S1.02 - Comparaison d’approches algorithmiques

BIRET-TOSCANO Esteban 4A / FERNANDEZ Mickaël 4A

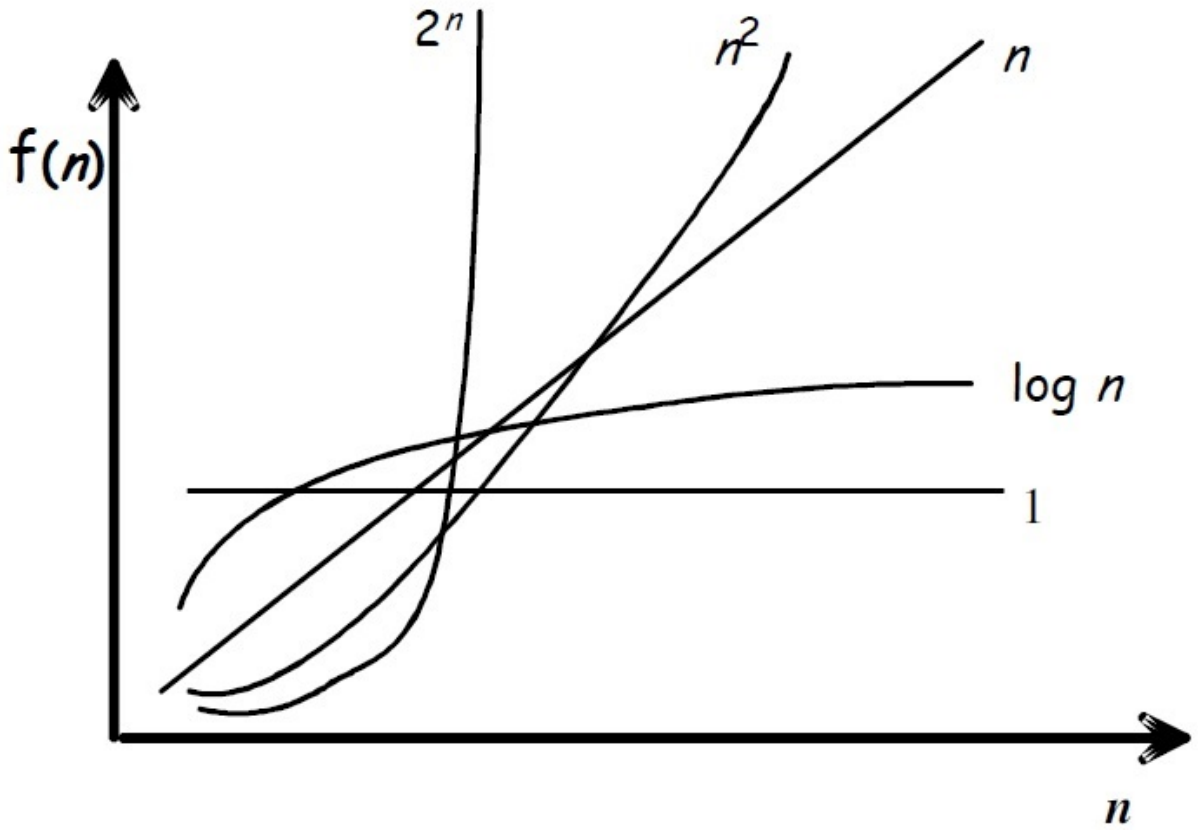


Table des matières :

I – Introduction

L’objectif de cette SAE est, par groupe, de comparer la complexité de différents algorithmes de recherche, afin de conclure sur leurs efficacités. Le projet consiste ainsi à réaliser trois algorithmes de recherche, dont la recherche par dichotomie, la recherche avec rupture et celle sans rupture puis de concevoir des jeux d’essais sur des échantillons de tailles variables afin de pouvoir effectuer ces comparatifs. Nous fournirons également à la fin de ce pdf un lexique en anglais du vocabulaire relatif à la comparaison d’algorithmes.

