COMPTE-RENDU DE TRAVAUX PRATIQUES (AC)

Travaux Pratiques 1 :

Analyse empirique de programmes

CHAUVEL Tom, JOSSO Célia
GROUPE TP1
ESIR 1 Info 2024-2025, Semestre 7

Tuteurs : Pierre MAUREL

SOMMAIRE

1. Introduction

Dans le cadre de ce premier travail pratique d'Algorithmique et Complexité, nous avons pour objectif d'analyser empiriquement les performances de différents algorithmes à travers la mesure de leur temps d'exécution. À l'aide de fichiers Java fournis, nous allons implanter et étudier les tris par insertion et par fusion, en mesurant leurs temps d'exécution pour des tableaux de tailles et de configurations variées (meilleur cas, pire cas, cas aléatoire). Cette analyse nous permettra de vérifier les complexités théoriques abordées en cours et de comprendre les comportements réels de ces algorithmes en fonction des données. En complément, nous explorerons la complexité d'une méthode de permutations et la fonction d'Ackermann.

2. Analyse de tris

2.1. Tri par insertion

2.1.1. Implémentation du tri par insertion

Nous avons implémenté le tri par insertion dans le fichier `Tri.java` conformément au pseudo-code présent dans l'exercice 2 du TD1.

```
1 pour j ← 2 à longueur de A faire
2         clé ← A[j]
3         i ← j−1
4         tant que i>0 et A[i]>clé faire
5                 A[i+1] ← A[i]
6                 i ← i−1
7         fin tant que
8         A[i+1] ← clé
9 fin pour
```

Pseudo-code

```java
/* Tri le tableau t via la méthode "Tri par insertion" */
public static void triInsertion(int[] t){
    for (int j = 1; j < t.length; j++ ) {
        int clef = t[j];
        int i = j-1;
        while (i>=0 && t[i] > clef){
            t[i+1] = t[i];
            i--;
        }
        t[i+1] = clef;
    }
}
```

Code implémenté en Java

Figure 1 : Implémentation du tri par insertion

Testons l'algorithme :

```
Avant le tri : [ 76  19  60  15  65  51  29  58  95  28 ]
Après le tri par insertion: [ 15  19  28  29  51  58  60  65  76  95 ]
```

Figure 2 : Implémentation du tri par insertion

En testant notre algorithme sur un tableau aléatoire avec la méthode main de `Mesure_tri`, nous remarquons que notre méthode est bien capable de trier un tableau aléatoire.

2.1.2. Analyse expérimentale

Nous allons analyser expérimentalement le temps d'exécution du tri par insertion selon les entrées.

- le meilleur des cas, lorsque le tableau est déjà trié dans l'ordre croissant
- le pire des cas, lorsque le tableau est déjà trié mais dans l'ordre décroissant
- le cas aléatoire, lorsque le tableau contient des valeurs aléatoires

Avant de la commencer, nous pouvons penser à l'initialisation des tableaux selon ces trois cas, dans le cadre d'une boucle finie pour `k` allant `0` jusqu'à `n`, la taille voulue pour le tableau.

- Pour le meilleur des cas (cas numéro 0), on peut dire que l'élément du tableau est simplement la valeur actuelle de l'index `k`. Cela donnera : `[0, 1, 2, …, n]`
- Pour le pire des cas (cas numéro 1), pour un tableau trié dans l'ordre décroissant, on peut dire que l'élément du tableau vaut `n-k`. Cela donnera : `[n, n-1, n-2, …, 0]`
- Pour le cas aléatoire, lorsque le tableau contient des valeurs aléatoires, nous pouvons penser à générer les éléments successivement de manière aléatoire entre `0` et `100`.

Voici donc à quoi ressemble l'initialisation du tableau selon les cas de figure :

```java
for(int k=0;k<t.length;k++){
    switch (type) {
        case 0:
            t[k] = k;
            break;
        case 1:
            t[k] = t.length - k;
            break;
        case 2:
            t[k]=(int) (Math.random()*100);
            break;
    }
}
```

Figure 3 : Initialisation du tableau pour le tri par insertion selon le cas de figure

2.1.2.1. Dans le meilleur des cas

2.1.2.1.1. Affichage des mesures effectuées

Pour rappel, le meilleur des cas survient lorsque le tableau est déjà trié dans l'ordre croissant.

Nous avons exécuté `15` fois le tri par insertion avec des tailles de tableau croissantes, allant de `100000000` éléments (`taille_init`) à `1000000000` éléments (`taille_fin`).

