Sylvain De Barros
Yohan Roux
27/09/2016

RAPPORT DE COMPLEXITE

Enseignant responsable : PERROT Kevin

Table des matières

- Algorithme de Kadane:
- Algorithme implémenté avec calcul de complexité
  - Algorithme 1
    - Pseudo code
    - Calcul de la complexité :
  - Algorithme 2
    - Pseudo code
    - Calcul de la complexité :
  - Algorithme 3
    - Pseudo Code
    - Calcul de la complexité :
  - Algorithme 4
    - Pseudo Code:
    - Calcul de la complexité :
- Résultat tests pratiques.
- Analyse des résultat obtenue
- Choix du langage
- Conclusion

Algorithme de Kadane:

Le but de ce tp est de calculer la sous séquence (éléments consécutifs) maximale dans un tableau d'entier de taille N.
Pour ce faire nous avons quatre algorithmes à faire avec des complexité décroissante allant de N3 a N ensuite nous devrons comparer leur temps d'exécution sur un jeu de donnée.
Le but de se tp étant de comprendre l'importance de la complexité d'un algorithme sur son temps d'exécution.

Algorithme implémenté avec calcul de complexité

Algorithme 1

Pseudo code

```
PROCEDURE aglo1 (tab , taille_tab)
        max = tab[0]
        indiceD = 0,
        indiceF = 0,
        indiceCourrantD = 0,
        indiceCourrantF = 0
Debut
        TANT_QUE indiceCourrantD < taille_tab
                TANT_QUE indiceCourrantF < taille_tab
                        Pour i de indiceCourrantD jusqu'à indiceCourrantF avec un pas de 1
                                maxTmp = maxTmp + tab[i]
                        Fin pour
                SI maxTmp > max ALORS                                          C
                        max = maxTmp;
                        indiceD = indiceCourrantD
                        indiceF = indiceCourrantF
                Fin_si
                indiceCourrantF = indiceCourrantF +1
                FIN_TANT_QUE
                indiceCourrantF = indiceCourrantD +1                           B
                indiceCourrantD = indiceCourrantD +1
        FIN_TANT_QUE
        AFFICHE("Tronçon  [" + indiceD + "-" + indiceF + "] max =" +  max)     A
Fin
```

Calcul de la complexité :

La boucle A s'exécute de 0 à la taille du tableau avec un pas de 1 ce qui nous donne une complexité de N. Notre boucle A appelle à chaque passage la boucle B.
La boucle B s'exécute de indiceCourrantF à la taille du tableau avec un pas de 1. La boucle b est appelé N fois au premier appel. IndiceCourrantF vaudra 0 puis 1 et ainsi de suite jusqu'à N.
Donc la boucle B va s'exécuter $\sum_{i=1}^{N} i$ ce qui nous donne une complexité de $\frac{N(N+1)}{2}$, elle est appelé à chaque passage la boucle C.
La boucle C s'exécute de l'indiceCourrantD jusqu'à indiceCourrantF ce qui nous faira au maximum N passage et ne fera que diminuer par la suite nous arrondissons donc sa complexité en N .
Ainsi la complexité de notre algorithme peut s'apparenter à une complexité de ½ (N3 + n2 + n)
Ce qui nous donne une complexité finale de $\Theta(N^3)$

Algorithme 2

Pseudo code

```
PROCEDURE aglo2 (tab , taille_tab)
        max = tab[0]
        indiceD = 0,
        indiceF = 0,
        indiceCourrantD = 0,
        indiceCourrantF = 0
        tmp = 0
Debut
        TANT_QUE indiceCourrantD < taille_tab
                tmp = 0
                TANT_QUE indiceCourrantF < taille_tab
                        maxTmp = tmp + t[indiceCourrantF]
                        SI maxTmp > max ALORS
                                max = maxTmp
                                indiceD = indiceCourrantD
                                indiceF = indiceCourrantF
                        FIN_SI
                        indiceCourrantF = indiceCourrantF +1
                FIN_TANT_QUE
                indiceCourrantF = indiceCourrantD +1
                indiceCourrantD = indiceCourrantD +1
        FIN_TANT_QUE
        AFFICHE("Tronçon  [" + indiceD + "-" + indiceF + "] max =" +  max)
Fin
```

B

A

Calcul de la complexité :

On considère N la taille du tableau.
En A la boucle va s'exécuter N Fois.
La boucle B va s'exécuter au premier appel 1 fois puis au deuxième 2 fois et ainsi de suite jusqu'au N^{ieme} appel ou elle sera appellé N fois.
La boucle B va donc s'exécuter $\sum_{i=1}^{N} i$ fois ce qui nous donne une complexité finale de $\frac{N(N+1)}{2}$. Ce qui nous donne un algorithme de complexité de O(N²).