II – Structures de données

Il était nécessaire, afin de procéder aux exécutions des algorithmes, de récupérer la liste des étudiants, sous forme d'un fichier `csv`, par le biais d'une fonction, ici nommée `getListe`, et de transformer cette liste d'étudiants, en un tableau de chaînes de caractères de la forme suivante : `{{"nom1", "prénom1", "groupeTD1", "groupeTP1"}, {"nom2", "prénom2", "groupeTD2", "groupeTP2"}, ...}`

A - Etudiant

Ainsi, il nous fallait réaliser un constructeur étant conforme aux informations du fichier `csv`. Le constructeur `Etudiant` était le plus représentatif de cette présentation des informations car il nous permettait ainsi, de contenir les différentes informations des différents étudiants (`nom, prénom, numgroupetd et numgroupetp`).

B - TNPEtudiants

Néanmoins, nous avions besoin d'un deuxième constructeur permettant de stocker sous forme de tableau, les informations relatives aux étudiants. C'est la fonction de `TNPEtudiants`, qui permet donc de stocker à la fois, les données des étudiants dans un tableau prénommé `tabEtu` ainsi que le nombre souhaité d'étudiants (`nbEtu`), en provenance du fichier csv.

III – Algorithmes de recherche

Ainsi, en ayant dans un tableau les informations des étudiants, les algorithmes peuvent être appliqués. Afin d'en effectuer une comparaison, un compteur est utilisé dans chacun de ces algorithmes. Celui-ci correspond au nombre d'itérations de l'algorithme exécuté. Effectivement, une comparaison par le temps d'exécution nous permettrait de réaliser une comparaison de complexités plus précise mais nous nous contenterons de les évaluer par le nombre d'itérations.

A – Recherche d'un élément sans rupture

Le premier algorithme, et sans doute le moins compliqué à réaliser, est l'algorithme de recherche d'un élément sans rupture.
Celui-ci consiste ainsi à chercher dans une liste d'éléments, un élément souhaité, figurant ou non dans cette liste. Une fois l'élément trouvé ou non, l'algorithme ne cesse pas son fonctionnement : il continue de parcourir le tableau dans son intégralité.
Ainsi, la fonction `rechercheSansRupture` permettait de faire cette recherche d'un étudiant. Celle-ci est de type booléenne puisqu'elle indique si l'étudiant est présent, malgré que le résultat soit certain.

Le tableau est parcouru à l'aide d'une boucle `for`. A chaque étudiant étudié, le compteur est incrémenté. Si l'étudiant est trouvé, on mentionne que celui-ci le soit sans pour autant arrêter l'exécution de la boucle. A la fin de l'évaluation du tableau, si l'étudiant n'est pas trouvé, un message le mentionne également et une indication sur le nombre d'itérations est affichée.

B - Recherche d’un élément avec rupture

Le deuxième algorithme est l’algorithme de recherche d’un élément avec rupture. Le principe est assez similaire au premier algorithme à la différence que, dans une liste d’éléments, si l’élément souhaité est trouvé, celui-ci arrête son exécution.

La fonction `rechercheAvecRupture` nous permet de faire cette recherche. De même que pour la première fonction, celle-ci est de type booléenne et utilise la même structure conditionnelle et la même structure de boucle. Le compteur est également incrémenté de la même manière. La seule différence que l’on puisse émettre est que, dès que l’élément est trouvé, le booléen change d’état et est directement retourné, arrêtant ainsi le parcours du tableau à l’aide de la boucle `for`.

C - Recherche d'un élément par dichotomie

Le troisième et dernier algorithme est l'algorithme de recherche d'un élément par dichotomie. Celui-ci possède un fonctionnement plus complexe à comparer des deux autres, mais reste dans sa logique assez accessible à comprendre. Cet algorithme consiste à diviser une liste d'éléments par deux et de comparer l'élément du milieu avec l'élément souhaité. Si les deux éléments sont égaux, l'algorithme se termine. Dans le cas contraire, on compare alors ces deux éléments de la manière suivante : est-ce que l'élément souhaité est plus grand que l'élément du milieu ? Si c'est le cas, l'algorithme recommence son exécution dans la deuxième moitié de la liste ou inversement.

La fonction `rechercheDicho` permet alors de réaliser cette recherche de l'élément souhaité. Elle est du même type que les deux autres fonctions mais possède ainsi un fonctionnement différent.