En traçant un nuage de points symbolisant le temps d'exécution en millisecondes en fonction de la taille du tableau, voici le résultat obtenu :

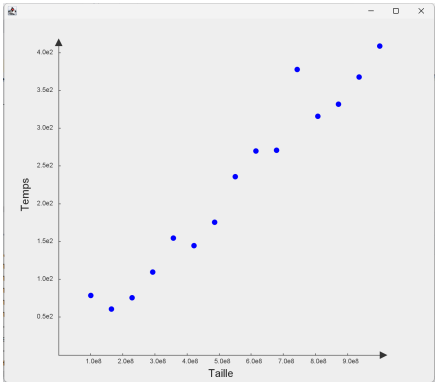

Figure 4 : Temps d'exécution (en ms) en fonction de la taille du tableau pour le tri par insertion (meilleur des cas)

En observant ce nuage de points, nous pouvons confirmer la cohérence de notre algorithme, dans la mesure où la courbe a une allure linéaire. Effectivement, nous savons que, dans le meilleur des cas pour le tri par insertion, la complexité est linéaire.

2.1.2.1.2. Régression manuelle

Nous allons désormais effectuer une régression manuelle dans le but d'approximer les coefficients du polynôme $T(n)$ expliquant le mieux nos mesures. $T(n)$ est la fonction représentant le temps en fonction de la taille du tableau.

Pour ce faire, puisque nous avons affaire à une complexité linéaire, alors nous avons :

$$T(n) = a \times n, \text{ soit } a = \frac{T(n)}{n}$$

Nous avons bien accès aux données $T(n)$ et n. Pour calculer facilement et efficacement a, pour pouvons faire un programme qui calcule la valeur moyenne de a pour chaque mesure et effectuer une moyenne de toutes les valeurs obtenues. Voici un programme Python qui permet de le faire :

```python
import math

def regression(filename, complexity):
    """
    Calculates an average value for the time complexity based on data points from a file.
    To approximate the coefficients of the polynomial T(n) that best explain the observed measurements.

    Args:
        filename (str): The name of the file containing data points in the format "n,t".
                        Each pair represents an input size 'n' and the corresponding time 't'.
        complexity (str): The type of complexity to calculate
                          Either 'linear' or 'quadratic'.

    Returns:
        float: The average value of the ratio according to the complexity,
               which approximates the coefficient of the polynomial T(n).

    Raises:
        ValueError: If an unknown complexity type is provided.
    """
    values = []

    with open(filename, 'r') as file:
        lst = file.read().split(" ")

        # Process each data point (input size and execution time) to approximate the coefficients
        for i in range(len(lst)):
            line = lst[i].split(",")
            n, t = int(line[0]), int(line[1])

            # Calculate the ratio based on the specified complexity to approximate the coefficient
            if complexity == "linear":
                values.append(t/n)
            elif complexity == "quasi-linear":
                values.append(t/(n*math.log(n)))
            elif complexity == "quadratic":
                values.append(t/n**2)
            elif complexity == "factorial":
                values.append(t/math.factorial(n))
            else:
                raise ValueError(f"Unknown complexity: {complexity}")

    # Calculate and return the average of the computed ratios to estimate the coefficient
    return sum(values) / len(values)

if __name__ == '__main__':
    filename = 'rapports/fusion/aleatoire.txt'
    complexity = 'quasi-linear'  # Choose between 'linear', 'quasi-linear' or 'quadratic' or 'factorial'
    print(regression(filename, complexity))
```

Figure 5 : Programme pour calculer la régression

Pour l'exécuter, nous avons stocké nos mesures sous un format `csv` (de la forme `taille,temps` pour chaque mesure, étant chacune séparée par un espace) dans un fichier texte.

Pour le tri par insertion dans le meilleur des cas, nous obtenons :

$$a = 4.2409638018682205 \times 10^{-7}$$

En traçant la courbe de régression à partir de la valeur de a et du type de régression (ici, linéaire), nous obtenons ce résultat :

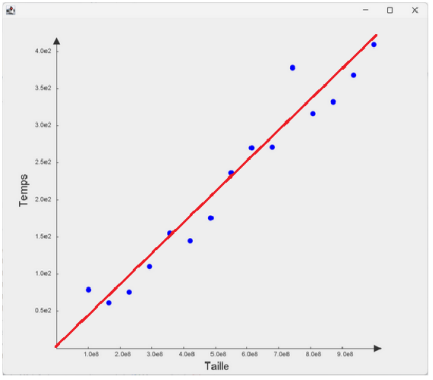

Figure 6 : Régression linéaire pour le tri par insertion (meilleur des cas)

Cette régression nous permet de nous rendre compte que, visuellement, le modèle linéaire est bel et bien le plus pertinent pour le tri par insertion dans le meilleur des cas.

