INF2010

Corrigé du contrôle périodique 1

Mon nom de famille est : _____

Mon prénom est : _____

Mon matricule est : _____

Directives :

- L'examen a une durée de 1h ;
- Aucune documentation permise ;
- Calculatrice non-programmable permise ;
- Veuillez répondre dans le cahier de l'énoncé dans les espaces réservés à cet effet ;
- Un cahier d'examen P vous est fourni pour servir de brouillon ;
- Le cahier d'examen P n'est pas évalué — seules les réponses dans les espaces réservés aux réponses dans l'énoncé seront corrigées ;
- L'examen est noté sur un total de 15 points et compte pour 15% de la note finale du cours ;
- Ne posez pas de question concernant l'examen. En cas de doute sur le sens d'une question, énoncez clairement toute supposition que vous faites.

Exercice 1 : Complexité algorithmique (4 points)

1.1) **(1 point)** Donnez, fonction de *n*, la complexité algorithmique de l'extrait de code Java qui suit. Justifiez clairement votre réponse.

```
for (int i = 0; i < n; i++){
   for (int j = i+1; j < n; j += j){
      System.out.println("(i, j) = " + "(" + i + ", " + j + ")");
   }
}
```

Nous avons ici deux boucles imbriquées, la première itère sur i de 0 à $n-1$ avec un incrément unitaire, la seconde itère sur j de $i+1$ à $n-1$ avec un incrément de j (on double la valeur à chaque itération). Le nombre d'itérations de la seconde boucle peut être approximé par $log(n-i-1)$. Nous aurons donc

$$T(n) \cong \sum_{i=0}^{n-1} \lceil \log(n-i-1) \rceil$$
$$\cong \log((n-1)!)$$

Une réponse O(nlog(n)) est une borne supérieure à cette approximation et elle est acceptée pour cette question. O(n^2) donne 0.25/1.

1.2) **(1 point)** Donnez, fonction de *n*, la complexité algorithmique de l'extrait de code Java qui suit. Justifiez clairement votre réponse.

```
for (int i = 0; i < n; i++){
   for (int j = i+1; j < n; j <<= (i+1)){
      System.out.println("(i, j) = " + "(" + i + ", " + j + ")");
   }
}
```

Nous avons ici deux boucles imbriquées, la première itère sur i de 0 à $n-1$ avec un incrément unitaire, la seconde itère sur j de $i+1$ à $n-1$ avec un facteur multiplication de 2^{i+1} sur j. Le nombre d'itérations de la seconde boucle se réduit à 1 très rapidement. On en déduit que le code s'exécute en O(n). O(nlog(n)) est une borne supérieure à cette approximation. O(n^2) donne 0.25/1.

Il convient de noter que Java ne permet pas d'exécuter ce code pour n > 27.

Le problème du voyageur de commerce consiste à trouver le plus court itinéraire pour visiter un ensemble de villes une seule fois, avant de retourner à la ville de départ.

Soit une fonction qui, pour un nombre de villes *n* donné, affiche l'ensemble des parcours possibles. Par exemple, si *n* = 4, la fonction affichera :

(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2), (2, 1, 3, 4), (2, 1, 4, 3), (2, 3, 1, 4), (2, 3, 4, 1), (2, 4, 1, 3), (2, 4, 3, 1), …, (4, 3, 2, 1)

Notons que le parcours (1, 2, 3, 4), (1, 4, 3, 2), (2, 3, 4, 1), (2, 1, 4, 3), (3, 4, 1, 2), (3, 2, 1, 4), (4, 1, 2, 3) et (4, 3, 2, 1) sont équivalents et ont la même longueur.

1.3) **(1 point)** Donnez T(*n*), le temps d'exécution de la fonction qui affiche l'ensemble des parcours possibles pour *n* villes. On supposera que l'affichage d'un parcours se fait en temps unitaire, et que la recherche du prochain parcours possible se fait en temps constant. Justifiez clairement votre réponse.

