PREFACE

Cette étude est à propos algorithmes de tri, la classification de ces algorithmes, les analyses de tris élémentaires, particulièrement sur le tri par seau « bucket sort », les algorithmes hybrides et l'amélioration sur le tri par seau. L'objet est de fournir au lecteur un arrangement plein des mécanismes principaux des algorithmes de tri et de présenter une nouvelle approche au tri par seau.

Je remercie le Professeur Murat Akın, de l'Institut des Sciences qui m'a proposé cette thèse et qui a encadré mon travail de ses conseils et de son exemple pendant la préparation de thèse.

Le 3 Janvier 2008 / Hüseyin Burak TAKMAZ

TABLE DES MATIERES

LISTE DES TABLEAUX

RESUME

Le problème de tri est un des problèmes fondamentaux dans le domaine informatique. Cet effet a plusieurs raisons : Plusieurs applications utilisent un algorithme de tri. On peut donner comme exemple Microsoft Excel qui trie les nombres d'indexes. Ou bien un programme de graphique, fait du traitement de signal sur une image, trie les couches de l'image par un algorithme de tri. De plus, les algorithmes de tri jouent un rôle important dans les domaines suivants : l'apprentissage de design et l'analyse des algorithmes, les structures de données et la programmation. Il y a encore pleine de recherches sur les algorithmes de tri. On peut aussi considérer le problème de tri comme un problème historique. Il y a eu un grand nombre de chercheurs qui sont travaille sur ce sujet-la et ont publies des livres.

Il n'est pas très facile de déterminer le choix de l'algorithme de tri pour un problème quelconque. Il est fréquent qu'une méthode de tri ne serve qu'une seule fois. Le choix de l'algorithme de tri est reliée a la capacité de mémoire, au nombre de d'éléments à trier, le nombre d'éléments répétés dans la liste à trier. Le choix change aussi si la liste est presque triée ou pas. A cause de ces effets-la, une méthode qui marche très bien avec un cas n'a pas la même performance dans un autre cas. Cette situation rend le problème de tri, un des problèmes fondamentaux pour lequel on fait des recherches depuis des années.

En général, on peut classifier les recherches sur le problème de tri dans quatre catégories : la première catégorie est l'application d'un modèle de statistique aux algorithmes et observation des résultats. La deuxième catégorie est le traitement de l'algorithme en parallèle. De cette façon, les machines qui sont mis en jeu, on l'accès a une mémoire partagée. Ces travaux consistent à mesurer la performance des algorithmes de tri parallèles. La troisième catégorie consiste à l'analyse des algorithmes de point de vue de complexités. Ils se focalisent sur la complexité moyenne, le cas pire et la complexité minimale des algorithmes de tri. Enfin, la dernière catégorie contient

les travaux qui sont concentres sur les algorithmes hybrides. De cette façon, on obtient des algorithmes en combinant différents types d'algorithmes.

Dans cette thèse, on représente une nouvelle approche à l'algorithme de seau (Bucket Sort). On sait déjà, que le tri par seau est un algorithme assez connu dans le domaine de tri. L'idée de base de l'algorithme est divisé puis conquérir. On distribue les nombres dans les seaux dont le nombre est détermine pendant la première étape de l'algorithme. Ensuite, on utilise le tri par insertion pour trier les seaux individuels.

Dans ce thèse, on fait des modifications a la trie par seau, au niveau d'implémentation pour augmenter la performance. De plus, on utilise le tri de Shell pour trier les seaux. Finalement, on obtient une nouvelle approche pour le tri par seau.

L'algorithme de tri par seau, étant un algorithme très flexible, est un bon choix pour appliquer diverses techniques d'implémentations. On propose de produire des objets d'une façon différente et on utilise des listes chaînées avec le langage C++ pour augmenter la performance. De plus, on change l'ordre des étapes de l'algorithme pour mieux profiter de la mémoire.

On sait qu'on utilise traditionnellement le tri par insertion pour le tri par seau. Par la nouvelle approche, on propose d'utiliser le tri de Shell – qui est quasiment un tri par insertion – pour trier les seaux. Pour cette raison, on fait des petites modifications sur le tri de Shell, pour la rendre utilisable dans le tri par seau. Enfin, on intègre le tri de Shell modifié au tri par seau.

On mesure la performance obtenue et on compare les temps d'exécution de la nouvelle approche par différents d'algorithmes : tri de Shell, tri par Seau (traditionnelle), tri à bulles. En conséquence, on constate que la nouvelle approche marche bien avec des listes (produites par hasard) de tailles différentes et grandes.

ABSTRACT

The sorting problem is one of the fundamental problems in the informatics space. This situation has a lot of reasons: many applications use a sorting algorithm. One can give the example of Microsoft Excel which sort the indexes numbers. Or another graphic application, process the signals of the images, sorts different layer of the image with a sorting algorithm. Moreover, the sorting algorithms play important role in the following domains: teaching of design and analysis of algorithms, data structures and programming. There are still great deals of ongoing researches which are related to sorting algorithms. Because, many heroes of informatics have studied and published books.

It is not too simple to decide a choice between sort algorithms for a random problem. It is frequent that a method works once and doesn't work another time. The choice of algorithm is strongly related to the memory capacity, the number of elements for sorting, and the number of redundant elements. The choice changes also in the case that whether the input list is almost sorted or not. Because of these arguments, a method which is working very well for a case doesn't have the same performance in another case. This situation the sorting problem, one of the fundamental problems that a lot of studies still continue for.

Generally, one can classify the studies on sorting problem into four categories : the first category is the application of a statistical model onto sorting algorithms and observation of results. The second category is the parallel processing of the algorithms. In this way, the machines in the domain have access to a shared memory. These studies aim to measure the performance of parallel sorts. The third category consists of the analysis of algorithms from the complexity point of view. They are focusing in the average complexity, the worst case and the minimal complexity of sorting algorithms. Finally, the last category includes the studies which are concentrated on hybrid algorithms. This can be done thanks to combination of different algorithms.

In this thesis, one represents a new approach to “Bucket Sort” algorithm. One already knows that the “Bucket Sort” is well known in the sorting domain. The basic idea comes from divide-and-conquer. One distributes the numbers into buckets whose capacities are determined during the first phase of the algorithm. Than, one uses the insertion for sorting the individual buckets.

In this thesis, one realizes some modifications in the Bucket Sorting in the implementation level for improving the performance. Additionally, one uses Shell Sort for sorting the buckets. In the consequence, one gets a new approach for the bucket sorting.

Bucket sorting, being a flexible algorithm, is a good choice for implementing different implementation techniques. One proposes creating objects in a usual way. One proposes linked lists with the C++ language for improving performance. Moreover, one changes the order of algorithm’s phases for better using the memory.

One knows that insertion sort is traditionally well used in bucket sorting. With the new approach, one proposes the use of Shell – which is a kind of insertion sort – for the buckets. For this reason, one makes some small modifications on Shell sort. Hence, one integrates the “Modified Shell” to the bucket sorting.

One measures the performance and compares execution times of the new approach with different algorithms: Shell sort, bucket sorting (traditional), bubble sort. Finally, one examines that the new approach works well with different size of lists as input.

ÖZET

Sıralama problemi enformatik alanındaki en temel problemlerden biridir. Bu durumun birçok sebebi vardır: Birçok uygulama sıralama algoritması kullanmaktadır. Bir resim üzerinde sinyal işleme yapan bir uygulama da, resmin katmanları üzerinde işlem yaparken sıralama algoritması kullanmaktadır. Buna ilaveten, sıralama algoritmaları, şu alanlarda da önemli bir rol oynamaktadır: algoritmaların tasarımı ve öğrenilmesi, veri yapıları ve programlama. Sıralama algoritmaları üzerinde halen yoğun bir araştırma sürdürülmektedir. Sıralama problemini tarihsel bir problem olarak da düşünülebilir. Zira birçok bilişim kahramanı bu konu üzerinde çalışmış ve kitaplar yayınlamıştır.

Rasgele bir problem için, sıralama algoritmalarından hangisinin kullanılacağına karar vermek çok kolay olmamaktadır. Algoritmanın seçimi; hafıza kapasitesi, sıralanacak elemanların sayısı, listede tekrarlanan eleman sayısı ve listenin tam ya da kısmi sıralı olması gibi birçok parametreye bağlıdır. Bu nedenlerden dolayı, belli bir durumda iyi bir performansla çalışan bir algoritma, başka durumlarda aynı performansı gösterememektedir. Bu durum, sıralama problemini senelerden beri üstünde araştırmalar yapılan bir alan haline getirmiştir.

Genelde, sıralama problemi üzerine yapılan çalışmaları dört kategoriye ayırabiliriz: ilk kategori, istatistikî bir modelin sıralama algoritmalarına uygulanması ve sonuçların gözlemlenmesidir. İkinci kategori, algoritmanın paralel şekilde çalıştırılmasıdır. Burada, paralel makinelerin paylaşılan bir hafıza ayarına erişmesi söz konusudur. Üçüncü kategori ise algoritmaların kompleksite açısından analizine dayanmaktadır. Bu çalışmalar ortalama kompleksite, en kötü durum ve minimum kompleksite hesapları ile meşgul olmaktadır. En son kategoride hibrid biçimde elde edilen algoritmaları kapsamaktadır. Hibrid algoritmalar ise, değişik algoritmaların birleşimlerinden oluşmaktadır.

