UNIVERSITÉ PARIS 13 | USPC Université Sorbonne Paris Cité

Structures de Données Avancées

Rapport TP1

Tables Dynamiques

Réalisé par :

Aoudjehane Sarah

KOUACHI Abdeldjalil

1) **définition pour la fonction potentielle dans le cas ou en multiplie la taille par un facteurα≥1**

$\Phi(i) =$ α n(T) – taille (T)/α -1

2) **Coût amorti de l'opération « insérer table » en fonction de ĉ**

on a ĉ i = c i + Φ i – Φ i-1

c i : le coût réel de la i ième opération

ĉ i : le coût amorti de la i ième opération

Φ i : fonction du potentiel avant l'opération

Φ i-1 : fonction potentielle après l'opération

on distingue deux cas:

cas 1 : pas d'extension de table :

Dans ce cas, on a : c i = 1, t i-1 = t i , n i-1 =n i - 1, t i = n i

ĉ i = ci +(αn_i – $taille_i$/α -1) –(α n_{i-1}– $taille_{i-1}$)/α -1

= 1+ (αn_i - ti)/(α - 1)- (αn_{i-1} - t_{i-1}/(α - 1)

=1+α(n_{i-1} +1) –ti/(α – 1) - αn_{i-1} + t_{i-1}/(α - 1)

=1+ αn_{i-1} +α -t_{i-1}/(α – 1) - αn_{i-1} + t_{i-1}/(α – 1)

ĉ i =1 + α/ α – 1

cas 2 : il y a extension de table

ti=αt_(i-1)

Ci=ti=ni

ni=n(i-1)+1=t(i-1)+1

ĉ i = ci +(αn_i – $taille_i$/α -1) –(α n_{i-1}– $taille_{i-1}$)/α -1

= t_i+ (αn_i - ti)/(α - 1)- (αn_{i-1} - t_{i-1}/(α – 1)

= t_{i-1}+ α(n_{i-1} +1) – α ti-1/(α – 1) - αn_{i-1} + t_{i-1}/(α - 1)

= t_{i-1+} (αn_{i-1} +α - α t_{i-1}/(α – 1) - αn_{i-1} + t_{i-1}/(α – 1)

ĉ i =α/ α – 1

3) les coûts réels et amortis de l’opération Inserer-Table

langage c

```
11928617@g207-15:~/sda-tp1$ cd C/
11928617@g207-15:~/sda-tp1/C$
11928617@g207-15:~/sda-tp1/C$ make
gcc    -c -o arraylist.o arraylist.c
gcc    -c -o analyzer.o analyzer.c
gcc    -c -o main.o main.c
gcc -Wall -ansi --pedantic -O3 -o arraylist_analysis arraylist.o analyzer.o main.o -lm
11928617@g207-15:~/sda-tp1/C$
11928617@g207-15:~/sda-tp1/C$ ./arraylist_analysis
Total cost: 25662148.000000
Average cost: 25.662148
Variance: 2186900802055.454160
Standard deviation: 1478817.366024
11928617@g207-15:~/sda-tp1/C$
11928617@g207-15:~/sda-tp1/C$ make clean
rm -rf *.o
rm -rf *~
rm -rf arraylist_analysis
```

langage c++

```
11928617@g207-15:~/sda-tp1/C$ cd ..
11928617@g207-15:~/sda-tp1$ # Compilation et exécution en C++:
11928617@g207-15:~/sda-tp1$
11928617@g207-15:~/sda-tp1$ cd CPP/
11928617@g207-15:~/sda-tp1/CPP$
11928617@g207-15:~/sda-tp1/CPP$ make
g++   -c analyzer.cpp -o analyzer.o
g++   -c main.cpp -o main.o
g++  -Wall -ansi --pedantic -g -o arraylist_analysis analyzer.o main.o
11928617@g207-15:~/sda-tp1/CPP$
11928617@g207-15:~/sda-tp1/CPP$ ./arraylist_analysis
Total cost :3.39689e+07
Average cost :33.9689
Variance :4.07223e+11
Standard deviation :638140
11928617@g207-15:~/sda-tp1/CPP$
11928617@g207-15:~/sda-tp1/CPP$ make clean
rm -rf *.o
rm -rf *~
rm -rf arraylist analysis
```

langage java

```
11928617@g207-15:~/sda-tp1$ # Compilation et exécution en Java:
11928617@g207-15:~/sda-tp1$
11928617@g207-15:~/sda-tp1$ cd Java
11928617@g207-15:~/sda-tp1/Java$
11928617@g207-15:~/sda-tp1/Java$ javac *
11928617@g207-15:~/sda-tp1/Java$
11928617@g207-15:~/sda-tp1/Java$ java Main
Total cost : 95095893
Average cost : 95.095893
Variance :4725254397596465.771134532551
Standard deviation :68740485.869656652212142944335937 5
```

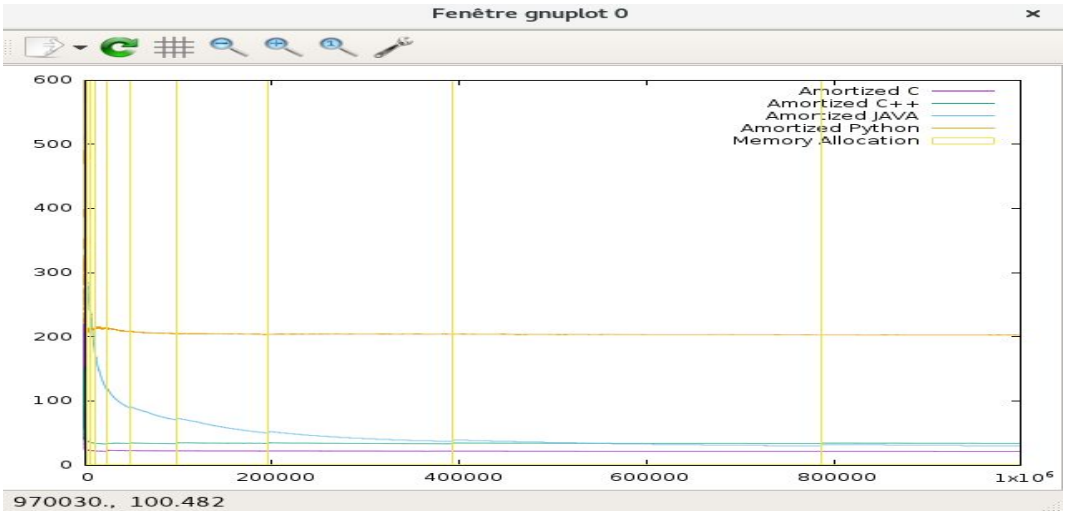
langage python

```
11928617@g207-15:~/sda-tp1$ # Exécution en Python
11928617@g207-15:~/sda-tp1$
11928617@g207-15:~/sda-tp1$ cd Python
11928617@g207-15:~/sda-tp1/Python$
11928617@g207-15:~/sda-tp1/Python$ python main.py
Total cost : 206575632.095
Average cost : 206.575632095
Variance :2.29811476475e+11
Standard deviation :479386.562677
11928617@g207-15:~/sda-tp1/Python$
```

Par conséquent on en déduit le tableau suivant :

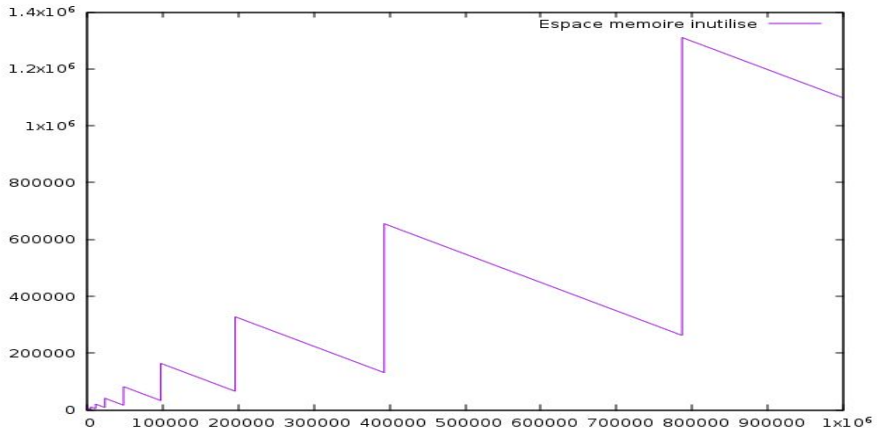
Langage	Total cost	Average cost
C	25662148.00	25.662148
C++	3.39689e+07	33.9689
Java	95095893	95.095
Python	206575632.095	206.57

Le graphe correspondant au coût amorti des 4 langages:

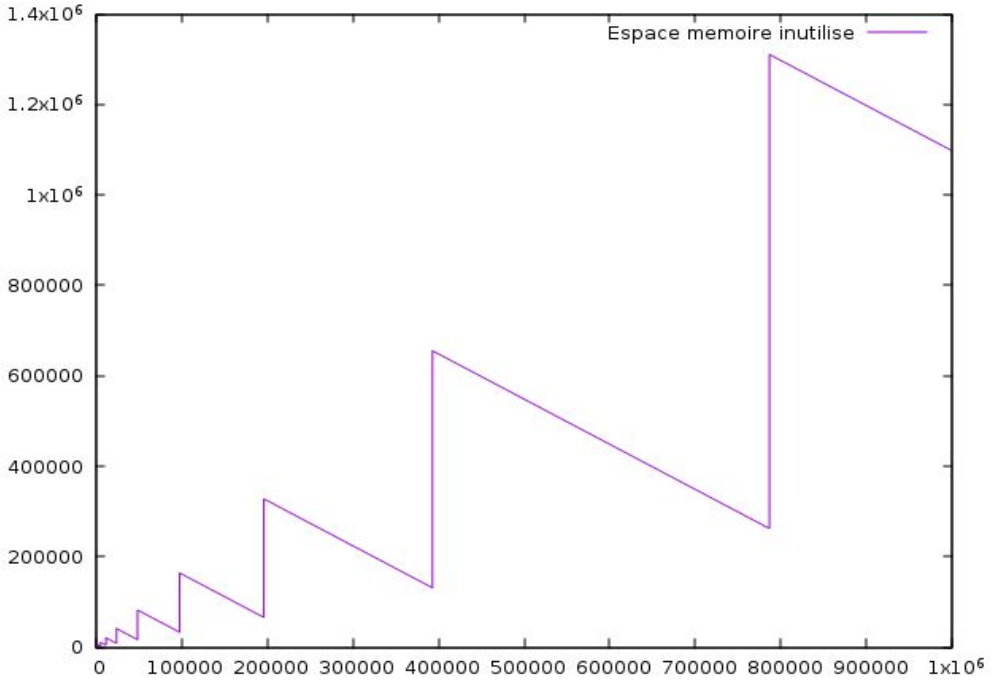


les graphes correspondant à l'espace mémoire inutilisés :