Il y a n ville dont on peut partir, desquelles il reste $n-1$ villes comme seconde ville, $n-2$ pour troisième ville, etc. Le temps d'exécution du programme est donc $T(n) = n!$

1.4) **(1 point)** Reprenez la question 1.3) en supposant cette fois que la fonction d'affichage évite les redondances. Par exemple, si la fonction affiche (1, 2, 3, 4), elle n'affichera pas les parcours redondants (1, 4, 3, 2), (2, 3, 4, 1), … (4, 3, 2, 1). Justifiez clairement votre réponse.

Pour éviter les redondances, on peut fixer la ville de départ et ne pas faire le tour des villes dans les deux sens. On trouve $T(n) = (n-1)!/2$

Exercice 2 : Table de dispersement **(6 points)**

Soit une table de dispersement où les collisions sont gérées par débordement progressif avec sondage quadratique et où Hash(clé) = (clé + i^2 + 2 i + 1) % N. On admettra que i correspond au nombre de collisions à date et que i débute à 0.

2.1) **(0.5 point)** Quelle est la complexité asymptotique en insertion de cette structure de données ? Considérez la complexité en cas moyen. Justifiez votre réponse.

O(1) puisque nous avons affaire à un table de dispersement par sondage quadratique.

2.2) **(1 point)** Quelle est la complexité asymptotique en insertion de cette structure de données si tous les éléments insérés génèrent la même valeur Hash(clé) à la première insertion (i=0) ? Justifiez votre réponse.

O(n) ayant un nombre linéairement croissant de collision pour chaque insertion.

2.3) **(1 point)** Compléter la fonction `findPos()` donnée à l'Annexe 1 et reproduite ci-après. Pour mémoire, la fonction doit implémenter Hash(clé) = (clé + i^2 + 2 i + 1) % N.

```
private int findPos( AnyType x ) {

   int offset =  3; // VOTRE REPONSE ICI

   int currentPos = myhash( x ) + 1;

   while( array[ currentPos ] != null &&
      !array[ currentPos ].element.equals( x ) )
   {
      currentPos += offset;  // Compute ith probe

      offset +=  2; // VOTRE REPONSE ICI

      if( currentPos >= array.length )
         currentPos -= array.length;
   }

   return currentPos;
}
```

2.4) **(2.5 point)** En vous servant du tableau ci-dessous, donnez l'état de la mémoire d'une table de taille N=13 après l'insertion des clés suivantes:

91, 56, 78, 65, 48.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Entrées		91			78	56				65	48		

2.5) **(1 point)** Si l'on continue à insérer de nouvelles clés, combien d'entrées la table contiendra-t-elle au moment où une nouvelle insertion provoquera un rehash ? Quelle sera la nouvelle taille de la table ? **Référez-vous au code Java donné à l'Annexe 1.**

La table contiendra 5 éléments au moment où une nouvelle insertion provoquera un rehash.

La nouvelle taille de la table sera: 29.

Exercice 3 : Tri en *n log*(*n*) **(6 points)**

Partie 1:

On désire exécuter l'algorithme « MergeSort » donné à l'Annexe 2 pour trier le vecteur suivant.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

3.1) **(1 point)** Illustrez les étapes de l'exécution de l'algorithme en vous servant du tableau ci-dessous :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1
Blocs de 2	14	16	10	12	6	8	2	4	13	15	9	11	5	7	1	3
Blocs de 4	10	12	14	16	2	4	6	8	9	11	13	15	1	3	5	7
Blocs de 8	2	4	6	8	10	12	14	16	1	3	5	7	9	11	13	15
Fin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3.2) **(1 point)** Au total, quel est le nombre de fois que la fonction récursive `mergesort` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void mergesort(AnyType [ ] a, AnyType [ ] tempArray, int left, int right)
```

Votre réponse: 31

3.3) **(1 point)** Au total, quel est le nombre de fois que la fonction `merge` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void merge (AnyType [ ] a, AnyType [ ] tempArray, int leftPos,
            int rightPos, int rightEnd)
```

Votre réponse: 15

Partie 2:

On désire maintenant étudier l'exécution de l'algorithme « QuickSort » donné à l'Annexe 2 pour trier le même vecteur, reproduit ci-après. On considère une valeur *cut-off* de 3.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

