Rapport de projet

Projet de Programmation fonctionnelle OCaml

21/12/2022

Table des matières

1 Introduction

Après avoir écrit les fonctions de tri du projet, il nous est demandé de réaliser une étude sur le temps d'exécution des fonctions et de choisir celle qui nous semble être la meilleure quand la taille de la liste à trier augmente.

Pour ce faire, nous avons utilisé la fonction donnée à la fin du projet permettant de mesurer le temps d'exécution d'une fonction de notre choix. De plus, par soucis de précision nous avons décidé de tester chaque fonction 10 fois et de prendre comme résultat la moyenne des 10 temps d'exécution de la fonction, nous récupérons également le temps maximal et minimal d'exécution afin de minorer et majorer notre résultat.

Nous avons travaillé sur le site Try-Ocaml lors du développement des fonctions de tris. Cependant, lors de la phase d'étude des fonctions nous avons décidé de les tester sur notre machine, par crainte de l'obtention de résultats biaisés. Pour ne pas obtenir de différences de calculs, tous les résultats ont été trouvés sur le même poste, lors de l'exécution des tests, tout processus tournant sur l'ordinateur et pouvant être arrêté, a été arrêté.

Afin d'analyser les temps d'exécutions des fonctions de tri, il a fallu générer des listes à trier, pour cela nous avons utilisé la fonction *liste_aleatoire* fournie dans l'énoncé, celle-ci prenant en compte deux paramètres : une longueur de liste et une borne maximale des nombres générés.

Ainsi, nous avons mené notre étude sur ces deux paramètres en procédant comme suit : Nous avons lancé chaque fonction avec sept bornes maximales différentes : 10, 100, 1000, 2000, 3000, 4000, 5000 et pour chacune de ces bornes, nous avons créé des listes aléatoires de taille : 10, 100, 1000, 2000, 3000, 4000, 5000.

Pour faciliter la génération des données et leur exploitation, nous avons pu développer une fonction permettant d'écrire les temps d'exécution dans un fichier ($print\_in\_file$) prenant en paramètres le temps d'exécution ainsi que le fichier dans lequel ce temps doit être écrit. Ou encore une fonction permettant de lancer n tests ($start\_test$).

2 Analyse du temps d'exécution des fonctions

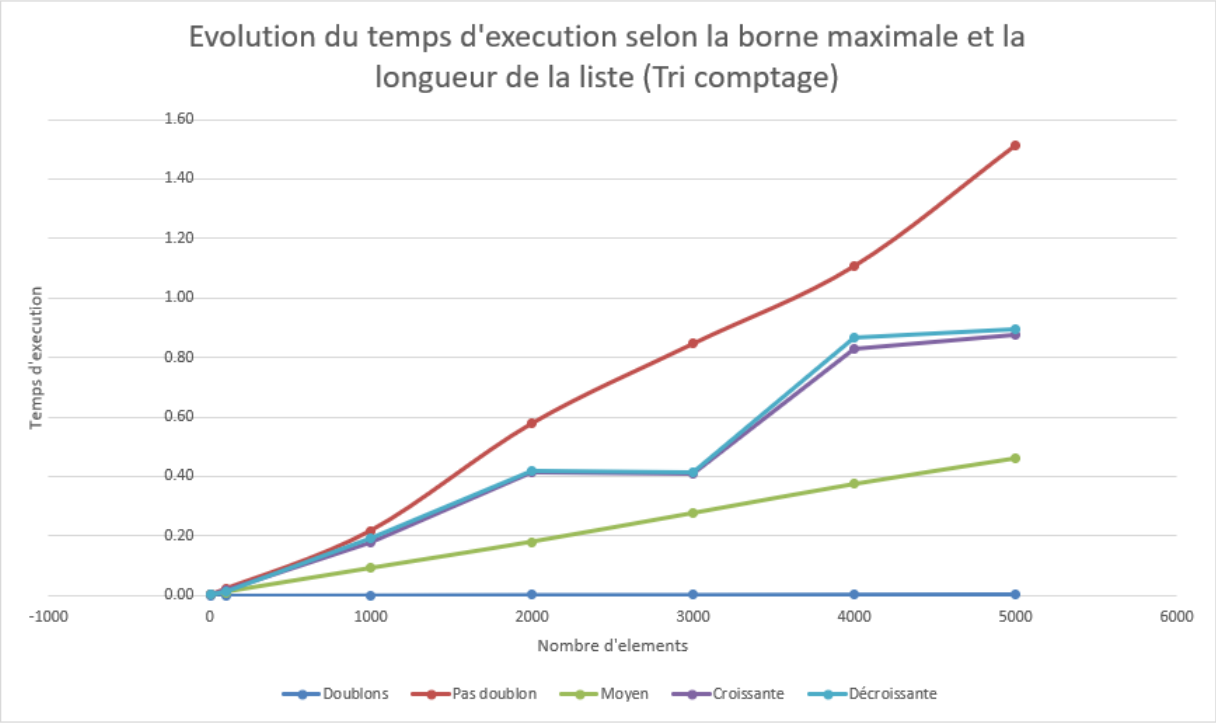
Nous avons étudié une à une le comportement de chaque fonction de tri en fonction de deux paramètres, la taille de la liste à trier ainsi que l'amplitude des éléments de cette liste.

On remarquera que le nombre de doublons dans la liste dépend de sa taille et de son amplitude. En effet, une liste de taille 1000 ayant pour borne maximale 10 aura énormément de doublons et au contraire une liste de taille 100 ayant pour borne maximale 5000 aura très peu de doublons.

Nous considérons dans nos schémas trois courbes différentes "peu doublon", "moyen" et "doublons", "peu doublon" a une borne max de 5000, "moyen" a une borne max de 2000 et enfin "doublons" a pour borne max 10.

De plus, nous présenterons les résultats de chaque fonction lorsqu'elle réalise un tri (croissant) sur une liste déjà triée par ordre décroissant ou par ordre croissant (nous considérerons des listes avec peu de doublons pour ces tests, borne max = 5000).