2.1.2.2. Dans le pire des cas

Pour rappel, le pire des cas survient lorsque le tableau est déjà trié mais dans l'ordre décroissant.

Nous avons exécuté `30` fois le tri par insertion avec des tailles de tableau croissantes, allant de `10000` éléments (`taille_init`) à `100000` éléments (`taille_fin`).

En traçant un nuage de points symbolisant le temps d'exécution en millisecondes en fonction de la taille du tableau, voici le résultat obtenu :

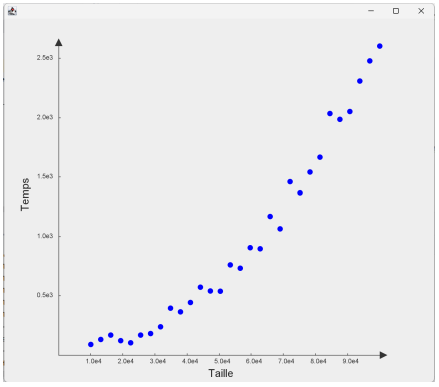

Figure 7 : Temps d'exécution (en ms) en fonction de la taille du tableau pour le tri par insertion (pire des cas)

En observant ce nuage de points, nous pouvons confirmer la cohérence de notre algorithme, dans la mesure où la courbe a une allure quadratique. Effectivement, nous savons que, dans le pire des cas pour le tri par insertion, la complexité est quadratique.

2.1.2.2.1. Régression manuelle

Maintenant, puisque nous avons affaire à une complexité quadratique, alors nous avons :

$$T(n) = a \times n^2, \text{ soit } a = \frac{T(n)}{n^2}$$

En exécutant le programme de tout à l'heure, pour le tri par insertion dans le pire des cas, nous obtenons :

$$a = 3.080013373906089 \times 10^{-7}$$

En traçant la courbe de régression à partir de la valeur de a et du type de régression (ici, quadratique), nous obtenons ce résultat :

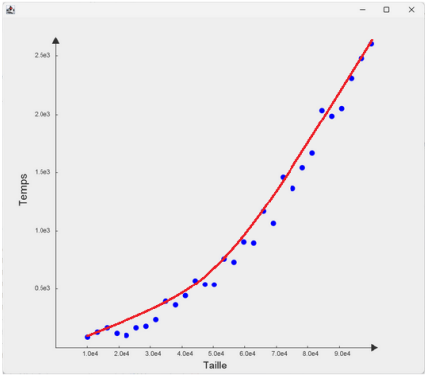

Figure 8 : Régression linéaire pour le tri par insertion (pire des cas)

Cette régression nous permet de nous rendre compte que, visuellement, le modèle quadratique est bel et bien le plus pertinent pour le tri par insertion dans le pire des cas.

2.1.2.3. Dans le cas aléatoire

Pour rappel, le cas aléatoire survient lorsque le tableau contient des valeurs aléatoires.

Nous avons exécuté `30` fois le tri par insertion avec des tailles de tableau croissantes, allant de `10000` éléments (`taille_init`) à `100000` éléments (`taille_fin`).

En traçant un nuage de points symbolisant le temps d'exécution en millisecondes en fonction de la taille du tableau, voici le résultat obtenu :

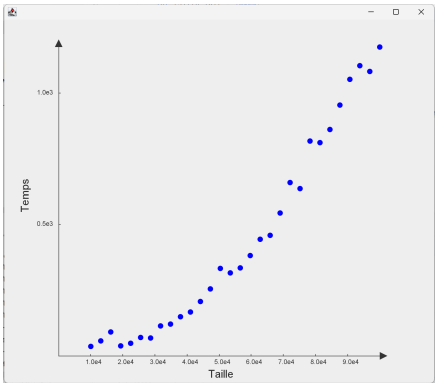

Figure 9 : Temps d'exécution (en ms) en fonction de la taille du tableau pour le tri par insertion (cas aléatoire)

En observant ce nuage de points, nous pouvons confirmer la cohérence de notre algorithme, dans la mesure où la courbe a une allure quadratique. Effectivement, nous savons que, dans le cas aléatoire pour le tri par insertion, la complexité est quadratique.