Bu tezde, kova sıralamasına yeni bir bakış açısı getirilmektedir. Kova sıralamasının sıralama alanında çok tanınan bir algoritma olduğu bilinmektedir. Algoritmanın temel mantığı böl-ve-yönettir. Sıralanacak elemanlar önce kovalara dağıtılmaktadır. Her kovadaki elemanın sayısı algoritmanın ilk etabında belirlenmektedir. Bunun devamında ise, kovalardaki elemanlar "insertion" sıralaması ile sıralanmaktadır.

Bu tezde, kova sıralama algoritmasında gerçekleme aşamalarında değişiklik yapılmaktadır. Ayrıca, kovaları sıralamak için "Shell" sıralama algoritması kullanılmaktadır. Sonuç olarak, kova sıralaması için yeni bir bakış açısı kazanılmaktadır.

Kova sıralaması, çok esnek bir algoritma olmasından dolayı, farklı gerçekleme teknikleri kullanmak için güzel bir seçim olma durumundadır. Bu çalışmada, nesnelerin yaratılması için farklı bir yaklaşım ve zincirlenmiş listeler kullanılmıştır. Ayrıca, algoritmanın etaplarının sıraları değiştirilerek, hafızadan daha fazla yararlanılmaktadır.

Kova sıralaması için, "insertion" sıralamasının yaygın olarak kullanıldığı bilinmektedir. Yeni yaklaşımda ise, "Shell" sıralama algoritmasının –genel manada "insertion" sıralaması olan bir algoritmadır- kullanılması teklif edilmektedir. Bu amaçla, "Shell" algoritması üzerinde bazı değişiklikler yapılarak, kova algoritmasında kullanılmaya uygun hale getirilmiştir. Daha sonra ise, değiştirilmiş "Shell" algoritması ile kova algoritması entegre edilmiştir.

Elde edilen performans ölçülmüş ve yeni yaklaşımın çalışma süresi diğer algoritmalar ile karşılaştırılmıştır. Bu algoritmalar şunlardır : "Shell" sıralaması, kova sıralaması, kabarcık sıralama. Sonuç olarak, yeni yaklaşımın orta ve büyük çaplı listelerde iyi çalıştığı gözlemlenmiştir.

1. Introduction

Depuis l'avènement de l'informatique, les ordinateurs traitent de plus en plus de l'information. Cela est du a l'évolution de mémoire et microprocesseur. En même temps, le génie logiciel a avance aussi avec de grands pas. Pendant les dernières décades, les études sur les algorithmes de tris, ont été l'une des domaines les plus étudiées à l'informatique.

On peut très bien profiter des algorithmes de tris, pour développer les méthodes d'implémentations, les calculs de complexités et les structures de donnée. Car, les algorithmes de tris souvent font l'usage des structures de donnée. On peut toujours utiliser de nouveaux méthodes ou structures de données dans ces algorithmes. On peut constater aussi que les algorithmes qui sont compliqués à comprendre et à implémenter, marchent plus vite par rapport aux autres. Dans d'autres mots, on peut dire aussi que plus l'algorithme devient complique, plus la performance augmente. D'autre part, on peut toujours faire l'analyse de l'usage de mémoire et de microprocesseurs pour les algorithmes de tris. Par exemple, si l'algorithme est récursif comme le tri « quick sort » ou bien « merge sort » l'usage de mémoire est excessivement grand quand le nombre à trier est de l'ordre de centaines de milliers.

Si on veut bien comprendre de plus prés les calculs de complexités, ce sont les algorithmes de tris qu'on peut bénéficier. Parce que, dans un algorithme de tri, on fait trop de tours, comparaisons et échanges. A cause de ça, les complexités deviennent quelquefois difficiles à calculer. On met en œuvre les calculs d'algèbre qui sont assez compliques. Donc, pour analyser les algorithmes de tri, on doit avoir aussi une bonne connaissance d'algèbre.

Cette thèse consiste à augmenter la performance de tri par seau appelé en Anglais comme « Bucket Sorting ». Cet algorithme est très modulaire et convenable pour

appliquer différentes techniques d'implémentation. On va exprimer toutes les étapes de la nouvelle approche dans les sections futures.

Pour mieux comprendre la nouvelle approche, on va examiner les algorithmes de tri tout d'abord. On va étudier les algorithmes de tris en deux phases. La première phase est assez théorique mais il donne de l'information qui est nécessaire pour connaitre de plus prés les algorithmes de tri. D'abord on va étudier les algorithmes de tri qui ont été utilises ou bien qui ont été référencés dans les sections. Pour cela, on va d'abord donner l'idée de base, de l'information générale et le pseudocode de l'algorithme. On va détailler aussi le fonctionnement de ces algorithmes qui sont assez connus dans le domaine informatique. Puis, on va analyser l'algorithme de point de vue complexité et performance.

Ensuite on va examiner les articles qu'on s'est servit pour la nouvelle approche. Ces articles ont construit l'infrastructure de cette thèse. On va résumer ces articles et expliquer ses points importants. Puis, on va dire les améliorations qu'on fait dans le tri par seau. On va aussi examiner le code écrit en C++ pour mieux comprendre, comment marche la nouvelle approche.

2. Les Algorithmes de Tri

Dans le domaine informatique, le tri est divisé en deux catégories : le tri numérique ou lexicographique. On va s'occuper seulement de tri numérique dans ce travail. Pour résoudre le problème de tri, on a propose beaucoup de solutions. On les appelle les algorithmes de tri. On va reconnaître les différents algorithmes et on va analyser ces algorithmes. Tout d'abord, on va caractériser les algorithmes de tri et grouper celles qui se ressemblent selon un paramètre.

2.1. Classification de Tris

On peut caractériser les algorithmes de tri de deux façons. Cette classification est importante, parce qu'elle permet de choisir l'algorithme le plus adapte au problème qu'on s'est rencontre.

2.1.1. Classification Selon Comparaison

On va se rappeler les caractéristiques tout d'abord des algorithmes de tris. On peut classifier les ces algorithmes en deux catégories :
Cette classification est basée à comparaison :

- Les tris *basés à comparaison*
- Les tris qui sont *basés à non-comparaison*

Si le tri marche en « comparant » les nombres avec les operateurs « < » et « > » le tri est dit « base a comparaison », en Anglais on dit « compared base sorting ». On peut donner comme exemple pour ce type d'algorithmes : tri par insertion, tri à bulles et tri par sélection, tri rapide (quicksort), tri par tas (heap sort) et tri de Shell.

Les tris bases a comparaison qui sont simples comme tri a bulles, sélection et insertion ont une complexité de $O(N^2)$ pour le cas pire et en moyenne. Les autres algorithmes qui sont plus compliqués comme tri quicksort, heapsort, mergesort ont un complexité de l'ordre de O(Nlog(N)). En fait, les algorithmes qui font l'usage de comparaison entre

les éléments de l'entrée de n éléments, nécessitent au minimum Ω(Nlong(N)) comparaisons[2].

Si le tri n'est pas base a comparaison, on dit que le tri est base a non comparaison. On peut donner comme exemple le tri par base (radix sort) qui est un tri linéaire. Ce tri prend le chiffre le moins significatif et trie la liste d'entrée selon ce chiffre. Il répète ces opérations avec chaque chiffre plus significatif.

Le tri comptage (counting sort) est aussi un algorithme qui est base à non-comparaison. On va examiner cet algorithme de plus près dans un article. Ce tri a une complexité linéaire de l'ordre de O(n). Par contre, son utilisation relève la condition que les éléments soient des entiers naturels.

Bien que, les complexités pour ces algorithmes soient précises, il y a eu beaucoup d'implémentations de ces algorithmes qui ont essayé d'augmenter les performances.

2.1.2. Classification Selon l'Usage de Mémoire

On peut classifier les algorithmes de tris en termes de mémoire. Il se trouve deux classes d'algorithmes selon l'usage de mémoire. Ces classes sont :

- *Les tris internes*
- *Les tris externes*

La plupart des algorithmes de tri peuvent être effectues dans la mémoire. Ils assument aussi que l'accès à la mémoire est rapide. Si la taille de la liste à trier est de l'ordre quelques millions, le tri peut marcher dans la mémoire complètement. Ces tris sont appelés des « tris internes ».

Comme les algorithmes assument que l'accès au mémoire est très rapide, ils ne sont pas le meilleur choix, quand la taille de la liste est trop grande. Particulièrement, si on ne peut pas tenir toute la liste dans la mémoire, on se sert des algorithmes appelés « tris externes ». Elles utilisent le disque dur pour les opérations. Par contre ces algorithmes ne sont pas références et utilises dans cette thèse. On peut trouver des variantes de tri fusion et tri rapide.

2.1.3. Classification Selon Complexité

Quand on détermine le choix d'un algorithme pour un problème quelconque, on regarde les complexités des algorithmes. Les complexités dans le cas pire, dans le cas meilleur et en moyenne ne sont pas égales dans certains algorithmes. Pour ceci, il faut examiner l'entrée. La décision change par rapport à l'entrée. Apres avoir examine l'entrée, on peut décider quel algorithme on va utiliser.

L'idée de base n'est pas mesure seulement le temps d'exécution et le nombre de comparaisons. Par contre, l'idée de base est de comparer le nombre d'opérations élémentaires (addition, soustraction, multiplication) et le nombre d'échanges.