Trois indices sont alors utilisés :

- un indice médian, indispensable dans la recherche de l'élément, étant constamment comparé à l'élément souhaité;
- un indice maximum, variable uniquement dans le cas où l'élément recherché se situe dans la première moitié du tableau. Dans ce cas, cet indice prendra l'indice médian - 1 car à la première exécution de la fonction, l'élément à l'indice médian a déjà été comparé.
- un indice minimum, variable uniquement dans le cas où l'élément recherché se situe dans la deuxième moitié du tableau. Celui-ci prendra alors l'indice médian + 1, toujours dans la même logique.

Ainsi, notre structure de contrôle diffère des autres car c'est une boucle `while` qui est utilisée. Celle-ci nous permet d'étudier la présence d'un étudiant et de toujours valider que l'indice minimum soit plus petit que l'indice maximum, sinon, cela signifie que l'algorithme arrive à sa fin.

Le compteur est également incrémenté de la même manière que les deux autres algorithmes de recherche. Ainsi, en rentrant dans cette structure de boucle, la première structure conditionnelle évalue si le nom

de l’étudiant recherché correspond au nom de l’étudiant à l’indice médian. Si c’est le cas, un message est précisé et la fonction prend fin. Dans l’autre cas, on évalue si le nom de l’étudiant est plus grand que le nom de l’étudiant à l’indice médian. Si les deux noms coïncident, ce seront alors les prénoms qui seront comparés. En fonction de l’état de l’étudiant recherché, l’indice minimum ou maximum changera.

IV – Tableaux de synthèse

Nous avons alors réalisé des jeux d'essais pour ces différents algorithmes de recherche. L'objectif de ces jeux d'essais est de nous procurer la complexité de ceux-ci à l'aide des deux principaux critères suivants :

- la taille du tableau d'étudiant;
- la moyenne d'itérations

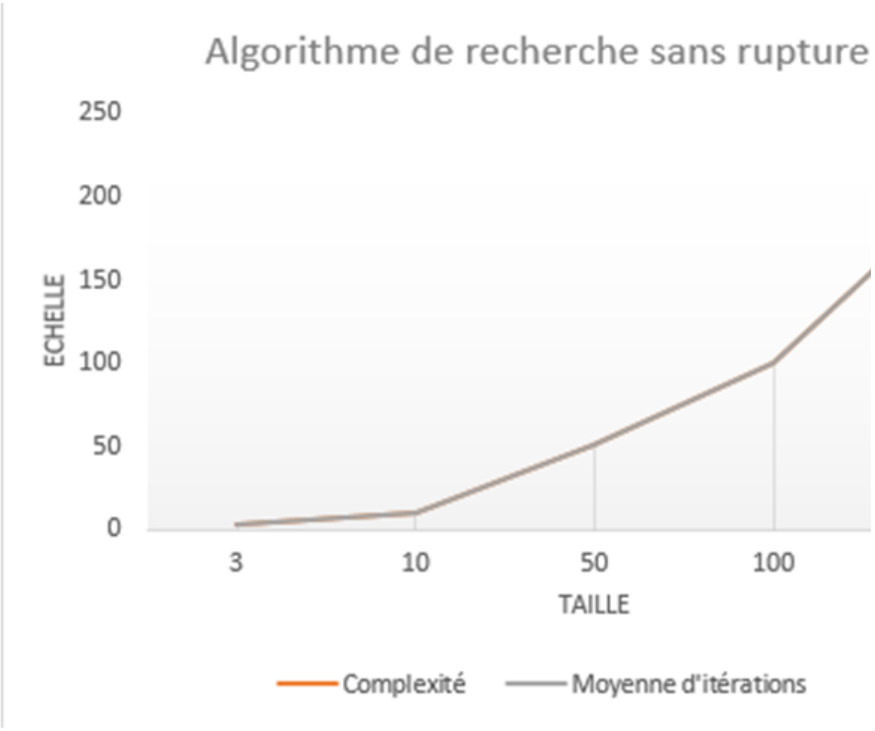
Pour la taille des tableaux, nous avons décidé de réaliser ces expériences pour des tailles variables de 3 étudiants, 10 étudiants, 50 étudiants, 100 étudiants et enfin la liste complète des étudiants (199).
Pour la moyenne des itérations, il fallait prendre en compte le nombre d'itérations pour les étudiants choisis. Nous avons alors choisi 4 étudiants différents, placés à des positions différentes :

- un étudiant placé à la première place du tableau;
- un second, placé à la moitié du tableau;
- un troisième, placé à la fin du tableau;
- et un dernier étudiant assez particulier, car cet étudiant ne figure pas dans le tableau.