Algorithme 3

Pseudo Code

```
FONCTION testSousSequence(Result resultat1,debutSeq1,finSeq1,Result resultat2,
         debutSeq2, finSeq2,tableau tab)
renvoie Result
    Result resultat;
    resultMaxTemp;
    //gauche ou droite
    SI resultat1.max < resultat2.max
        resultat.max = resultat2.max
        resultat.debut = resultat2.debut
        resultat.fin = resultat2.fin
    FIN_SI
    SINON
        resultat.max = resultat1.max
        resultat.debut = resultat1.debut
        resultat.fin = resultat1.fin
    FIN_SINON
    Result resultatMilieu
    //coler au 2
    SI (resultat1.fin == finSeq1 && resultat2.debut == debutSeq2)
        resultatMilieu.max = resultat1.max + resultat2.max
        resultatMilieu.debut = resultat1.debut
        resultatMilieu.fin = resultat2.fin
    FIN_SI

    SINON_SI (resultat1.fin == finSeq1)
        resultatMilieu = collerGauche(resultat1,resultat2,debutSeq2,tab)
    FIN_SI                                                                  1

    SINON_SI (resultat2.debut == debutSeq2)
        resultatMilieu = collerDroite(resultat2,resultat1,finSeq1,tab)
    FIN_SINON                                                               2

        resultatMilieu.max = tab[finSeq1] + tab[debutSeq2]
        resultatMilieu.debut = finSeq1
        resultatMilieu.fin = debutSeq2
        i = finSeq1 -1

        resultMaxTemp = resultatMilieu.max
        TANT_QUE (i >= resultat1.debut )
                 resultMaxTemp += tab[i];
                 SI(resultMaxTemp > resultatMilieu.max)
                   resultatMilieu.max = resultMaxTemp
                   resultatMilieu.debut = i
                 FIN_SI
                 i = i-1
        FIN_TANT_QUE                                                        A

        resultMaxTemp = resultatMilieu.max
        i = debutSeq2 +1
        TANT_QUE(i <= resultat2.fin )
            resultMaxTemp += tab[i]
            SI(resultMaxTemp > resultatMilieu.max){
                 resultatMilieu.max = resultMaxTemp
                 resultatMilieu.fin = i
            FIN_SI
            i = i +1
        FIN_TANT_QUE                                                        B
FIN_SINON
 SI (resultatMilieu.max > resultat.max)
    retourner resultatMilieu                                                3
 FIN_SI
retourner resultat
FIN
```

```
FONCTION collerGauche(Result resultat1,Result resultat2,debutSeq2,tableau tab)
renvoie Result
DEBUT
    i = 0;
    Result resultatMilieu;
    resultatMilieu.max = tab[debutSeq2]+ resultat1.max
    resultatMilieu.debut = resultat1.debut
    resultatMilieu.fin = debutSeq2
    int resultMaxTemp =  resultatMilieu.max
    i = debutSeq2 +1
    resultMaxTemp = resultatMilieu.max
    TANT_QUE i <= resultat2.fin
        resultMaxTemp = resultMaxTemp + tab[i]
        SI(resultMaxTemp > resultatMilieu.max)
            resultatMilieu.max = resultMaxTemp
            resultatMilieu.fin = i
        FIN_SI
        i = i +1
    FIN_TANT_QUE
    RETOURNER resultatMilieu;
FIN

FONCTION collerDroite (Result resultat2,Result resultat1,finSeq1,tableau tab)
renvoie Result
DEBUT
    i = 0
    Result resultatMilieu
    resultatMilieu.max = tab[finSeq1] + resultat2.max
    resultatMilieu.debut = finSeq1
    resultatMilieu.fin = resultat2.fin
    int resultMaxTemp =  resultatMilieu.max
    //si le milieu est améliore par la  sous séquence a gauche
    i = finSeq1 -1
    resultMaxTemp = resultatMilieu.max
    TANT_QUE i >= resultat1.debut
        resultMaxTemp = resultMaxTemp + tab[i]
        SI(resultMaxTemp > resultatMilieu.max){
            resultatMilieu.max = resultMaxTemp
            resultatMilieu.debut = i
        FIN_SI
        --i;
    FIN_TANT_QUE
    RETOUR resultatMilieu;
FIN
```

