DS 2

Option informatique, première année

Julien Reichert

Questions de cours :

Question 1 : Comment introduit-on une fonction récursive en Caml ?

Question 2 : Expliquer en quelques lignes, avec au moins un exemple incorporant un calcul de complexité, le principe du paradigme « Diviser pour régner ».

Question 3 : Décrire brièvement la structure de dictionnaire en listant les opérations élémentaires sur cette structure avec leur signature (plusieurs signatures sont possibles, il faudra qu'elles soient cohérentes entre elles et avec la description).

Exercices :

Exercice 1 : Afin de produire un certain nombre de copies d'un passage, il est possible de sélectionner une partie et de copier-coller. Écrire un programme dynamique (même si une solution directe est possible) déterminant pour toute valeur le nombre minimal d'opérations nécessaires pour créer toutes ces copies, une opération pouvant être soit l'écriture d'un élément, soit la copie de l'ensemble des éléments dans la mémoire, soit le collage de la mémoire.

Indication : le tableau mémorisant le nombre optimal d'opérations devra être à double entrée, un pour le nombre d'éléments disponibles et un pour la taille de la mémoire. Par ailleurs, il n'est pas demandé que le programme retourne la liste des opérations nécessaires dans l'ordre (suggestion de travail pour ceux qui ont terminé en avance).

Pour information, il s'agit de la suite A005245 de l'encyclopédie OEIS.

Exercice 2 : Implémenter la structure de pile d'entiers sous la forme d'un type enregistrement comme précisé ci-après. La tête d'une pile vide peut être n'importe quel entier. Quatre opérations de base suffisent, sans écrire celles qui s'en déduisent.

```
type pile = { mutable vide : bool; mutable tete : int; mutable reste : int list };;
```

Exercice 3 : Implémenter la structure de file d'entiers sous la forme d'un type enregistrement comme précisé ci-après. Les informations associées sont la taille et la position de la tête. La fonction de création fournit la capacité de la file, et les ajouts et retraits peuvent être gérés comme dans la version du cours, afin de limiter la complexité.

```
type file = { mutable taille : int; mutable tete : int; elements : int array };;
```

Exercice 4 : Expliquer la structure de données implémentée ci-après ainsi que les programmes associés.

```
type 'a elem_ldc = { valeur : 'a; mutable suivant : 'a elem_ldc option;
    mutable precedent : 'a elem_ldc option };;

type 'a ldc = { mutable vide : bool; mutable debut : 'a elem_ldc option;
    mutable fin : 'a elem_ldc option };;

let creer_ldc () = { vide = true; debut = None; fin = None };;

let ajouter_debut_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.suivant <- l.debut;
      match l.debut with
      | Some el -> el.precedent <- Some ebis
      | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.fin <- Some ebis );
  l.debut <- Some ebis;;

let ajouter_fin_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.precedent <- l.fin;
      match l.fin with
      | Some el -> el.suivant <- Some ebis
      | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.debut <- Some ebis; );
  l.fin <- Some ebis;;

let retirer_debut_ldc l =
  match l.vide, l.debut with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.debut <- ebis.suivant;
      (match l.debut with
      | None -> l.vide <- true; l.fin <- None;
      | Some deb -> deb.precedent <- None);
  e;;

let retirer_fin_ldc l =
  match l.vide, l.fin with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.fin <- ebis.precedent;
      (match l.fin with
      | None -> l.vide <- true; l.debut <- None;
      | Some fi -> fi.suivant <- None);
  e;;
```