Anne Sophie Ganguillet,

Julien Bard

& Maxime Scharwath

Date 01.03.2020

**ASD1**

**Rapport Laboratoire :**

**Complexité**

# chercherPosition

## Théorique

La fonction implémente une recherche linéaire dans un tableau de taille *n*, sa complexité est donc de *O(n)*.

Dans le pire des cas, si la valeur est le dernier élément du tableau ou qu’elle ne s’y trouve pas, la complexité est de O(n). Dans le cas moyen il s’agit également d’une complexité moyenne de O(n). Finalement dans le meilleur cas où la valeur est le premier élément du tableau, la complexité est de O(1).

## Pratique

-

# trier

## Théorique

La fonction parcourt tous les éléments successifs du tableau de taille *n* pour les comparer, et elle effectue ce parcours pour tous les éléments du tableau. Ainsi avec comparaisons effectuées au total, sa complexité est de *O()*.

La complexité de cette fonction est indépendante du tableau passé en entrée, car peu importe le résultat de la comparaison, il y a toujours comparaisons.

## Pratique

|  |  |
| --- | --- |
| Taille du tableau | Nombre de comparaisons |
| 6 | 4032 |
| 7 | 16256 |
| 8 | 65280 |
| 9 | 261632 |
| 10 | 1047552 |
| 11 | 4192256 |

La pente que l’on peut observer sur ce graphique à échelle logarithmique ne correspond pas parfaitement à notre estimation de complexité de l’ordre de *O()*. En effet l’évolution observée est de l’ordre de *O()*, donc à mi-chemin entre linéaire et quadratique. Néanmoins nous remarquons que plus les valeurs sont grandes, plus l’évolution tend vers la quadratique ; il s’agit donc d’une limite qui sera atteinte par des valeurs plus importantes.

# chercherSiContient

## Théorique

La fonction implémente une recherche binaire dans un tableau de taille *n*, sa complexité est donc de *O((n))*

Dans le pire des cas, si la valeur est le premier ou dernier élément du tableau ou alors qu’elle ne s’y trouve pas, la complexité est de *O((n))*. Dans le cas moyen il s’agit également d’une complexité moyenne de *O((n))*. Finalement dans le meilleur cas où la valeur est l’élément au milieu du tableau, la complexité est de O(1).

## Pratique

|  |  |
| --- | --- |
| Taille du tableau | Nombre d’itérations |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | 11 |

# f

## Théorique

La fonction s’appelle récursivement 3 fois et ces appels récursifs se font de la valeur initiale *n* jusqu’à atteindre la valeur 1, donc *n* fois. La complexité de cette fonction correspond donc au nombre de branches d’un arbre ternaire de profondeur *n*, c’est-à-dire une complexité de *O(3n)*. Le fait que 2 additions soit effectués à chaque appel n’est qu’un facteur multiplicatif de la complexité.

Cette fonction reçoit comme argument uniquement la valeur entière *n*. Elle est indépendante.

## Pratique

|  |  |
| --- | --- |
| N | Nombre d’additions |
| 11 | 177147 |
| 12 | 531441 |
| 13 | 1594323 |
| 14 | 4782969 |
| 15 | 14348907 |
| 16 | 43046721 |
| 17 | 129140163 |
| 18 | 387420489 |

Comme nous pouvons l’observer avec l’échelle logarithmique de ce graphique, la fonction évolue de façon exponentielle comme nous l’avons estimé. Sa pente ne correspond pas exactement à notre estimation théorique mais elle en est proche et respecte l’ordre de grandeur *O(3n)* estimé.

# g

## Théorique

La fonction parcourt tous les éléments du tableau de taille *n*, et pour chacun de ces éléments … Ainsi effectuant additions au total, sa complexité est de *O()*.

La complexité de cette fonction est indépendante du tableau passé en entrée, car peu importe la valeur des éléments, il y a toujours additions.

## Pratique

|  |  |
| --- | --- |
| Taille du tableau | Nombre d'additions |
| 6561 | 85293 |
| 19683 | 295245 |
| 59049 | 944784 |
| 177147 | 3188646 |
| 531441 | 10628820 |
| 1594323 | 33480783 |

Les résultats obtenus que l’on peut voir sur ce graphique linéarithmique évoluent exactement avec la complexité *O()* comme nous l’avons estimé de manière théorique.

# random

## Théorique

La fonction génère aléatoirement *n* éléments à insérer à la fin du tableau. La génération de ces éléments ainsi que l’insertion des éléments à la fin du tableau sont des opérations amorties constantes et elles sont effectuées pour chacun des éléments. Le temps d’exécution est donc de l’ordre de *O(n)*.

Le temps d’exécution de cette fonction est indépendant de la valeur aléatoire de chaque élément, il est donc toujours de l’ordre *O(n)*.

## Pratique

|  |  |
| --- | --- |
| Nombre d’éléments | Temps moyen |
| 177147 | 9609000 |
| 531441 | 31241400 |
| 1594323 | 96852200 |
| 4782969 | 284821800 |
| 14348907 | 809747800 |
| 43046721 | 2433408400 |

# random2

## Théorique

La fonction génère aléatoirement *n* éléments à insérer au début du tableau. La génération de ces éléments est une opération amortie constante mais l’insertion des éléments au début du tableau est une opération linéaire au nombre d’éléments déjà présents dans le tableau, de plus ces opérations sont effectuées pour chacun des éléments. Le temps d’exécution est donc de l’ordre de *O(n!)*.

Le temps d’exécution de cette fonction est indépendant de la valeur aléatoire de chaque élément, il est donc toujours de l’ordre de *O(n!)*.

## Pratique

|  |  |
| --- | --- |
| Nombre d’éléments | Temps moyen |
| 5000 | 6256000 |
| 15000 | 46871800 |
| 30000 | 181206600 |
| 50000 | 480430600 |
| 75000 | 1077662400 |
| 105000 | 2179729600 |

Nous pouvons observer sur ce graphique à l’échelle logarithmique la rapide augmentation du temps moyen nécessaire à l’exécution de la fonction random2. Cette évolution correspond à notre estimation théorique de complexité de l’ordre de *O(n!)*.