Devoir en temps limité n°3 – 3h

Calculatrices autorisées

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.

1 Questions de cours

1. *Rappeler les primitives qui caractérisent la structure de pile.*
   Les primitives sont créer une pile vide, dépiler l'élément le plus récent, empiler un élément, tester si la pile est vide et détruire la pile.
2. *Rappeler une manières d'implémenter une file vues dans le cours. Quelques phrases d'explication (et pourquoi pas un dessin) sont attendues.*
   On pouvait proposer d'implémenter une file dans un tableau circulaire avec deux indices ou avec deux piles (on a vu encore d'autres méthodes en TD et TP).
3. *Comment implémenter un type permettant de représenter les arbres binaires en Ocaml ?*
   ```
   type 'a arbrebin = Vide|N of 'a arbrebin * 'a * 'a arbrebin;;
   ```
4. *Quelle est la formule récursive définissant la hauteur d'un arbre binaire ?*
   Hauteur(Vide) = -1 Hauteur(N(g,x,d))=1+max(Hauteur(g),Hauteur(d))
5. *Écrire en Ocaml une fonction **récursive** qui calcule la somme des éléments d'une liste.*
   On considère ici que la somme des éléments de la liste vide est 0. C'est une convention.
   ```
   let rec somme l = match l with
     |[] -> 0
     |t::q -> t+(somme q);;
   ```
6. *Donner sa signature (son type).* `int list -> int`
7. *Montrer la terminaison de votre fonction.*
   La longueur de la liste l est un variant des appels récursifs. En effet c'est une quantité entière et positive et l'appel récursif est effectué sur q la queue, dont la longueur est par définition la longueur de l moins 1.
8. *Montrer la correction de votre fonction.*
   On doit montrer la correction pour le cas de base et montrer que pour une entrée l donnée, si l'appel sur q est correct alors l'appel sur l aussi.
   Cas de base : si $l = []$ on renvoie 0. On a dit que par convention c'était bien la somme de la liste vide.
   Supposons que l'appel récursif sur q soit correct et renvoie la somme des éléments de q. Alors notre fonction renvoie t plus la somme des éléments de q, ce qui donne la somme des éléments de l.
   La fonction est donc correcte.
9. *Calculer la complexité de votre fonction en écrivant la formule de récurrence de sa complexité.* On note $C(n)$ la complexité de lap fonction sur une liste de taille n.
   On a alors $C(0) = 1$ et $C(n) = 1 + C(n-1)$.
   Cette suite est arithmétique. SOn expression générale est $C(n) = 1+n$. On en déduit que notre fonction a une compleixté asymptotique linéaire.

2 Un peu de poissons

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.

10. *Écrire une fonction* `compte_poissons : poisson array -> int` *qui compte combien de vrais poissons la pêcheuse a capturé.*
    ```
    let compte_poissons seau =
      let nb = ref 0 in
      let n = Array.length seau in
      while !nb<n && seau.(!nb) <> faux_poisson do (*On s'arrête quand on atteint la fin ou un faux poisson*)
        nb:=!nb+1
      done;
      !nb;;
    ```

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

```
let compte_argent seau =
  let res = ref 0 in
  let n = compte_poissons seau in
  for i=0 to n-1 do (*Additionner les valeurs des vrais poissons*)
    res:=!res+seau.(i).valeur
  done;
```

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.*

```
let peut_porter seau x =
  let poids = compte_poids seau in
  if poids>x then false
  else true;;
```

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.*

```
let rec  insere l p = match l with
  |[] -> p
  |t::q when t.valeur > p.valeur -> p::l
  |t::q -> t::(insere p q);;
```

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 = compte_poissons seau in
  let poids_total = compte_poids seau in
  let l = ref [] in (*liste des poissons qu'on pourrait retirer*)

  for i = 0 to nb_poissons-1 do (*pour chaque poisson*)
    if poids_total - seau.(i).poids <= x then l:=insere !l seau.(i)
  done;