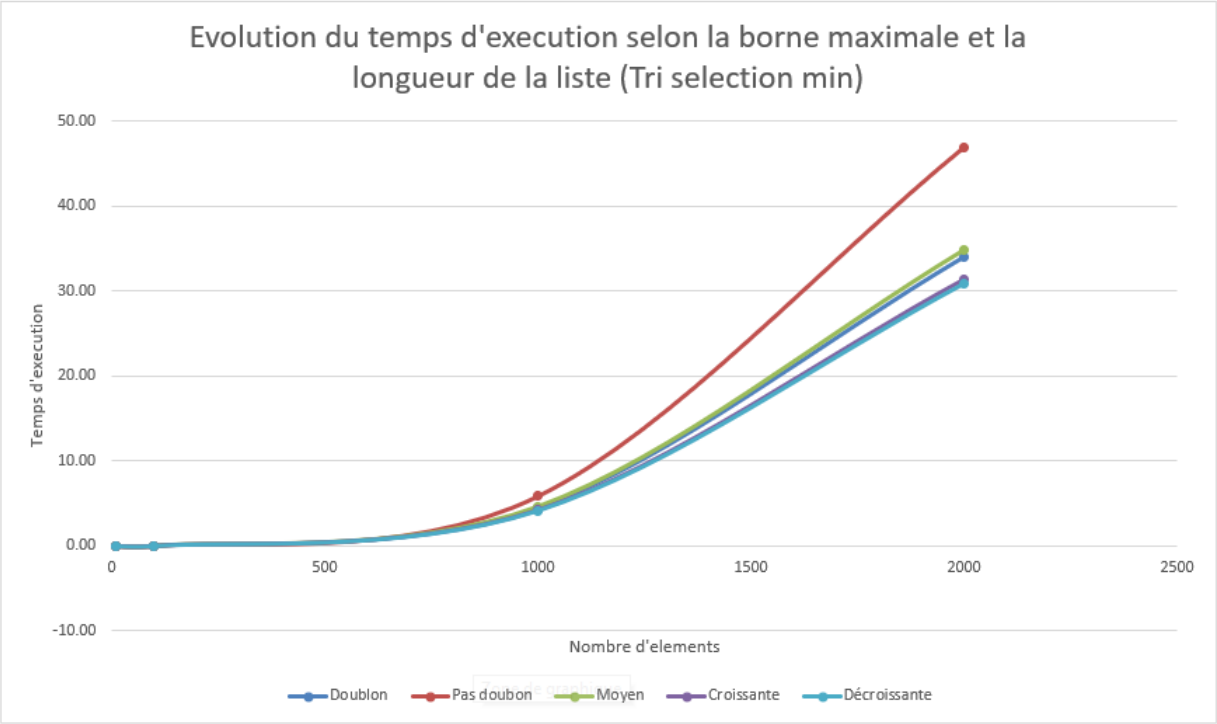
2.1 Tri Comptage



Cette fonction est très sensible au nombre de doublons dans la liste. En effet, nous pouvons remarquer que son temps d'exécution sur une liste possédant beaucoup de doublons est proche de 0 secondes qu'importe son nombre d'éléments. Le temps d'exécution de la fonction augmente de manière significative sur une liste possédant peu de doublons. De plus, on remarque que le temps d'exécution de la fonction diminue lorsque celle ci traite des listes triées.

Ceci peut s'expliquer par le fait que la fonction *tri_comptage* compte le nombre de d'occurrences de chaque élément dans la liste à l'aide de la fonction *nb_chaque*. Ainsi, le nombre de couples renvoyé sera plus grand et donc le nombre d'appels aux fonctions aussi, lorsque la liste contient peu de doublons.

2.2 Tri sélection min

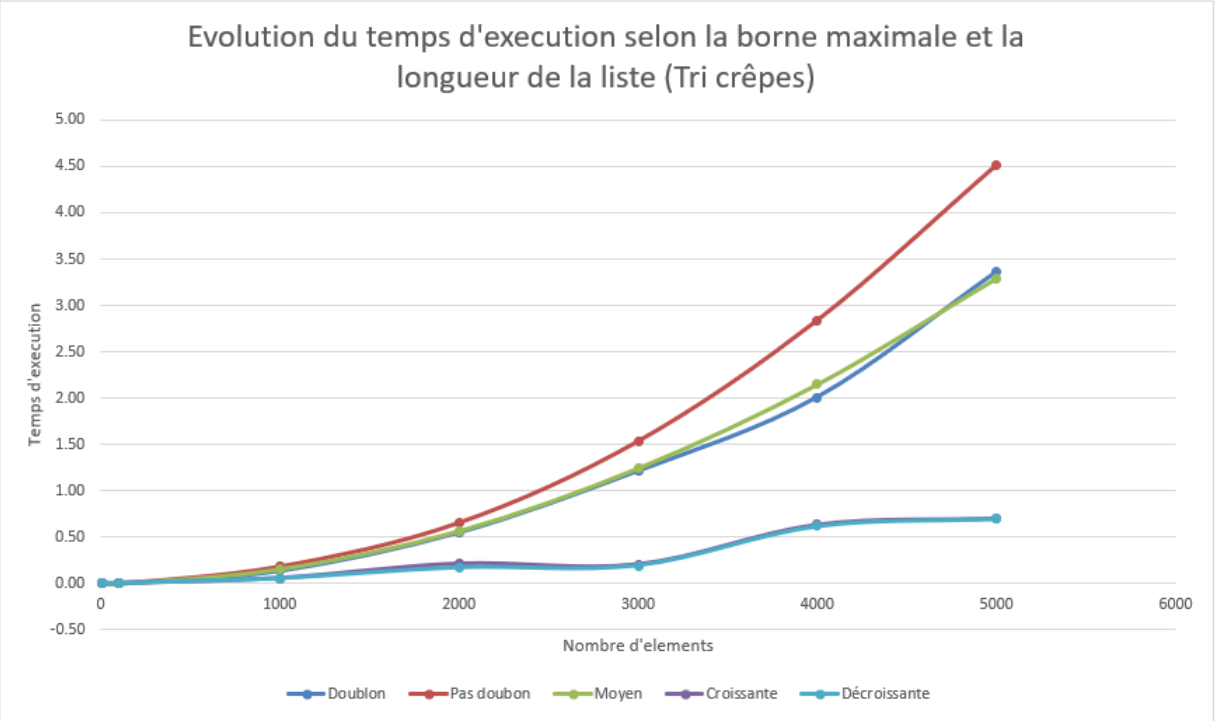


On observe que cette fonction ne dépend pas du nombre de doublons, étant donné l'allure identique des courbes. On remarque que la fonction a un temps d'exécution d'environ 40 secondes pour trier une liste de 2000 éléments, ce qui est très long.