2.1.2.3.1. Régression manuelle

De même que pour le pire des cas, puisque nous avons affaire à une complexité quadratique, alors nous avons :

$$a = \frac{T(n)}{n^2}$$

En exécutant le programme de tout à l'heure, pour le tri par insertion dans le cas aléatoire, nous obtenons :

$$a = 1.3589388926951665 \times 10^{-7}$$

En traçant la courbe de régression à partir de la valeur de a et du type de régression (ici, quadratique), nous obtenons ce résultat :

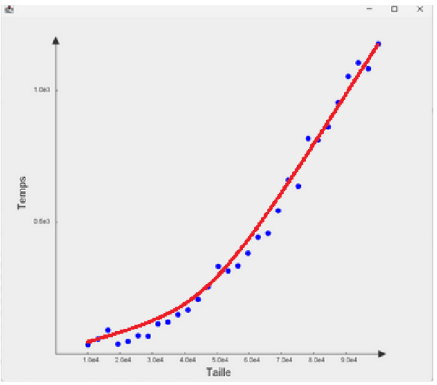

Figure 10 : Régression quadratique pour le tri par insertion (cas aléatoire)

Cette régression nous permet de nous rendre compte que, visuellement, le modèle quadratique est bel et bien le plus pertinent pour le tri par insertion dans le cas aléatoire.

2.1.3. Temps estimé pour trier les numéros de sécurité sociale français

Pour estimer le temps pour trier les numéros de sécurité sociale français, nous pouvons considérer que nous sommes dans le cas aléatoire, soit une complexité quadratique. Soit la formule :

$$T(n) = a \times n^2$$

avec : $a = 1.3589388926951665 \times 10^{-7}$

Il y a 67 millions d'habitants en France, soit 67 millions de numéros de sécurité sociale. Le tableau contiendrait donc 67 millions d'éléments. Nous aurions donc :

$$T(6.7 \times 10^7) = a \times (6.7 \times 10^7)^2 = 1.3589388926951665 \times 10^{-7} \times (6.7 \times 10^7)^2$$

Soit :

$$T(6.7 \times 10^7) \simeq 6.1 \times 10^8\, ms$$

Donnons cette estimation en unités plus compréhensibles.

En convertissant en secondes, nous avons :

$$6.1 \times 10^8\, ms = 6.1 \times 10^5\, s$$

En divisant par 60, nous obtenons :

$$6.1 \times 10^5\, s \simeq 10167\, min$$

En redivisant par 60, nous obtenons :

$$10167\, min \simeq 169h$$

En divisant par 24, nous obtenons :

$$169h \simeq 7\, jours$$

Donc il faudra environ 7 jours pour trier les numéros de sécurité sociale de la population française avec le tri par insertion.

2.2. Tri fusion

2.2.1. Implémentation du tri fusion

Nous avons implémenté le tri fusion dans le fichier `Tri.java` conformément au pseudo-code présent dans l'exercice 5 du TD2.

```
1  TRI–FUSION(A, debut, fin)
2      si debut<fin alors
3          milieu ← partie entière de (debut+fin)/2
4          TRI–FUSION(A, debut, milieu)
5          TRI–FUSION(A, milieu+1, fin)
6          FUSIONNER(A, debut, milieu, fin)
```

Pseudo-code

```java
/*Tri le tableau t via la méthode "Tri fusion" */
public static void triFusion(int[] t){
    if (t.length>0)
        triFusion(t, 0, t.length-1);
}

/* Sous-fonction (récursive) pour le tri fusion
 * Trie le sous-tableau t[debut]..t[fin] */
private static void triFusion(int[] t, int debut, int fin){
    if (debut < fin){
        int milieu = (debut + fin) / 2;
        triFusion(t,debut,milieu);
        triFusion(t,milieu+1,fin);
        fusionner(t,debut,milieu,fin);
    }
}
```

Code implémenté en Java

Figure 11 : Implémentation du tri par insertion

2.2.2. Analyse expérimentale

Cette fois-ci, l'analyse expérimentale ne se fera uniquement dans le cas de tableaux aléatoires. En effet, le tri fusion a toujours la même complexité, quelle que soit la disposition du tableau.

Nous avons exécuté `30` fois le tri par insertion avec des tailles de tableau croissantes, allant de `10000000` éléments (`taille_init`) à `100000000` éléments (`taille_fin`).