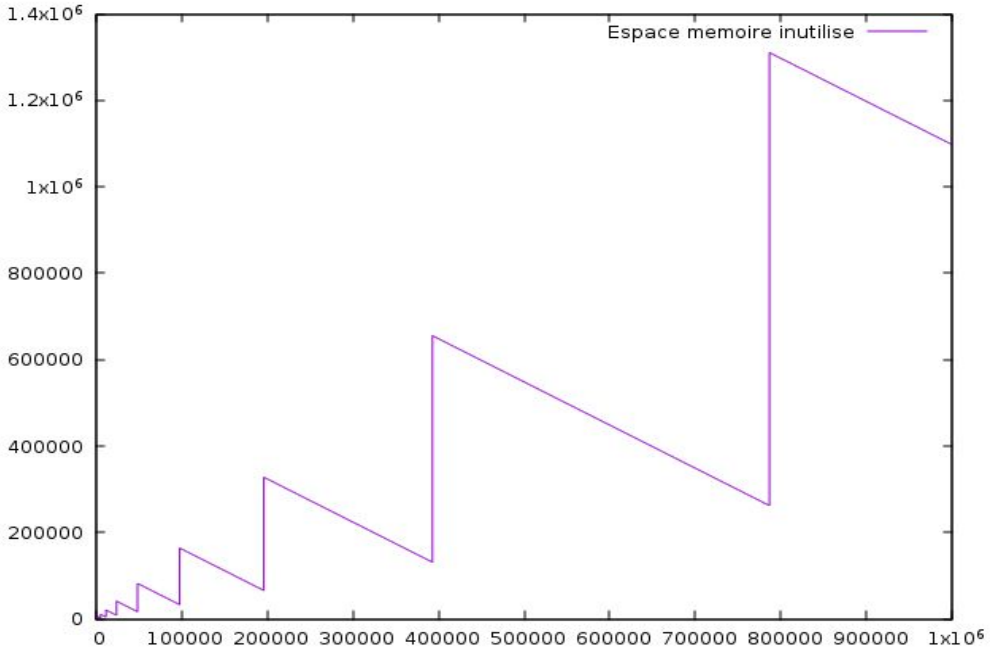
C



C++

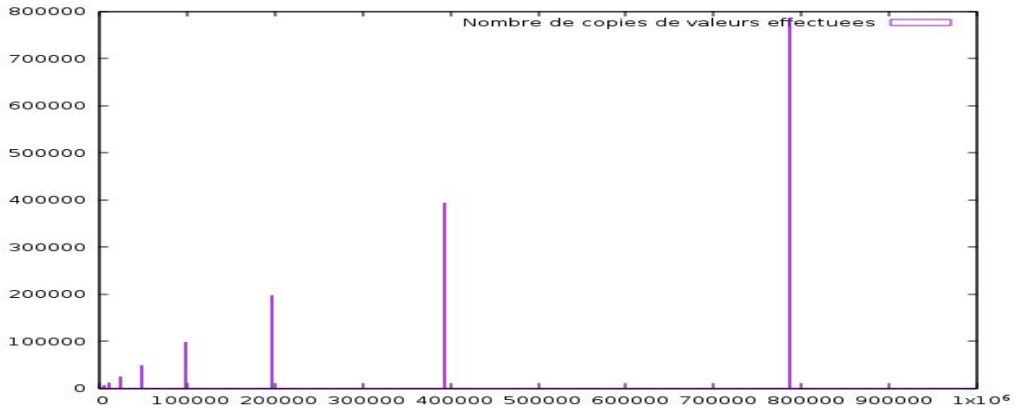


JAVA

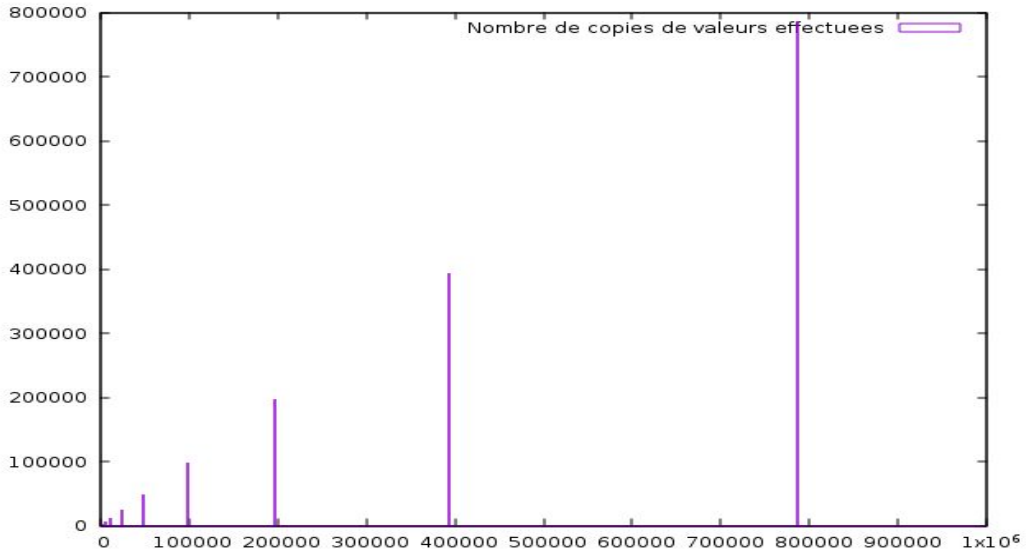


les graphes correspondant au nombre de copie dans chaque langage:

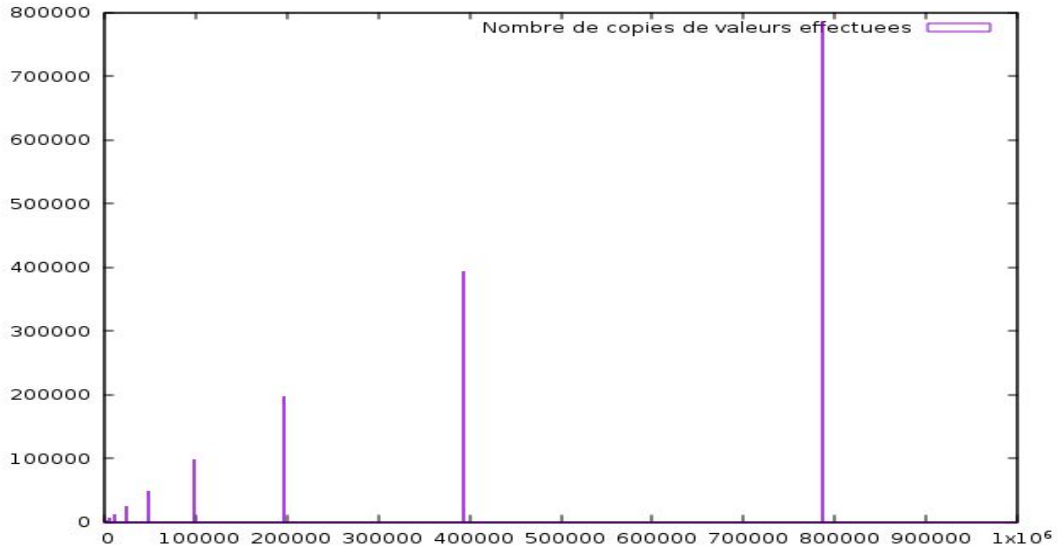
C


Nombre de copies de valeurs effectuees
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

C++


Nombre de copies de valeurs effectuees
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

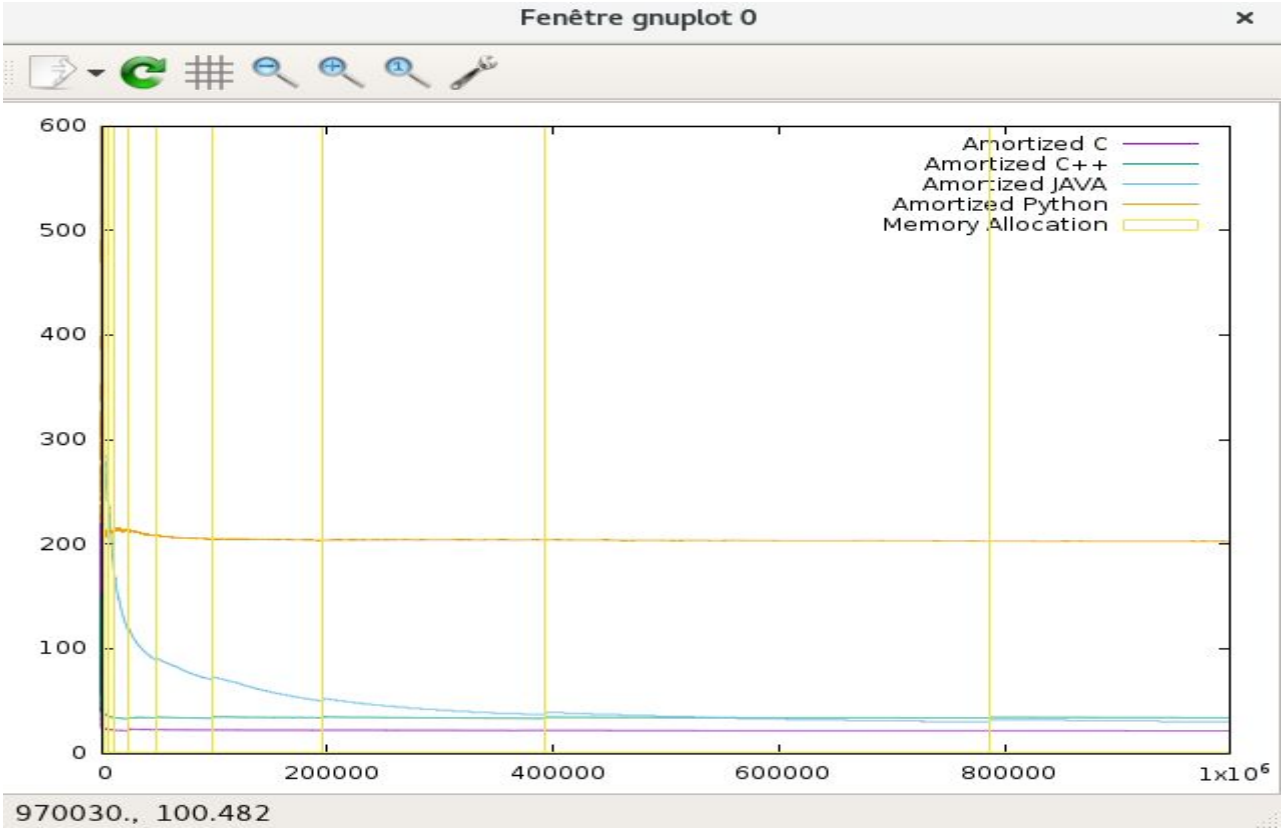
JAVA


Nombre de copies de valeurs effectuees
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

a) Pour voir le morceau de code le plus lent on regarde le code”sda/c/arrayList.h”.On remarque que toutes les fonctions se font en temps constant O(1) sauf celle qui permet d'insérer desélément“arraylist-append”qui prend beaucoups plus de temps quand il y a une extension sa complexité est en O(size) cad dans le pire des cas et dans le meilleur des cas“y a pas d'extension” c’est en 0(1).

```
for(i = 0; i < 1000000 ; i++){
    // Ajout d'un élément et mesure du temps pris par l'opération.
    before = System.nanoTime();
    memory_allocation = a.append(i);
    after = System.nanoTime();

    // Enregistrement du temps pris par l'opération
    time_analysis.append(after - before);
    // Enregistrement du nombre de copies efféctuées par l'opération.
    // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
    copy_analysis.append( (memory_allocation == true)? i: 1);
    // Enregistrement de l'espace mémoire non-utilisé.
    memory_analysis.append( a.capacity() - a.size() );
}
```

b)



