Master BioInformatique	ANNÉE : 2011/2012 SESSION DE DÉCEMBRE 2011 **PARCOURS :** Master 1 **UE :** Algorithmes et structures de données **Épreuve :** Examen **Date :** Vendredi 16 décembre 2011 **Heure :** 10 heures **Durée :** 2 heures Documents : autorisés Épreuve de M. Alain GRIFFAULT

SUJET + CORRIGE

Avertissement

- La plupart des questions sont indépendantes, et le barème total est de 22 points.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Exercice 1 (Files à l'aide de Piles (8 points))
Nous avons vu en cours une implémentation d'un pile par un tableau borné.

```
CreerPileVide(N){
  P.T : objet[N];   // T est un tableau de N "objets"
  P.sommet <- -1;    // sommet est l'indice du dernier objet depose.
  retourner P;
}

PileVide(P){
  retourner (P.sommet < 0);
}

PilePleine(P){
  retourner (P.sommet = P.T.longueur -1);
}

SommetPile(P){
  si (non PileVide(P))
  alors
    retourner P.T[P.sommet];
  sinon
    Ecrire "Impossible, la pile est vide";
}

Empiler(P,X){
  si (non PilePleine(P))
  alors {
    P.sommet <- P.sommet + 1;
    P.T[P.sommet] <- X;
  }
  sinon
    Ecrire "Impossible, la pile est pleine";
}

Depiler(P){
  si (non PileVide(P))
  alors
    P.sommet <- P.sommet - 1;
  sinon
    Ecrire "Impossible, la pile est vide";
}
```

Nous avons également vu en cours une implémentation d'une file par un tableau circulaire. Dans cet exercice, nous allons implémenter une file de taille N à l'aide de deux piles de taille N. L'idée est la suivante : Le sommet de la première pile correspond à l'avant de la file, tandis que le sommet de la seconde pile correspond à l'arrière de la file. Lorsque la première pile est vide, la seconde peut être "retournée" sur la première. Ce retournement est utilisé pour retirer l'élément en tête de file lorsque la pile correspondant à l'avant est vide. Ainsi la création d'une file s'écrit :

```
CreerFile(N){
  F.Tete <- CreerPile(N);   // le sommet de F.Tete est la tete de la file F
  F.Queue <- CreerPile(N);  // le sommet de F.Queue est la queue de la file F
  retourner F;
}
```

Les tests de file vide et de file pleine s'écrivent :

```
FileVide(F){
  // La file est vide si les deux piles sont vides.
  retourner PileVide(F.tete) && PileVide(F.Queue);
}

FilePleine(F){
  // la file est pleine si la pile de queue est pleine (c'est un choix)
  retourner PilePleine(F.Queue);
}
```

La tete de file est obtenue par :

```
TeteFile(F){
  si (non FileVide(F))
  alors {
    si (non PileVide(F.Tete))
    alors
      retourner SommetPile(F.Tete);
    sinon
      retourner F.Queue.T[0];
  }
  sinon
    Ecrire "Impossible, la file est vide";
}
```

Question 1.1 (2 points) *Complétez le code de la primitive* `Enfiler(F)`.

Réponse :

```
Enfiler(F,X){
  // si la file n'est pas pleine, les objets sont deposes dans la pile de queue.
  si (non FilePleine(F))
  alors
    Empiler(F.Queue,X);
  sinon
    Ecrire "Impossible, la file est vide";
}
```

Question 1.2 (3 points) *Complétez le code de la primitive* `Defiler(F)`.

Réponse :

```
Defiler(F){
  // Il est possible de defiler si la file n'est pas vide.
  // Si la pile de tete est vide,
  //   - il faut "retourner" la file de queue dans la file de tete
  // ainsi, la pile de tete n'est pas vide et il est possible de "defiler"
  si (non FileVide(F))
  alors {
```

```
      si (PileVide(F.Tete))
      alors{ // Il faut "retourner" la pile de queue dans la pile de tete.
        tant que (non PileVide(F.Queue)) faire {
          Empiler(F.Tete, SommetPile(F.Queue));
          Depiler(F.Queue);
        }
      }
      Depiler(F.Tete);
    }
    sinon
      Ecrire "Impossible, la file est vide";
}
```

Question 1.3 (1 points) *Donnez une suite d'appels contenant des* `Enfiler(F,X)` *et des* `Defiler(F)` *possible avec une implémentation d'une file de taille 3 à l'aide de deux piles de taille 3, et qui n'est pas possible avec l'implémentation vue en cours.*

Réponse : *Avec les deux piles de taille 3, il est possible d'avoir plus de 3 objets simultanément dans la file. La séquence suivante est impossible avec une file de taille 3 implémentée avec un tableau circulaire.*
`Enfiler(F,X);Enfiler(F,Y);Enfiler(F,Z);Defiler(F);Enfiler(F,T);Enfiler(F,U);`

Question 1.4 (2 points) *Donnez une nouvelle version de la primitive* `FilePleine` *pour corriger le problème de la question précédente.*