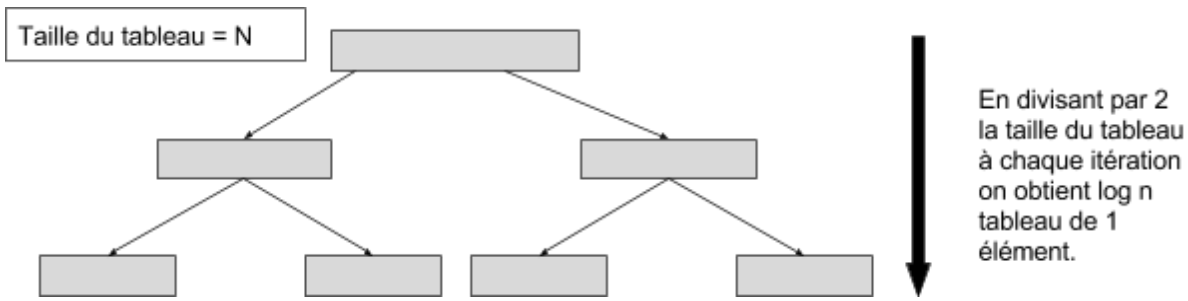
Calcul complexité fonction testSouséquence:

La fonction test sous séquence comporte 3 cas distinct représenter dans les carré 1, 2 et 3 nous nous plaçons dans le pire des cas le cas numéro 3.
Dans ce cas nous voyons qu'il y a 2 boucles la boucle A et la boucle B nous pouvons remarquer que le première boucle parcourt le tableau à partir de l'indice vers la gauche et que la deuxième le parcours vers la droite. Donc notre algorithme dans le pires des cas va parcourir le tableau entièrement donc notre fonction est en O(N)

```
Fonction Result diviserPourRegner(tableau,debut,fin)
DEBUT
        Result r
        SI debut == fin
                r.max = tableau[debut]
                r.debut = debut
                r.fin = fin
        FIN_SI
        SINON SI debut < fin
                Result r1
                Result r2
                r1 = diviserPourRegner(tableau,debut,(debut+fin)/2)          1
                r2 = diviserPourRegner(tableau,((debut+fin)/2)+1,fin)        2
                r = testSousSequence(r1,debut,((debut+fin)/2),r2,(((debut+fin)/2)+1),fin,tableau)
                                                                             A
        FIN_SI
        retourne r;
FIN
```

Calcul de la complexité :

La fonction diviserPourRegner est récursive.
Nous divisons donc notre tableau en 2 partie et nous rappelons la même fonction dessus
En 1 (ci-dessous) nous traitons le tableau sur la première moitié et en 2 sur la seconde moitié.



À ce stade là, je ne fais que remonter ma fonction grâce à la récursivité.
Ayant log n tableau, et la fonction testSousSequence ayant une complexité de 0(n) dans le pire des cas.
On a une complexitée total de 0(n log n) pour ma fonction diviser pour régner.

Algorithme 4

Pseudo Code:

