques phrases d'explication (et pourquoi pas un dessin) sont attendues.
3. Comment implémenter un type permettant de représenter les arbres binaires en Ocaml ?
4. Quelle est la formule récurrente définissant la hauteur d'un arbre binaire ?
5. Écrire en Ocaml une fonction **récursive** qui calcule la somme des éléments d'une liste.
6. Donner sa signature (son type).
7. Montrer la terminaison de votre fonction.
8. Montrer la correction de votre fonction.
9. Calculer la complexité de votre fonction en écrivant la formule de récurrence de sa complexité. **Vous devriez tomber sur un type de suite que vous connaissez très bien**.

2 Un peu de poissons

Dans cette partie, on s'intéresse à une pêcheuse amateure pêchant des poissons pour les vendre.

Un poisson est représenté en Ocaml par le type enregistrement suivant :

```
type poisson = {
  espece : string;
  poids : int; (*poids exprimé en grammes*)
  valeur : int;
};;
```

Rappel : en Ocaml, pour accéder à un champ d'un type enregistrement, on écrit `variable.champ`. Ex : `p.espece`, `p3.valeur`, ...

Par exemple on peut définir un saumon : `let saumon1 = {espece = "saumon"; poids = 5000; valeur = 60};;`, une truite : `let truite1 = {espece = "truite"; poids = 500; valeur = 25};;` et une carpe koi : `let carpekoi1 = {espece = "carpe koi"; poids = 200; valeur = 200};;`.

Le seau de de la pêcheuse (là où elle stocke les poissons) est représenté par un tableau de poissons type `poisson array`. Le seau a une taille maximale de n, qui est déterminée par les quotas en vigueur (il ne faut pas trop pêcher). n est supposé fixe dans le reste de l'exercice.

Pour indiquer le fait que le seau n'est pas complètement rempli, on mettra des faux-poissons : `let faux_poisson = {espece = "faux"; poids = 0; valeur = 0}`.

Par exemple pour $n = 3$, le seau de la pêcheuse peut être `[|saumon1;faux_poisson;faux_poisson|]`. Le seau contient alors un saumon et c'est tout.

On garantira que les vrais poissons sont tous rangés au début du tableau, ainsi dès qu'on voit un faux poisson, on est assurés que le reste du tableau ne contient que des faux poissons.

10. Écrire une fonction `compte_poissons : poisson array -> int` qui compte combien de vrais poissons la pêcheuse a capturé.
11. Écrire une fonction `compte_argent : poisson array -> int` qui compte combien d'argent la pêcheuse va gagner avec son seau de poissons.

En plus d'être limitée par les quotas, la pêcheuse est également limitée par sa capacité à transporter le seau. On fixe donc un poids x tel que, si le seau fait strictement plus que x grammes, la pêcheuse ne peut plus le porter.

On suppose écrite une fonction `compte_poids : poisson array -> int` qui compte le poids total du seau (c'est presque la même fonction que `compte_argent`).

12. Écrire une fonction `peut_porter : poisson array -> int -> bool` qui prend en entrée le seau et x et renvoie vrai si la pêcheuse peut porter le seau et faux sinon.

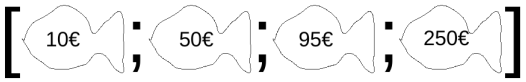
Si la pêcheuse ne peut plus porter son seau, il faut qu'elle retire un poisson pour que le seau pèse moins lourd. Cependant la pêcheuse aimerait gagner autant d'argent que possible, donc elle va essayer de retirer le poisson qui vaut le moins cher.

Par exemple si le seau contient le saumon (5kg), la truite (500g) et la carpe koi (200g) précedemment définis et que x vaut 5.5kg, alors le seau pèse 5.7 kg et retirer n'importe quel poisson permet de repasser en-dessous de la limite. On va donc choisir celui qui vaut le moins cher : la truite.