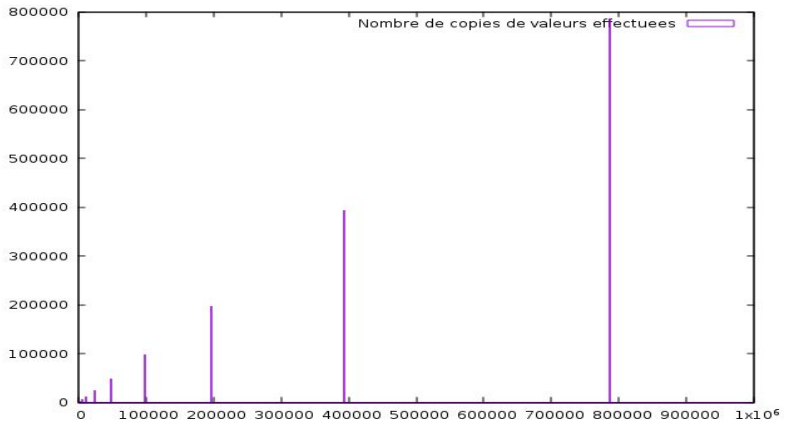
Nous remarquons dans ce graphe que python a un coût amorti supérieur par rapport aux autres langages,suivi de java puis C++ et enfin le C. On remarque aussi que le coût amorti est élevé au départ et à chaque fois que le nombre d'opération(insertion) augmente le coût se réduit jusqu'à ce qu'il devienne constant et ça pour tous les langages.

justification :

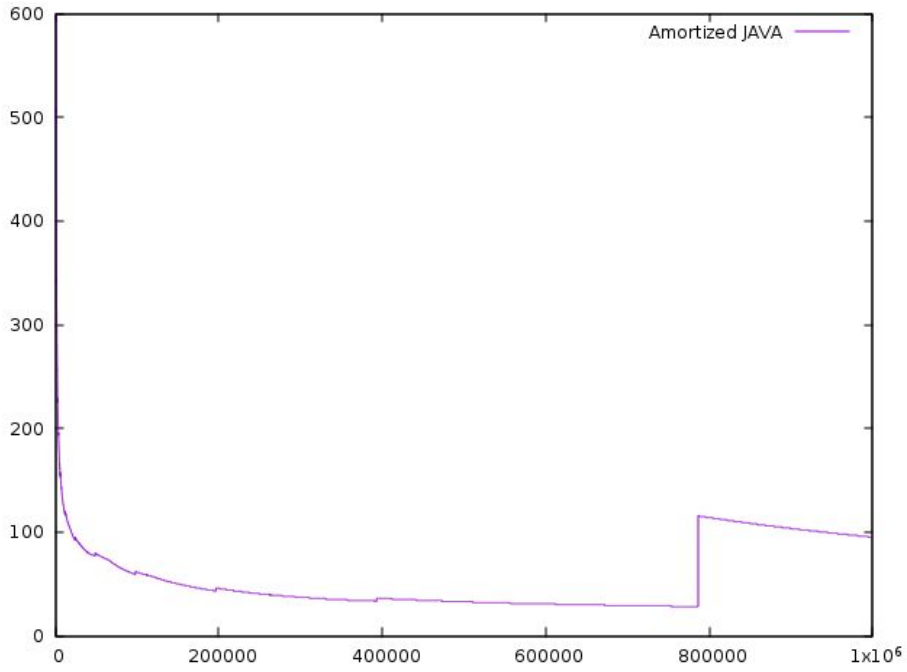
Nous constatons que python a un coût amorti plus grand que les autres langages car c'est un langage interprété, qui par conséquent ne nécessite donc pas d'être compilé pour fonctionner. il ne s'agit pas d'un langage compilé comme le C, ce qui rend ce langage un peu plus lent, ce qui explique l'augmentation de son coût amorti.

Nous constatons aussi que java est derrière lui suivi de c++ car ce sont des langages orientés objets donc qui doivent être compilé. Exemple java utilise le compilateur Java et ne produit pas directement un fichier exécutable mais un bytecode. Pour cela il fait appel à la machine virtuelle Java qui va interpréter le code intermédiaire en vue de l'exécution du programme ce qui fait de lui un langage qui se situe entre un langage interprété et compilé.

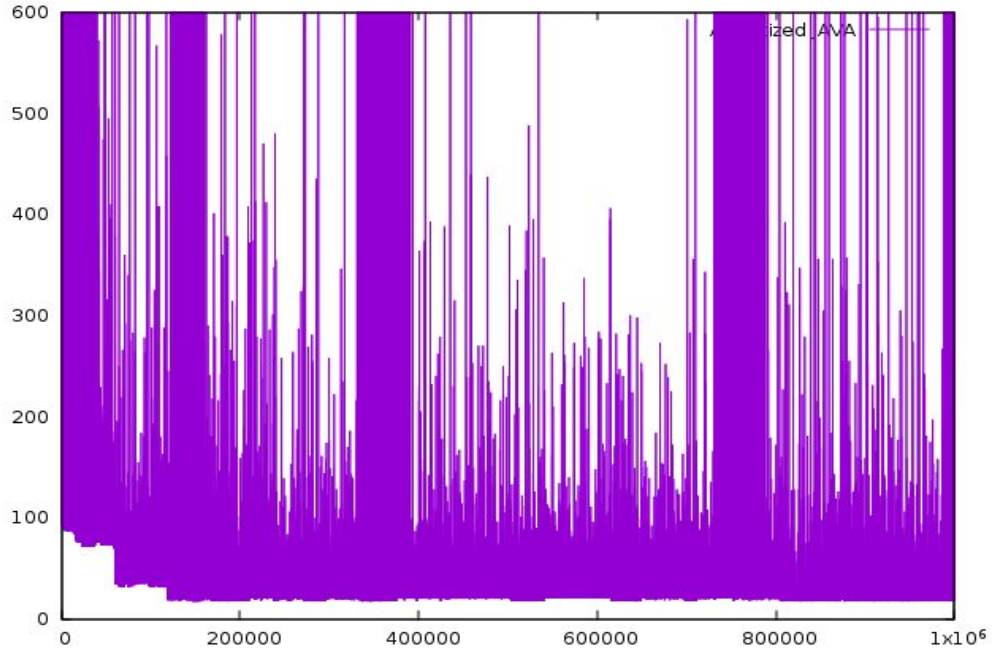
c) le nombre de copie pour Java :



le coût amorti pour Java :



le coût réel pour Java :




```
gnuplot> plot [0:1000000][0:600]'dynamic_array_time_java.plot' using 1:2 w lines
 title "Amortized JAVA",
```

Nous constatons dans le premier graphe que le nombre de copie augmente au fur à mesure qu'on ajoute des éléments dans la table, ce à chaque fois que le nombre d'éléments dans la table est doublé.

Dans le deuxième graphe le coût amorti est élevé au départ, et décroît à chaque fois qu'on ajoute des éléments dans la table jusqu'à ce qu'il se stabilise et devient constant.

Lors de l'insertion d'un élément dans la table, le coût réel vaut 1 si la table n'est pas pleine, donc le nombre de copie c'est 1, s'il y a une extension de table le coût réel dépend du nombre élément copiées de l'ancienne table plus le nouveau élément inséré.donc le coût réel est égal au nombre de copie effectuées.

Remarque:

Plus on fait de copie,plus on gagne en mémoire gaspillé cad on n'alloue pas de l'espace a chaque fois

d)

Tableau comparatif de total cost :

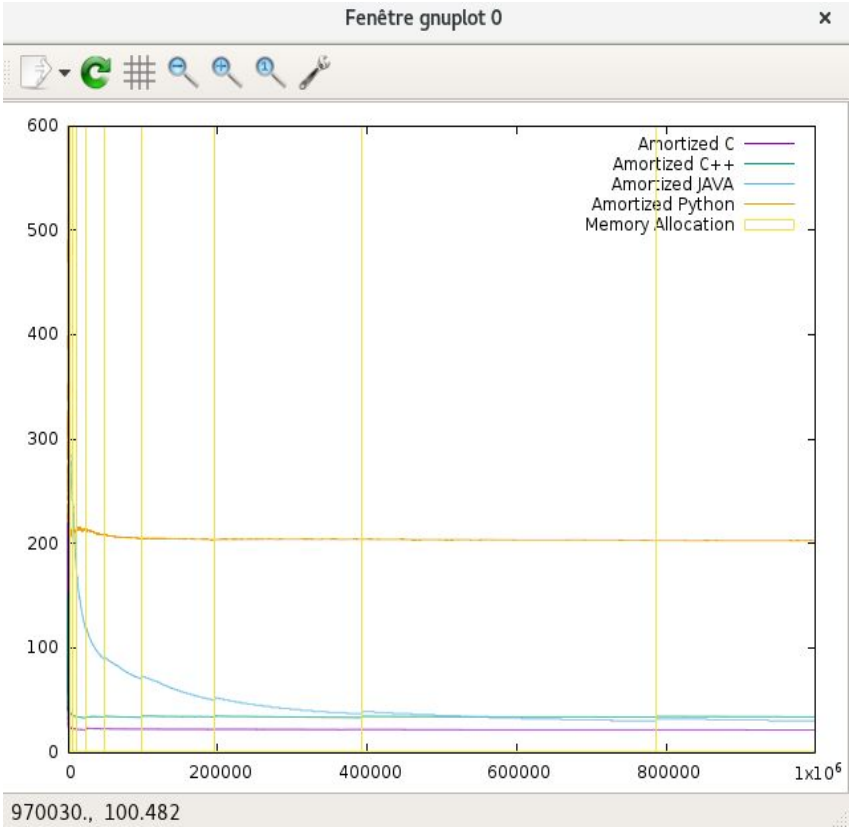
Langage	Expérience 1	Expérience 2	Expérience 3
C	39912305.000000	40731505.000000	37091523.000000
C++	6.15466e+07	5.98249e+07	5.78387e+07
JAVA	47651259	53714226	53092967
Python	449557304.382	449746370.316	442247867.584

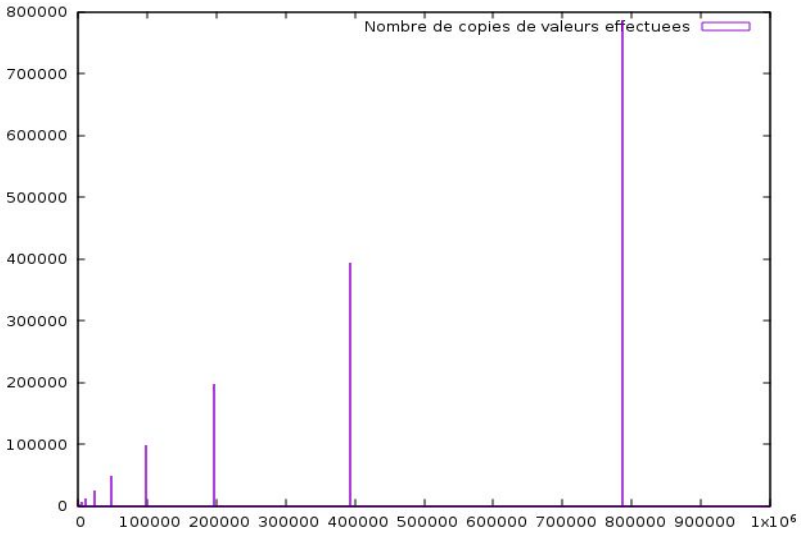
Tableau comparatif de Average cost :