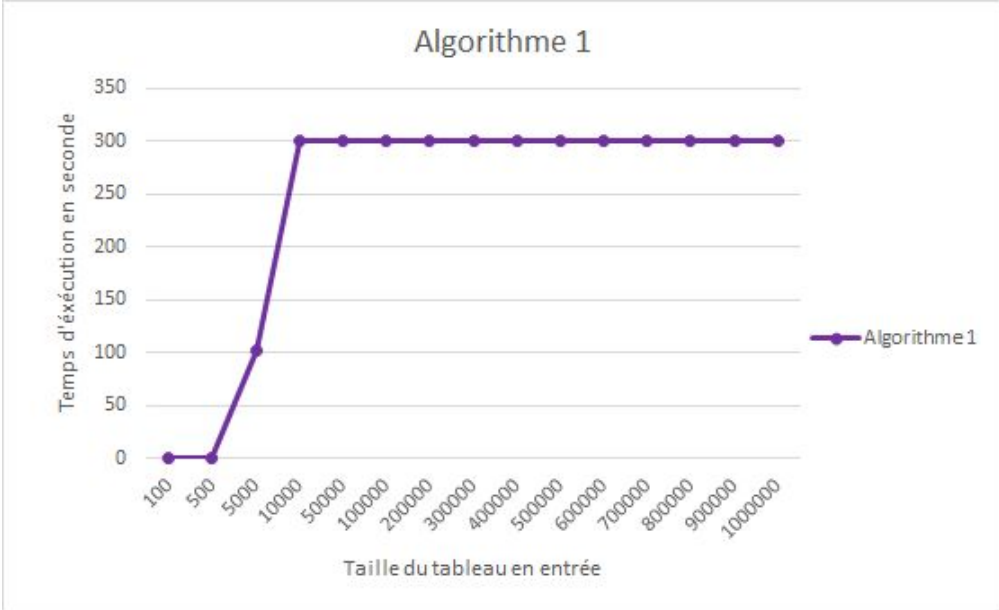
```
FONCTION algo4(tableau[],taille)
Renvoie Result
DEBUT
    Result resultat
    resultat.max = tableau[0]
    resultat.debut = 0
    resultat.fin = 0
    somme = 0
    debutTmp = 0
    POUR i de 0 jusqu'a taille avec un pas de 1
        SI somme < 0
          somme = 0
          debutTmp = i
        FIN_SI
        somme = somme + tableau[i]
        SI resultat.max < somme
            resultat.max = somme
            resultat.debut = debutTmp
            resultat.fin = i
        FIN_SI
    FIN_POUR
    renvoi resultat
FIN
```

A

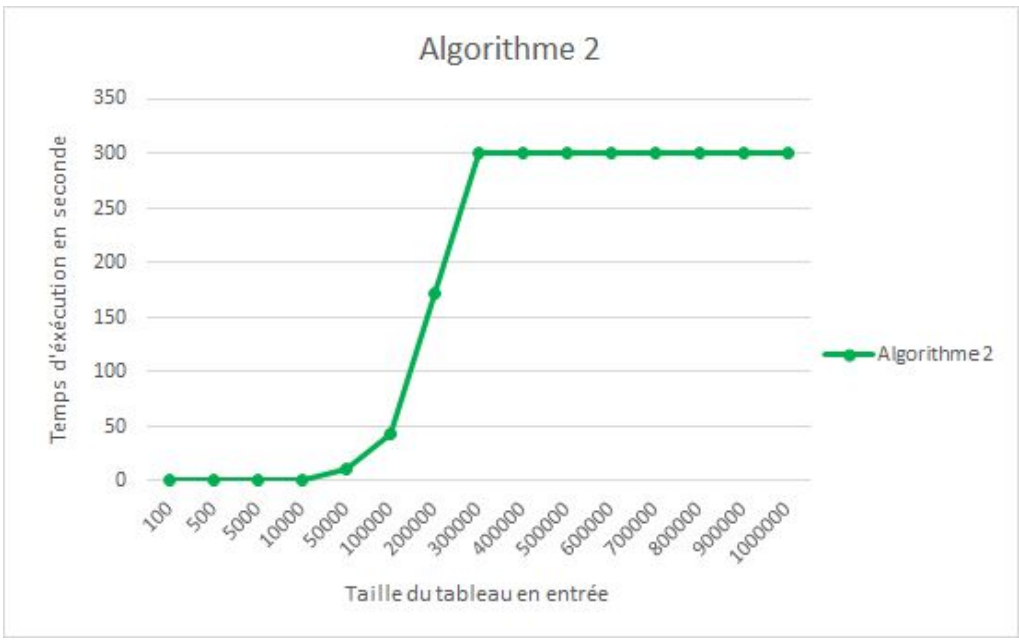
Calcul de la complexité :

Notre algorithme comporte qu'une seule boucle A
La boucle s'exécute de i allant de 0 jusqu'à la taille du tableau donc notre algorithme a une complexité en Θ(N)