En traçant un nuage de points symbolisant le temps d'exécution en millisecondes en fonction de la taille du tableau, voici le résultat obtenu :

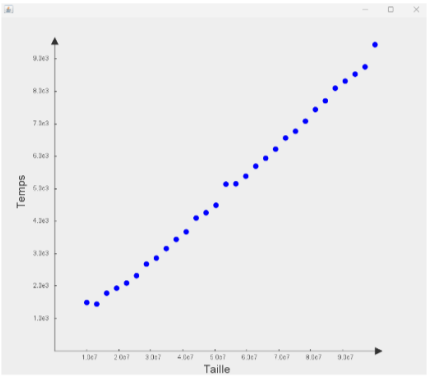

Figure 12 : Temps d'exécution (en ms) en fonction de la taille du tableau pour le tri fusion

En observant ce nuage de points, nous pouvons confirmer la cohérence de notre algorithme, dans la mesure où la courbe a une allure quasi-linéaire. Effectivement, nous savons que, dans le cas aléatoire pour le tri par insertion, la complexité est quasi-linéaire c'est à dire $O(n\ ln(n))$.

2.2.3. Régression manuelle

De même que pour le pire des cas, puisque nous avons affaire à une complexité quasi-linéaire, alors nous avons :

$$a = \frac{T(n)}{n\ ln(n)}$$

En exécutant le programme de tout à l'heure, pour le tri fusion, nous obtenons :

$$a = 7.920894301673746 \times 10^{-6}$$

En traçant la courbe de régression à partir de la valeur de a et du type de régression (ici, quasi-linéaire), nous obtenons ce résultat :

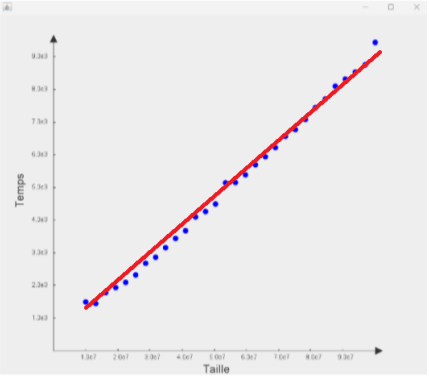

Figure 13 : Régression quasi-linéaire pour le tri fusion

Cette régression nous permet de nous rendre compte que, visuellement, le modèle quasi-linéaire est bel et bien le plus pertinent pour le tri par insertion dans le cas aléatoire.

2.2.4. Temps estimé pour trier les numéros de sécurité sociale français

Pour estimer le temps pour trier les numéros de sécurité sociale français, nous pouvons le faire de la même manière. Soit la formule dans le cadre du tri fusion de complexité quasi-linéaire :

$$T(n) = a \times n\, ln(n)$$

avec : $a = 7.920894301673746 \times 10^{-6}$

Il y a 67 millions d'habitants en France, soit 67 millions de numéros de sécurité sociale. Le tableau contiendrait donc 67 millions d'éléments.

Nous aurions donc :

$$T(6.7 \times 10^{7}) = a \times n \times log(n) = 7.920894301673746 \times 10^{-6} \times 6.7 \times 10^{7} * ln(6.7 \times 10^{7})$$

Soit :

$$T(6.7 \times 10^{7}) \simeq 9563\, ms$$

Donnons cette estimation en unités plus compréhensibles.

En convertissant en secondes, nous avons :

$$9563\, ms \simeq 10\, s$$

Donc il faudra environ 10 millisecondes pour trier les numéros de sécurité sociale de la population française avec le tri fusion. Il est donc beaucoup plus rapide que le tri par insertion dans ce cas de figure.

3. Permutations

Il nous est fourni dans le fichier `Permutations.java` une méthode permutation. Voici le fichier :

```java
public class Permutations {
    public static void permutation(String target, String original,boolean afficher){
        int i;
        String target1, original1;
        if (original.length() == 0 && afficher){
            System.out.println(target);
        }
        else {
            i = 0;
            while (i < original.length()){
                target1 = target + original.substring(i,i+1);
                original1 = original.substring(0,i) + original.substring(i+1,original.length());
                permutation(target1,original1,afficher);
                i = i + 1;
            }
        }
    }
    public static void main(String[] args) {
        String x="";
        String alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        int n=3;
        String y=alphabet.substring(0, n);
        permutation(x,y,true);
    }
}
```

Figure 14 : Fichier Permutations.java

Testons cette fonction à l'aide du main fourni. Voici l'affichage que nous obtenons en l'exécutant :

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Répondons désormais aux questions

3.1. Que fait cette fonction ?

Cette fonction génère et affiche toutes les permutations possibles d'une chaîne de caractères donnée.