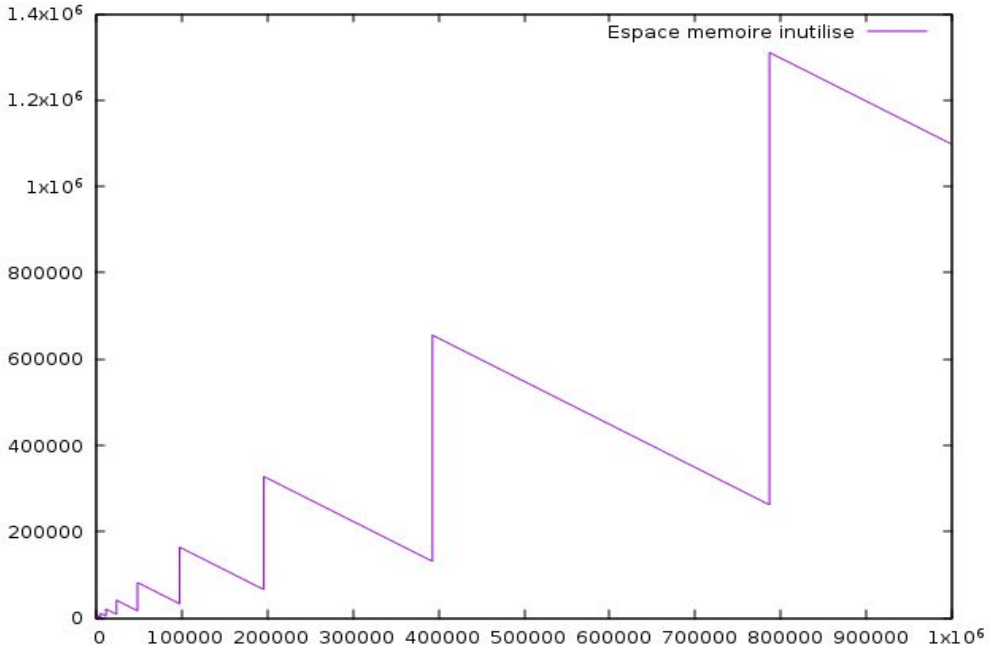
Langage	Expérience 1	Expérience 2	Expérience 3
C	39.912305	40.731505	37.091523
C++	61.5466	59.8249	57.8387
JAVA	47.651259	53.714226	53.092967
Python	449.557304382	449.746370316	442.247867584

Comparatif de graphe Coût amorti :

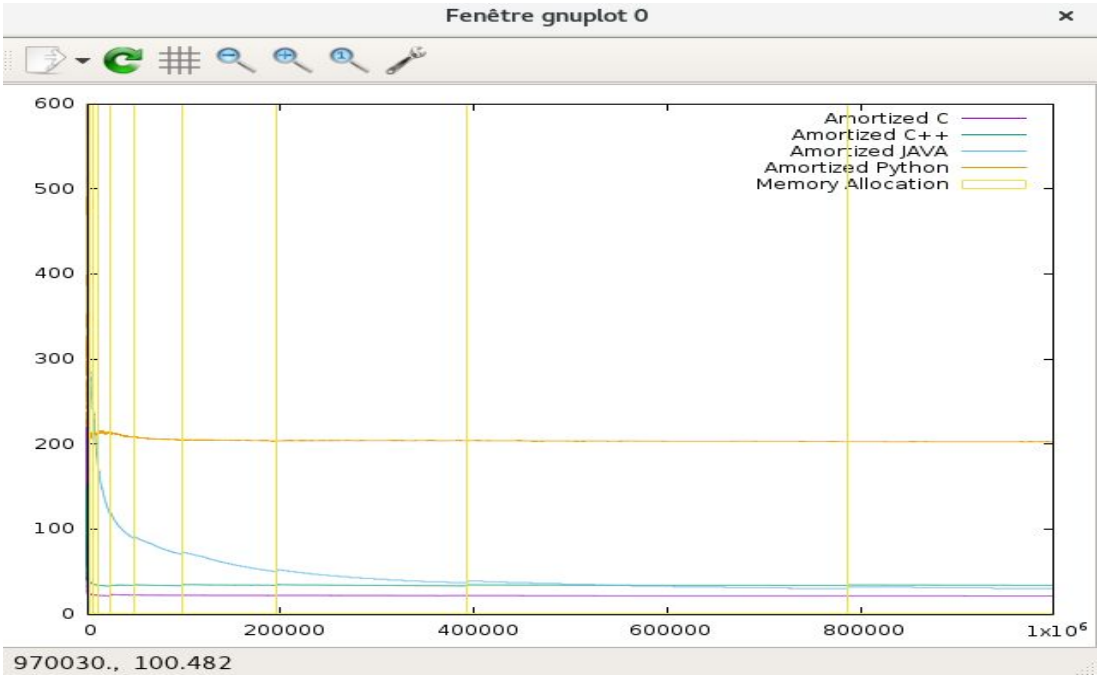
Expérience 1 :

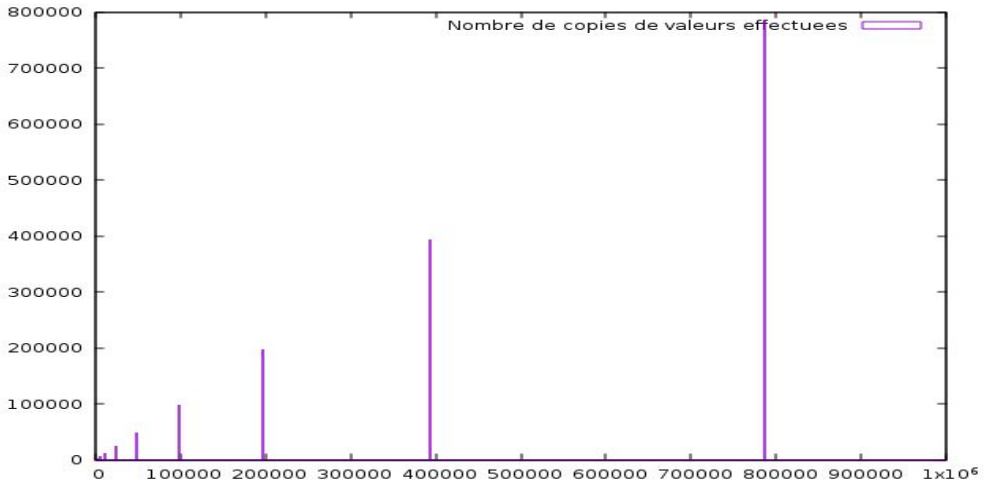


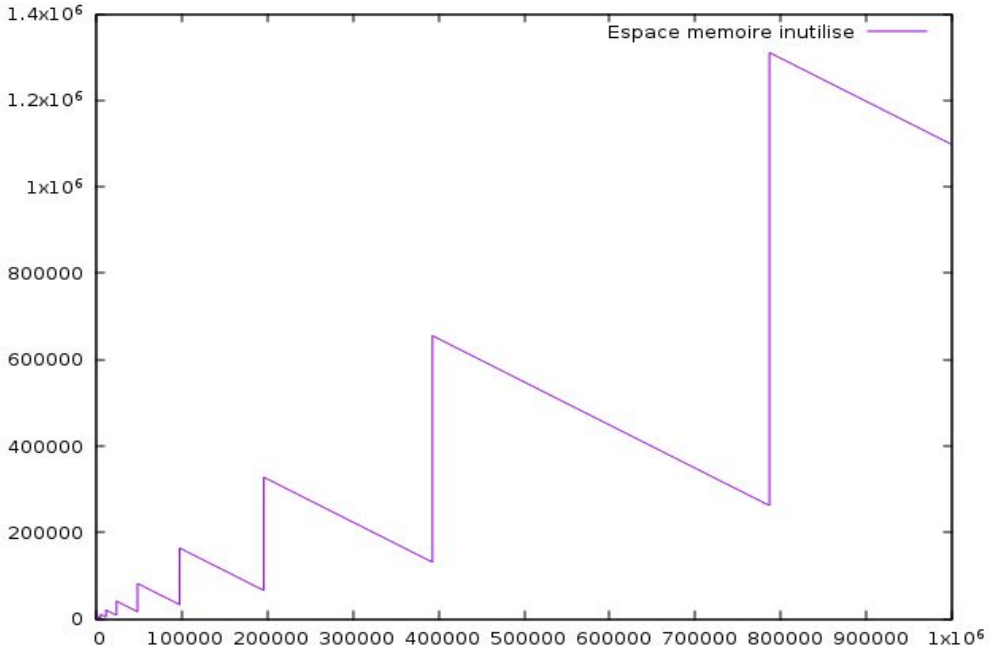




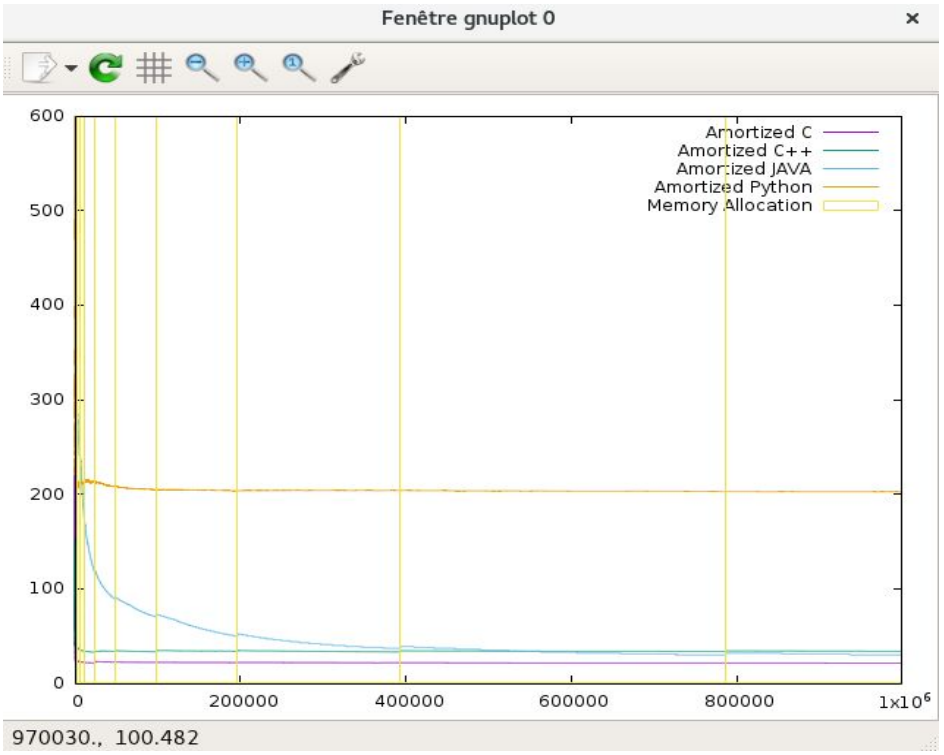
Expérience 2 :