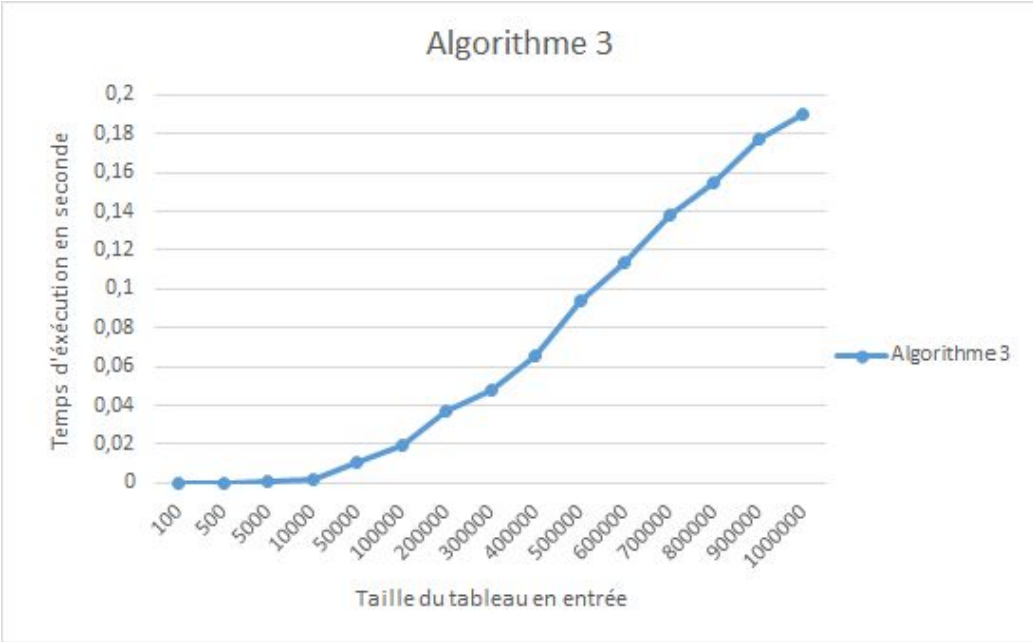
Résultat tests pratiques.



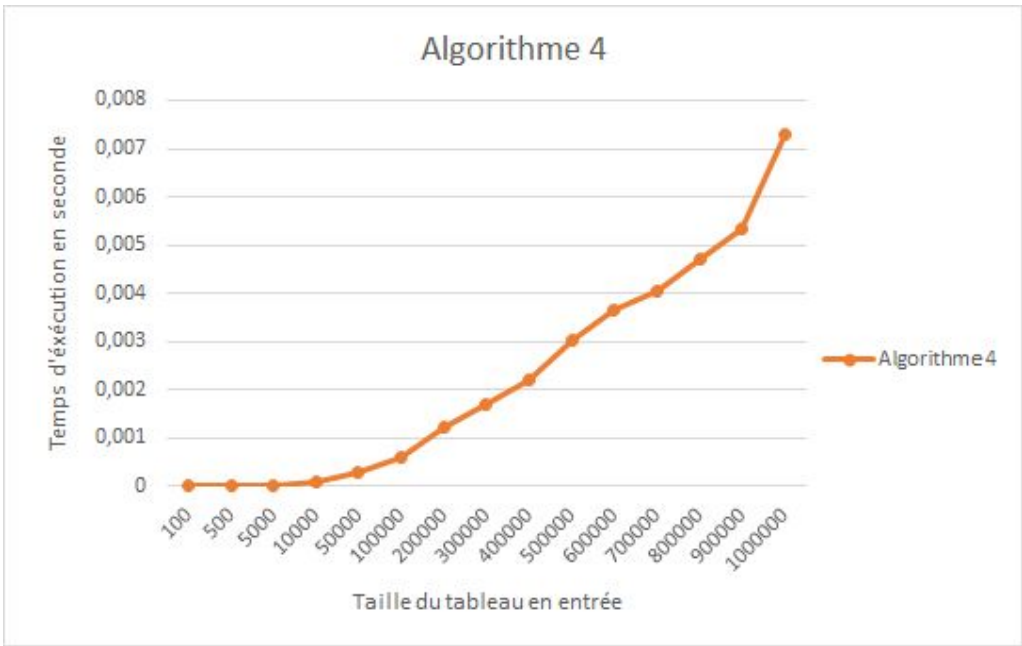
On voit que l'algorithme est rapidement dépassé par le nombre d'entrée. A 10.000 nombre, l'algorithme va mettre plus de 5min.L'algorithme s'arrête à ce moment là, sinon le temps de finir les tests va être trop important. On arrive donc facilement à constater que la complexité de cet algorithme est plus que quadratique. Donc cubique.