Par exemple, si la liste a peu près mise en ordre. C'est-à-dire il faut échanger seulement un petit pourcentage des éléments, on peut utiliser le tri par insertion. Il faut bien connaitre le comportement de l'algorithme dans le cas pire, le cas meilleur et en moyenne. En conclusion, il faut comparer les complexités des algorithmes dans ces cas pour la liste d'entrée.

2.1.3.1. Comparaison des Algorithmes Selon Complexité

Pour mieux comprendre on va décrire les complexités des algorithmes dans un tableau. Dans le tableau, le nombre à trier est n. On donne les valeurs des complexités en moyenne, dans le pire des cas et spatiale [8].

Table 2.1 : Complexités des algorithmes de tri

Algorithme De Tri	**En moyenne**	**Le cas pire**	**Spatiale**
Tri à bulles	-	$O(n^2)$	O(1)
Tri par sélection	$O(n^2)$	$O(n^2)$	O(1)
Tri par insertion	$O(n^2)$	$O(n^2)$	O(1)
Tri de Shell	-	$O(n\log^2 n)$	O(1)
Tri rapide	O(nlog(n))	$O(n^2)$	O(logn)
Tri par fusion	O(nlog(n))	O(nlog(n))	O(n)
Tri par tas	O(nlog(n))	O(nlog(n))	O(1)

- *La complexité en moyenne :* C'est le nombre des opérations élémentaires pour l'algorithme. Le calcul de ces opérations élémentaires est parfois long et difficile à calculer.

- *La complexité dans le pire des cas :* C'est le nombre supérieur d'opérations qui sont nécessaires.

- *La complexité spatiale :* C'est l'utilisation de mémoire nécessaire pour l'algorithme.

La première remarque qu'on peut faire, c'est que le complexité au minimum pour les complexités des algorithmes bases a comparaison est de O(nlogn). Donc, au minimum, la complexité est logarithmique. Ce sont des algorithmes plus élaborés que les autres algorithmes.

On constate aussi des complexités quadratiques. Mais, ces algorithmes marchent plus lentement que les autres.

Les complexités servent à mesurer la rapidité des algorithmes. Pour cette situation, donnons un exemple, le complexité de tri par insertion est $O(n^2)$. Il existe aussi le tri par tas, dont le complexité est de l'ordre de O(nlog(n)). Il reste à trouver l'intervalle ou la complexité de l'une des algorithmes est plus grande que l'autre. Quand on fait des calculs, on observe qu'a partir de n = 5000, le tri par tas est plus rapide.

Quant à la complexité spatiale, c'est soi O(n) soi O(1). Les complexités de grandeur O(1), n'ont pas besoin d'un extra espace de mémoire. Seule, la mémoire utilise pour l'échange de deux nombres est obligatoire. Le tri rapide est récursif et a besoin du mémoire pour marcher l'algorithme. De même, le tri fusion a besoin un espace de mémoire O(n), car il est un algorithme récursif et nécessite une liste supplémentaire pour trier les nombres.

2.2. Etat de l'Art

2.2.1. Tris Principales

2.2.1.1. Tri à Bulles

Cet algorithme fameux envoie le plus grand nombre des nombres dépassés à la fin à chaque tour. Au total on effectue la longueur de la liste – 1 tours. On compare tous les nombres voisins et on échange les places des nombres, si le nombre dont l'index est plus petit que l'autre, est plus grand que l'autre nombre.

L'algorithme est appelé tri à bulles, car il envoie les grands nombres au bout de la liste, en faisant des bulles.

Pseudocode :
Comme entrée on a A[0, ...j-1]

```
FOR  k = j to N-1 DO
        FOR i = 0 to k DO
                IF  A[i] > A[i+]  THEN
                        SWAP(A[i], A[i+1])
                ENDIF
        ENDFOR
ENDFOR
```

Analyse

La complexité du tri à bulles est obtenue après des longs calculs. On peut dire qu'il est domine par $O(N^2)$ [1]. Mais le complexité totale contient aussi des termes en terme O(nlog(n)). En conclusion, cet algorithme fait fatiguer beaucoup les processeurs. Le mauvais cas est rencontre quand les plus petits nombres se trouvent a la fin de la liste. Les résultats montrent qu'il marche plus mauvais par rapport aux tris par insertion et sélection.

Dans certains tours on peut examiner qu'on fait des comparaisons qui ne sont pas nécessaires. Let tri « Coctail-Shaker » essaie d'éviter ces comparaisons, en détectant la frontière qu'on ne doit pas dépasser ou on ne doit plus faire des comparaisons [1].

2.2.1.2. Tri Par Sélection

Cet algorithme traite tous les éléments de la liste. A chaque tour, elle change les places de l'élément choisi pour le tour avec le nombre le plus petit des nombres qui se trouvent a droite de l'élément choisi.

Pseudocode :

Comme entrée on a A[0, ...j-1]

```
FOR i = 0 to j DO
        min = i ;
        FOR k = i + 1 to j DO
                IF A[k] < A[min] THEN
                        min = k;
                ENDIF
        ENDFOR

        SWAP(A[k], min) ;

ENDFOR
```

Analyse

La complexité de l'algorithme est la somme du nombre de comparaison et de nombre d'échanges.

Le nombre de comparaison est la combinaison de N nombres a 2 éléments. Ce qui fait ½ N(N-1).

Le nombre d'échange de nombre est équivalent à N3/4 [1].

Les tests montrent que la performance de tri par sélection est mieux (%60) par rapport à celle de tri à bulles, mais mauvais que le tri par insertion De plus, il est très simple à programmer.

2.2.1.3. Tri Par Insertion

Le tri par insertion insère les nombres dans leurs propres places à chaque tour de l'algorithme. A chaque tour, l'algorithme compare les nombres qui se trouvent à gauche nombre appelé « index » et puis elle met le nombre à sa place propre. Finalement, on obtient les nombres qui sont triés.

Pseudocode :
Comme entrée on a A[0, ...j-1]

```
FOR i = 1 to j DO
        index = A[i] ;
        k = i ;
        WHILE  k > 0  AND A[k-1] > index  DO
                        A[k] = A[k-1] ;
                        k = k-1 ;
        ENDWHILE

        A[k] = index ;

ENDFOR
```

Analyse

Pendant le grand tour, on compare le nième élément avec j/2 éléments au moyen. Alors, le nombre de comparaison au total est (1 + 2 + ... + N)/2 ~ N2/4. C'est équivalent aussi à N2.

On doit aussi se rappeler que le tri par insertion est plus vite par rapport au tri à bulles et tri par sélection.

Il y a aussi le tri par insertion binaire qui est une sorte de tri par insertion. Dans ce tri, le nième élément est compare au maximum log(i) fois. Par exemple A[128] est compare d'abord avec A[64]. Si A[128] est plus grand que A[64], A[128] est compare avec A[96]. Ça continue jusqu'a ce qu'on trouve la place correcte pour A[128].

Au total la complexité pour le tri par insertion binaire est égale au nombre de comparaisons pour les éléments, ce qui fait :
N x log(N) ~ O(N x log(N).

Par contre, on doit aussi déplace les éléments a leurs places propres, mais ça prend encore beaucoup de temps. Cette étape augmente aussi la complexité de l'algorithme jusqu'a O (N2).

2.2.1.4. Tri de Shell

Le tri de Shell est propose par Donald Shell en 1959. Ce tri essaie d'envoyer les nombres les petits nombres vers le début de la liste le plus vite possible. Le tri fait un certain nombre de passes sur la liste, a chaque passe il tri des ensembles de nombre dont les tailles sont égales. A chaque passe de la liste, il trie les ensembles de nombres en utilisant le tri par insertion. De même, la cardinalité des ensembles diminuent à chaque passe. Le choix des cardinalités est définit par un nombre qu'on appelle l'incrément.

Par exemple, si on trie une liste a 12 éléments. Supposons aussi qu'on commence par un incrément de 4.
Dans la première passe, on trie les triplets A[0], A[4], A[8] ; A[1], A[5], A[9] ; A[2], A[6], A[10] ; A[3], A[7], A[11] .

On divise l'incrément qui est égale à 4 par 2. L'incrément est 2 maintenant.

Dans la seconde passe, on tris les six-uplets A[0], A[2], A[4], A[6], A[8], A[10]; A[1], A[3], A[5], A[7], A[9], A[11].

Dans la dernière passe, l'incrément est 1. On effectue un tri par insertion sur la liste. Comme les petits nombres se sont bousculés au début, le tri par insertion marche bien sur cette liste.

Pseudocode :

```
Increment = x ;
Il y a des conseilles pour le choix de x qui sont proposes dans le passe.

WHILE increment > 0 DO
        FOR i = 0 to j DO
                k = i ;
                temp = A[i] ;
                WHILE k >= increment  AND  A[k-increment] > temp DO
                        A[k] = A[k-increment];
                        k = k – increment;
                ENDWHILE
                A[k] = temp;
        ENDFOR
        increment = increment/2 ;
ENDWHILE
```

Analyse

Le tri de Shell est un bon exemple d'un algorithme simple, dont l'analyse est très complexe. Plusieurs chercheurs comme Knuth, Weiss et Sedgwick on fait des recherches sur la complexité de l'algorithme Shell. Le point important est que, le complexité est strictement lie à la séquences d'incréments.