3.4) (**0.5 point**) Donnez :

Les trois valeurs de « Median3 » à la première récursion :

16, 2, 1

La valeur de la médiane (pivot) :

2

3.5) (**0.5 point**) Donnez l'état du vecteur après l'exécution de Median3 de la première récursion :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	1	14	12	10	8	6	4	3	15	13	11	9	7	5	2	16

3.6) (**1 point**) Donnez l'état du vecteur après l'exécution du partitionnement de la première récursion :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeurs	1	2	12	10	8	6	4	3	15	13	11	9	7	5	14	16

3.7) **(1 point)** Au total, quel est le nombre de fois que la fonction récursive `quicksort` aura été appelée pour exécuter le tri ? Pour éviter tout ambigüité, la signature de la fonction est reproduite ci-après.

Signature de la fonction considérée :

```
private static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a, int left, int right )
```

Votre réponse: 13

Annexe 1

```
public class MyQuadraticProbingHashTable<AnyType> {
    public MyQuadraticProbingHashTable( ) {
        this( DEFAULT_TABLE_SIZE );
    }

    public QuadraticProbingHashTable( int size ) {
        allocateArray( size );  makeEmpty( );
    }

    public void insert( AnyType x ) {
        int currentPos = findPos( x );
        if( isActive( currentPos ) )  return;

        array[ currentPos ] = new MyHashEntry<AnyType>( x, true );

        // Rehash
        if( ++currentSize >= array.length / 2 )  rehash( );
    }

    private void rehash( ) {
        MyHashEntry<AnyType> [ ] oldArray = array;

        allocateArray( nextPrime( 2 * oldArray.length ) );
        currentSize = 0;

        for( int i = 0; i < oldArray.length; i++ )
            if( oldArray[ i ] != null && oldArray[ i ].isActive )
                insert( oldArray[ i ].element );
    }

    private int findPos( AnyType x ) {
        int offset =  ; // masqué pour l'exercice
        int currentPos = myhash( x ) + 1;

        while( array[ currentPos ] != null &&
                !array[ currentPos ].element.equals( x ) )
        {
            currentPos += offset;  // Compute ith probe
            offset +=  ; // masqué pour l'exercice
            if( currentPos >= array.length )
                currentPos -= array.length;
        }

        return currentPos;
    }

    public void remove( AnyType x ) {
        int currentPos = findPos( x );
        if( isActive( currentPos ) )
            array[ currentPos ].isActive = false;
    }

    public boolean contains( AnyType x ) {
        int currentPos = findPos( x );
        return isActive( currentPos );
    }
```

```
    private boolean isActive( int currentPos ) {
        return array[ currentPos ] != null && array[ currentPos ].isActive;
    }

    public void makeEmpty( ) {
        currentSize = 0;
        for( int i = 0; i < array.length; i++ )
            array[ i ] = null;
    }

    private int myhash( AnyType x ) {
        int hashVal = x.hashCode( ) + 1;

        hashVal %= array.length;
        if( hashVal < 0 ) hashVal += array.length;

        return hashVal;
    }

    private static class MyHashEntry<AnyType> {
        public AnyType  element;  // the element
        public boolean isActive;  // false if marked deleted

        public MyHashEntry( AnyType e ) { this( e, true ); }

        public MyHashEntry( AnyType e, boolean i ) {
            element  = e;  isActive = i;
        }
    }

    private static final int DEFAULT_TABLE_SIZE = 11;
    private MyHashEntry<AnyType> [ ] array; // Tableau des éléments
    private int currentSize;                // Nombre d'alvéoles occupées

    @SuppressWarnings("unchecked")
    private void allocateArray( int arraySize ) {
        array = new MyHashEntry[ nextPrime( arraySize ) ];
    }

    private static int nextPrime( int n ) {
        if( n <= 0 )
            n = 3;
        if( n % 2 == 0 )
            n++;
        for( ; !isPrime( n ); n += 2 )  ;

        return n;
    }

    private static boolean isPrime( int n ) {
        if( n == 2 || n == 3 )
            return true;
        if( n == 1 || n % 2 == 0 )
            return false;
        for( int i = 3; i * i <= n; i += 2 )
            if( n % i == 0 )
                return false;
        return true;
    }
}
```