En comparaison avec l'algorithme 1, voit que celui-ci commence à être dépassé par le nombre d'entrée à partir de 300.000. Ce qui est beaucoup plus que l'algo1. On peut constater que la complexité de l'algorithme est quadratique



En comparaison avec les 2 derniers algorithmes on voit que celui-ci fini les tests de toutes les tailles de tableau en moins d'une seconde chacun. Pour un tableau de 1.000.000 d'éléments, l'algorithme a fini de s'exécuter en seulement 0.2 secondes. On peut voir avec l'allure de la courbe, que cet algorithme à une complexité presque linéaire. En n log n pour être précis.



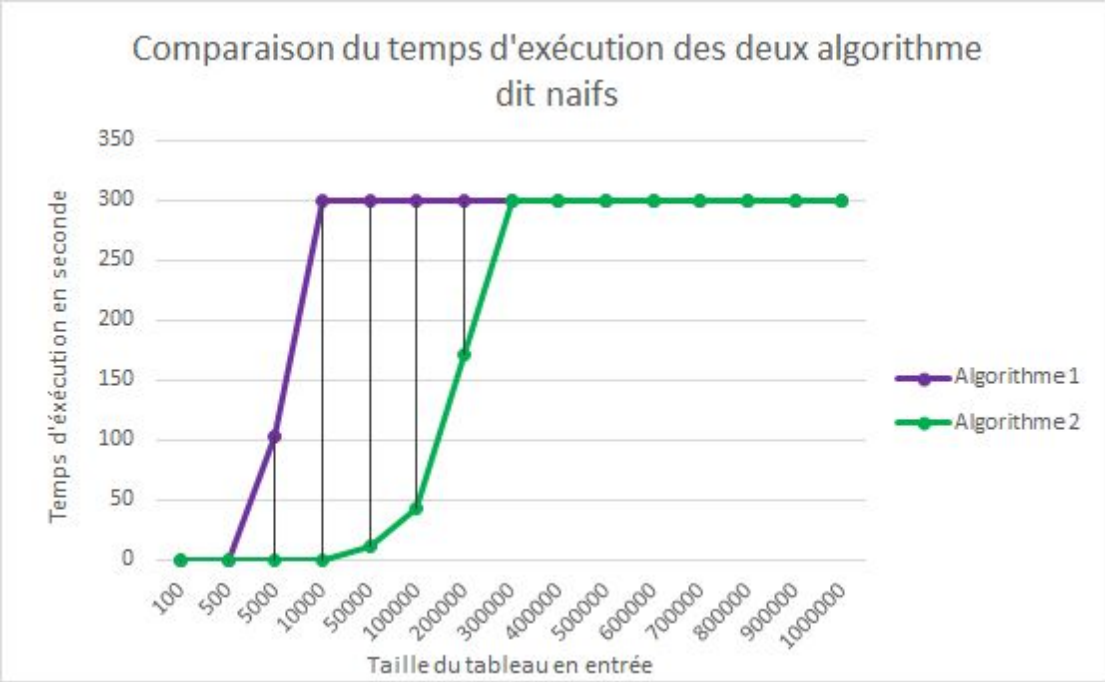
Comparée à l'algorithme précédent, l'algo est beaucoup plus rapide. Pour un tableau de 1.000.000 d'éléments, il a fini son exécution en moins de 0.008 seconde soit presque 100

fois plus rapide l'algo avec une complexité en n log n. L'allure de la courbe nous indique un tracé presque régulier. On peut en déduire que la complexité de l'algorithme est linéaire (O(n)).