La complexité du tri de Shell, dans le cas pire, en utilisant la séquence d'incrément de Shell est de $O(N^2)$ [2].

La complexité dans le cas moyenne change par rapport aux séquences d'incréments. Par exemple, Weiss propose le complexité de l'ordre de $O(N^{5/4})$ [2] avec la séquence de Hibbard qui sont 2k – 1 [2]. Sedgewick a calculé un complexité moyenne qui est égale a $O(N^{7/6})$.

Le choix de séquence d'incrément est décidé par celui qui implémenté cet algorithme. Il y a des considérations pour des grands et petits nombres.

2.2.2. Tri Par Indexes

L'article [3] représente un nouveau algorithme Self-Indexed Sort qui est un type de tri base a non comparaison. On dit que la complexité de temps d'exécution est O(n) ou n le nombre des éléments à trier. On ajoute aussi que le complexité en espace est de O(n+m) ou m est le max des nombres a trier. On rappelle que les tris sont classifies de point de vue de comparaison a deux catégories :

- Base à comparaison : Pour ce type, on peut donner le tri par sélection, le tri rapide et le tri à bulles. Car dans ces tris, on compare les nombres pour arriver à la solution finale.
- Base à non comparaison : Pour ce type, on peut donner comme exemple le tri Self-Indexes Sort. On va donner les détails dans les paragraphes suivants.

2.2.2.1. L'algorithme

L'algorithme consiste a faire le mapping des éléments à trier dans un nouveau liste en comptant le nombre de fois apparaît un nombre dans la liste initiale. Ensuite, on met nombres de départs à leurs places correctes en regardant le nombre de fois qu'ils s'apparaissent.

2.2.2.2. Les étapes de l'algorithme

On pose le problème comme ceci :

Entrée : Une liste aléatoire : A = (a_0 a_1 .....a_{m-1} ;

La liste de mapping qui contiennent les indices : S = (s, s, ...s);

On divise l'algorithme en trois étapes :

- **Initialisation :** On assigne 0 à tous les éléments S dont le nombre des éléments est le max.

  Pseudocode :

  FOR j = a to a_{m-1} do where m = (max a)

  S[j] = 0 ;

  ENDFOR

- **Arrangement self-indexed :** On tourne dans une boucle, ou on passe par les éléments de la liste de départ en incrémentant l'indice de ce nombre par 1 dans la liste S.

  Pseudocode :

  FOR i = 0 to n-1

  INCREMENT(S[A[i])) ;

  ENDF OR

- **La compression d'ordre préservée :** On itère dans la liste S. Ça veut dire, on regarde la valeur des éléments. Si elle est plus grande que 0, on met ce nombre dans la liste initiale et on décrémente le nombre par 1.

Pseudocode :

```
i = 0 ;
FOR  j = s to DO
        WHILE  S[j] >0  DO
                        A[i] = j;
                        DECREMENT(S[j]);
                        INCREMENT(i) ;
        ENDWHILE
ENDFOR
```

Un exemple pour arrangement auto-indexe :

1ere étape : initialisation

	0	1	2	3		i-1	i		i-1
A[i]	2	1	0	2	.....	k	m-1	.....	k

ou m est le max de A[i]

	0	1	2		k-1	k	k+1		m-2	m-1
S[j]	0	0	0	.....	0	0	0	...	0	0

2eme étape : arrangement auto-indexe

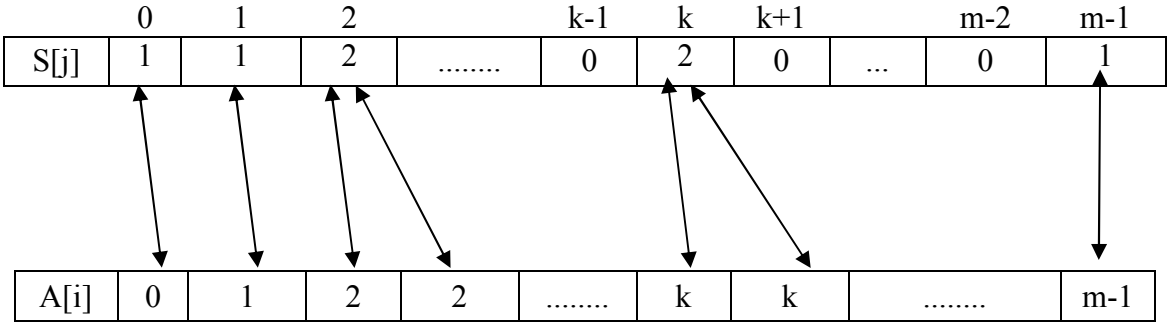
On incrémente les indices des nombres à trier dans S.

	0	1	2		k-1	k	k+1		m-2	m-1
S[j]	1	1	2	.....	0	2	0	...	0	1

A[i]	2	1	0	2	.....	k	m-1	.....	k

3eme étape : La compression des indices

On met les nombres de façon triée a la liste A.



2.2.2.3. Complexité de l'algorithme

Complexité en terme d'espace : S(n) = O(n+m) ou n est le nombre des éléments a trier et m le max des éléments.

Complexité en termes d'exécution : La complexité totale est la somme des complexités de l'arrangement auto-indexe et la compression des indices.

Ce qui fait, O(n) pour la complexité de l'arrangement auto-indexe. Car, on traverse les nombres de la liste initiale une seule fois. Pour la compression des indices, l'article dit qu'on fait n fois la compression des indices. Au totale, ça fait O(n) + O(n) qui est équivalent a O(n).

L'article montre des résultats obtenus avec des listes de capacités différentes qui vont de n = 100 jusqu'a 1000. A chaque fois, le temps d'exécution est linéaire est plus vite par rapport aux tris rapide, de Shell et a bulles.

2.2.3. Commentaires à Propos de Tri Par Indexes

Cet article [9] est basée sur l'article précèdent. On constate que l'article précèdent a été publie dans une magasine. Par contre, dan Comment on Self-Indexed Sort, les auteurs Arne Anderson et Andrej Brodnik démontrent qu'il y a des arguments incorrectes a. On étudie l'article précèdent en terme d'analyse, d'origine de l'algorithme et les résultats de tests.

2.2.3.1. L'analyse

On suppose que le complexité en espace est de O(n+m). C'est sans doute vrai. Par contre on suppose que la complexité en temps est de O(n). Mais, ça ne présente pas la vraie complexité.

On rappelle que le tri Self-Indexed était divisé en trois phases :

- Initialisation
- Arrangement self-indexed
- Compression d'ordre préservée

On fait m fois d'initialisations. Ce qui fait en termes de complexité O(m). On passe par tous les éléments de la liste initiale. Cette phase coûte à une complexité de O (n). La dernière phase contient un boucle qui manipule m éléments mais qui exécute une opération seulement n fois. Au total ça fait O(2n) + O(m) qui est équivalent a O(n) + O(m) et en tout cas le complexité en temps est de O(n+m).

2.2.3.2. L'origine de l'algo Self-Indexed

Malgré le fait que Sunny Y.Wang dit que l'algorithme est nouveau. On peut trouver le même algorithme avec de noms différents :

- « Bucket sorting » : Aho, Hopcroft et Ullman [7]
- « Counting sorting » : Cormen, Leiserson et Rivest [6]
- « Distribution sorting » : Sedgewick [10]

Dans ces algorithmes, il s'agit de deux façons : parfois on utilise des listes chaînées ou les indices des éléments.

En simplifiant l'algorithme de Wang on peut se rencontrer avec les algorithmes qu'on a parle juste au-dessus. Anderson et Brodnik prouve que Wang utilise des éléments qui ne sont pas nécessaires dans la liste chaînée.

2.2.3.3. Les Résultats de Tests

Comme on travaille avec des nombres de 16, 24 et 32 bits. On peut dire que le maximum d'une liste à trier est 16777216 pour les nombres de 24 bits. Donc, le temps

de complexité pour m va dominer le temps de complexité pour n qui est de l'ordre 20k. En partant de cette idée, Andersson et Brodnik calculent la complexité en temps pour une liste dont le nombre d'éléments est 100. Ils trouvent un résultat qui n'à rien rapport a celui de Wang, de même pour des listes de nombres différents. En conséquence, ils démontrent que les résultats cités dans l'article de Wang ne sont pas des résultats réels.

2.2.4. La Transition de Tri a Bulles au Tri de Shell

Cet article [4] représente une transition de tri à bulles au tri de Shell. C'est un mixe de tri a bulle et de Shell. La nouvelle idée de cette nouvelle méthode est d'utiliser des incréments dans les comparaisons de tri à bulles. Par conséquent, cette méthode n'est pas aussi efficace.

2.2.4.1. Tri à Bulles

Le tri par, comme on le sait déjà, marche en comparant les deux nombres adjacents. S'ils ne sont pas triés, on change les positions. Donc, le grand nombre passe vers la droite.

La complexité du tri à bulles est égal a aux nombres de comparaison pour un tour multiplie par le nombre de tours. On compare au maximum un nombre N-1 fois dans un tour et on change les positions des nombres s'il est nécessaire. Le nombre de tour est de N-1, car on commence du début et on continue jusqu'à la fin. Finalement, on obtient la complexité comme ceci :

$O(n-1) * O(n) \sim O(n2)$

2.2.4.2. Tri de Burp

L'inefficacité de tri à bulles est liée aux mouvements des nombres avec des petits pas – d'une seule position – en tout cas. Le tri Shaker et de Batcher essaient d'augmenter la rapidité du tri à bulles avec des stratégies différents.