Pour généraliser ceci on va suivre les étapes suivants :

- Créer une liste l vide.
- Pour chaque poisson du seau, identifier si le retirer permet de repasser en-dessous du poids limite. Si oui :
  - Créer une nouvelle liste l' en ajoutant le poisson à l, en l'insérant de telle sorte à ce que l reste rangée par valeur de poisson croissante.
- Le premier élément de la liste finale est alors le moins cher dont le retrait permet de respecter la limite de poids.

13. Écrire une fonction `insere : poisson list -> poisson -> poisson list` qui prend en entrée la liste l triée selon les valeurs croissantes et un poisson p et range le poisson p dans la liste l à sa place, c'est à dire en préservant le tri par valeurs.
    Par exemple si on veut rajouter un poisson valant 35 euros dans la liste suivante (pour clarifier on écrit que la valeur du poisson et pas son poids ni son nom) :



    On obtient :

14. Compléter la fonction `quoi_retirer : poisson array -> int -> string` suivante qui prend en entrée le seau et x et renvoie le nom du poisson retiré.
    S'il n'est pas possible de ramener le poids du seau en dessous de x en retirant un seul poisson, on fera une erreur.

```
let quoi_retirer seau x =
  let nb_poissons = ... in
  let poids_total = ... in
  let l = ref [] in (*liste des poissons qu'on pourrait retirer*)

  for i = 0 to ... do (*pour chaque poisson*)
    if ... then ...
  done;

  (*Conclure sur quel poisson on va retirer*)
  if ... then failwith "il faudrait retirer plusieurs poissons"
  else ...;;
```

3 Un peu de chainage

Dans cette partie on va étudier en C une structure chainée pour former une grille.

Dans une grille chaque case peut avoir jusqu'à 4 voisins (au nord, au sud, à l'est et à l'ouest). On va donc écrire un type de maillon qui possède 4 pointeurs pointant vers les voisins de la case. Une case contiendra aussi une valeur entière.

```
struct Case {
 int valeur;
 struct Case* nord;
 struct Case* sud;
 struct Case* ouest;
 struct Case* est;
}

typedef struct Case case;
```

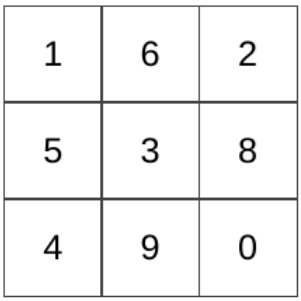


FIGURE 1 – Une grille

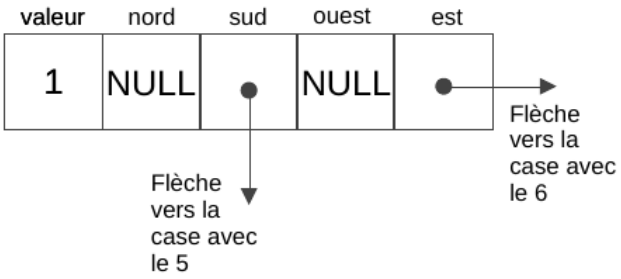


FIGURE 2 – Représentation mémoire de la case (0,0)

Si une case n'a pas de voisine dans une des directions, on mettra le pointeur NULL dans le champ correspondant. Dans l'exemple de la Figure 2, la case (0,0) n'a pas de voisine au nord, donc son champ `nord` vaut NULL.

Dans la suite on confondra la case et le pointeur qui pointe dessus. Ainsi une variable `case*` représente une case.

On rappelle que si `c` est de type `case*`, alors pour accéder à sa voisine `nord` on écrit `c->nord` (qui est de type `case*`) et pour accéder à sa valeur on écrit `c->valeur` (qui est de type `int`).

15. Écrire une fonction `bool est_bord(case* c)` qui détermine si la case `c` est sur un bord de la grille. Indication : les cases sur le bord n'ont jamais 4 voisines.
16. Écrire une fonction `case* nouvelle_case(int v, case* n, case* o, case* s, case* e)` qui crée une nouvelle case dont la valeur est v et qui pointe vers les cases indiquées (n= voisine nord, etc...).