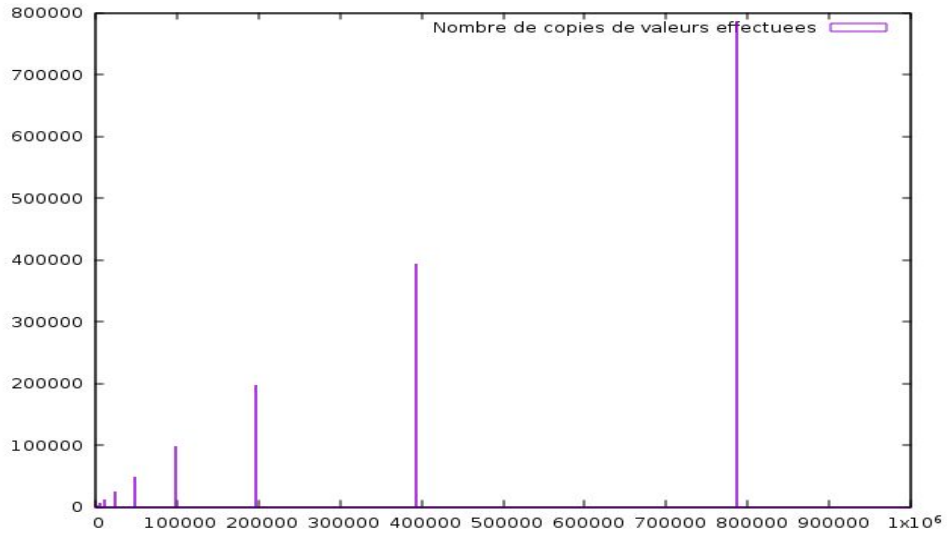


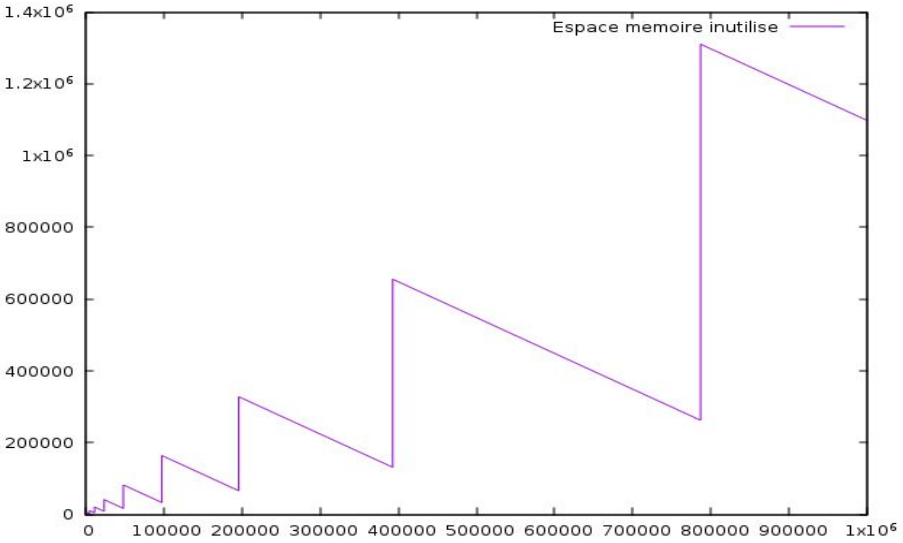




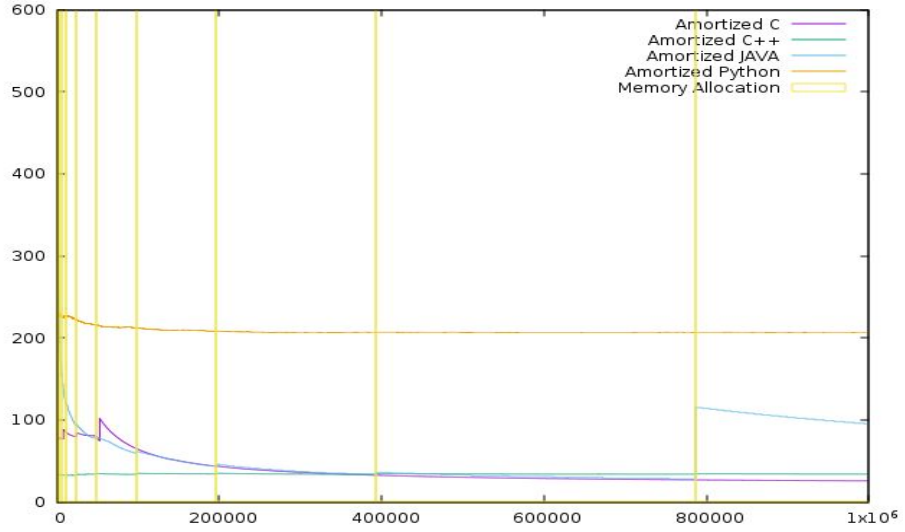
Expérience 3 :

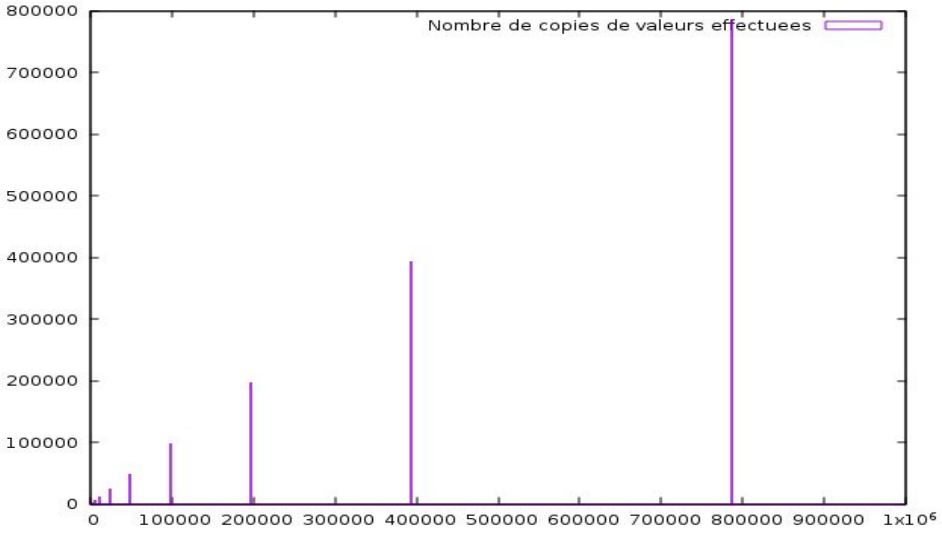

Fenêtre gnuplot 0
Amortized C
Amortized C++
Amortized JAVA
Amortized Python
Memory Allocation
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6
970030., 100.482

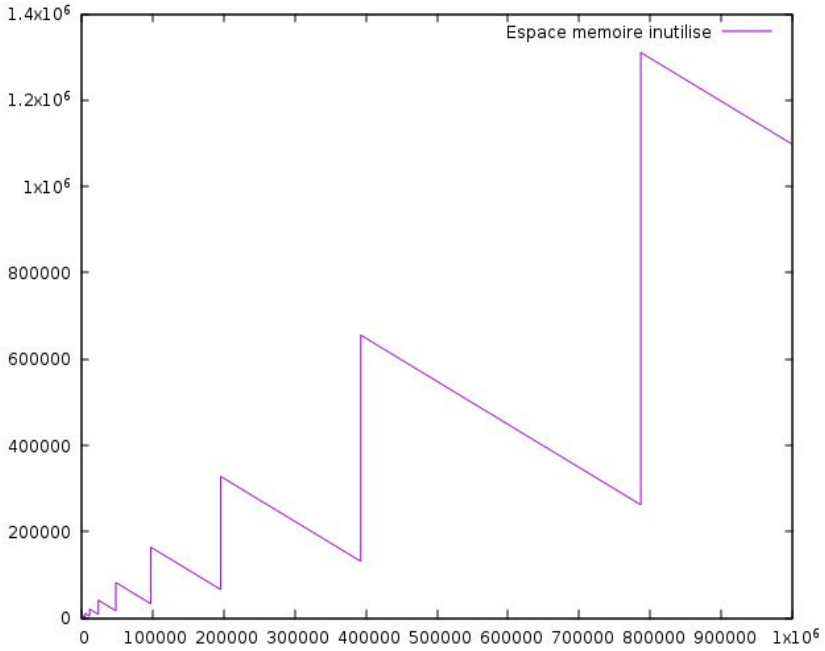

Nombre de copies de valeurs effectuees
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6



Expérience 4







le Constat :

- D'après le tableau,on remarque que le coût réel change a chaque fois qu'on refait l'expérience.
- le nombre de copie ainsi l'espace inutilisé reste constant.

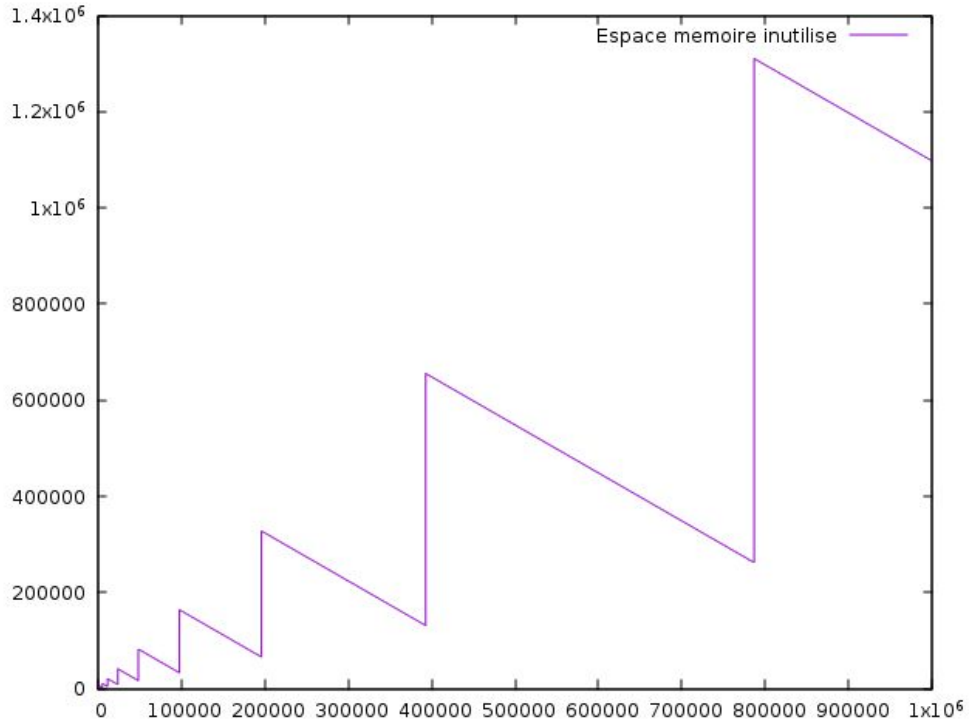
- D'après nos graphes on remarque que le coût amorti est élevé pour python (plus lent) puis java suivi de c++et enfin c (plus rapide) dans les 4 expériences.
- Pour java on remarque une variation à partir de expérience 4.

justification : Le temps d'exécution change d'une expérience à une autre car Quand nos programmes s'exécutent, il y a d'autres processus qui s'exécutent à la fois, d'où ce changement.

E)Comparaison des laganges

- le C et C++ sont des langages totalement compilés.
- Par contre Java et Python sont des langages interprétés, il y a un programme qui lit le code et l'interprète ce qui prend du temps. Le garbage Collector vérifie lui aussi s'il y a de la mémoire à vider.
- Java accelere a un certain moment afin de compiler les bouts de code réutilisés.
- C++, Java et Python on a réécrit une classe qui appelle une autre, qui fait les mêmes tests.

F) Espace mémoire inutilisé :



Dans ce graphe on remarque que l'espace mémoire inutilisé est le même pour tout les langages.Il augmente à chaque fois qu'on insère des éléments dans la table et se double lorsqu'on fait une extension.Un scénario qui pourrait poser problème est lorsque notre machine ne dispose plus de mémoire.