Elle contient les paramètres suivants :

- `target`, une string qui contient la permutation partielle en cours de construction.
- `original`, une string qui contient les caractères restants à utiliser pour générer des permutations.
- `afficher`, un booléen qui permet de contrôler si les permutations générées doivent être affichées.

Voici son fonctionnement :

- Si `original` est vide et que le paramètre `afficher` vaut true, alors une permutation complète a été construite, et elle est alors affichée (c'est la chaîne de caractères présente dans la variable `target`).
- Si `original` n'est pas vide, la fonction effectue une boucle sur chaque caractère de `original` pour ajouter le caractère à `target` (pour former une nouvelle permutation partielle), puis appelle récursivement la fonction avec le reste des caractères (sans le caractère ajouté).

Enfin, dans le main, on extrait une chaîne de longueur `n` de l'alphabet, puis toutes ses permutations sont générées et affichées. Ici, l'exemple de base est avec une valeur de `n` égale à `3`. C'est pour cela que la fonction nous a affiché les `6` permutations possibles de la chaîne de caractères `"ABC"`.

3.2. Quelle est sa complexité ?

Pour trouver la complexité de la fonction, nous pouvons analyser le nombre de permutations possibles d'une chaîne de longueur n.

Nous savons que le nombre total de permutations d'une chaîne de n caractères est $n!$. Nous avons donc $n!$ appels récursifs. Par exemple, pour une chaîne de 3 caractères, il y a $3! = 6$ permutations.

À chaque niveau de récursion, la fonction effectue une boucle sur les caractères restants dans original. On a donc un facteur de multiplication de taille décroissante à chaque étape de récursion.

Finalement, la complexité de la fonction est $O(n!)$, car il y a $n!$ permutations à générer. Chaque permutation demande un nombre constant d'opérations pour être construite, ce qui ne modifie pas l'ordre de grandeur de la complexité.

3.3. Evolution du temps de calcul en fonction de n

Pour répondre à cette question, nous pouvons d'abord désactivé l'affichage des différentes permutations car non seulement il n'est pas utile pour répondre à la question mais aussi car il sera très conséquent. Pour ce faire, il suffit simplement de passer le paramètre afficher présent dans le main à false au lieu de true.

Ensuite, pour mesurer le temps d'exécution en fonction de la taille de l'ensemble des lettres `n`, nous avons utilisé une approche similaire à celle employée pour l'analyse de la complexité des algorithmes de tri des parties précédentes. Nous avons effectué des mesures pour différentes valeurs de `n` allant de `1` à `12`, car au-delà de `12`, le temps d'exécution devient trop long. Cela est dû au fait que le nombre de permutations à générer augmente de façon exponentielle, ce qui rend les calculs très coûteux en temps.

Voici le code modifié, qui effectue les mesures de temps pour des valeurs de `n` jusqu'à `12` :

```java
import java.util.ArrayList;
import java.util.List;

public class Permutations {
    public static void permutation(String target, String original,boolean afficher){
        int i;
        String target1, original1;
        if (original.length() == 0 && afficher){
            System.out.println(target);
        }
        else {
            i = 0;
            while (i < original.length()){
                target1 = target + original.substring(i,i+1);
                original1 = original.substring(0,i) + original.substring(i+1,original.length());
                permutation(target1,original1,afficher);
                i = i + 1;
            }
        }
    }

    public static void main(String[] args) {
        String x = "";
        String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        List<Integer> tab_n = new ArrayList<>();
        List<Integer> tab_temps = new ArrayList<>();

        // Mesurer le temps d'exécution pour n de 1 à 26
        for (int n = 1; n <= 12; n++) {
            String y = alphabet.substring(0, n);
            tab_n.add(n);

            // Mesure du temps d'exécution
```

```java
            long startTime = System.currentTimeMillis();
            permutation(x, y, false);  // désactiver l'affichage avec "false"
            long endTime = System.currentTimeMillis();

            int elapsedTime = (int) (endTime - startTime);
            tab_temps.add(elapsedTime);

            System.out.println("Temps de calcul pour n=" + n + " : " + elapsedTime + " millisecondes.");
        }

        Graph g = new Graph(tab_n, tab_temps);
        g.display();
    }

}
```