Réponse :

```
FilePleineTailleN(F){
  // la file est pleine si le nombre d'objets dans les deux piles est de N
  retourner ((F.Tete.sommet + 1) + (F.Queue.sommet + 1) = F.Tete.longueur);
}
```

Exercice 2 (Tri par base (8 points))
Nous avons vu en cours de nombreux algorithmes pour trier des objets contenus dans un tableau.
Dans cet exercice, nous allons implémenter un nouvel algorithme de tri. Cet algorithme a été inventé il y a bien longtemps pour trier les cartes perforées. Nous allons l'écrire pour trier des personnes suivant leur date de naissance. Pour cela nous allons supposer que les personnes sont stockées sous forme de tableaux.

```
// Une personne est representee par sa date de naissance.
// Personne[0] est le jour de naissance.
// Personne[1] est le mois de naissance.
// Personne[2] est l'annee de naissance.
Personne : Entier[3];
```

Question 2.1 (2 points) *Écrivez une fonction* `NeAvant(P1,P2)` *de comparaison de deux personnes* `P1` *et* `P2` *qui retourne* `Faux` *si* `P2` *est né strictement après* `P1`*, et* `Vrai` *sinon. (Pensez aux prédicats !)*

Réponse :

```
NeAvant(P1, P2){
  retourner ((P1[2] < P2[2]) ||
             ((P1[2] = P2[2]) && (P1[1] < P2[1])) ||
             ((P1[2] = P2[2]) && (P1[1] = P2[1]) && (P1[0] <= P2[0])));
}
```

Cette fonction `NeAvant(P1,P2)` *peut être utilisée dans n'importe lequel des algorithmes de tri vus en cours pour trier un tableau de N personnes suivant leur date de naissance. Il suffit de remplacer les tests* `T[i]<=T[j]` *par* `NeAvant(T[i],T[j])`.
Un algorithme de tri est dit "stable" si l'ordre des indices de deux valeurs égales est inchangé dans le tableau trié. Par exemple si $T[5] = T[36]$ dans le tableau non trié, alors si i et j sont les nouveaux indices de $T[5]$ et de $T[36]$ dans le tableau trié, alors $i < j$.
Dans la suite, on acceptera les propriétés suivantes :

1. *Les algorithmes vus en cours de tri par insertion, par sélection, à bulles et par fusion sont des algorithmes stables.*

2. Les algorithmes vu en cours de tri rapide et de tri par tas ne sont pas stables.

L'idée du tri par base appliquée au date de naissance est d'effectuer séquentiellement trois tris :

1. *Trier (avec un tri stable) suivant le jour de naissance.*
2. *Trier (avec un tri stable) suivant le mois de naissance.*
3. *Trier (avec un tri stable) suivant l'année de naissance.*

Question 2.2 (2 points) *Soit le tableau suivant de sept personnes avec leurs dates de naissances.*

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
jour	*8*	*13*	*9*	*21*	*8*	*16*	*13*
mois	*9*	*2*	*2*	*10*	*7*	*4*	*6*
année	*1977*	*1984*	*1980*	*1984*	*1990*	*1979*	*1981*

Complétez les tableaux suivants après chacune des trois étapes.

Réponse :

Après le tri stable des jours de naissance.

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
jour	*8*	*8*	*9*	*13*	*13*	*16*	*21*
mois	*9*	*7*	*2*	*2*	*6*	*4*	*10*
année	*1977*	*1990*	*1980*	*1984*	*1981*	*1979*	*1984*

Après le tri stable des mois de naissance.

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
jour	*9*	*13*	*16*	*13*	*8*	*8*	*21*
mois	*2*	*2*	*4*	*6*	*7*	*9*	*10*
année	*1980*	*1984*	*1979*	*1981*	*1990*	*1977*	*1984*

Après le tri stable des années de naissance.

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
jour	*8*	*16*	*9*	*13*	*13*	*21*	*8*
mois	*9*	*4*	*2*	*6*	*2*	*10*	*7*
année	*1977*	*1979*	*1980*	*1981*	*1984*	*1984*	*1990*

Question 2.3 (1 point) *Expliquez ou bien montrer sur un exemple pourquoi le tri par base n'est pas correct si le tri utilisé n'est pas stable.*

Réponse : *Par exemple, la dernière étape peut donner avec un tri non stable :*

Après un tri **non stable** *des années de naissance.*

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
jour	*8*	*16*	*9*	*13*	**21**	**13**	*8*
mois	*9*	*4*	*2*	*6*	**10**	**2**	*7*
année	*1977*	*1979*	*1980*	*1981*	**1984**	**1984**	*1990*

Ce qui démontre que le tri base doit n'utiliser que des tris stables.