4)

- Nous avons choisi de travailler en langage Java.
- le nombre d'éléments du tableau doit être égale à sa capacité afin que la fonction do_we_need_to_enlarge_capacity ne se déclenche que lorsque le tableau est plein.

illustration:

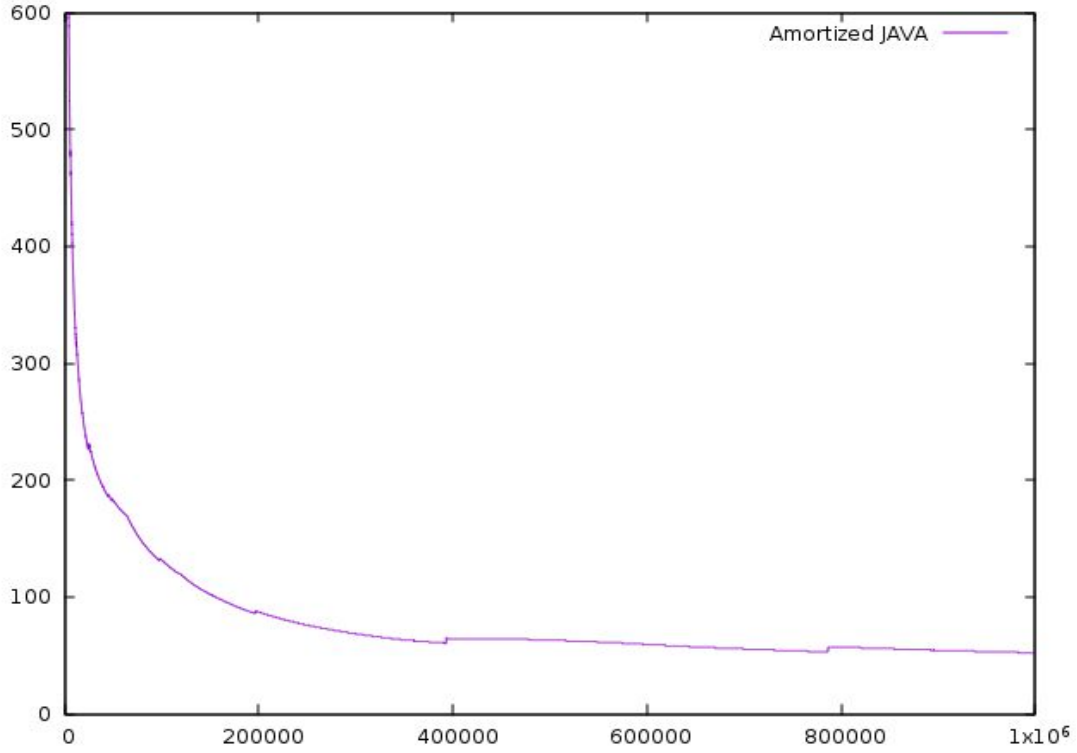
```
 Cette fonction détermine la règle selon laquelle un espace mémoire plus grand sera alloué ou non.
 @returns true si le tableau doit être agrandi, false sinon.
 */
private boolean do_we_need_to_enlarge_capacity() {
   // return size >= (capacity * 3)/4;
    reuturn size == capacity;
}
```

Résultat de l'exécution :

```
11928626@g210-2:~/Documents/sda-tp1/Java$ java Main
Total cost : 52130521
Average cost : 52.130521
Variance :16978418568415.408780268559
Standard deviation :4120487.661480787210166454315185546875
11928626@g210-2:~/Documents/sda-tp1/Java$
```

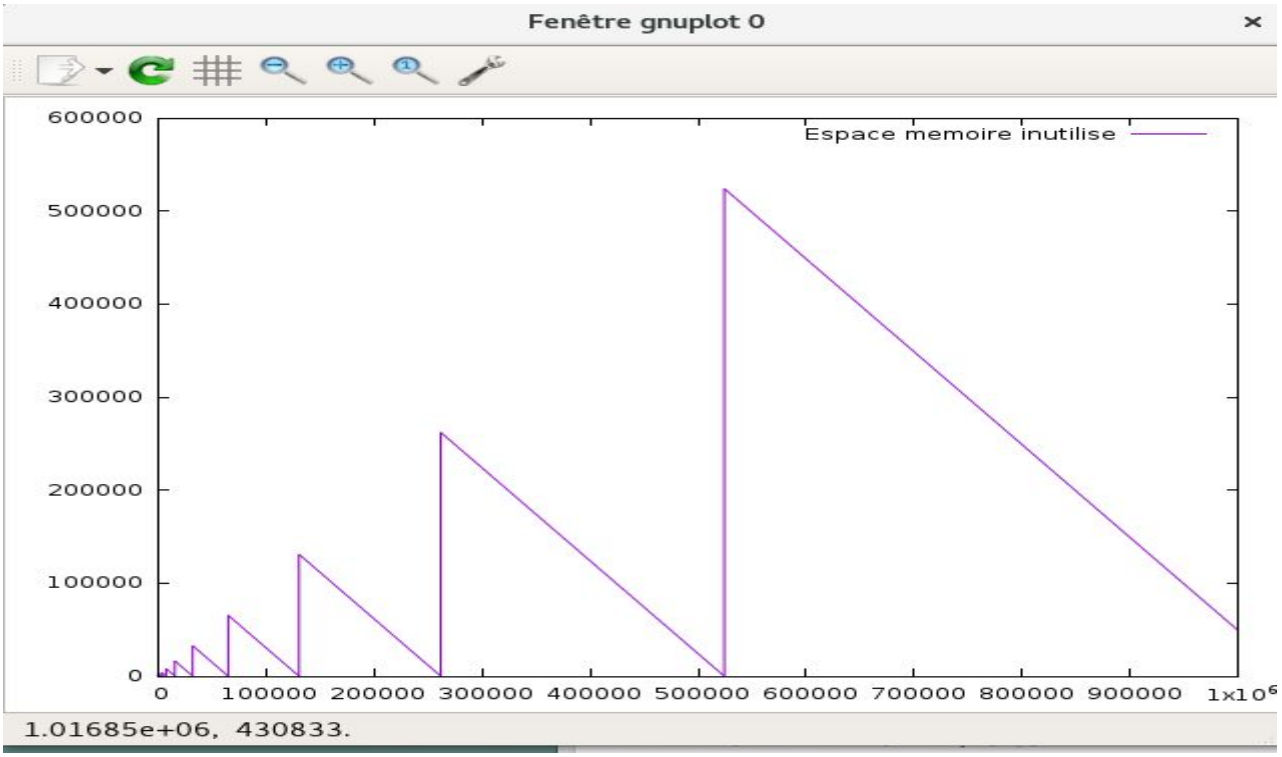
On remarque que le coût total est réduit par rapport aux expérience précédentes

Le coût amorti :



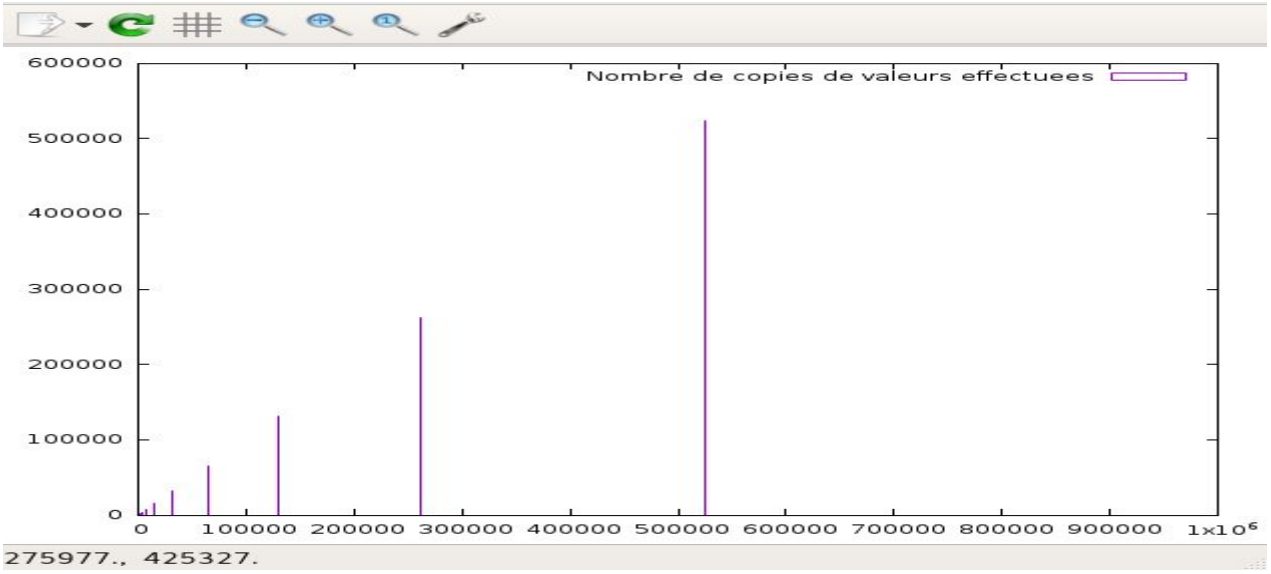
Nous constatons que le coût amorti change , il est élevé au départ puis il diminue au fur à mesure et quand n tend vers l'infinie, il se stabilise.

L'espace inutilisé:



l'espace mémoire inutilisé est différent par rapport à la première expérience, cette espace augmente quand le tableau est pleins (l'espace mémoire inutilisé est nul).

Nombre de copies :



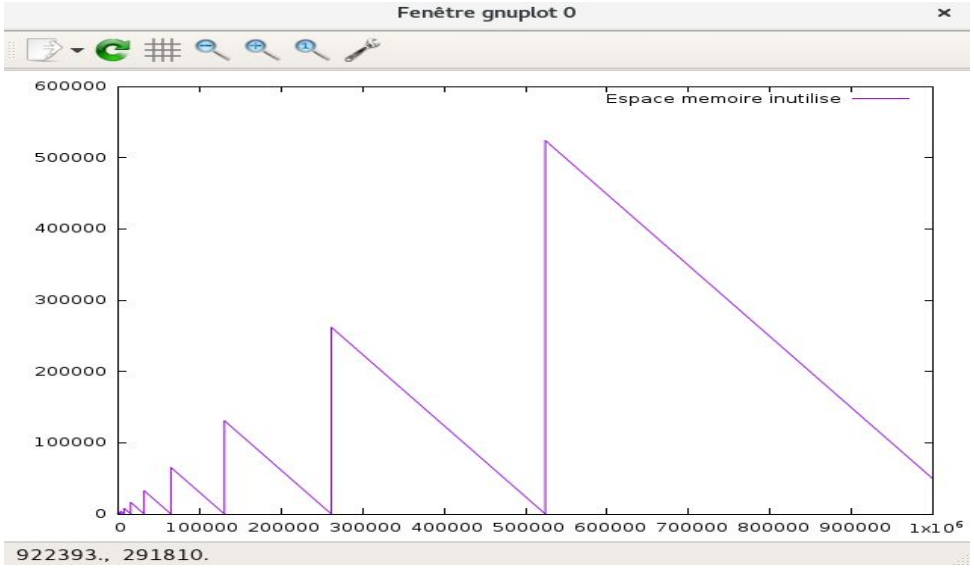
En ce qui concerne le nombre de copies,on constate que le nombre de copies augmente à chaque fois qu'on insère des éléments dans la table.