Ce dernier choix est également pertinent puisqu'il nous permet aussi d'évaluer cette complexité, d'une manière tout aussi juste.

A - Evaluation de l’algorithme de recherche sans rupture

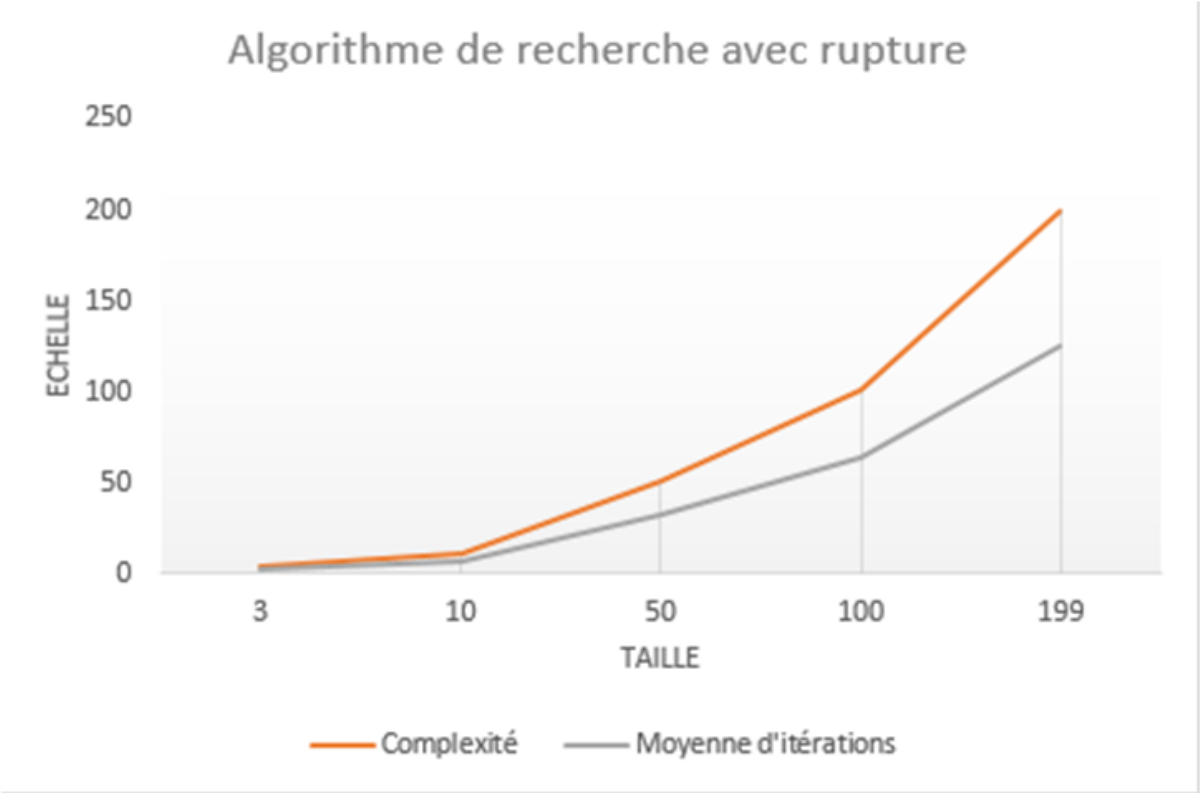
Algorithme de recherche sans rupture						
	Nombre d'itérations					
Taille	Début	Milieu	Fin	Absent	Complexité	Moyenne d'itérations
3	3	3	3	3	3	3
10	10	10	10	10	10	10
50	50	50	50	50	50	50
100	100	100	100	100	100	100
199	199	199	199	199	199	199



D’après la courbe sur le graphique de l’algorithme de recherche sans rupture, celle-ci porte à confusion car il semblerait que la complexité évalue de manière exponentielle, de la même manière que la moyenne d’itérations. Pour autant, la complexité correspond à la taille du tableau ce qui signifie que sur un échantillon plus large, la courbe serait de tendance linéaire. Comme cette complexité est toujours la même, puisqu’elle vaut constamment la taille du tableau, celle-ci vaut alors O(n).