Nous pouvons expliquer l'insensibilité aux doublons de cette fonction par le fait que celle-ci recherche le plus petit élément dans la liste pas encore triée. Ainsi, *tri_selection_min* appelle *min_list_indice* et recherche l'élément le plus petit dans la liste sur l'intervalle de liste pas encore triée.

De plus, nous avons pu remarquer que cette fonction de tri est très sensible au nombre d'éléments à trier. En effet, la fonction compare chaque élément e à tous les éléments de la liste l' (liste pas encore triée) et ceci pour chaque élément e de cette liste. Cela induit un nombre d'appels récursifs très grand réduisant l'efficacité de cet algorithme et le rendant très coûteux en temps.

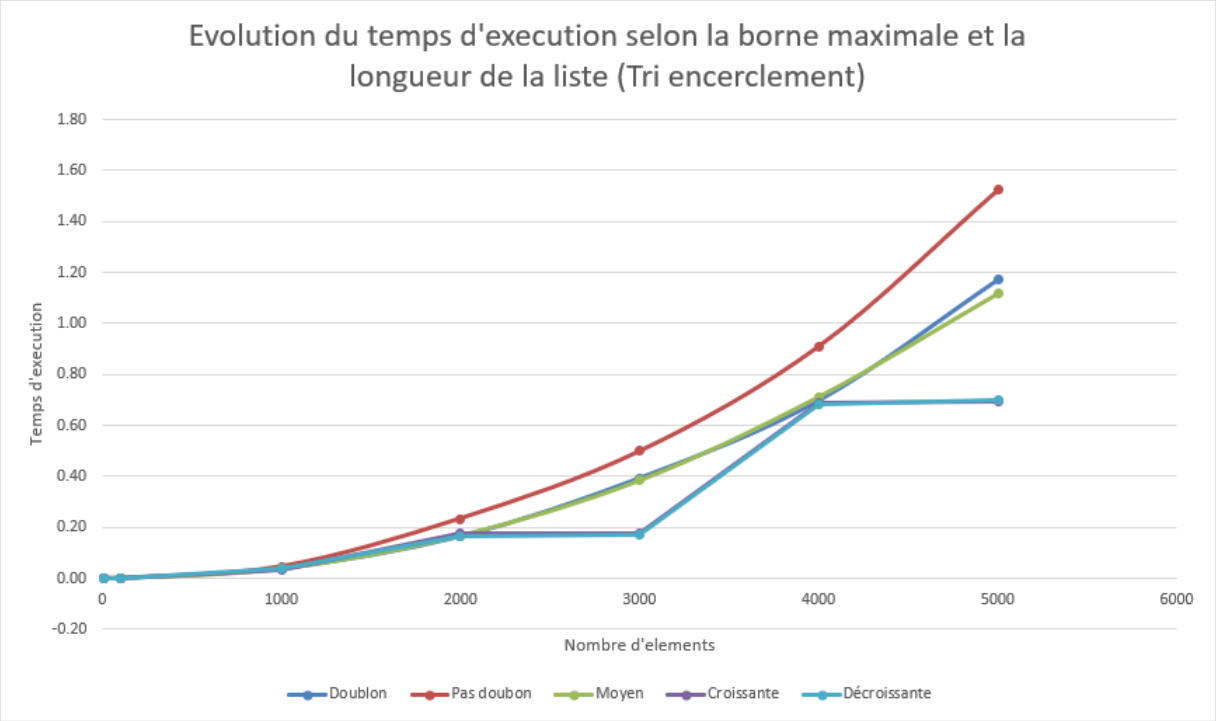
2.3 Tri crêpes



Nous pouvons remarquer que cette fonction est insensible aux nombres de doublons, en effet l'allure des trois courbes est identique. Nous pouvons également observer que peu importe le cas (peu, ou beaucoup de doublons), le temps d'exécution de la fonction augmente significativement lorsque celle-ci trie de grandes listes. De plus, le temps d'exécution est nettement réduit lorsque la fonction traite des listes déjà triées.

Cette fonction de tri est insensible aux doublons, en effet *tri_crepes* effectue des retournements de la liste en insérant judicieusement la spatule (*inserer_spatule*) pour pouvoir la trier. De plus, le temps d'exécution croît en fonction du nombre d'éléments puisque pour insérer la spatule, la fonction doit rechercher le plus grand élément dans l'intervalle de la liste pas encore triée.

2.4 Tri Encerclement



Nous remarquons donc une fois encore que la fonction est insensible aux doublons. En effet, les listes comportant un grand nombre de doublons n'influe pas sur le temps d'exécution. Son temps moyen d'exécution est d'environ 1 seconde.