5) Résultats de variation de α

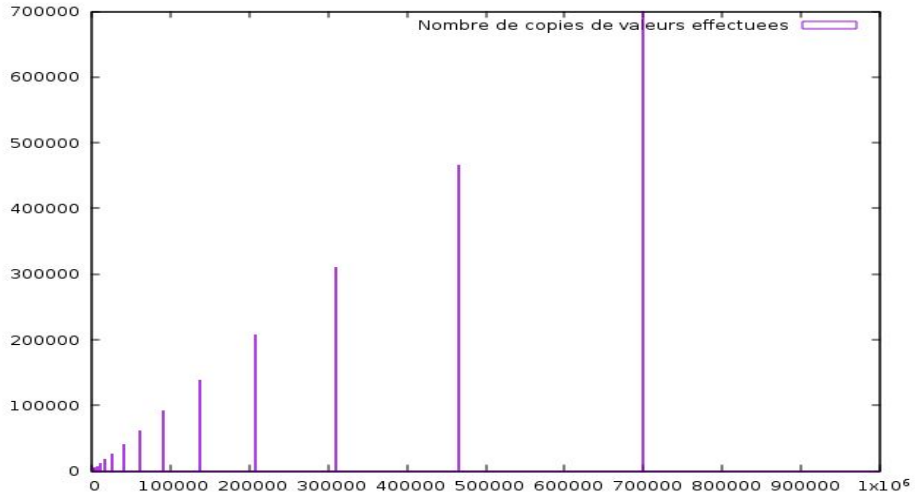
α=1.1

```java
private void enlarge_capacity(){
  data = java.util.Arrays.copyOf(data, capacity*2);
  capacity *= 1.1;
}
```

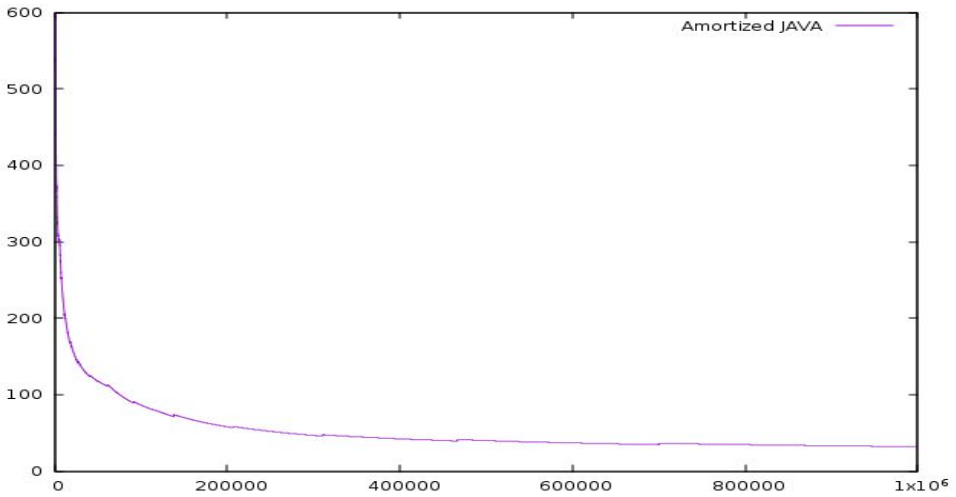
espace mémoire non utilisé



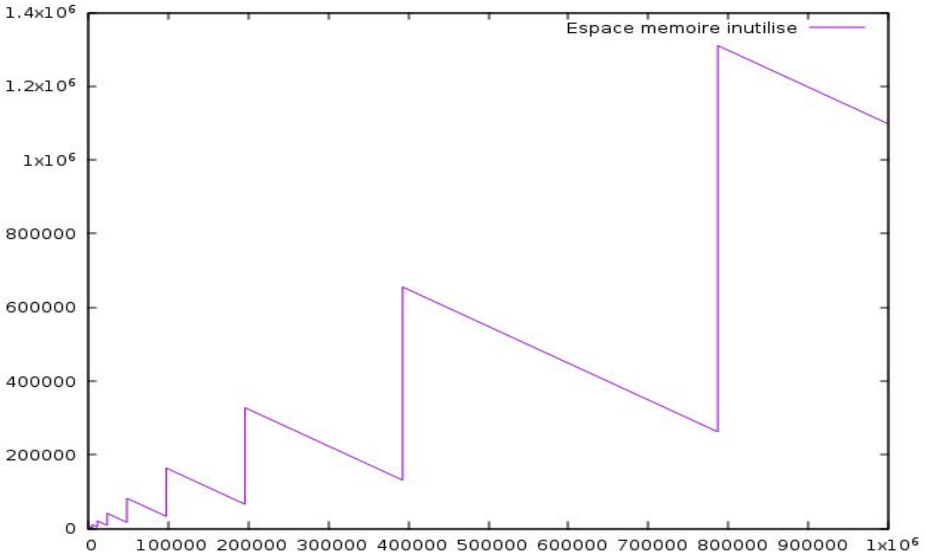
nombre de copies:



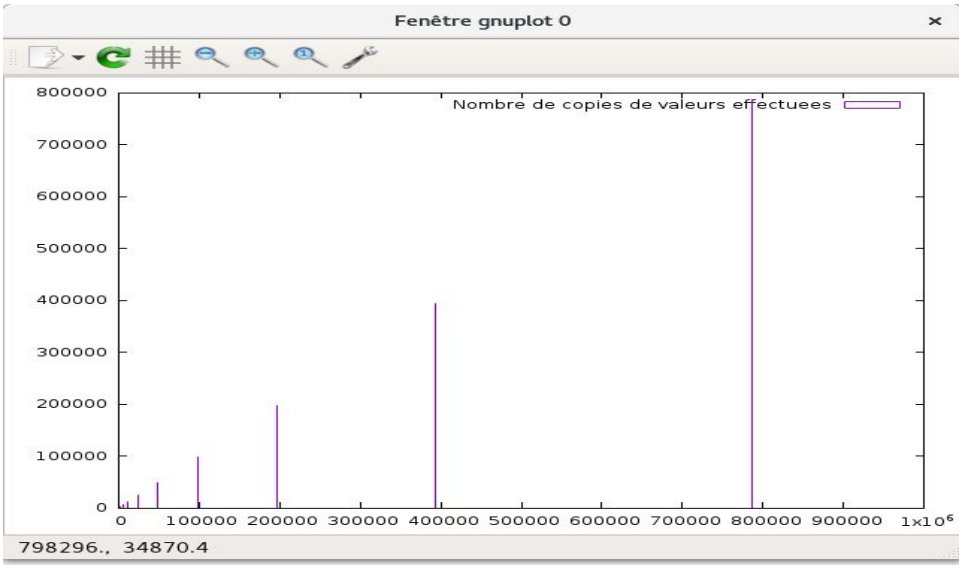
coût amorti:

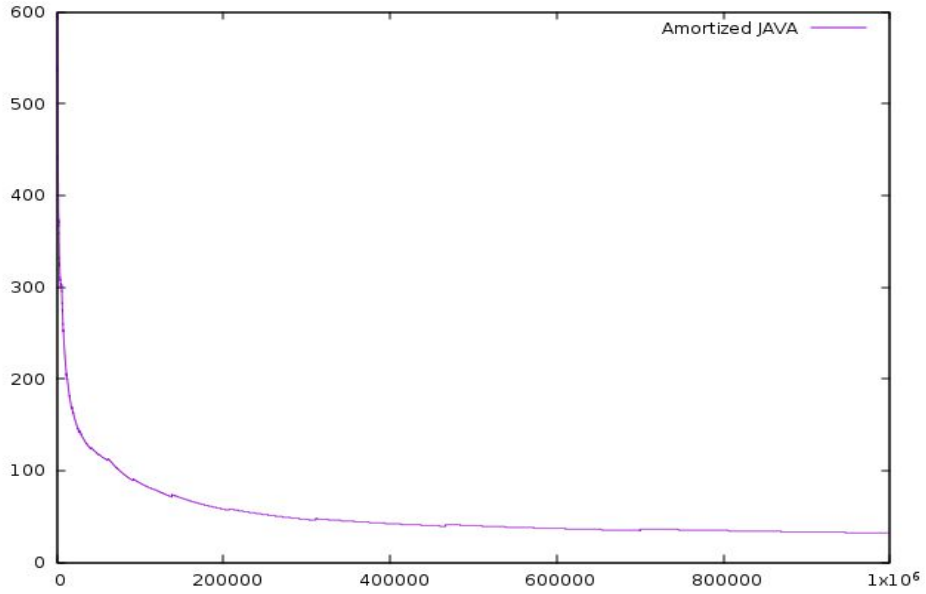


pour α=1.5

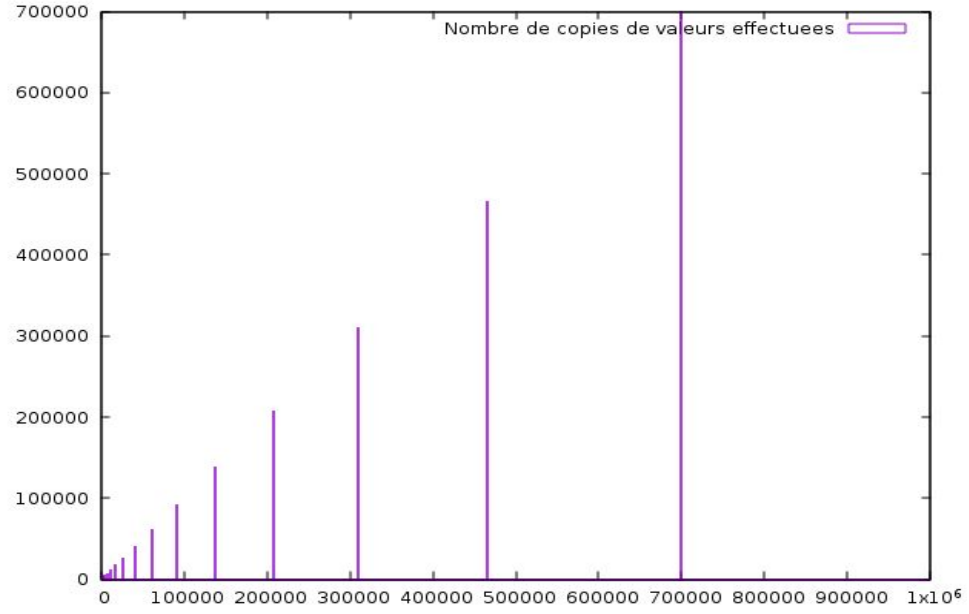


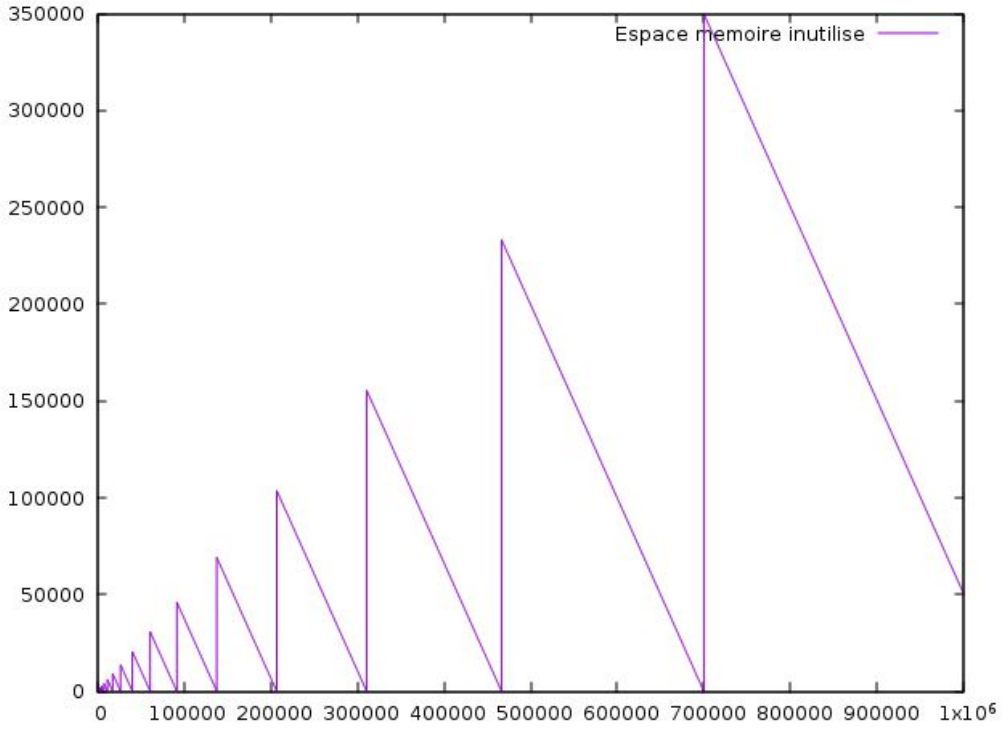
nombre de copies


Fenêtre gnuplot 0
Nombre de copies de valeurs effectuees
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶
798296., 34870.4