Une grille sera représentée par son coin en nord-ouest (aussi appelé case NO ou case (0,0)) et on supposera qu'une grille est toujours rectangulaire. On notera ses dimensions n et m. On écrit le type `grille` suivant :

```
struct Grille {
 case* coinNO;
}
typedef struct Grille grille;
```

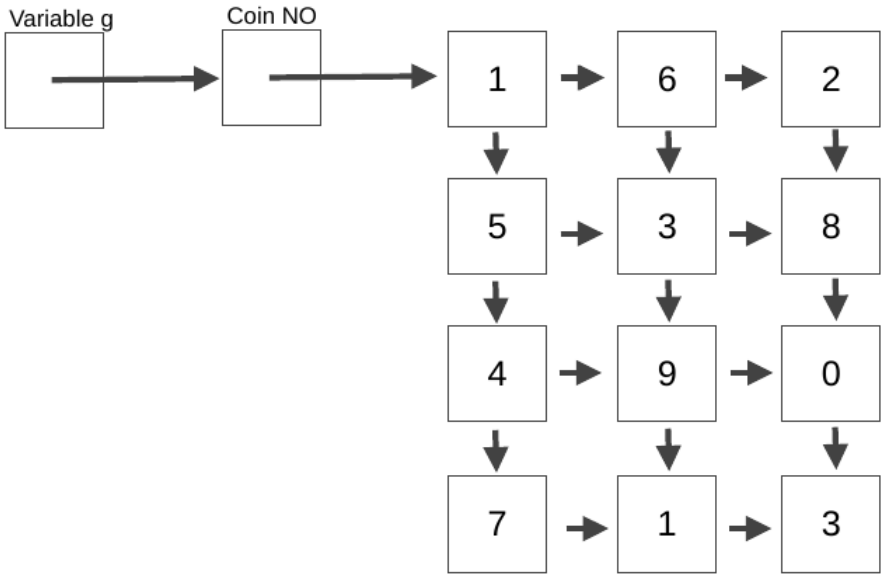


FIGURE 3 – Exemple de grille* g

17. On suppose qu'on dispose du pointeur g de la Figure 3. Quelle instruction permet d'obtenir la valeur de la case (0,1)? De la case (1,2)? Du coin sud-est? La réponse attendue est de la forme `g->machin->bidule->...`.
18. Écrire une fonction `int nb_colonnes(grille* g)` qui calcule le nombre de colonnes de g. Indication : il faut compter combien de fois on peut aller à l'est depuis la case NO.
19. Écrire une fonction `int nb_lignes(grille* g)` qui calcule le nombre de lignes de g.
20. Quelle est la complexité de la fonction précédente?
21. Écrire une fonction `int valeur_case(grille* g, int i, int j)` qui renvoie la valeur de la case (i, j). On utilisera `assert` pour faire une erreur si la case n'existe pas.

22. Quelle est la complexité de la fonction précédente en fonction de i et j ?
23. Écrire une fonction `int somme_grille(grille* g)` qui effectue la somme des éléments de la grille.
24. Quelle est la complexité de la fonction précédente ?

4 Tri avec des files

Le but de cet exercice est de trier une liste d'entiers strictement positifs sans répétitions $L = [s_1; s_2; ...; s_n]$ avec $n \in \mathbb{N}$ en utilisant un réseau de k files en parallèle.

Dans toute la suite, même si on omet de le préciser, les s_i considérées sont des entiers strictement positifs et tous différents.

1. Réseau de files

Un réseau de k files en parallèle est composé de $k + 2$ files :

- Une file **donnée**
- k files intermédiaires numérotées F_1, ..., F_k.
- Une file **résultat**