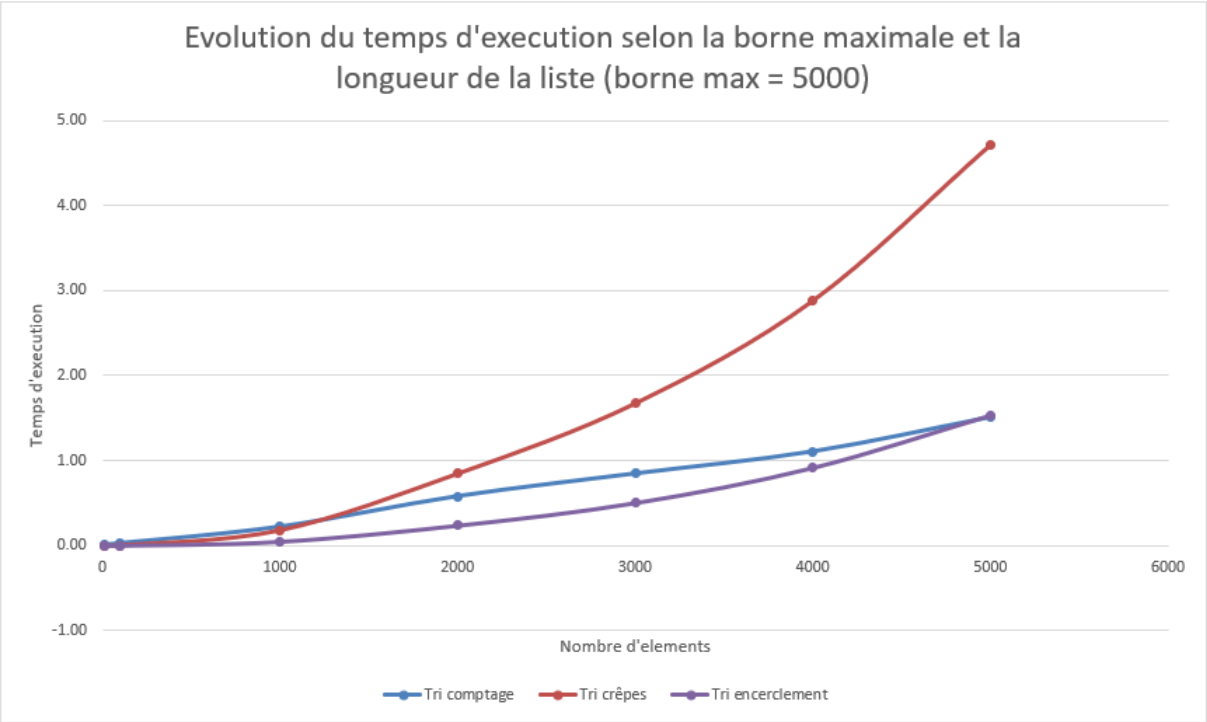
Cette fonction est insensible aux doublons notamment car elle consiste a créer autant de listes à un élément que d'éléments dans la liste. Les doublons ne faisant pas exception, leur présence n'influe en rien sur le temps d'exécution de la fonction, si ce n'est qu'il ne sont pas échangés lorsque la liste est encerclée par la fonction *encercler*.

3 Comparaison des temps d'exécution des fonctions

Nous avons comparé le temps d'exécution de chaque fonction entre elles également pour pouvoir trouver celle qui est la plus rapide et dans quelle situation elle l'est.

Les courbes que nous étudierons ci-dessous ne présente pas la fonction *tri_selection_min* car celle-ci possède un temps d'exécution bien trop grand et occulte les résultats des autres fonctions, c'est pourquoi nous fournirons les courbes contenant le temps de la fonction *tri_selection_min* en annexe.

3.1 Sur liste avec peu de doublons

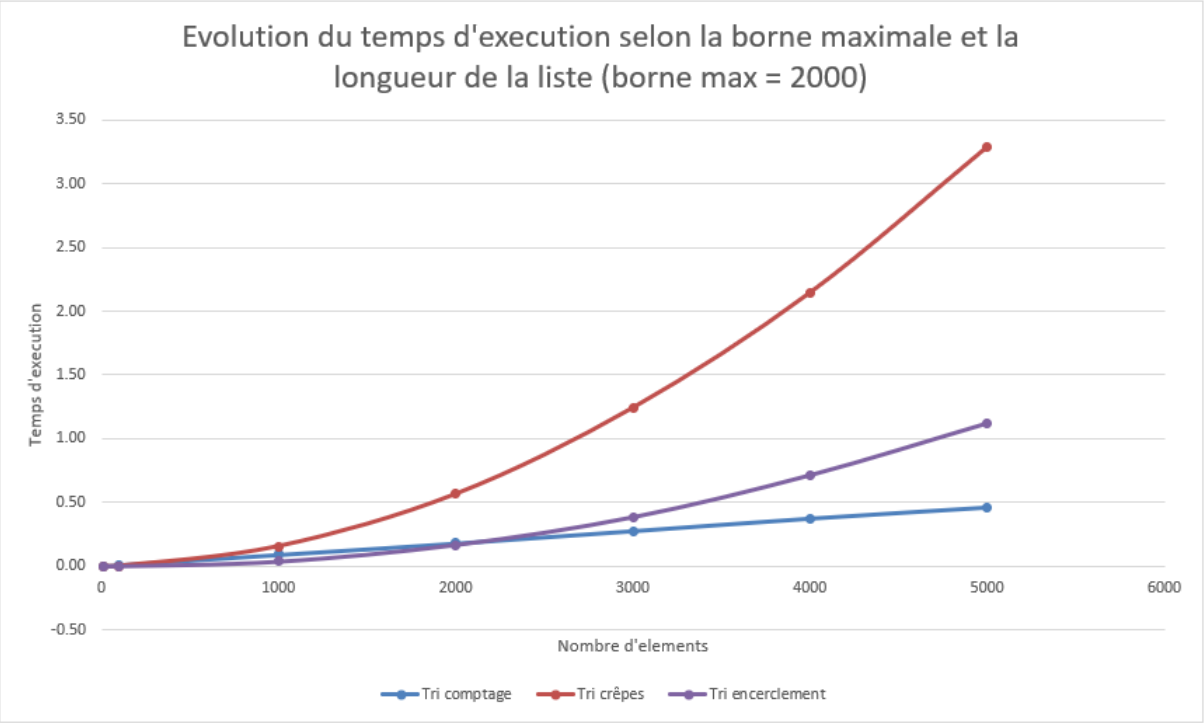


L'analyse effectuée sur les temps d'exécution des algorithmes, est confirmée par la comparaison des fonctions entre elles. On retrouve bien, dans ce cas de listes avec peu de doublons, les fonctions $tri_comptage$ et $tri_encerclement$ qui possèdent les temps d'exécution les plus faibles (1.6 secondes pour 5000 éléments).

Viens ensuite, la fonction tri_crepes, avec un temps d'exécution d'environ 4.5 secondes sur les plus grandes listes.