Annexe 2

```
public final class IntraSort {

   @SuppressWarnings("unchecked")
   public static <AnyType extends Comparable<? super AnyType>>
   void mergesort( AnyType [ ] a ) {
      AnyType [ ] tmpArray = (AnyType[]) new Comparable[ a.length ];

      mergesort( a, tmpArray, 0, a.length - 1 );
   }

   private static <AnyType extends Comparable<? super AnyType>>
   void mergesort( AnyType [ ] a, AnyType [ ] tmpArray, int left, int right ) {
      if( left < right ) {
         int center = ( left + right ) / 2;
         mergesort( a, tmpArray, left, center );
         mergesort( a, tmpArray, center + 1, right );
         merge( a, tmpArray, left, center + 1, right );
      }
   }

   private static <AnyType extends Comparable<? super AnyType>>
   void merge( AnyType [ ] a, AnyType [ ] tmpArray, int leftPos, int rightPos, int rightEnd ) {
      int leftEnd = rightPos - 1;
      int tmpPos = leftPos;
      int numElements = rightEnd - leftPos + 1;

      while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];
      else
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      while( leftPos <= leftEnd )    // Copy rest of first half
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];

      while( rightPos <= rightEnd )  // Copy rest of right half
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      for( int i = 0; i < numElements; i++, rightEnd-- )
         a[ rightEnd ] = tmpArray[ rightEnd ];
   }

   private static final int CUTOFF = 3;

   public static <AnyType extends Comparable<? super AnyType>>
   void quicksort( AnyType [ ] a ) {
      quicksort( a, 0, a.length - 1 );
   }

   private static <AnyType extends Comparable<? super AnyType>>
   void quicksort( AnyType [ ] a, int left, int right ) {
      if( left + CUTOFF <= right ) {
         AnyType pivot = median3( a, left, right );

         // partitionnement
         int i = left, j = right - 1;
         for( ; ; ) {
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
```

```
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j )  swapReferences( a, i, j );
            else break;
         }

         swapReferences( a, i, right - 1 );
         // fin du partitionnement

         // recursions
         quicksort( a, left, i - 1 );
         quicksort( a, i + 1, right );
      }
      else
         insertionSort( a, left, right );
   }

   public static <AnyType> void
   swapReferences( AnyType [ ] a, int index1, int index2 ) {
      AnyType tmp = a[ index1 ];
      a[ index1 ] = a[ index2 ];
      a[ index2 ] = tmp;
   }

   private static <AnyType extends Comparable<? super AnyType>>
   AnyType median3( AnyType [ ] a, int left, int right ) {
      int center = ( left + right ) / 2;
      if( a[ center ].compareTo( a[ left ] ) < 0 )
         swapReferences( a, left, center );
      if( a[ right ].compareTo( a[ left ] ) < 0 )
         swapReferences( a, left, right );
      if( a[ right ].compareTo( a[ center ] ) < 0 )
         swapReferences( a, center, right );

      swapReferences( a, center, right - 1 );
      return a[ right - 1 ];
   }

   private static <AnyType extends Comparable<? super AnyType>>
   void insertionSort( AnyType [ ] a, int left, int right ) {
      for( int p = left + 1; p <= right; p++ ) {
         AnyType tmp = a[ p ]; int j;
         for( j = p; j > left && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
         a[ j ] = a[ j - 1 ];
         a[ j ] = tmp;
      }
   }

   public static void main( String [ ] args ) {
      Integer [ ] a = {16, 14, 12, 10, 8, 6, 4, 2, 15, 13, 11,  9, 7, 5, 3, 1},
                  b = {16, 14, 12, 10, 8, 6, 4, 2, 15, 13, 11,  9, 7, 5, 3, 1};

      mergesort( a );
      quicksort( b );

      for(Integer valeur : a) System.out.print(valeur + " ");
      System.out.println();
      for(Integer valeur : b) System.out.print(valeur + " ");

   }
}
```