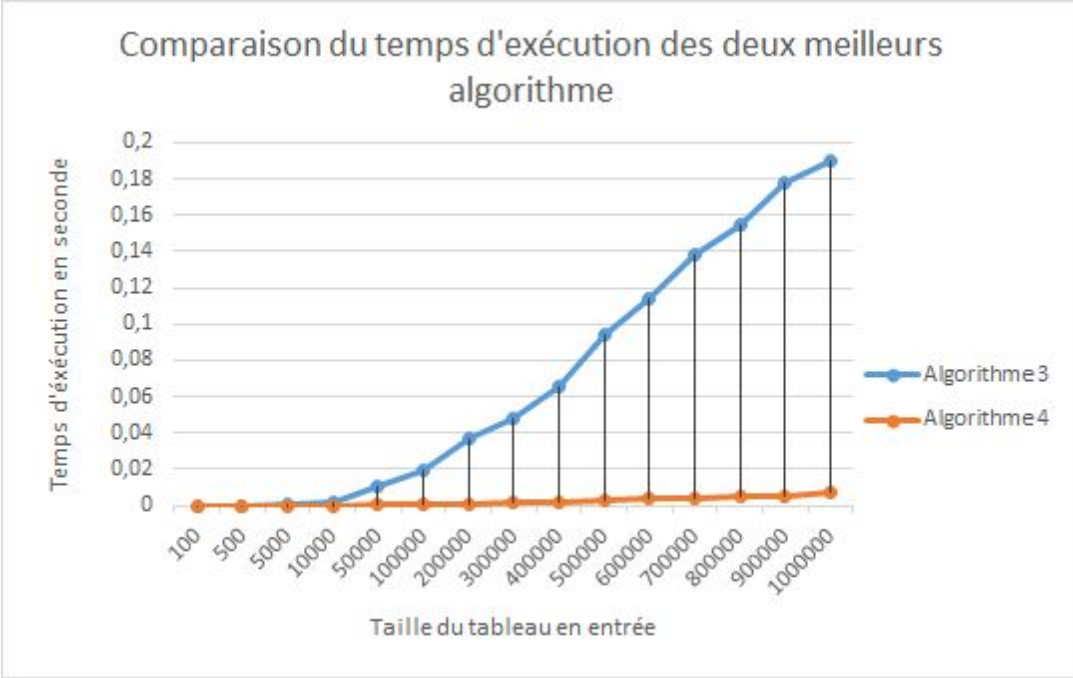
Analyse des résultat obtenue

Nom\Taille	100	500	5000	10000	50000	100000	200000
algo1	0.001127	0.101403	102.603049	300	300	300	300
algo2	0.000046	0.001105	0.107627	0.432029	10.777609	43.027651	172.123258
algo3	0.000016	0.000092	0.000783	0.00161	0.010748	0.019322	0.036592
algo4	0.000003	0.000007	0.000035	0.000098	0.000294	0.000621	0.001224

300000	400000	500000	600000	700000	800000	900000	1000000
300	300	300	300	300	300	300	300
300	300	300	300	300	300	300	300
0.047855	0.065674	0.094105	0.113708	0.137798	0.154749	0.177485	0.189983
0.001698	0.002197	0.003024	0.003641	0.004067	0.004697	0.005341	0.007294



On voit grâce aux allures des courbes, que la croissance de l'algorithme 1 est bien plus rapide que celle de l'algorithme 2. On voit très clairement que la complexité de l'algo 1 est en $0(n^3)$ et en $0(n^2)$ pour l'algo 2.



Pour l'algorithme 3 et 4, on voit bien ci-dessus que le 4 est beaucoup plus rapide que le 3. Comparer comme ceci, on voit bien que la complexité de l'algo 4 est linéaire. Tandis que pour l'algo 3, à partir de 50.000 éléments, on voit que le temps mis pour finir est de plus en plus important par rapport à l'algo 4. Malgrès ce décalage de temps, les performances de l'algorithme 3 est plus que correct par rapport aux algorithmes 1 et 2.

Choix du langage

Nous avons choisi le langage C pour faire ce projet, car nous avons des facilités à coder avec ce langage.
En C, il est plus facile d'implémenter un système d'alarme pour arrêter le programme lorsque celui-ci met trop de temps.
Nous utilisons aussi, tous les coeurs de l'ordinateur qui exécute le programme, pour optimiser le temps de calcul. Avec ce procédé, nous pouvons avoir des résultat plus rapidement sans pour autant être gêné par les limites de l'ordinateur.

Conclusion

Nous avons donc vu au cour de ce projet que la complexité est importante pour un algoritme. Pour un même problème, nous avons créé 4 algorithmes avec des complexité totalement différentes, et donc un temps d'exécution très différents. Il est donc important de créer des algorithme qui ont une faible complexité.