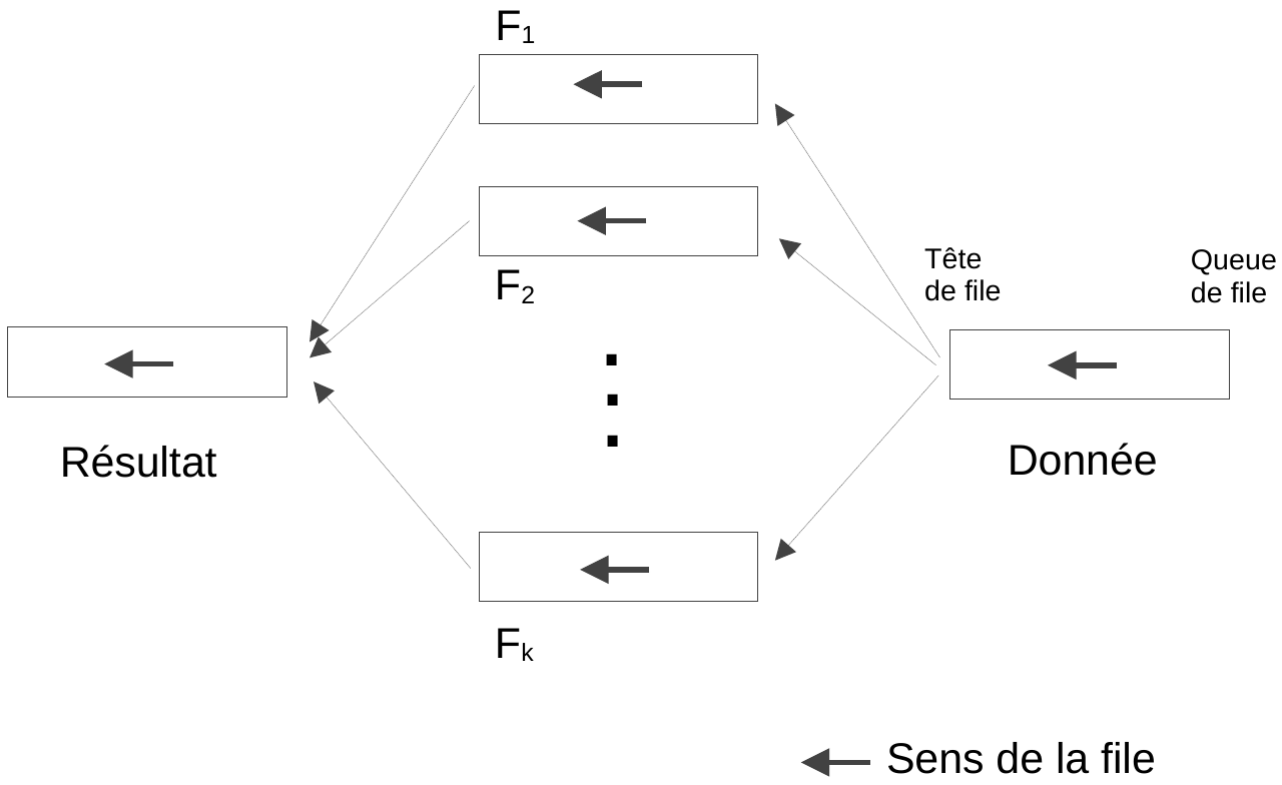


Figure 4 – Illustration d'un réseau à k files

Initialement tous les éléments de la liste sont mis dans la file **donnée** avec s_1 en tête de file et s_n en queue de file.

Ensuite, on déplace les entiers dans le réseau avec un déplacement à la fois et un entier à la fois.

Il y a deux types de déplacement possible :

- les déplacements d'entrée consistent à défiler la file **donnée** et enfiler l'élément s obtenu dans une des files F_i avec $i \in [|1, k|]$ choisi par le programmeur.
  Un tel déplacement est noté $In(i)$.
- les déplacements de sortie consistent à défiler une des files F_i, $i \in [|1, k|]$ et enfiler l'élément s obtenu dans la file **résultat**.
  Un tel déplacement est noté $Out(i)$.

25. Dans cette question on considère $k = 2$. Dessiner l'état final du réseau si on commence avec la liste $L = [3; 1; 0; 2]$ dans la file **donnée** et qu'on effectue les déplacements $In(1), In(1), In(2), Out(2), Out(1)$.
    On représentera les éléments restants dans les 4 files du réseau.

On va programmer le réseau en Ocaml. Pour représenter les déplacements on disposera du type `type deplacement : In of int | Out of int`.