Enfin, la fonction *tri_selection_min* placée en annexe possède un temps d'exécution de loin supérieur aux autres on peut donc remarquer que la fonction *tri_selection_min* est la moins performante sur les quatre avec un temps d'exécution d

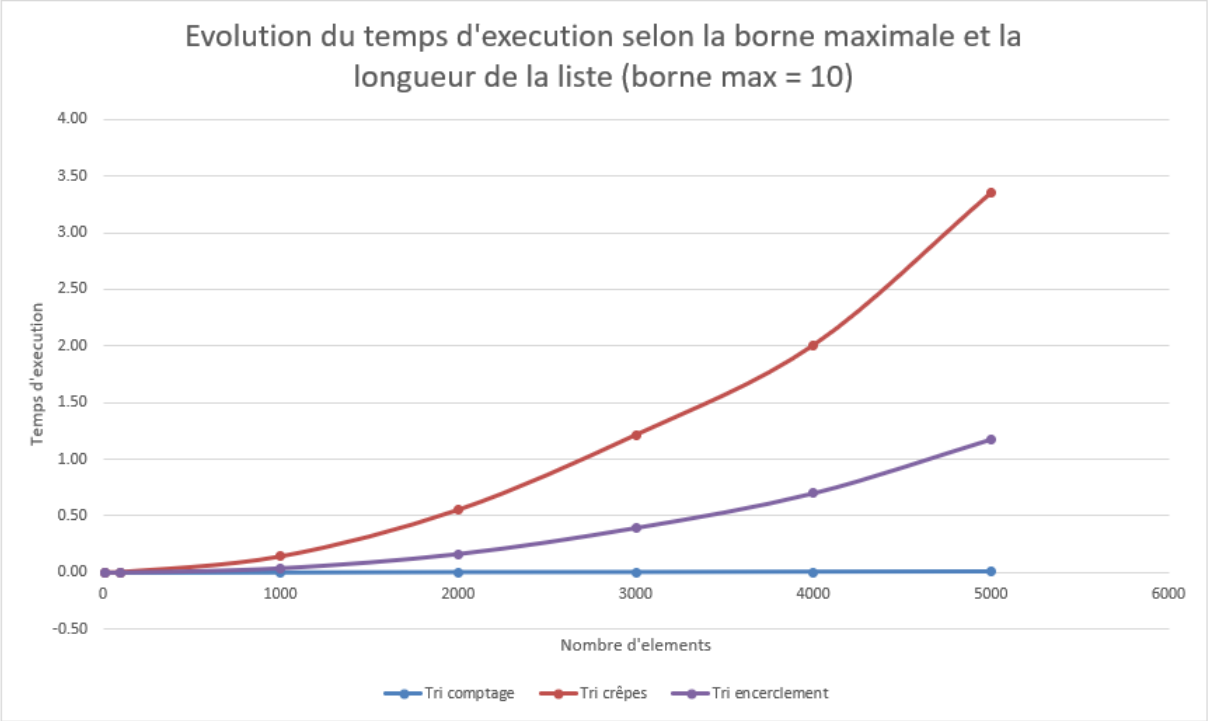
3.2 Sur liste avec un nombre de doublons moyen



Nous pouvons remarquer ici que la fonction la plus performante est *tri_comptage* avec un temps négligeable d'environ 0.5 seconde, ceci est expliquer par la présence d'un nombre de doublons plus grand et donc un temps d'exécution optimisé pour cette fonction.

Viens ensuite, la fonction *tri_encerclement* avec un temps légèrement supérieur, suivi de la fonction *tri_crepes*. Enfin la fonction *tri_selection_min* qui est donc encore la moins performante lors des tries de listes comportant un nombre moyen de doublons.

3.3 Sur liste avec beaucoup de doublons



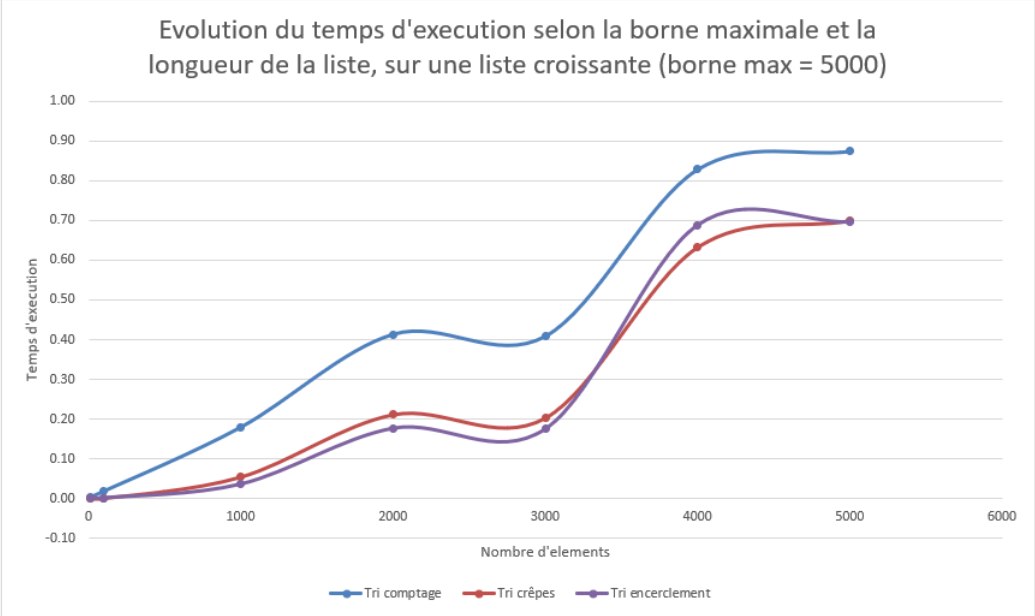
Les constats fait précédemment se confirme, en effet sur des listes contenant beaucoup de doublons a fonction $tri_comptage$ réduit encore sont temps d'exécution la rendant de loin la fonction la plus performante avec un temps d'exécution tendant vers 0 ;