Figure 15 : Fichier Permutations.java modifié

En traçant un nuage de points symbolisant le temps d'exécution en millisecondes en fonction de la taille de la chaîne de caractères, voici le résultat obtenu :

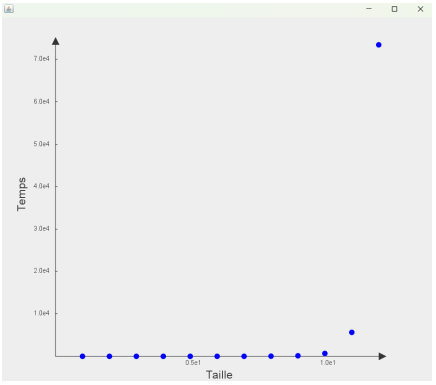

Figure 16: Temps d'exécution (en ms) en fonction de la taille de la chaîne de caractères pour la fonction permutations

En observant ce nuage de points, nous pouvons confirmer la cohérence de notre raisonnement, dans la mesure où la courbe a une allure similaire à celle de la fonction factorielle (augmentation croissante et brutale).

Ensuite, en traçant la courbe de régression à partir de la valeur de a et du type de régression (ici, factorielle), nous obtenons ce résultat :

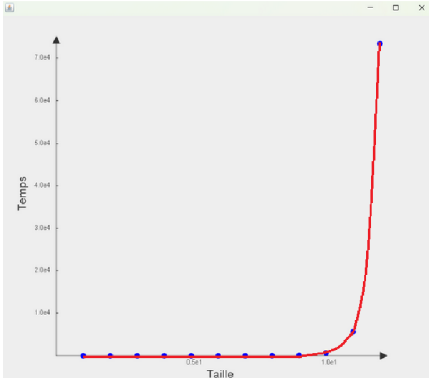

Figure 17 : Régression factorielle pour la fonction permutations

Cette régression nous permet de nous rendre compte que, visuellement, le modèle factoriel est bel et bien le plus pertinent pour la fonction `permutations`.

3.4. Temps nécessaire pour n=26

Puisque nous avons affaire à une complexité en factorielle, alors nous avons :

$$a = \frac{T(n)}{n!}$$

En analysant les premières valeurs de sortie de la fonction permutations, nous nous sommes aperçues qu'elles sont trop proches, et ne définissent pas à proprement parler les fonctions factorielles. Elles ne permettent pas vraiment de calculer le coefficient de régression.

En exécutant le programme de tout à l'heure, pour le programme de permutation, nous obtenons :

$$a = 2.1060054914221583 \times 10^{-4}$$

Pour estimer le temps pour calculer toutes les permutations possibles sur la chaîne des lettres de l'alphabet, nous pouvons le faire de la même manière que pour le tri des numéros de sécurité sociale. Soit la formule dans le cadre de la complexité factorielle :

$$T(n) = a \times n!$$

avec : $a = 2.1060054914221583 \times 10^{-4} \simeq 2.1 \times 10^{-4}$

Nous aurions donc :

$$T(26) = a \times 26! = a \times 4 \times 10^{26} = 2.1 \times 10^{-4} \times 4 \times 10^{26}$$

Soit :

$$T(26) \simeq 8.4 \times 10^{22}\, ms$$

Donnons cette estimation en unités plus compréhensibles.

En convertissant en secondes, nous avons alors :

$$8.4 \times 10^{22}\, ms = 8.4 \times 10^{19}\, s$$

En divisant par 60, nous obtenons :

$$8.4 \times 10^{19}\, s = 1.45 \times 10^{18}\, min$$

En redivisant par 60, nous obtenons :

$$1.45 \times 10^{18}\, min = 2.42 \times 10^{16}\, h$$

En divisant par 24, nous obtenons :

$$2.42 \times 10^{16}\, h = 1 \times 10^{15}\, jours$$

En divisant par 365.25, nous avons :

$$1 \times 10^{15}\ jours = 2.76 \times 10^{12}\, années$$

La date de création de l'univers est estimée à il y a environ 13.7 milliards d'années, soit 1.37×10^{10} années. Cela veut dire que la durée pour calculer toutes les permutations possibles sur la chaîne des lettres de l'alphabet est 200 fois plus longue que l'âge de l'univers !