  (*Conclure sur quel poisson on va retirer*)
  if !l=[] then failwith "il faudrait retirer plusieurs poissons"
  else (List.hd !l).nom;;
```

3 Un peu de chainage

15. *Écrire une fonction* `bool est_bord(case* c)` *qui détermine si la case* `c` *est sur un bord de la grille.*

```
bool est_bord(case* c){
  if (c->nord==NULL||c->sud==NULL||c->est==NULL||c->ouest==NULL){return true;}
  return false;
}
```

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...).*

```
case* nouvelle_case(int v, case* n, case* o, case* s, case* e){
  case* res = malloc(sizeof(case));
  res-> valeur = v;
  res->nord = n;
  res->sud = s;
  res->ouest = o;
  res->est=e;
  return res;
}
```

17. *On suppose qu'on dispose du pointeur g de la Figure 3. Comment obtenir la valeur de la case (0,1) ? De la case (1,2) ? Du coin sud-est ?*
    Pour la case (0,1) : `g->coinNO->est` (à l'est de la case NO)
    Pour la case (1,2) : `g->coinNO->est->est->sud` (il faut aller deux fois à l'est et une fois au sud, l'ordre n'est pas important)
    Pour le coin sud-est : `g->coinNO->est->est->sud->sud->sud`.
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.*

```
int nb_colonnes(grille* g){
  int res = 0
  case* case_actuelle = g->coinNO;
  while (case_actuelle!=NULL){
    res+=1;
    case_actuelle = case_actuelle->est;
  }
  return res;
}
```

19. *Écrire une fonction* `int nb_lignes(grille* g)` *qui calcule le nombre de lignes de g.*
    C'est pareil en allant vers le sud plutôt que l'est.
20. *Quelle est la complexité de la fonction précédente ?*
    La boucle while s'effectue autant de fois que le nombre de lignes. Dans la boucle while on ne fait que des opérations élémentaires. La fonction est linéaire en le nombre de lignes.
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.*
    Le principal piège dans cette question est qu'on utilise des boucles séparées : on se balade vers le sud i fois puis vers l'est j fois.

```
int valeur_case(grille* g, int i, int j)
  case* case_actuelle = g->coinNO;
  for(int x=0; x<i; x+=1){
    assert (case_actuelle!=NULL);
    case_actuelle = case_actuelle->sud;
  }
  for(int y=0; y<j; y+=1){
    assert (case_actuelle!=NULL);
    case_actuelle = case_actuelle->est;
  }
  assert (case_actuelle!=NULL);
  return case_actuelle->valeur;
}
```

22. *Quelle est la complexité de la fonction précédente en fonction de i et j ?*
    On a une boucle qui fait $i - 1$ tours (avec que des opérations élémentaires) et une boucle qui fait $j - 1$ tours (avec que des opérations élémentaires. La complexité finale est en $O(i + j) = O(max(i, j))$.
23. *Écrire une fonction* `int somme_grille(grille* g)` *qui effectue la somme des éléments de la grille.*
    On parcourt ligne par ligne en allant vers l'est autant que possible. Quand on arrive en bout de ligne, on va une fois au sud et autant de fois que possible à l'ouest et on recommence en sommant les cases au fur et à mesure.

```
int somme_grille(grille* g){
  int res=0;
  case* case_actuelle = g->coinNO;

  int nb_lignes = nb_lignes(g);
  int nb_colonnes = nb_colonnes(g);
  //Parcours des lignes
  for(int i=0;i<nb_lignes;i+=1){

    //Aller vers l'est en sommant
    for(int j=0;j<nb_colonnes;j+=1){
      res+=case_actuelle.valeur;
      case_actuelle=case_actuelle->est;
    }