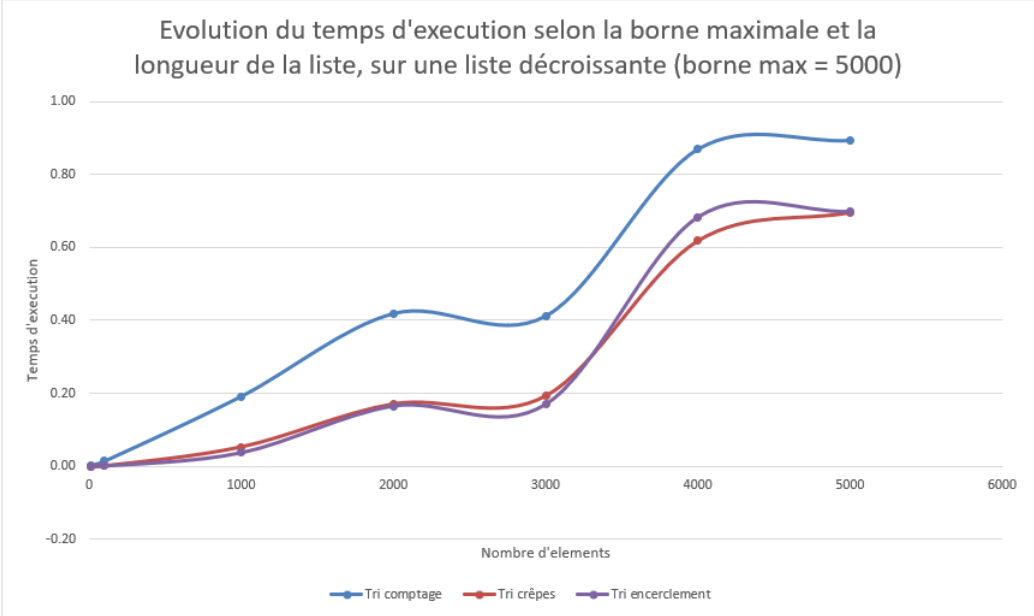
Les autres fonctions ne dépendant pas du nombre de doublons, possèdent le même temps d'exécution et donc leur performance reste inchangée.

3.4 Sur des listes déjà triée

3.4.1 Liste trié aléatoire

Sans exception chacune des fonctions ont vu leur temps d'exécution augmenter (par rapport au listes non triées (quelque soit l'ordre de trie).





Nous avons relevé les temps d'exécution de chaque fonction lorsque l'on trie dans l'ordre croissant une liste déjà triée dans l'ordre décroissant. Les résultats sont similaires aux temps trouvés, lorsque les fonctions trient dans l'ordre croissant, des listes déjà triées dans l'ordre croissant.

4 Conclusion

Après avoir analysé consciencieusement chacune des fonctions du projet, une fonction semble dominer les autres au niveau du temps d'exécution : la fonction *tri_comptage*.

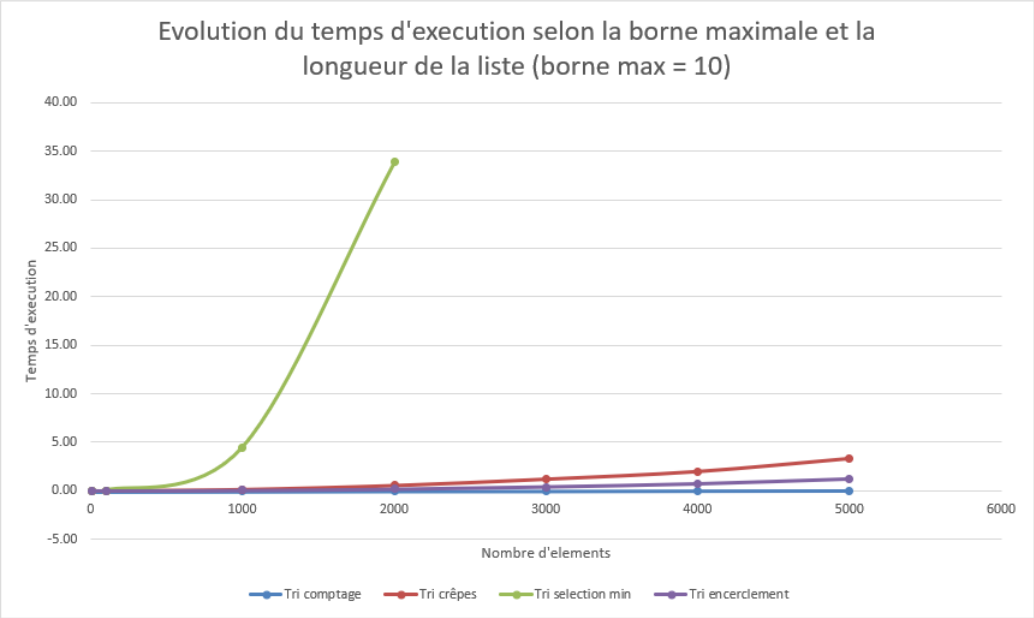
En effet, cette fonction est très performante lorsqu'elle trie des listes avec beaucoup de doublons, mais est également performante comparée aux autres fonctions sur des listes contenant peu de doublons. La seule fonction ayant un temps d'exécution très proche de *tri_comptage* est la fonction *tri_encerclement* lorsque celle-ci traite des listes avec peu de doublons.

De plus, nous avons pu remarquer que sur des listes déjà triées avec peu de doublons, le temps d'exécution de cette fonction est très proche des temps de *tri_crepes* et *tri_encerclement* bien que celle-ci soit légèrement plus rapide.

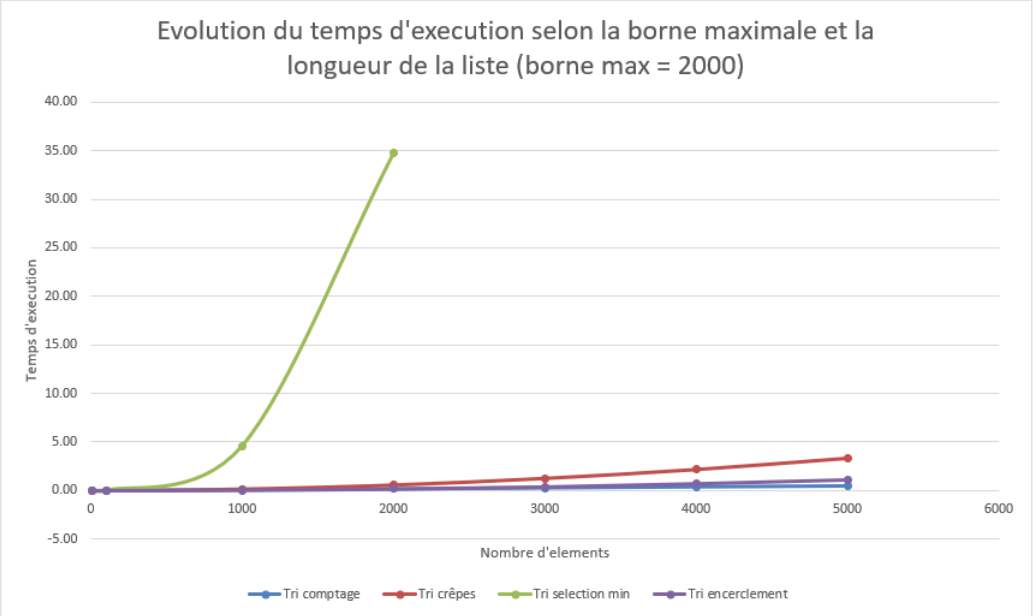
Nous avons donc décidé, de choisir *tri_comptage* comme fonction la plus performante, de par sa rapidité et son faible temps d'exécution quel que soit l'organisation de la liste (liste triée, liste avec ou sans doublons).