Les auteurs appliquent l'idée de déplacer les positions des nombres de plus vite. Pour cela, comme on le fait dans le tri de Shell, ils déplacent les nombres dans les tours avec des incréments.

Dans les premières boucles de l'algorithme on essaie d'envoyer les grands nombres vers la droite. Dans les boucles finales, comme les grands nombres se trouvent au bout de la liste, on n'a pas besoin de déplacer les nombres dans la majorité des cas.

Les auteurs comparent les résultats de tri de burp avec des tris différents. Le nouveau tri est plus vite que le tri à bulles, mais moins vite que les tris par sélection, par insertion et de Shell.

Table 3.1 : La performance du tri « Burp » avec les autres algorithmes

N	Bulle	*Burp*	Sélection	Insertion	Shell
50	20	***10***	20	20	10
100	90	***70***	50	50	20
200	400	***260***	190	190	50
400	1580	***1050***	770	770	110
800	6400	***4200***	3090	3090	260
1600	25490	***16890***	12030	12030	580
3200	103670	***69010***	48740	48740	1340

2.2.4.3. Conclusion

Le tri « Burp » a besoin d'une démonstration. Car, il n'est pas très clair comme les algorithmes. Par contre, il sert de noter de points :

- Est-il possible de trouver des algorithmes hybrides comme cet article propose
- Il est toujours possible d'augmenter la performance des algorithmes de tri.

On rappelle aussi que l'idée de commination des algorithmes dans cet article est utilise dans cette thèse.

2.2.5. Variations sur le Tri Par Seau

Cet article [5] est dédié à l'implémentation de tri par sceau. On sait déjà que le complexité de tri par Bucket est de O(n). Par contre, on peut toujours faire des améliorations en observant de plus pré cet algorithme au niveau d'implémentation et de mémoire.

2.2.5.1. Tri par Seau

Le tri par seau est divisé en trois étapes. Etant donne qu'on a n nombres appartenant a l'intervalle [0,100] par exemple, on met distribue les nombres dans les intervalles de même longueurs comme ceci : [0,10), [10,20), ….. [90,100]. On applique la même méthode si les nombres appartiennent à l'intervalle [0,100], sauf qu'on multiplie par 10. Dans ce cas, les intervalles sont : [0, 100), [100, 200),….. [900,1000]. Le choix du nombre des seaux est décidé par l'implémentation. Ensuite, on applique le tri par insertion à chaque seau.

Les étapes du tri par seaux :
Entrée : la liste a
Sortie : la liste a qui est triée

1ere étape :

```
FOR i = 1 to n DO
    METTRE a[i] in seau auquel il appartient
ENDFOR
```

Seconde étape:

```
FOR i = 1 to n DO
    FAIRE un tri par insertion a le seau numéroté par i
ENDFOR
```

Troisième étape :

FOR i = 1 to n DO

COPIER le contenu de seau numéroté par i a la liste triée

ENDFOR

2.2.5.2. Amélioration a la Tri par Seau

Les auteurs Corwin & Logar expliquent ce qu'il faut faire pour augmenter la performance de tri par Seau.

2.2.5.3. Allocation de Mémoire

Le tri par seau accepte des éléments d'une liste comme entrée. Pour chaque élément, on alloue un nœud. Par contre, allouer un nœud à chaque fois prend beaucoup de temps. Au lieu de ça, on peut allouer à un coup toute la mémoire nécessaire pour les nœuds.

2.2.5.4. Changement de l'ordre Des Etape de Tri par Seau

En générale, on sait qu'on a des systèmes de 32 bits. Grâce à ces systèmes, on peut adresser des données de tailles de milliards d'octets. Ce qui veut dire qu'on peut adresser des centaines de millions d'entrées. Si l'adresse de mémoire n'est pas suffisante pour l'application, le disque dure est utilise comme une mémoire.

Dans le cas de tri par seau, la mémoire RAM est suffisante pour faire l'adressage de l'entrée. Mais, comme le tri par seau accède aux locations très dispersées, on se sert souvent de Translation Look aside Buffer(TLB) qui retient les pages utilises récemment. TLB est utilise pour minimiser la translation des pages virtuelles aux adresses physiques. Cette opération diminue la rapidité du tri par seau.

Le tri par seau est divisé en trois étapes. On distribue les éléments dans les seaux. On trie les éléments par le tri par insertion. Puis, on remet les éléments triés dans les seaux, à la liste initiale. Mais, si on change l'ordre de ces étapes, de distribution, tri, remise a distribution, remise et trie ? Le temps d'exécution est diminue a la quatrième de l'algorithme initiale. Quand on remet les éléments à liste initiale, on place les éléments dans une zone de mémoire consécutive, contrairement aux seaux. C'est à cause

fonctionnement d'une liste dans la mémoire. Grâce à ça, les pages de TLB ne changent pas souvent.

2.2.5.5. Conclusion

Les améliorations faites dans cet article forme la source d'inspiration de cette thèse. Si la taille de mémoire virtuelle est suffisante, le tri par sceau peut être assez puissant dans des listes à des petites tailles. Quand la taille augmente, le tri rapide est moins puissant. Pour des listes à très grandes tailles, le tri rapide est encore très efficace par rapport a tri par seau et aux autres algorithmes aussi.

2.2.6. Nouvelles Approches Pour les Listes Presque Triées

Dans cet article [11], les auteurs Cook et Kim présentent la performance des différents algorithmes de tri qu'ils ont exécutés sur les listes qui sont partiellement triées. Ainsi, les auteurs développent une norme qui mesure combien la liste n'est pas triée. En outre, ils développent un algorithme qui combine le tri par insertion et tri rapide (mais qui partitionne la liste en choisissant la médiane). Cette idée est aussi un autre exemple de combinaison comme on a fait dans notre travail. Finalement, ils comparent leurs algorithmes avec d'autres algorithmes.

Mesure de « sortedness »

Les auteurs cherchent d'abord a compter le nombre d'éléments qu'on doit enlever de la liste au point que la liste devienne triée.

Formule de « sortedness » : Si on appelle k, le nombre a enlever et si le nombre de la liste est n. Alors, la fonction S qui prend comme entrée une liste a trier, est comme ci-dessous.

- S(liste) = k/N

Plus S s'approche à 1, plus la liste est loin d'être triée. Par contre, plus S s'approche vers 0, plus la liste est partiellement triée. Cette fonction est très utile, car on fait des tests sur des listes partiellement triées.

Exemple : A = {0, 2, 8, 5, 3, 10, 16, 12, 29, 38, 30, 89}
Pour la liste ci-dessous 8, 3, 12 et 30 ne sont pas dans leurs propres places. Ainsi
S(A) = 4 / 12 = 0,33.

2.2.6.1. Création des Listes Partiellement Triées

Il y a diverses techniques pour créer différentes listes qui sont partiellement triées. Pour créer une liste de ce type, on crée une permutation randomisée de taille m. Ensuite, on relève m éléments de permutation d'identité qui a N éléments. Puis, on remplace ces m éléments avec les éléments qui sont produites par la permutation randomisée.

2.2.6.2. Mesure des Résultats

Pour mesurer la performance, on calcule le nombre d'opérations élémentaires. Ces opérations élémentaires sont comparaison, déplacement et échange. On assume aussi qu'une opération d'échange coûte deux fois plus qu'une opération de comparaison et de déplacement.

On fait des tests de taille 50, 100, 200, 500, 1000, 2000 éléments et avec des « sortedness » de 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 et 20. On utilise le tri par insertion, le tri rapide (qui choisit comme pivot le médiane), le tri fusion, le tri par tas.

On prétend toujours que le tri par insertion marche bien avec les listes partiellement triées. Quand on examine les résultats, on peut dire que le tri par insertion et le tri rapide marchent bien avec les listes partiellement triées. Quand le « sortednees » et la taille de liste augmentent, le tri rapide devient plus rapide que le tri par insertion. On voit aussi que le pire des cas pour le tri par insertion est le cas ou la liste s'éloigne d'être triée. Dans ces cas, sa performance est plus mauvaise que les autres algorithmes. Cette étude prouve que le tri par insertion et le tri rapide marchent bien avec les listes partiellement triées.

Les valeurs pour le tri fusion et le tri par tas ne diffèrent pas beaucoup quand le « sortedness » ou la taille de la liste changent.

2.2.6.3. Le Nouvel Algorithme et Conclusion

Le nouvel algorithme combine le tri par insertion et le tri rapide. Les étapes de l'algorithme sont ainsi :

- On balaie la liste et on met le premier couple qui n'est pas dans le bon ordre, dan une liste.
- Apres, regarde le couple précédent au couple juste relève et on met ce couple s'il n'est pas dans le bon ordre, dans la liste des nombres qui ne sont pas triées.
- On répète ces étapes pour tous les éléments de la liste.
- On trie la liste qui contiennent les éléments qui ne sont pas tries. Si la taille de la liste est plus petite que 30, on utilise le tri par insertion. Sinon on utilise le tri rapide.
- Finalement, on fusionne les deux listes.

Les résultats de ce nouvel algorithme sont assez satisfaisants. Il marche mieux que tous les autres tris.