Question 2.4 (3 points) *Complétez le code suivant du tri par base, en utilisant l'algorithme de tri de votre choix comme tri stable. Vous préciserez les critères de comparaison utilisés.*

Réponse :

```
TriBase(T){
  // T est un tableau de Personnes.
  si (T.longueur > 1)
  alors {
    pour clef de 0 a 2 faire {
      TriStable(T, clef);
  }
}

TriStable(T, clef) est soit :
  - le tri par insertion ,
  - le tri par selection ,
  - le tris a bulles ,
  - le tri par fusion
en utilisant T[i][clef] pour comparer les personnes.
```

Question 2.5 (2 points) *Donnez et justifiez la complexité de votre algorithme.*

Réponse : *Le tri appelle un nombre de fois déterminé (ici 3) un algorithme de tri stable. La complexité du tri par base est donc celle du tri stable utilisé soit :*
– $\Theta(n^2)$ *pour les tris par sélection, insertion, à bulles.*
– $\Theta(nlog_2(n))$ *pour le tri par fusion.*
Quelque fois, les clefs des différentes étapes d'un tri base sont bien connues et sur un petit domaine fini. Pour notre exemple, les jours sont entre 0 et 31, les mois entre 0 et 12. Pour ce type de clef, on peut également envisager d'utiliser un tri par dénombrement, qui est stable, qui est de complexité linéaire mais qui demande plus de mémoire. Lorsque toutes les clefs sont sur des petits domaines, il est possible d'avoir une complexité $\Theta(n)$ *en utilsisant un tri par dénombrement stable.*

Exercice 3 (Récursivité et complexité (4 points))

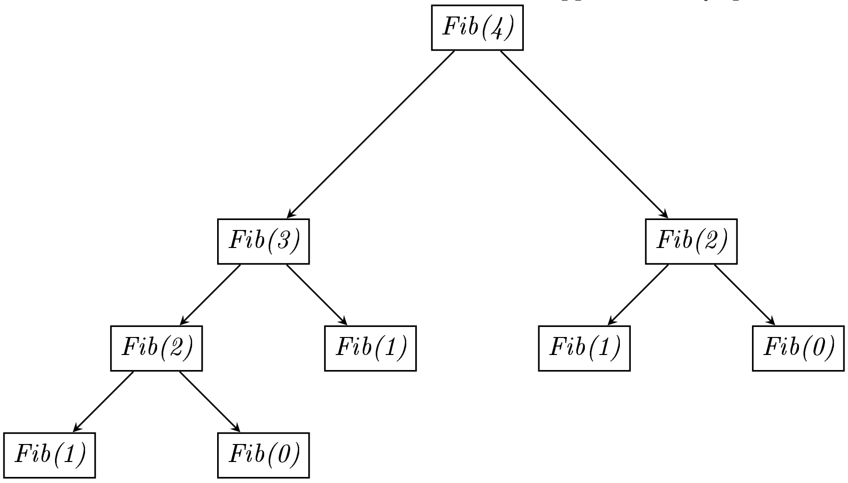
Question 3.1 (2 points) *Soit l'algorithme récursif qui implémente la fonction de Fibonacci.*

```
Fibonacci(n){
  si (n>1)
  alors
    retourner Fibonacci(n-1) + Fibonacci(n-2);
  sinon
    retourner 1;
}
```

Dessinez l'arbre des appels pour `Fibonacci(4)`, *puis donnez le nombre d'appels récursif pour le calcul de* `Fibonacci(n)`.

Réponse :

FIGURE *1 – Les appels récursifs pour Fibonacci(4)*



La seule constante utilisée est 1 *(c'est la valeur de* $Fib(0)$ *et de* $Fib(1)$*), et les seules opérations sont des additions. Le nombre de feuilles dans l'arbre (qui valent soit* $Fib(0$ *soit* $Fib(1)$ *) est donc* $Fib(n)$*. Le nombre de noeuds est donc* $Fib(n) - 1$*. Le nombre d'appels récursifs est donc* $2Fib(n) - 1$*.*
Une autre réponse intéressante est la suivante :
– *La branche la plus à gauche est la plus longue de l'arbre des appels. Sa longueur est* n*, le nombre d'appels est donc inférieur à* 2^n*.*
– *La branche la plus à droite est la plus courte de l'arbre des appels. Sa longueur est* $n/2$*, le nombre d'appels est donc supérieur à* $2^{n/2}$*.*

Question 3.2 (1 point) *Soit l'algorithme récursif qui implémente la fonction de Syracuse.*

```
Syracuse(n){
  si (n=1)
  alors
    retourner 1;
  sinon
    si (n est pair)
```

```
    alors
        retourner Syracuse(n/2);
    sinon
        retourner Syracuse(3*n + 1);
}
```

Donnez la suite des appels pour `Syracuse(13)`.

Réponse :

Question 3.3 (1 point) *Malgré de nombreuses recherches, nous ne savons toujours pas si* `Syracuse(n)=1` *pour tout entier* `n`.
Donnez la complexité de l'algorithme `Syracuse(n)`.

Réponse : *Le résultat dit que nous ne savons pas si l'algorithme termine pour toute valeur de n. La complexité de cette fonction est donc inconnue aujourd'hui.*