5 Annexe

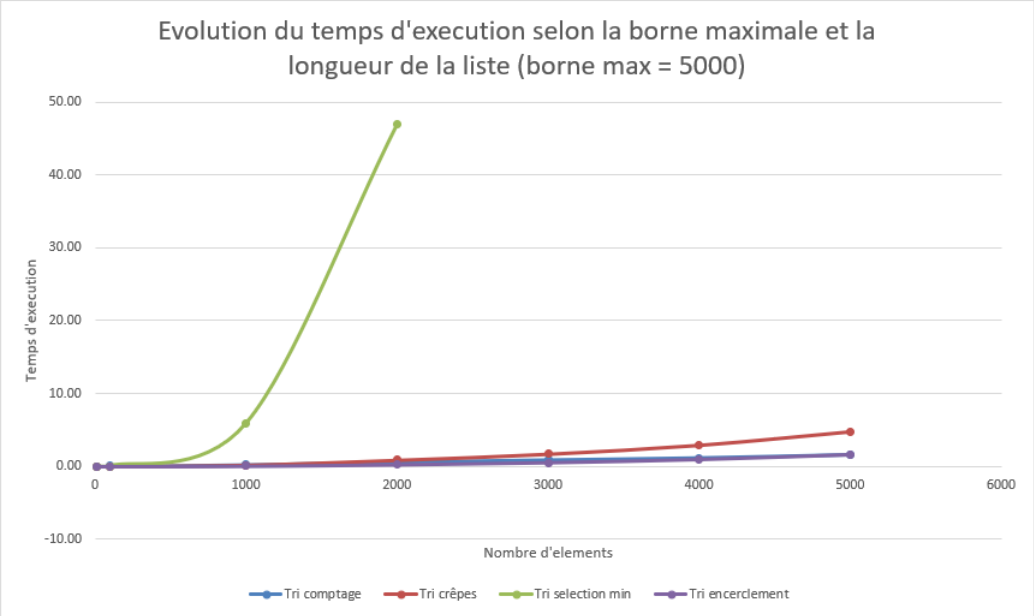
5.1 Sur liste avec peu de doublons



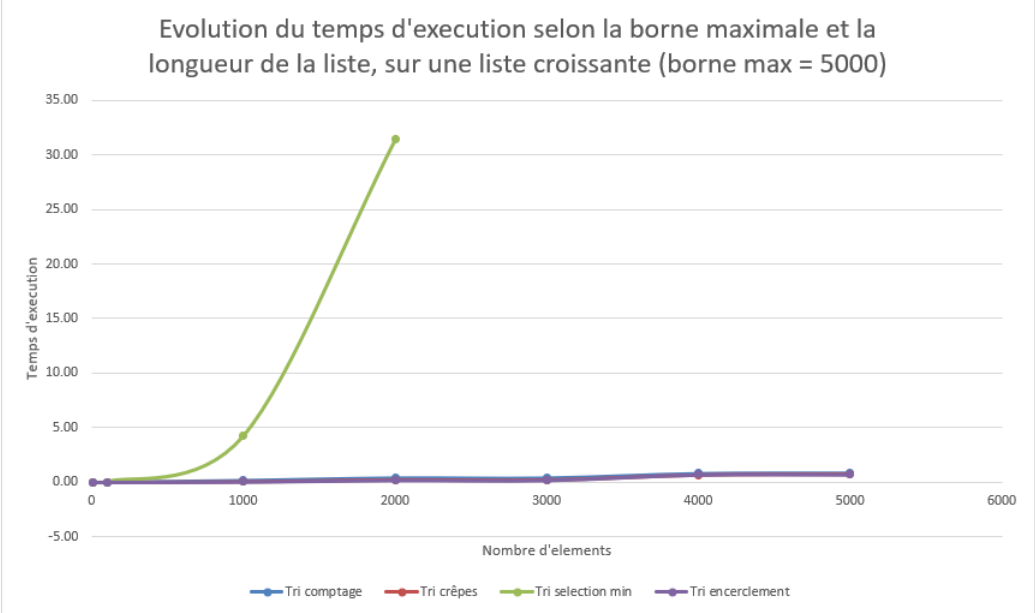
5.2 Sur liste avec un nombre moyen de doublons