Voici quelques remarques qu'on peut faire sur cet article :

- Il faut ajouter aussi que, les listes partiellement triées ou complètement triées sont très fréquentes.
- L'idée de combinaison est une bonne idée pour améliorer la performance
- Les auteurs ont seulement exécuté les tris sur des listes de taille petites. Il faudrait aussi voir les tableaux de performance pour des listes grandes.

3. La Méthode de Solution

3.1. L'algorithme de Tri Par Seau Mise En Jeu

Le terme tri par seau (ou bien « bucket sort ») est employée pour différents algorithmes. Aho, Hopcroft et Ullman ont employé la sorte de seau (« bucket sort ») [7] dans leur livre. Le tri par seau qu'ils expliquent est également connu comme tri casier. Il consiste d'abord classer les nombres et compter le nombre d'éléments par l'intermédiaire des ndex.

Le tri par seau utilisée en cet article se fonde sur le tri par seau (Bucket Sort) mentionnée en introduction aux algorithmes par Cormen, Leiserson, Rivest et Stein [6].

L'idée principale derrière cet algorithme est ; rassembler d'abord les nombres d'une liste dans des seaux appartenant aux mêmes intervalles. La deuxième phase est de trier les seaux individuellement. La dernière phase est écrire les nombres dans les seaux dans la liste initiale.

Plus en détail, on peut définir trois étapes principales pour la sorte de seau en tant que ci-dessous :

On considère un choix de nombres dans un intervalle appartenant à [1..n].

Première étape : À la première étape, les nombres sont groupés dans des intervalles classés égaux : [0, 1/n), [1/n, 2/n), [2/n, 3/n) et ainsi de suite.

On a utilise des intervalles de sceau comme ceci : [0, 10), [10, 20), etc. contrairement a au dessous. En fait, les intervalles qu'on utilise sont relies a l'intervalle des éléments. C'est-à-dire si les nombres a tri appartiennent a l'intervalle [0, 100]. On peut avoir des intervalles de seau qui finissent et commencent avec les multiples de « 10 ». Si les

nombres appartiennent à l'intervalle [0, 1000]. On peut avoir des intervalles de seau qui finissent et commencent par les multiples de 100.
Deuxième Étape : À la deuxième étape, les nombres sont tries en utilisant une sorte d'insertion ou une sorte de casier.

Troisième Étape : À l'étape finale, les nombres qui ont été tries dans les seaux sont réécrits dans la liste initiale. Ainsi, nous obtenons la liste initiale avec les éléments tries.

3.2. La Nouvelle Approche Proposée

3.2.1. L'optimisation au Niveau de Mémoire Dans la Tri Par Seau

Dans [5] Edouard Corwin et Antonette Logar ont mentionné qu'on pourrait faire des modifications avec les étapes de tri par seau qui peuvent améliorer des résultats. Si on appelle la première étape en tant que « distribution », deuxième étape en tant que « tri » et troisième étape comme « remis ». En changeant de nouveau l'ordre d'étapes de la « distribution, tri, remise » à « distribution, remise, tri » peuvent réduire le temps de tri de manière significative. C'est dû au ce, l'opération qui met les valeurs dans des endroits consécutifs dans la mémoire. Car les éléments d'une liste se trouvent dans les endroits séquentiels.

On applique cette idée dans l'implémentation de tri par seau. Alors, on change l'ordre des étapes de l'implémentation de tri par seau.

3.2.2. L'utilisation de Tri de Shell dans la Tri Par Seau

Le tri par insertion et le tri de casier ont été appliqués largement au tri par seau à la deuxième étape. On propose d'utiliser le tri de Shell au lieu des algorithmes de tris mentionnes.

Dans [4] on parle un passage de tri a bulle au tri de Shell. Dans [11], on développe un nouvel algorithme qui est base au tri par insertion et au tri rapide. Dans les deux articles, l'idée de base est faire l'utilisation de tris hybrides. On peut se profiter de cette idée dans le tri par seau.

On sait que la sorte d'insertion est utilisée comme « de facto » dans les seaux qu'on utilise pour le tri par sceau. Le tri Shell fonctionne avec une bonne performance sur des listes de taille moyenne. Puisque, le tri de Shell est un genre de tri par insertion. Nous pouvons prévoir qu'elle va fonctionner efficacement sur les seaux de taille moyenne. Le principe est utilisé des algorithmes hybrides respectant le paragraphe précédent. Ainsi, on utilise le tri de Shell dans les seaux.

Afin de trier les seaux à l'étape finale, on a modifié le tri de Shell, de telle manière que la version modifiée fonctionne seulement pour un nombre limité d'éléments de la liste.

3.3. Les Classes Utilises Pour la Programmation

On utilise C++, comme le langage de programmation. Car il marche très vite et les structures de données sont plus faciles et rigoureux a implémenter.

On a trois classes et une structure. On va présenter les interfaces de ces classes dans les sections suivantes. On peut comprendre le fonctionnement si on regarde juste les dossiers de definition.

3.3.1. Classe Timer

```
class Timer
{
  private :
    clock_t mStart ;
    clock_t mStop ;
    struct timeval mStartTimeVal ;
    struct timeval mStopTimeVal ;
    struct timezone mStopTimeZone ;
    struct timezone mStartTimeZone ;
    double mDuration ;

  public :
    void elapseTime() ;
```

```
    void stopTime() ;

    void printDuration() ;
} ;
```

Les variables membres de l'objet servent à maintenir le temps initial et le temps final d'un algorithme de tri. On obtient des mesures de l'ordre de microsecondes. C'est du a « gettimeofday » fonction qui est une fonction de POSIX. On peut trouver la définition de cette fonction dans <sys/time.h>.

De même, les fonctions membres de la classes sont utilisé à chaque fois qu'on commence a exécuter un algorithme, on arrête un algorithme et on affiche le temps d'exécution.

3.3.2. Classe SortAlgorithms

```
class SortAlgorithms
{
        private:

                int* mArray;
                int mSize;
                int* mIndexArray;
                int mIndexArraySize;
                int mLowend;        // minimum element
                int mHighend;      //max element
                int mInterval;     //number of intervals
```

```
public:
        SortAlgorithms();

        ~SortAlgorithms();

        void setArrayAndSize(int* array, int size);

        void setSize(int size);

        int getSize();

        int* getArray();

        void printArray();

        int findMaxWithIncrements(int index, int increment);

        int findMax();

        void swap(int& x, int& y);

        void initializeIndexArray();

        void shellSort();

        void insertionSort();

        void bubbleSort();

        void regularBucketSort();

        void shellMixedBucketSort();
```

```
        void shellSortPartOfArray(int startIndex, int stopIndex);

        void insertionSortPartOfArray(int startIndex, int stopIndex);

        void selfIndexedSort() ;
} ;
```

Cette classe est charge d'exécuter des tris sur des listes. Ces opérations sont : remplissage de la liste, tri de la liste, trouver le maximum, etc. Les éléments pour les listes sont produits de façon randomisée avec les fonctions srand et rand. Les fonctions de tri exécutent les tris élémentaires ; le tri Shell, le tri à bulles, le tri par insertion, le tri de la nouvelle approche, le tri par seau, etc. Certaines fonctions exécutent seulement juste sur des portions de la liste. Le code complet est donnes dans les Appendices.

3.3.3. Structure Element

```
struct Element
{
    public:
    int value;
    Element* next;
    Element()
    {
        next = NULL;
        value = -1 ;
    }
} ;
```

Cette structure est une abstraction d'un élément qui se trouve dans un seau. On effectue le tri par seau grâce à cette structure.

3.3.4. Class Bucket

```
typedef struct Element Node ;
class Bucket
{
        public:

                Node* mHead;
                int mSize;
                Bucket();
                void addElement(Node* element);
};
```

Cette classe décrit les fonctions d'un seau qu'on utilise dans le tri par seau. Elle est composée des structures « Elements ». On ajoute des éléments dans le seau qui est en tout cas une liste chaînée.

3.3.5. L'algorithme Finale Obtenue

Nous pouvons diviser la nouvelle approche en 5 étapes.

1. Première étape : Nous créons les objets.

Le nombre de seaux est trouvé avec l'opération suivante :

Nombre de seau = (maximum de la liste – minimu la liste) / intervalle

On a employé les nombres entre 0 et 100. Et on a décidé de prendre l'intervalle en tant que 10. Donc, on utilise 10 seaux.

2. Deuxième étape : Plus tard, nous créons des éléments de n à une commande simple :

*Element * éléments = new Element[n] ;*

Les objets d'élément tiennent la valeur des éléments de rangée.

3. Troisième étape : On remplit les seaux. On distribue les nombres dans des seaux concernant la valeur de l'objet d'élément.
On détermine le « bucketIndex », qui est l'index du seau où l'objet d'élément sera mis.
L'index de seau est trouvé de cette façon :

bucketIndex = la valeur de l'objet d'élément / de intervalle

4. Quatrième étape : On écrit les valeurs des objets de chaque élément dans les seaux, vers la liste initiale. En conséquence, les éléments sont dans des endroits consécutifs dans la mémoire. Maintenant la rangée initiale se compose de sections tel qu'où les éléments de chaque section appartiennent à un intervalle [0, 1/n), [1/n, 2/n), [2/n, 3/n) et ainsi de suite.