On représentera le réseau à k files en parallèle par un tableau de $k+2$ files. La file **donnée** sera dans la case 0 du tableau. La file F_i avec $i \in [|1, k|]$ sera dans la case i. La file **résultat** sera dans la case $k+1$ du tableau.

Les files seront implémentées avec le module `Queue`. Les primitives sont rappelées en annexe. Un réseau sera donc de type `int Queue.t array` (pour certaines questions, obtenir un `'a Queue.t array` est acceptable).

26. Écrire une fonction `cree_reseau_vide : int -> int Queue.t array` qui prend en entrée k et crée un réseau qui pour le moment ne contient aucun élément.
27. Écrire une fonction `charge_liste : int list -> int Queue.t array -> unit` qui prend en entrée une liste d'éléments et un réseau et met les éléments dans la file **donnée**, dans le bon ordre.
28. Écrire une fonction `execute_sequence : deplacement list -> int Queue.t array -> unit` qui prend en entrée une liste de déplacements et un réseau et effectue les déplacements demandés (In ou Out)

2. Tri de listes

On va utiliser le réseau pour trier une liste en effectuant uniquement des déplacements de la forme In et Out.

Le tri se termine lorsque tous les s_i se retrouvent dans la file **résultat**, triés dans l'ordre croissant de la tête de la file vers la queue de la file.

On appelle scénario de tri une séquence de déplacements qui amène tous les éléments dans la file **résultat** rangés dans l'ordre croissant.

Par exemple, si on considère $k = 2$ et $n = 3$, le scénario $[In(1), In(2), Out(2), Out(1), In(1), Out(1)]$ trie la liste [9 ; 3 ; 20] avec 2 files intermédiaires.

Remarque : on dira qu'une file est croissante (resp. décroissante) si les éléments qu'elle contient, considérés dans l'ordre de leur arrivée en commençant par le plus ancien (le prochain à sortir), sont rangés dans l'ordre croissant (resp. décroissant).

29. En utilisant $k = 1$ files, donner un scénario qui permet de trier la liste [1 ;2].
30. En utilisant $k = 3$ files, donner un scénario qui permet de trier la liste [3 ;5 ;2 ;7 ;1 ;8 ;9].
31. Combien de déplacements contient un scénario de tri ? Justifier.
32. Justifier que pour tout scénario de tri T, on peut construire un scénario de tri T' qui utilise les mêmes déplacements mais où tous les In sont faits avant les Out.

Dans la suite on supposera que tous les scénarios de tri effectuent les In avant les Out.

33. Montrez qu'à chaque étape d'un scénario de tri, chacune des files intermédiaires (les F_i) est soit vide, soit triée dans l'ordre croissant.
34. Déduisez-en que, dans un scénario de tri, deux éléments s_i et s_j tels que $i < j$ mais $s_i > s_j$ ne peuvent pas aller dans la même file intermédiaire. Montrez que si L contient une sous-séquence décroissante de longueur m, avec $m \geq 2$, il faut au moins m files en parallèle pour trier L.

    Remarque : une sous-séquence de $[s_1; ...; s_n]$ est une liste de $m \leq n$ éléments $[s_{i_1}, ...s_{i_m}]$ telle que $1 \leq i_1 < ... < i_m \leq n$. C'est très similaire à l'idée d'une suite extraite, appliquée dans le cadre d'une "suite finie" (la liste).
35. En exploitant les propriétés remarquées dans les questions précédentes, donner un algorithme en pseudo-code permettant de trouver une scénario de tri pour une liste L quelconque.
    Il est interdit d'utiliser d'autres structures files et piles dans l'algorithme. Il est permis d'utiliser des variables numériques, des listes et des tableaux si besoin.

5 Annexe

Module Queue

- `Queue.create : unit -> 'a Queue.t` qui crée une file vide
- `Queue.is_empty : 'a Queue.t -> bool` qui teste si une file est vide
- `Queue.push : 'a -> 'a Queue.t -> unit` qui ajoute un élément
- `Queue.pop : 'a Queue.t -> 'a` qui retire et renvoie l'élément le plus ancien
- `Queue.peek : 'a Queue.t -> 'a` qui renvoie sans retirer l'élément le plus ancien