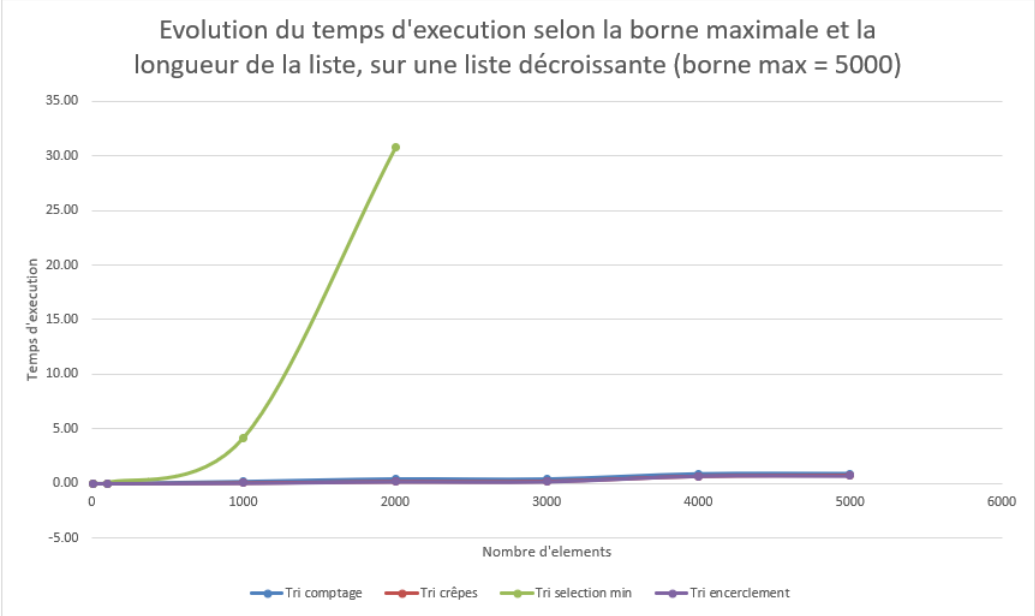


5.3 Sur liste avec beaucoup de doublons



5.4 Sur liste triée





5.5 Tableau des données

5.5.1 Liste avec et sans doublons

Algo	borne_max	10	100	1000	2000	3000	4000	5000
Tri comptage	10	0.0000199000	0.0001010000	0.0008191000	0.0020217000	0.0022987000	0.0033448000	0.0040873000
Tri comptage	100	0.0000686000	0.0006492000	0.0055766000	0.0106056000	0.0162686000	0.0228659000	0.0270609000
Tri comptage	1000	0.0005889000	0.0043078000	0.0508933000	0.0971357000	0.1436432000	0.1998486000	0.2503350000
Tri comptage	2000	0.0015716000	0.0104369000	0.0911232000	0.1801654000	0.2773333000	0.3755777000	0.4625386000
Tri comptage	3000	0.0032485000	0.0170955000	0.1662185000	0.3291986000	0.4980738000	0.7162036000	0.9305625000
Tri comptage	4000	0.0031621000	0.0240006000	0.2095032000	0.3698177000	0.6732635000	0.8924639000	0.8924639000
Tri comptage	5000	0.0038866000	0.0226088000	0.2179140000	0.5782659000	0.8459889000	1.1068879000	1.5127533000
Tri crepes	10	0.0000289000	0.0015336000	0.1403406000	0.5525421000	1.2157925000	2.0091273000	3.3565807000
Tri crepes	100	0.0000274000	0.0020668000	0.1577154000	0.6447540000	1.3285618000	2.1667641000	3.4888124000
Tri crepes	1000	0.0000346000	0.0017507000	0.1444998000	0.5845643000	1.2062062000	2.4674468000	3.4979828000
Tri crepes	2000	0.0000354000	0.0019620000	0.1567407000	0.5668652000	1.2436343000	2.1473632000	3.2916975000
Tri crepes	3000	0.0000348000	0.0018690000	0.1863210000	0.6914650000	1.6743613000	3.0291858000	4.6944109000
Tri crepes	4000	0.0000359000	0.0024032000	0.1858842000	0.6574230000	1.5361966000	2.8393861000	4.5130030000
Tri crepes	5000	0.0000317000	0.0016947000	0.1782800000	0.8448895000	1.6709605000	2.8792577000	4.7132308000
Tri selection min	10	0.0000394000	0.0068169000	4.4888884000	33.9301731000	ERR	ERR	ERR
Tri selection min	100	0.0000425000	0.0076122000	4.4675216000	33.9783175000	ERR	ERR	ERR
Tri selection min	1000	0.0000371000	0.0063690000	4.5988662000	33.4169489000	ERR	ERR	ERR
Tri selection min	2000	0.0000406000	0.0078539000	4.6184570000	34.8174086000	ERR	ERR	ERR
Tri selection min	3000	0.0000449000	0.0084535000	5.9428078000	50.3505558333	ERR	ERR	ERR
Tri selection min	4000	0.0000428000	0.0109668000	6.0016590000	39.4106854000	ERR	ERR	ERR
Tri selection min	5000	0.0000438000	0.0073419000	5.8752784000	46.9171372000	ERR	ERR	ERR
Tri encerclement	10	0.0000380000	0.0008049000	0.0383233000	0.1636398000	0.3927872000	0.6974740000	1.1715145000
Tri encerclement	100	0.0000372000	0.0005924000	0.0392946000	0.1584231000	0.4075384000	0.6652127000	1.1345176000
Tri encerclement	1000	0.0000377000	0.0005702000	0.0380479000	0.1708724000	0.3835302000	0.6914098000	1.1412576000
Tri encerclement	2000	0.0000411000	0.0006529000	0.0364446000	0.1667603000	0.3841854000	0.7126564000	1.1186747000
Tri encerclement	3000	0.0000424000	0.0006786000	0.0467592000	0.2087513000	0.4871458000	0.9323735000	1.4569865000
Tri encerclement	4000	0.0000431000	0.0009055000	0.0485163000	0.1944637000	0.4825568000	0.9419835000	1.4547505000
Tri encerclement	5000	0.0000402000	0.0006070000	0.0461683000	0.2344606000	0.4987455000	0.9100003000	1.5238677000

5.6 Liste triée

LISTE CROISSANTE

Algo	borne_max	10	100	1000	2000	3000	4000	5000
Tri comptage	5000	0.002917	0.0180218	0.1794479	0.4134707	0.4084688	0.8287061	0.8743286
Tri crepes	5000	0.000011	0.000331	0.054268	0.211159	0.202748	0.632651	0.699339
Tri selection min	5000	0.0000219	0.0022286	4.2181394	31.4231891	ERR	ERR	ERR
Tri encerclement	5000	0.0000262	0.0006625	0.0353606	0.1761546	0.1744942	0.6888525	0.6964626

LISTE DECROISSANTE

Algo	borne_max	10	100	1000	2000	3000	4000	5000
Tri comptage	5000	0.0021704	0.0138211	0.1916936	0.4185761	0.4117346	0.8689273	0.8927938
Tri crepes	5000	0.000013	0.000408	0.052595	0.170851	0.192839	0.619226	0.696961
Tri selection min	5000	0.0000246	0.0033729	4.1532889	30.8252664	ERR	ERR	ERR
Tri encerclement	5000	0.0000266	0.0005158	0.0370893	0.1652559	0.1699665	0.6827731	0.6980341