    //Fin de la ligne, retourner en (i,0) en allant à l'ouest sans sommer
```

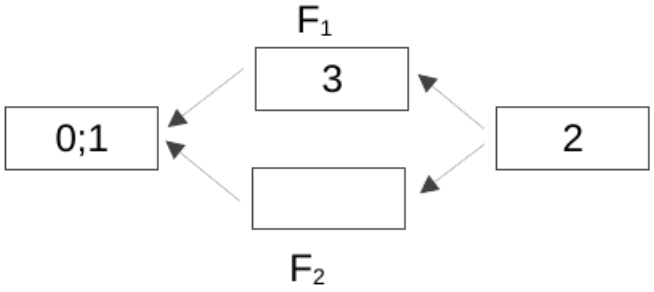
```
    for(int j=0;j<nb_colonnes;j+=1){
      case_actuelle=case_actuelle->ouest;
    }
    //Aller une fois au sud pour la ligne suivante
    case_actuelle = case_actuelle->sud;
  }
```

24. *Quelle est la complexité de la fonction précédente ?*
    On a deux boucles for imbriquées qui effectuent des opérations élémentaires. En notant n le nombre de ligne set m le nombre de colonnes, la complexité est en $O(n \times m)$.

4 Tri avec des files

1. Réseau de files

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



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.*
    Attention, les files crées dans le Array.make ci-dessous ne sont aps indépendantes, c'est pourquoi on les remplace (même problème qu'avec les matrices)

```
let cree_reseau_vide () =
  let reseau = Array.make (k+2) (Queue.make ()) in
  for i = 0 to n-1 do
    reseau.(i)<-Queue. make ()
  done;
  reseau;;
```

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**.*

```
let rec charge_liste l reseau = match l with
  |[] -> ()
  |t::q -> Queue.push t reseau.(0); charge_liste q;;
```

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)*