5. Cinquième étape : *On trie chaque seau avec le tri de Shell modifie.*

Le tri de Shell prend deux paramètres : startIndex et stopIndex. Chaque fois le tri de Shell tri seulement un morceau déterminé de liste considérant au nombre d'éléments dans les seaux qui ont été rempli à la deuxième étape.

3.3.6. Le Tri de Shell Modifié

Appelons A, rangée d'initiale à assortir. Le pseudo code pour le tri de Shell modifie coquille est comme suit :

SHELLSORTMODIFIED (parameters: int startindex, int stop index);

```
int temp;
// On constate les bordures des éléments à trier
int size = stopIndex – startIndex + 1;
// On divise l'incrément par deux a chaque étape
int increment = size / 2;
```

```
WHILE increment >0 DO

        FOR i = startIndex + increment TO  stopIndex DO
                j = i;
                temp = A[i];
                WHILE (j  >= increment) AND (mArray[j-increment] > temp) DO
                        A[j] = A[j-increment] ;
                        j = j- increment ;
                ENDWHILE
        A[j] = temp ;
        ENDFOR
        increment /= 2 ;

ENDWHILE
```

4. Application

4.1. Etude du Système

Le système est compose d'un seul ordinateur qui possède un RAM de 256 Mb et un processeur Intel Pentium III. Le système d'exploitation de la machine est Linux (Fedora 4).

4.2. La Production de Nombres

Les nombres sont produits par la méthode « setSize » de la classe SortAlgorithms.
Cette méthode produit des nombres dont la taille est donne par un paramètre appelé « size ».

Chaque nombre est le résultat du nombre par la fonction rand modulo 100. On rappelle que la fonction rand retourne un nombre aléatoire. Ensuite, les nombres sont enregistrés dans la liste.

Grace a la fonction « srand » :

- On peut éviter d'obtenir différents nombres dans les appels successives a la fonction rand.
- De même, on peut avoir les mêmes nombres à la fin des appels successifs de la fonction rand.

On donne comme paramètres time(NULL), pour obtenir différents nombres à chaque fois qu'on construit une liste. Mais, les fonctions de tris se sont exécutées sur la même liste.

On donne le contenu de la fonction setSize ci-dessous :

```
void SortAlgorithms::setSize(int size)
{
     mSize = size;
     mArray = (int*)malloc(sizeof(int)*size);

     srand(time(NULL));

     for(int j=0; j < mSize; j++)
     {
          mArray[j] = rand()  % 100;
     }
}
```

4.3. Exécution des Algorithmes de Tris

Les méthodes de la classe SortAlgorithms sont appelées consécutivement sur la liste dont les éléments se sont obtenus comme dans la section 6.2.

On fait courir les algorithmes suivants sur les listes :

- Tri à bulles
- Tri par seau
- Tri de Shell
- La nouvelle approche (5.3.6)

Les détails de ces fonctions sont donnes dans l'Appendice B.

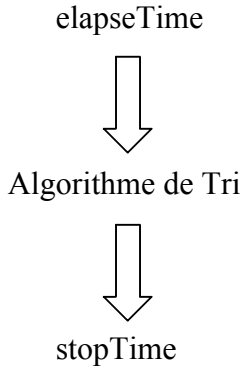
Les tailles des listes varient de 500 jusqu'à 50000. On mesure les temps d'exécution et la performance des algorithmes pour chaque liste.

4.4. Mesure de Temps d'Exécution

Quand on fait commencer un algorithme, on exécute la fonction elapseTime de la classe Timer.

A la fin de l'algorithme, quand la liste devient triée. On exécute la fonction stopTime de la classe Timer.

De point de vue générale, les lignes s'aperçoivent comme ci-dessous :



Les fonctions stopTime et elapseTime sont données dans l'Appendice A.

4.5. Comparaison au Niveau de Temps

On a effectue les mêmes tests 50 fois pour obtenir des résultats en moyenne. Pour chaque algorithme, on a écrit la moyenne comme résultat de temps d'exécution.

Table 6.1 : La performance de la nouvelle approche

Taille	**Algorithme Utilise (Résultats en moyenne)**			
	Tri à bulles (µs)	**Le tri par seau régulier (µs)**	**Le tri de Shell (µs)**	**La nouvelle approche (µs)**
500	2195	230	143	**163**
1000	9332	495	210	**328**
2000	43596	1050	568	**594**
5000	248725	4576	1447	**1350**
7500	562598	10101	2121	**2035**
10000	994813	21534	3489	**3034**
15000	2 seconds + 180126	41454	4489	**4179**
50000	-	394112	192226	**16210**

Le tri par seau est un tri efficace pour des listes de petite et moyenne taille. On réduit le temps d'exécution de sorte en combinant le tri de Shell et le tri par seau. En plus on a

suivi les instructions dans [7.] L'exécution globale obtenue est meilleure que le tri par seau régulier (ou chaque seau est trie avec le tri par insertion) et naturellement le tri a bulles. On peut déterminer également que l'exécution de la nouvelle approche est meilleure que le tri de Shell, quand la taille de la liste est plus grande que 2000 éléments.

Cet effet est du a l'allocation de mémoire des seaux. Pendant ce processus, on dépense un temps en moyenne pour chaque test des listes de tailles différentes. Dans d'autres mots, on fait des allocations de mémoire pour chaque liste de tailles différentes. Mais, ce processus étant un processus standard. A partir de 2000 éléments, le total cette opération plus le temps d'exécution de la nouvelle approche est plus petite que le tri de Shell. En conclusion, cette opération d'allocation de mémoire est compense avec la rapidité de la nouvelle approche par rapport à la performance du tri de Shell.

5. Conclusion

D’abord ce travail, on a essaye plusieurs hybrides algorithmes, notamment avec le tri et le tri de Shell. Mais, ces algorithmes n’ont pas marche à cause de problèmes de mémoire, etc.

L’exécution est le point clé dans des algorithmes de tri. À l'avenir, le même genre d'améliorations peut être fait sur d'autres algorithmes de sorte en changeant des étapes de l’implémentation. Deuxièmement, des algorithmes de sorte hybrides peuvent être inventés et appliqués pour augmenter la performance des algorithmes de tri. Ainsi de suite, on peut faire des nouvelles recherches sur les implémentations de tris.

Dans les étapes suivantes ;

- On peut faire plusieurs tests avec des listes de différentes tailles et constater un mesure en termes de taille, dans le cas ou la nouvelle approche marche bien par rapport aux autres algorithmes,
- On peut tester la nouvelle approche sur les listes tries partiellement,
- On peut combiner le tri par seau avec d’autres algorithmes,
- On peut concentrer sur le tri de Shell et faire des modifications pendant le tri des nombres dans l’étape des incréments,
- Finalement, on peut également calculer et étudier les complexités des ces algorithmes de plus proche.

BIBLIOGRAPHIE

[1] Knuth, D.E., *The Art of Computer Programming Vol3 - Sorting and Searching*, Addison-Wesley, Eighth Printing, Reading, (2001).

[2] Weiss, M.A., "*Data Structures & Algorithm Analysis In C++",* Addison-Wesley, Secon Edition, (1999).

[3] Wang, S.Y., "A New Sort Algorithm : Self-indexed Sort", *ACM SIGPLAN Notices*, 31 (3), (1996).

[4] Joseph B. Klerlein and Curtis Fullbright, "A Transition From Bubble Sort To Shell Sort", ACM SIGCSE Bulltein, 183-184, (1988).

[5] Edward Corwin and Antonette Logar, "Sorting In Linear Time – Variations On The Bucket Sort", *JCSC* 20-1, (2004).

[6] Cormen, Leiserson, Rivest and Stein, *Introduction To Algorithms*, 2nd Edition, McGraw-Hill, (2001).

[7] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design And Analysis of Computer Algorithms*, Addison-Wesley, (1974).

[8]http://dept-info.labri.fr/ENSEIGNEMENT/INITINFO/initinfo/supports/book/node44.html.

[9] Andersson A. and Brodnik A., "Comment on Self-Indexed-Sort", *ACM SIGPLAN Notices*, 31, (1996).

[10] R.Sedgewick, *Algorithms*, Addison-Wesley, Reading, (1988).

[11] C.R. Cook and D.J.Kim, "Best Sorting Algorithms for Nearly Sorted Lists", *Communications of the ASM*, Volume 23, Number 11, (1980).