B - Evaluation de l’algorithme de recherche avec rupture

Algorithme de recherche avec rupture						
	Nombre d'itérations					
Taille	Début	Milieu	Fin	Absent	Complexité	Moyenne d'itérations
3	1	2	3	3	3	2,25
10	1	5	10	10	10	6,5
50	1	25	50	50	50	31,5
100	1	50	100	100	100	62,75
199	1	100	199	199	199	124,75

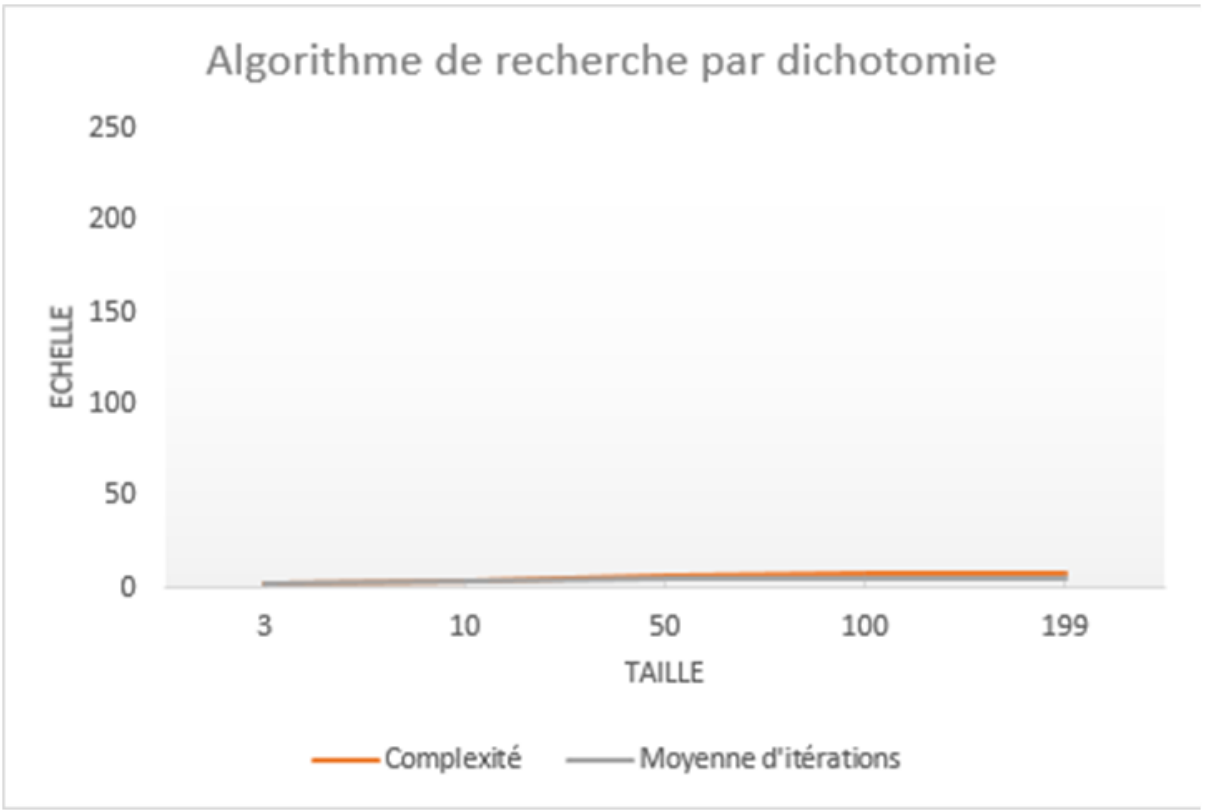


De même que pour l’algorithme de recherche sans rupture, il semblerait que la courbe de la complexité soit de tendance exponentielle mais en réalité, comme cette complexité correspond également à la taille du tableau, celle-ci correspond à une complexité linéaire O(n). Pour autant, sa performance est plus efficace, à en observer la moyenne d’itérations de l’algorithme puisque celle-ci est deux fois plus performante que l’algorithme de recherche sans rupture. Il est logique d’en constater cette différence, car pour une comparaison d’un étudiant situé en première position d’une liste, l’algorithme de recherche sans rupture va parcourir