```
let rec execute_sequence l reseau = match l with
  |[] -> ()
  |In(i)::q -> Queue.push (Queue.pop reseau.(0)) reseau.(i); execute_sequence q
  |Out(i)::q -> let kplus2 = Array.length reseau in Queue.push (Queue.pop reseau.(i)) reseau.(kplus2-1);
                execute_sequence q
```

2. Tri de listes

29. *En utilisant $k = 1$ files, donner un scénario qui permet de trier la liste [1 ;2]. $In(1), In(1), Out(1), Out(1)$*
30. *En utilisant $k = 3$ files, donner un scénario qui permet de trier la liste [3 ;5 ;2 ;7 ;1 ;8 ;9].*
    $In(1), In(1), In(2), In(1), In(3), Out(3), Out(2), Out(1), Out(1), Out(1), In(1), In(1), Out(1), Out(1)$.
31. *Combien de déplacements contient un scénario de tri ? Justifier.*
    Un scénario de tri contient toujours $2n$ déplacements (on rappelle que $n = |L|$). En effet chaque élément subi un seul déplacement d'entrée et un seul déplacement de sortie.

32. *Justifier que pour tout scénario T, on peut construire un scénario T' qui utilise les mêmes déplacements mais où tous les In sont faits avant les Out.*

    Si on a dans le scénario un $Out(i)$ puis un $In(j)$ alors :

    - Si $i \neq j$ échanger les deux déplacements ne change rien : ce qui se passe dans la file i ne peut pas affecter ce qui se passe dans la file j et réciproquement.
    - Si $i = j$, alors étudions ce qui se passe si on échange les deux mouvement. On note e l'élément en tête de la file F_i juste avant l'exécution de $Out(i)$ dans le scénario original. On note e' l'élément en tête de la file **donnée** juste avant l'exécution de $In(i)$ dans le scénario original. On remarque que e' est également en tête de **donnée** juste avant l'exécution de $Out(i)$ dans le scénario original

      Scénario original : on défile e et on met e' dans F_i.

      Scénario modifié : on met e' dans F_i et c'est toujours e qui est défilé car la file est une structure FIFO.

      Conclusion : inverser deux évènement Out et In consécutifs ne change rien. En répétant ce échanges qui ne changent rien au comportement du scénario suffisament de fois, on arrive à une situation où le scénario fait tous 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.*

    Pour que le scénario se termine, chaque file sauf la file **résultat** doit être vide. Donc tout élément qui est à un moment dans une des files intermédiaires est à un autre moment mis dans résultat.

    Supposons que durant l'algorithme il existe un instant où une certaine file F_i n'est pas triée dans l'ordre croissant, c'est à dire qu'elle contient deux éléments e et e' tels que e a été ajouté avant e' dans la file mais $e' < e$. Alors ces deux élements vont être mis dans **résultat** et pour respecter la propriété FIFO, e apparaitra avant e'. Donc la file **résultat** ne sera pas triée à la fin, ce qui est absurde, le scénario proposé est incorrect.

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 $m \in \mathbb{N}^*$, il faut au moins m files en parallèle pour trier L.*

    Si on met deux éléments s_i et s_j tels que $i < j$ mais $s_i > s_j$ dans la même file intermédiaire, celle-ci n'est pas croissante. De plus s_i ne peut être lo'bjet d'un déplacement de sortie avec s_j, sinon **résultat** ne serait pas triée. D'après la question précédente, aucun scénario de tri ne peut fonctionner ainsi.

    Montrons maintenant par récurrence sur m que si L contient une sous-séquence décroissante de longueur m, avec $m \in \mathbb{N}^*$, il faut au moins m files en parallèle pour trier L, peut importe le nombre total k de files.

    On initialise pour $m = 2$. On a donc deux éléments s_i et s_j tels que $i < j$ mais $s_i > s_j$. Comme on vient de le dire, ces deux éléments ne peuvent pas aller dans la même file. Il nous faut donc au moins 2 files pour trier L.

    Supposons la propriété vraie pour $m < n$. Soit L contenant une sous-séquence décroissante de longueur $m+1$: $s_{i_1}, \dots s_{i_m}, s_{m+1}$.

    Comme trier plus d'éléments ne peut qu'utiliser des files en plus et jamais des files en moins, on va se ramener au cas où L ne contient que $s_{i_1}, \dots s_{i_m}, s_{m+1}$.

    s_{i_1} va à un moment être ajouté à une file, sans perte de généralité on peut considérer que c'est F_1. Il ne peut pas en ressortir avant que tous les autres éléments de la sous-séquence aient été mis dans résultat et aucun autre élément de la sous-séquence ne peut être mis dans F_1 d'après la question précédente.

    On va donc faire comme si F_1 n'existait plus.

    Ramenons-nous au cas où on a $k-1$ files (on ignore F_1) et la liste à trier est $[s_{i_2}, \dots s_{i_m}, s_{m+1}]$. Il s'agit d'une liste contenant une sous-séquence décroissante de taille m, donc par hypothèse de récurrence il faut au moins m files pour la trier.

    Finalement il nous faut au moins $m+1$ files pour trier L.

35. *En utilisant les propriétés déterminées dans les questions précédentes, donner un algorithme en pseudo-code permettant de trier L avec le réseau.*

    L'idée poussée par les observations précédentes est qu'il faut faire attention aux sous-séquences décroissantes et maintenir les files intermédiaires dans un ordre croissant.

    On va donc essayer de toujours utiliser le moins de files possibles pour avoir des files disponibles pour y mettre des éléments de sous-séquences décroissantes quand on les rencontre.

    Pour savoir quels éléments peuvent être mis sans danger dans quelle file, on peut garder un tableau indiquant le dernier élément enfilé dans chaque file intermédiaire. Les files intermédiaires devant toujours rester croissantes, cela nous suffit.

    Enfin, grâce à la question 32 on sait qu'on peut d'abord faire tous les In puis tous les Out, ce qui évite des vérifications.

    Pseudo code :

    ```
    Créer le réseau.
    Charger la liste dans donnée.
    ```

```
Créer un tableau der rempli de -1 de taille k.
Pour i allant de 1 à n :
    Défiler un élément e de donnée
    Regarder s'il existe une file intermédiaire non vide i telle que e est plus grand que der[i].
    Si ça n'existe pas , mettre e dans la première file vide
    Mettre à jour der
Pour i allant de 1 à n :
    Trouver la plus petite tête parmi toutes les files intermédiaires
    Défiler et enfiler dans résultat
```