APPENDICE A

Timer.cpp

```
#include "Timer.h"

void Timer::elapseTime()
{
   //mStart = clock();
   gettimeofday(&mStartTimeVal, &mStartTimeZone);
   cout << "Start second:\t\t" << mStartTimeVal.tv_sec <<
endl;
   cout << "Start microsecond:\t" << mStartTimeVal.tv_usec
<< endl;
}

void Timer::stopTime()
{
   //mStop = clock();
   gettimeofday(&mStopTimeVal, &mStopTimeZone);
   cout << "Stop second:\t\t" << mStopTimeVal.tv_sec <<
endl;
   cout << "Stop microsecond:\t" << mStopTimeVal.tv_usec <<
endl;
   //mDuration = ((double)mStop - (double)mStart) /
CLOCKS_PER_SEC;
   //mDuration = difftime(mStop, mStart) / CLOCKS_PER_SEC;
}

void Timer::printDuration()
{
   int microSeconds = mStopTimeVal.tv_usec -
mStartTimeVal.tv_usec;
   int seconds = mStopTimeVal.tv_sec -
mStartTimeVal.tv_sec;

   if (microSeconds < 0)
      {
          microSeconds = 1000000 + microSeconds;
          seconds -= 1;
      }

   setprecision(5);
```

```
   cout << "Total duration:\t" << seconds << "\tsecond
and\t";
   cout << microSeconds << "\tmicroseconds." << endl;
   //std::cout << "Duration:  " << mDuration << std::endl;
}
```

APPENDİCE B

SortAlgorithms.cpp

```
#include "SortAlgorithms.h"

SortAlgorithms::SortAlgorithms()
{
     mArray = NULL;
     mSize = 0;
     mLowend = 0;
     mHighend = 100;
     mInterval = 10;
}

SortAlgorithms::~SortAlgorithms()
{
     if (mArray != NULL)
     {
          delete mArray;
     }
}

int* SortAlgorithms::getArray()
{
     return this->mArray;
}

int SortAlgorithms::getSize()
{
     return mSize;
}

void SortAlgorithms::printArray()
{
     cout << "Printing numbers:" << endl;

     for(int i=0;i < mSize;i++)
     {    cout << mArray[i] << "\t";

     }
     cout << endl;
}
```

```
void SortAlgorithms::setSize(int size)
{
     mSize = size;
     mArray = (int*)malloc(sizeof(int)*size);

     srand(time(NULL));

     for(int j=0; j < mSize; j++)
     {
          mArray[j] = rand()  % 100;
     }
}

void SortAlgorithms::setArrayAndSize(int* array, int size)
{
     mArray = array;
     mSize =  size;
}

void SortAlgorithms::setVectorAndSize(vector <int> v, int
size)
{
     mSize = size;
     mArray = (int*)malloc(sizeof(int)*mSize);

     for(unsigned int i = 0; i < size; i++)
     {
          mArray[i] = v[i];

     }
}

void SortAlgorithms::quickSort(int left, int right)
{
     int middle;
     if (left < right)
     {
          middle = partition(left, right);
          quicksort(left, middle);
          quicksort(middle+1, right);
     }

     return;
}

int SortAlgorithms::partition(int left, int right)
{
     int x = mArray[left];
     int i = left - 1;
```

```
     int j = right + 1;
     int temp;

     do
     {
          do
          {
               j--;
          }while (x > mArray[j]);

          do
          {
               i++;
          }while (x < mArray[i]);

          if (i < j)
          {
               swap(mArray[i], mArray[j]);
          }
     }while (i < j);

     return j;
}

void SortAlgorithms::swap(int& first_index, int&
second_index)
{
     int temp;
     temp = first_index;
     first_index = second_index;
     second_index = temp;
}

void SortAlgorithms::shellSort()
{
     int increment;
     int i;
     int j;
     int temp;
     increment = mSize/2;

     while (increment>0)
     {
          for(i=increment; i < mSize; i++)
          {
               j = i;
               temp = mArray[i];
```

```
                while ((j>=increment) && (mArray[j-
                increment]>temp))
                {
                     mArray[j] = mArray[j-increment];
                     j = j-increment;
                }
                mArray[j] = temp;
           }
           increment /= 2;
     }
}

void SortAlgorithms::initializeIndexArray()
{
   // Size of the index-mArray, (maximum of the mArray) +
(1 for the maximum element)
   mIndexArraySize = this->findMax() + 1;

   // Initialization of the index mArray
   mIndexArray = new int[mIndexArraySize];
   for (int i = 0; i < mIndexArraySize; i++)
     {
          mIndexArray[i] = 0;
     }
}


void SortAlgorithms::selfIndexedSort()
{
   // Size of the index-mArray, (maximum of the mArray) +
(1 for the maximum element)
   int indexArraySize = this->findMax() + 1;

   // Initialization of the index mArray
   int* indexArray = new int[indexArraySize];

   for (int i=0; i < indexArraySize; i++)
     indexArray[i] = 0;

   // Self-indexed arrangement
   for (int j=0; j < mSize; j++)
   {
     indexArray[mArray[j]]++;
   }

   // Order-kept compression
   int orderedArrayIndex = 0;
```

```
     for (int l=0; l < indexArraySize; l++)
   {
        if (indexArray[l] > 0)
        {
             for (int k=0; k < indexArray[l]; k++)
             {
                  mArray[orderedArrayIndex] = l;
                  orderedArrayIndex++;
             }
        }
   }

   delete []indexArray;
}

void SortAlgorithms::insertionSort()
{
     int i;
     int j;

     for (j = 1; j < mSize; j++)
     {
          int index = mArray[j];

          i = j;
          while ( i > 0 && mArray[i-1] > index )
          {
               mArray[i] = mArray[i-1];
               i = i-1;
          }
          mArray[i] = index;
     }
}

void SortAlgorithms::bubbleSort()
{
     for (int i = 0; i < (mSize-1); i++)
     {
          for (int j = 1; j < mSize; j++)
          {
               if (mArray[j] < mArray[j-1])
               {
                    int temp = mArray[j];
                    mArray[j] = mArray[j-1];
                    mArray[j-1] = temp;
               }
          }
     }
}
```

```
void Bucket::addElement(Node* element)
{
     if (mHead == NULL)
     {
          mHead = element;
          mSize++;
     }
     else
     {
          element->next = mHead;
          mHead = element;
          mSize++;
     }
}

Bucket::Bucket()
{
     mHead = NULL;
     mSize = 0;
}

void SortAlgorithms::shellSortPartOfArray(int startIndex,
int stopIndex)
{
     //cout << "StartIndex: " << startIndex << " StopIndex:
" << stopIndex << endl;
     int increment;
     int i;
     int j;
     int temp;
     int size = stopIndex - startIndex + 1;
     increment = size/2;

     while (increment > 0)
     {
          for(i = startIndex + increment; i < stopIndex;
          i++)
          {
               j = i;
               temp = mArray[i];

               while ((j >= increment) && (mArray[j-
increment] > temp))
               {
                    mArray[j] = mArray[j-increment];
                    j = j - increment;
               }
               mArray[j] = temp;
```

```
          }
          increment /= 2;
     }
}

void SortAlgorithms::shellMixedBucketSort()
{
     // Create the buckets
     int noBuckets = (mHighend - mLowend) / mInterval;
     //number of buckets required
     //cout << "--------------------------------" << endl;
     //cout << "noBuckets:   " << noBuckets << endl;

     vector <Bucket*> buckets;
     for (int l = 0; l < noBuckets; l++)
     {
          Bucket* bucket = new Bucket();
          buckets.push_back(bucket);
     }

     //cout << "--------------------------------" << endl;
     //cout << "Created the buckets" << endl;

     // Allocate nodes
     Element* elements  = new Element[mSize];
     //cout << "--------------------------------" << endl;
     //cout << "Allocated the nodes" << endl;

     // Fill the buckets
     for (int i = 0; i < mSize; i++)
     {
          // Put the values into the nodes
          elements[i].value = mArray[i];
          //cout << "Deger: " << elements[i].value << endl;

          // Link elements[i] into bucket floor(n*a[i])
          int bucketIndex = elements[i].value/mInterval;
          buckets[bucketIndex]->addElement(elements+i);
     }

     // Sort the buckets
     for (int t = 0; t < noBuckets; t++)
     {
          Node* temp2 = buckets[t]->mHead;
          while(temp2 != NULL)
          {
               //cout << "value:" << temp2->value;
               temp2 = temp2->next;
          }
```

```
          //cout << endl;
     }

     int i = 0;
     for(int u = 0; u < noBuckets; u++)
     {
          Node* temp3 = buckets[u]->mHead;
          while (temp3 != NULL)
          {
               mArray[i] = temp3->value;
               temp3 = temp3->next;
               i++;
          }
     }

     //printArray();
     int startIndex = 0;
     int stopIndex = 0;
     // Sort the buckets
     for (int t = 0; t < noBuckets; t++)
     {
          //cout << "mSize:" << buckets[t]->mSize;
          stopIndex = buckets[t]->mSize-1;
          shellSortPartOfArray(startIndex, startIndex +
stopIndex);
          startIndex += stopIndex + 1;
     }
}


int SortAlgorithms::findMaxWithIncrements(int index, int
increment)
{
   int max = mArray[index];

   for (int i = index; i < mSize; i = i + increment)
     {
     if (mArray[i] > max)
        {
          max = mArray[i];
        }
     }

   //cout << "Max\t" << max << endl;

   return max;
}
```

```
int SortAlgorithms::findMax()
{
   int max = mArray[0];

   for (int i=0; i < mSize; i++)
     {
        if (mArray[i] > max)
        {
          max = mArray[i];
        }
     }

   return max;
}
```

BIOGRAPHIE

L'auteur Hüseyin Burak TAKMAZ est né le 1 Février 1981 à Izmir. De 1992 jusqu'à 1999, il a continue au lycée Saint-Joseph Izmir. Apres le lycée, il a été étudiant de 1999 à 2003 l'université de Galatasaray au Département de Génie Informatique. Depuis le Septembre 2005, il est étudiant dans le programme de Master de Génie Informatique. De l'Aout 2005, il est chargé comme chercheur, « TÜBİTAK / Ulusal Elektronik ve Kriptoloji Enstitüsü » ou il s'occupe de développement et test de logiciel.