4. Bonus : Fonction d'Ackermann

4.1. Qu'est ce que la fonction d'Ackermann ?

En se renseignant sur Wikipédia, nous avons trouvé que la fonction d'Ackermann est une fonction mathématique bien connue pour son comportement de récursivité profonde et rapide. C'est un exemple de fonction qui n'est pas calculable par une simple récursion primitive, ce qui signifie que sa croissance dépasse celle de fonctions récursives basiques (comme les factoriels ou les puissances). Elle est souvent utilisée pour tester les limites des systèmes informatiques et des algorithmes de récursion en raison de la profondeur et de la complexité des appels qu'elle génère.
La fonction d'Ackermann est définie de manière récursive selon les trois règles suivantes :

- `Si m == 0,` `alors A(m, n) = n + 1`
- `Si n == 0 et m > 0,` `alors A(m, n) = A(m - 1, 1)`
- `Si m > 0 et n > 0,` `alors A(m, n) = A(m - 1, A(m, n - 1))`

4.2. Test sur plusieurs valeurs

En testant la fonction d'Ackermann sur l'exemple fourni de base dans le code (`n=3` et `m=3`), nous obtenons l'affichage suivant : `ack(3,3) = 61`

Or, en mettant `n=3` et `m=4`, nous obtenons une erreur de dépassement de pile.

Ensuite, en fixant `m=4` et en augmentant `n` progressivement, nous nous rendons compte que l'erreur de dépassement de pile survient à partir de `n=13`.

4.3. Augmentation de la taille de la pile

Nous pouvons la taille de la pile, en ajoutant une taille de pile dans les arguments de la machine virtuelle. Avec l'IDE VSCode, cela peut se faire en créant un fichier `launch.json` et en y ajoutant par exemple la ligne `"vmArgs": "-Xss1m"` dans la configuration de la classe `Ackermann` pour augmenter la taille de la pile à 1 Mo.

Sur ce principe, voici un tableau des limites de valeurs de `m` (avant de se retrouver avec une erreur de dépassement de pile) selon la taille de la pile en fixant `n` à `3`.

taille pile (Mo)	initial	1	2	3	4	5	6	7	8	9	10
limite pour `m`	3	3	3	3	3	3	3	3	3	3	3

Nous nous rendons compte que même avec 10Mo de pile, nous ne parvenons pas à augmenter la limite pour `m` avec `n` fixé.

De façon analogique, voici un tableau des limites de valeurs de `n` (avant de se retrouver avec une erreur de dépassement de pile) selon la taille de la pile en fixant `m` à `3`.

taille pile (Mo)	initial	1	2	3	4	5	6	7	8	9	10
limite pour `n`	12	12	13	13	14	14	14	15	15	15	15

Ici, nous parvenons à augmenter la limite de `n` avec `m` fixé, mais cette augmentation n'est pas flagrante.

4.4. Interprétation

Nous pouvons en déduire que la fonction d'Ackermann a une croissance si rapide qu'elle limite les appels récursifs. Cela rend difficile l'évaluation pour des valeurs de `m` élevées, tandis que les valeurs de `n` montrent une légère amélioration avec une taille de pile accrue.

5. Conclusion

Ce travail a permis d'explorer les performances des algorithmes de tri par insertion et fusion, en analysant empiriquement leurs temps d'exécution dans divers scénarios. Les résultats ont confirmé les complexités théoriques. Pour rappel, le tri par insertion se comporte de manière linéaire dans le meilleur des cas et de manière quadratique dans le pire des cas et le cas aléatoire. Quant au tri fusion, il maintient une complexité quasi-linéaire. Ce dernier est efficace pour trier de grandes quantités de données lorsque nous avons étudié le cas du tri des numéros de sécurité sociale français.

Nous avons ensuite étudié une méthode de permutations qui nous a montré la puissance de la récursion dans la génération de toutes les permutations d'une chaîne de caractères.

Enfin, nous nous sommes penchés sur la fonction d'Ackermann, une fonction récursive qui met en lumière les limites des calculs en termes de complexité.

Toutes ces observations nous ont permis de nous rendre compte de l'importance de produire des algorithmes de moindre complexités pour rendre leur exécution assez rapide, ou du moins réalisable. Or, nous pouvons penser que l'émergence des ordinateurs quantiques pourrait être révolutionnaire dans ce domaine car elle promet des vitesses de calcul exponentiellement plus rapides pour des problèmes complexes. Nous pouvons donc nous poser des questions concernant l'évolution de l'efficacité de nos méthodes actuelles face aux nouvelles capacités offertes par l'informatique quantique.