Retranscriptions algorithmiques du TP : Vins

Afin de structurer le fichier, nous allons déterminer une taille fixe pour chacune des lignes de ce fichier. Et Pour ce faire, nous allons créer un descripteur sur ce fichier qui va nous permettre de nous positionner dedans et le considérer comme un tableau géant de caractère. On va du coup créer de nouvelles lignes, de nouvelles chaînes avec une taille fixe et parcourir ce tableau géant afin de pouvoir recopier chacune des lignes à l'intérieur. On trouve pour le coup 3 boucles : la première boucle permet de lire le fichier ligne par ligne et dans cette boucle là, on trouvera 2 nouvelles boucles. La première qui va permettre de recopier chacun des caractères dans la nouvelle ligne de taille fixe. La 2nde boucle permettra de remplir le reste de cette nouvelle ligne par des espaces.

```
@Fonction structurer inputFilepath: Texte -> Texte
    @DebutBloc
    fixedFinalLength: nombre <- 100
    raf: Descripteur <- créer Descripteur sur inputFilepath
    result: Texte <- "top.cpy"
    raf2: Descripteur <- créer Descripteur sur result

    line: Texte <- Vide
    @TantQue line <- raf.readLine != Vide
        @DebutBloc

        L: lettre[] <- line
        O: lettre[] <- créer char[fixedFinalLength]
        @Pour i: nombre <- 0, i < L.longueur, i <- i +1
            @DebutBloc
            O[i] <- L[i]
            @FinBloc

        @Pour i: nombre <- L.longueur, i < O.longueur, i <- i +1
            @DebutBloc
            O[i] <- ' '
            @FinBloc
                O[fixedFinalLength -2] <- '\r'
                O[fixedFinalLength -1] <- '\n'
                Ecrire O dans raf2

        @FinBloc

    @Résultat: result
    @FinBloc
```

```
private static String structurer(String inputFilepath) throws IOException {
    int fixedFinalLength = 100;
    RandomAccessFile raf = new RandomAccessFile(inputFilepath, "r");
    String result = "top.cpy";
    RandomAccessFile raf2 = new RandomAccessFile(result, "rw");

    String line = null;
    while ((line = raf.readLine()) != null) {

        char[] L = line.toCharArray();
        char[] O = new char[fixedFinalLength];
        for (int i = 0; i < L.length; ++i) {
            O[i] = L[i];
        }
        for (int i = L.length; i < O.length; ++i) {
            O[i] = ' ';
        }
        O[fixedFinalLength -2] = '\r';
        O[fixedFinalLength -1] = '\n';
        raf2.writeBytes(new String(O));
    }

    return result;
}
```

Nous avons également créé un nouvel algorithme pour déverser un tableau dans un autre tableau. On trouve ici une boucle qui permet de parcourir le premier tableau et de recopier tous les éléments dans le 2nd.

```
@Fonction deverser octet[] A, octet[] B -> Rien
    @DebutBloc

    @Pour i: nombre <- 0, i < A.longueur, i <- i +1
        @DebutBloc
            B[i] <- A[i]
        @FinBloc

    @FinBloc
```

```
private static void deverser(byte[] A, byte[] B) {

    for (int i = 0; i < A.length; ++i) {
        B[i] = A[i];
    }

}
```

Ici, on s'intéresse à pouvoir comparer 2 segments de lettres et pour ce faire, on va les convertir en texte pour pouvoir les comparer car les textes viennent déjà avec une relation d'ordre.

```
@Fonction comparerLignes octet[] A, octet[] B -> nombre
    @DebutBloc
    a: Texte <- créer Texte pour A
    b: Texte <- créer Texte pour B
    @Résultat: comparer a et b
    @FinBloc
```

```
private static int comparerLignes(byte[] A, byte[] B) {
    String a = new String(A);
    String b = new String(B);
    return a.compareTo(b);
}
```

Dans cet algorithme, nous cherchons à récupérer une ligne du fichier structuré Pour ce faire, comme nous savons que chacune des lignes a une taille fixe, nous pouvons nous adresser directement au caractère qui se situe au début de chacune des lignes. Nous fonctionnons donc comme l'adressage direct d'un tableau ordinaire, utilisant le fait que chacun de ses éléments a une taille fixe pour calculer l'adresse mémoire de chacun des éléments.

```
@Fonction recupererLigne raf: Descripteur, i: nombre -> octet[]
    @DebutBloc

    tailleLigne: nombre <- 100
    Positioner le pointeur dans raf sur  tailleLigne * i

    octet[] result <- créer octet[tailleLigne]
    Lire result du raf
    @Résultat: result

    @FinBloc
```

```
private static byte[] recupererLigne(RandomAccessFile raf, int i) throws IOException {

    int tailleLigne = 100;
    raf.seek(tailleLigne * i);

    byte[] result = new byte[tailleLigne];
    raf.read(result);
    return result;
}
```