toute la liste tandis que l'algorithme de recherche avec rupture n'aura qu'une seule itération.

C - Evaluation de l'algorithme de recherche par dichotomie

Algorithme de recherche par dichotomie						
	Nombre d'itérations					
Taille	Début	Milieu	Fin	Absent	Complexité	Moyenne d'itérations
3	2	1	2	2	2	1,75
10	3	1	4	4	4	3
50	5	1	6	6	6	4,5
100	6	1	7	7	7	5,25
199	7	1	8	6	8	5,5



Cette fois-ci, à comparer des deux autres algorithmes, l'algorithme de recherche par dichotomie possède une complexité complètement différente. En effet, cette complexité semble être assez faible, en

observant sa courbe sur le graphique. En découpant un tableau en deux à chaque étape, symbolisé par n/2^k avec n, la taille du tableau en question et k, le nombre d'étapes - 1, on s'aperçoit qu'on arrive à une profondeur maximale du tableau correspondant à log2(n). La complexité de la recherche dichotomique dans le pire des cas correspond ainsi à O(log2(n)). Ainsi, cet algorithme est le plus efficace et le plus performant, quelle que soit la taille du tableau même si un cas est à déplorer : si l'étudiant recherché est situé en première position, les algorithmes de recherche avec et sans rupture seront plus efficaces car ils commencent à parcourir le tableau à partir du premier élément. L'algorithme de recherche par dichotomie commence par le milieu, ce qui demande plus d'étapes pour celui-ci.

Si nous cherchons alors un élément positionné à une place inférieure à log2(n), ce sont les deux autres algorithmes qui sont les plus efficaces.

V – Lexique en anglais

Voici un lexique en anglais comprenant des termes en rapport avec la complexité algorithmique avec leurs descriptions.

Algorithmic complexity	It's a measure of how long an algorithm would take to complete given an input of size n. If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n.
A logarithmic algorithm – O(logn)	Runtime grows logarithmically in proportion to n.
A linear algorithm – O(n)	Runtime grows directly in proportion to n. The linear search is an algorithm for finding a value in a list. It simply looks at the elements of the list one after the other, until the element is found, or all cells have been read.

A superlinear algorithm – O(nlogn)	Runtime grows in proportion to n.
Algorithm	Its the description of a sequence of steps allowing to obtain a result from elements provided as input. For example, a cooking recipe is an algorithm for obtaining a dish from its ingredients.
Comparison	Fact of considering together (two or more objects of thought) to seek their differences or similarities.
Test game	A test game allows you to verify that your program works correctly. We check that the result of the program is the one expected, we must think of all the possible scenarios to be completely sure that the program is 100% functional.
Iteration	In a program, an iteration is the repetition of a block of instructions, this makes it possible to evaluate the algorithmic complexity of a program.
Running time	The running time is the period of time during which a program is running. It starts when a program is executed and ends when the program is stopped or closed.
Elementary operation	In linear algebra, the elementary operations on a family of vectors are algebraic manipulations which do not modify the properties of linear independence. They are easy to describe in code and allow the writing of algorithms.

VI – Conclusion

Suite à cette SAE, nous avons pu constater qu'en termes de complexité, l'algorithme de recherche par dichotomie est le plus efficace même si une nuance s'impose, lorsque la position de l'étudiant recherché est inférieure à log2(n).

Nous avons pu, grâce à cette SAE, renforcer notre niveau en Java, en étudiant de nouvelles notions dans ce langage, mais également en s'intéressant à l'efficacité d'un programme, nous indiquant à la fois la complexité de celui-ci mais également la résolution d'un programme que l'on peut optimiser.