Amortized JAVA
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10⁶

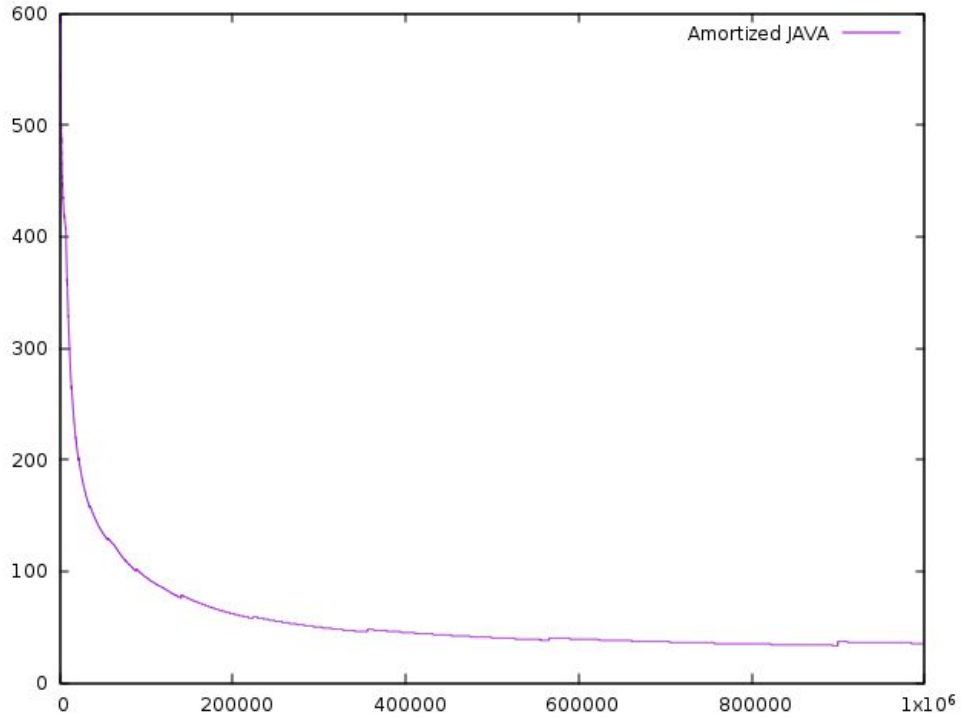
pour α=1.0



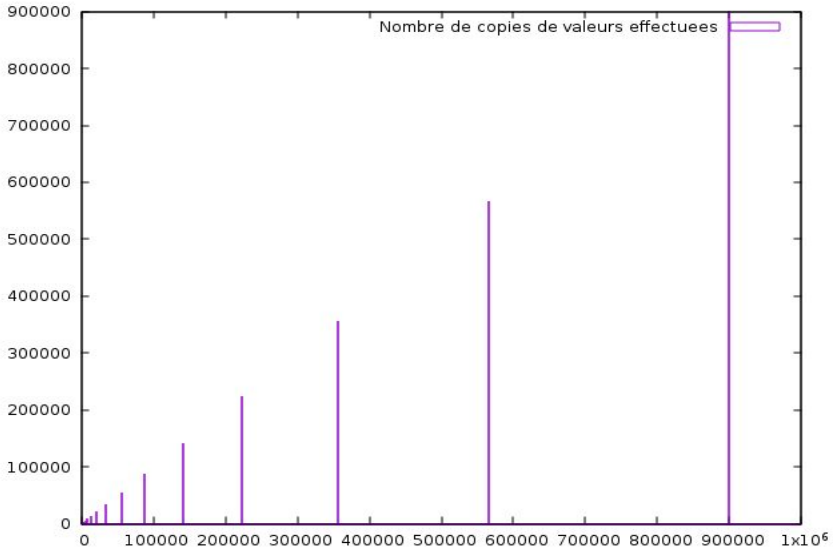


pour α=1.59 :

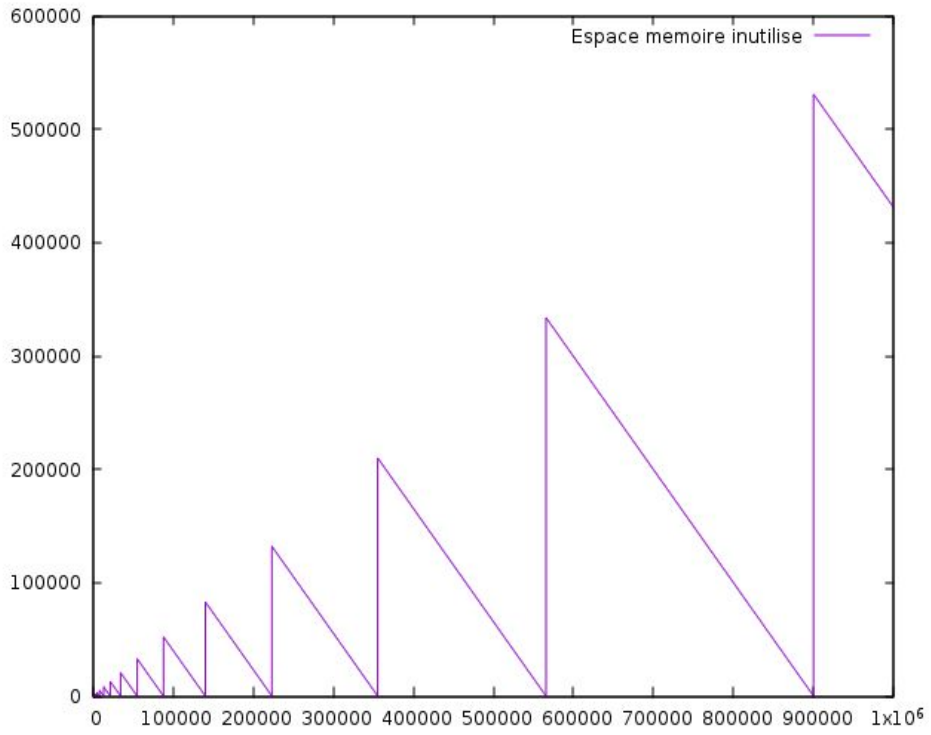
Coût amorti



nombre de copies



espace inutilisé



Conclusion

Quand α est petit alors on a beaucoup de copies qui se font c'est ce qui n'est pas efficace en temps d'exécution.Par contre l'espace gaspillé est petit.
quand α est grand on a pas beaucoup de copies donc c'est efficace en temps mais on gaspille beaucoup de mémoire.

6)

Nous avons modifié la fonction enlarge_capacity() :

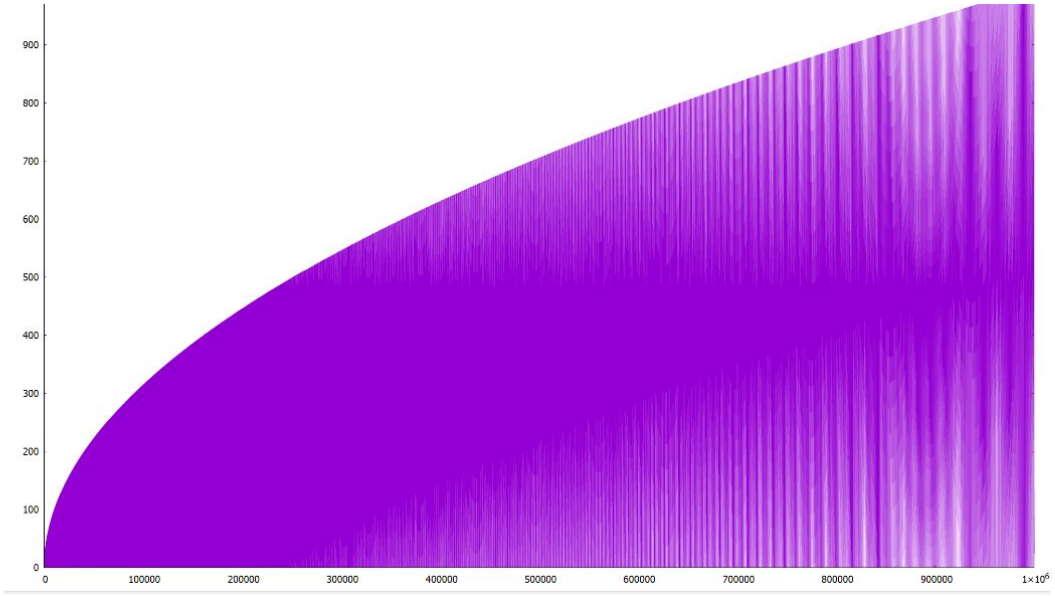
```
/**
   Cette fonction augmente la capacité du tableau.
*/

private void enlarge_capacity(){
    data = java.util.Arrays.copyOf(data, capacity*2);
    //   capacity *= 1.0;
    capacity = capacity + (int) Math.sqrt(capacity);

}
```

le résultat de compilation est le suivant :

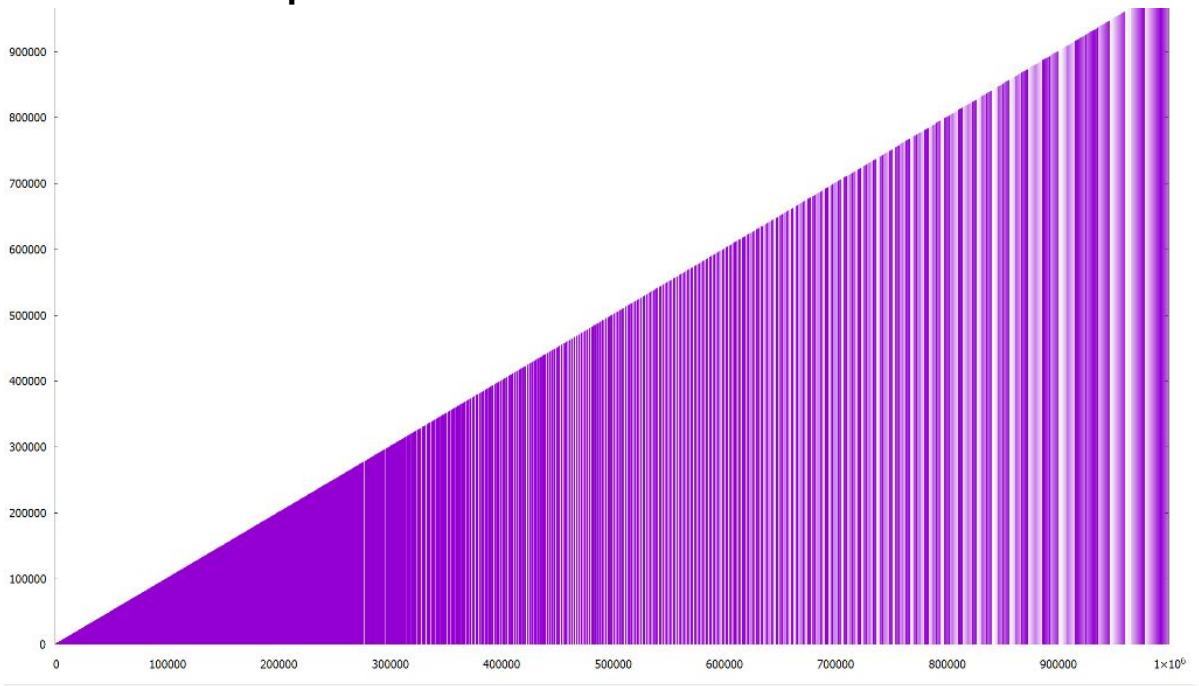
```
pedro@pedro-Inspiron-3542:~/Téléchargements/SDA/s
Total cost : 4041096494
Average cost : 4041.096494
Variance :748640879004450339.126180907964
Standard deviation :865240359.093616485595703125
```

Nous obtenons par la suite :

l’espace mémoire inutilisé :



le nombre de copies :



Constat

On remarque, que l’espace mémoire non-utilisée varie très rapidement de façon exponentielle.Le nombre de copie augmente aussi ce qui va être très coûteux en temps .On doit allouer un nouveau tableau plusieurs fois et copier les éléments de l’ancien dans la nouvelle table .