Nous mettons ici en place un tri non-performant pour pouvoir trier l'ensemble des lignes du fichier. nous cherchons au fur et à mesure le minimum de chacune des lignes pour le remonter tout en haut du fichier. De cette manière, nous aurons une suite croissante. Remarquez ici que nous faisons appel à un algorithme que nous avons élaboré avant, qui s'appelle déverser. Cet algorithme permet de simuler une sorte d'affectation entre des tableaux et nous permet ainsi de permuter des lignes dans ce fichier.

```
@Fonction trierFichier filepath: Texte -> Rien
    @DebutBloc

    tailleLigne: nombre <- 100
    Descripteur raf <- créer Descripteur sur filepath
    long n <- raf.longueur / tailleLigne
    Lt: octet[] <- créer octet[tailleLigne]

    @Pour i: nombre <- 0, i < n -1, i <- i +1
        @DebutBloc
        Li: octet[] <- recupererLigneraf, i
        @Pour j: nombre <- i +1, j < n, j <- j +1
            @DebutBloc
            Lj: octet[] <- recupererLigneraf, j
            @Si (@Appeler comparerLignes @Avec Li, Lj) > 0
                @DebutBloc
                @Appeler deverser @Avec Li, Lt
                @Appeler deverser @Avec Lj, Li
                @Appeler deverser @Avec Lt, Lj

                Positioner le pointeur dans le raf sur i * tailleLigne
                Ecrire Li dans le raf
                Positioner le pointeur dans le raf sur j * tailleLigne
                Ecrire Lj dans le raf
                @FinBloc
            @FinBloc
        @FinBloc
    @FinBloc
```

```
private static void trierFichier(String filepath) throws IOException {
    int tailleLigne = 100;

    RandomAccessFile raf = new RandomAccessFile(filepath, "rw");

    long n = raf.length() / tailleLigne;
    byte[] Lt = new byte[tailleLigne];
    for (int i = 0; i < n -1; ++i) {
        byte[] Li = recupererLigne(raf, i);
        for (int j = i +1; j < n; ++j) {
            byte[] Lj = recupererLigne(raf, j);
            if (comparerLignes(Li, Lj) > 0) {
                deverser(Li, Lt);
                deverser(Lj, Li);
                deverser(Lt, Lj);

                raf.seek((long) i * tailleLigne);
                raf.write(Li);
                raf.seek((long) j * tailleLigne);
                raf.write(Lj);
            }
        }

    }
}
```

Sur la recherche dichotomique nous faisons appel à la notion de récursivité. La récursivité est tout simplement le fait qu'un algorithme peut s'appeler lui-même avec les mêmes paramètres, mais avec des valeurs différentes. Dans le cadre de cette récursivité, nous séparons le fichier des vins en 2, en plein milieu, et nous comparons la donnée recherchée avec la ligne trouvée. Si la valeur recherchée est inférieure à la ligne qui a été actuellement trouvée, nous refaisons une recherche, mais en considérant seulement la première moitié de ce fichier. À contrario, si le vin recherché est supérieur à la ligne actuellement trouvé, nous refaisons la recherche, mais cette fois-ci dans la 2nde partie de ce fichier. on réitère ce process si bien qu'on coupe ce fichier en des parts de plus en plus petites, l'algorithme s'arrête lorsqu'il ne peut plus rien découper.

```
@Fonction chercherRec octet[] V, int i, int j, Descripteur raf -> octet[]
    @DebutBloc
    @Si j - i > 1
        @DebutBloc
            milieuIndex: nombre <- i + j / 2
            octet[] milieuLigne <- recupererLigneraf, milieuIndex
            @Si comparerLignesV, milieuLigne < 0
                @DebutBloc
                @Résultat: chercherRecV, i, milieuIndex, raf
                @FinBloc
            @Sinon
                @DebutBloc
                @Résultat: chercherRecV, milieuIndex, j, raf
                @FinBloc
        @FinBloc
    @Sinon
        @DebutBloc
            Li: octet[] <- @Appeler recupererLigne @Avec raf, i
            Lj: octet[] <- @Appeler recupererLigne @Avec raf, j

            @Si comparerLignesV, Li = 0
                @DebutBloc
                @Résultat: Li
                @FinBloc
            @Sinon @Si (@Appeler comparerLignes @Avec V, Lj) = 0
                @DebutBloc
                @Résultat: Lj
                @FinBloc
            @Sinon
                @DebutBloc
                @Résultat: Lj
                @FinBloc
        @FinBloc
    @FinBloc
```

```
private static byte[] chercherRec(byte[] V, int i, int j, RandomAccessFile raf) throws IOException {
    if (j - i > 1) {
        int milieuIndex = (i + j) / 2;
        byte[] milieuLigne = recupererLigne(raf, milieuIndex);
        if (comparerLignes(V, milieuLigne) < 0) {
            return chercherRec(V, i, milieuIndex, raf);
        }
        else {
            return chercherRec(V, milieuIndex, j, raf);
        }
    }
    else {
        byte[] Li = recupererLigne(raf, i);
        byte[] Lj = recupererLigne(raf, j);

        if (comparerLignes(V, Li) == 0) {
            return Li;
        }
        else if (comparerLignes(V, Lj) == 0) {
            return Lj;
        }
        else {
            return Lj;
        }
    }
}
```

Pour appeler cet algorithme de manière plus simple, nous créons un nouvel algorithme qui fera appel à cet algorithme, récursif.

```
@Fonction search filepath: Texte, value: Texte -> Texte
    @DebutBloc
    tailleLigne: nombre <- 100
    raf: Descripteur <- créer Descripteur sur filepath
    result: octet[] <- @Appeler chercherRec @Avec value, 0, raf.longueur / tailleLigne, raf
    @Résultat: créer Texte result
    @FinBloc
```

```
public static String search(String filepath, String value) throws IOException {
    int tailleLigne = 100;
    RandomAccessFile raf = new RandomAccessFile(filepath, "r");
    byte[] result = chercherRec(value.getBytes(), 0, (int) raf.length() / tailleLigne, raf);
    return new String(result);
}
```

Enfin, nous terminons sur l'algorithme principal avec la structuration du fichier d’entrée, le tri de ce de ce fichier structuré et enfin la recherche de la chaîne « cava » dans ce dernier.

```
 result: Texte <- Appeler structurer @Avec "VINStp.DON"
Appeler trierFichier @Avec result
Afficher @Appeler search @Avec result, "Cava"
```

```
public static void main(String[] args) {
    try {
        String result = structurer("VINStp.DON");
        trierFichier(result);
        System.out.println(search(result, "Cava"));
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```