Tp1

(1) Considérons que l'on ne double pas la taille de la table quand celle-ci est pleine, mais qu'on en multiplie la taille par un facteur α ≥1, donner une définition pour la fonction potentielle dans ce cas.

Φ c'est la fonction potentielle qui s'applique sur les structures de et qui renvoie un nombre réel positif.

αi = (nombre d'éléments/taille du tableau) le facteur de remplissage de la table après la ième opération .

$\Phi(i) = X*x_i - Y*t_i$.

Après :

$X*x_i - Y*t_i = 0$, $X*x_i - Y*\alpha*x_i = 0$, $X = Y*\alpha$.

Avant :

$X*x_i - Y*t_i = x_i$, $X*x_i - Y*x_i = x_i$, $X = 1+Y$.

donc $Y = 1/\alpha - 1$ et $X = \alpha / \alpha - 1$.

Alors : $\Phi(i) = (\alpha / \alpha - 1) x_i - (1/\alpha - 1) t_i$.

pour $\alpha = 2$, $\Phi(i) = 2 * x_i - t_i$.

(2) Donner le coût amorti de l'opération Insérer-Table en fonction de α.

$\hat{} ci = ci + \Phi i - \Phi i-1$.

Cas 1 : pas d'extension

$\hat{} ci = ci + (\alpha / \alpha - 1) x_i - (1/\alpha - 1) t_i - (\alpha / \alpha - 1) x_{i-1} - (1/\alpha - 1) t_{i-1}$

$\hat{} ci = 1 + (\alpha / (\alpha - 1)) x_i - (1/(\alpha - 1)) t_i - (\alpha / (\alpha - 1)) x_i - 1 + (\alpha / (\alpha - 1)) + (1/(\alpha - 1)) t_i$

$\hat{} ci = (\alpha / (\alpha - 1))$

$\hat{} ci = O(\alpha)$, alors $\hat{} ci = O(1)$.

Cas: avec l'extension

$\hat{} ci = ci + \Phi i - \Phi i-1$.

$\hat{} ci = x_i + (\alpha / \alpha - 1) 2*x_i - (1/\alpha - 1) t_i - (\alpha / \alpha - 1)2* x_{i-1} - (1/\alpha - 1) t_{i-1}$

$\hat{} ci = x_i + (\alpha / \alpha - 1)2* x_i - (1/\alpha - 1)2* x_i +2 - (\alpha / \alpha - 1)2* x_i +2 - (1/\alpha - 1)x_i +1$

$\hat{} ci = x_i + (\alpha / \alpha - 1) x_i - (1/\alpha - 1) t_i - (\alpha / \alpha - 1)2 x_i +1 - (1/\alpha - 1)x_i +1$

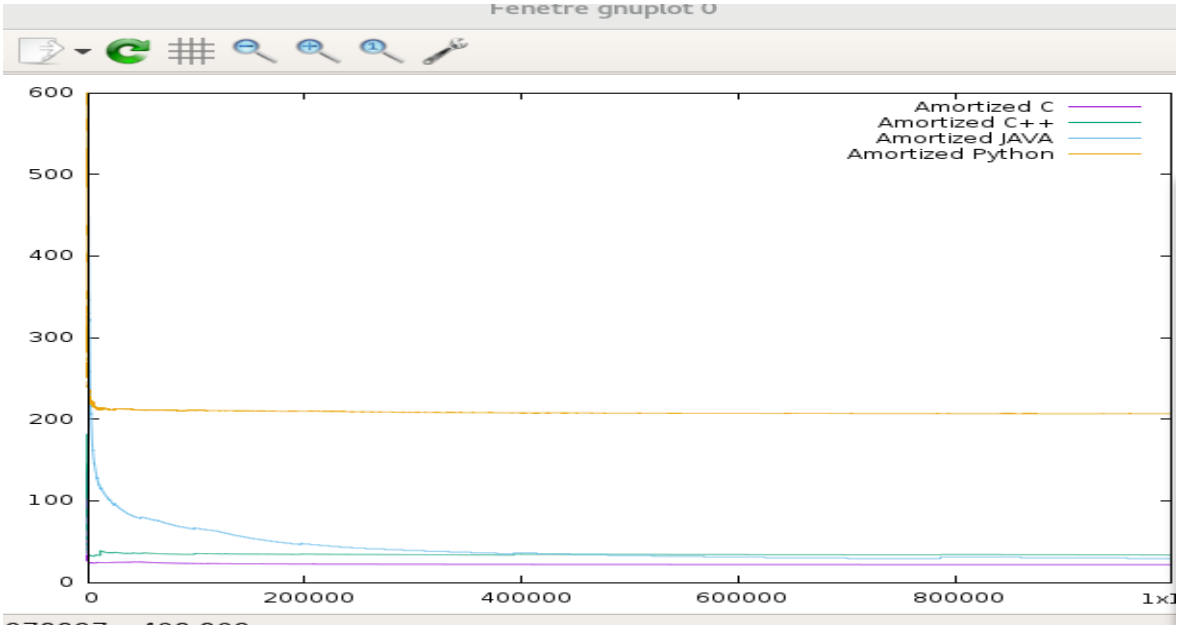
ˆ ci = O(α)

ˆ ci = O(1).

Nous avons donc montrer que dans les deux cas le coût amorti est égal à 1 + (α / (α - 1)).

On conclut que le coût réel total de n opérations insérer est majoré par O(α*n) = O(n).

(3)



a) Lors de l'exécution des programmes, quelle est le morceau de code qui semble prendre le plus de temps à s'exécuter. Quelle est la complexité de ces fonctions ? Pourquoi ce morceau de code est-il plus lent que le reste ?

On remarque qu il y a des morceaux de code qui prennent plus du temps à s'exécuter et cela revient à la fonction append qui prend plus du temps.

Cette fonction permet d'ajouter un élément dans un tableau mais lorsque ce dernier est plein on doit l'élargir sa taille puis recopier tous les éléments de ce tableau dans le nouveau et cela prend du temps surtout si on a beaucoup d'éléments à recopier.

(b) Observez maintenant le coût amorti en temps dans les différents langages. A quel moment le coût amorti augmente-t-il ? Pourquoi ? (La réponse dépend du langage)

Le cout amorti augmente à chaque fois qu il faut élargir le tableau car on construit à chaque fois un nouveau tableau plus grand que l'ancien et cela dans

tous les graphiques (quel que soit le langage par lequel on a codé notre fonction append).

Par exemple en Java on multiplie la taille mémoire par le nombre d'or qui est égal à 1,6.

En C++ on multiplie la taille mémoire par deux.

On remarque que pour certains langages (les langages interprétés comme Java et Python) y a des augmentations en dehors des allocations mémoires car les langages interprétés sont plus lents à s'exécuter.

Dans ces langages, le code source est interprété, par un logiciel qu'on appelle *interpréteur*. Celui-ci va utiliser le code source et les données d'entrée pour calculer les données de sortie.

L'interprétation du code source est un processus « pas à pas » : l'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire par la suite.

En java ces augmentations sont dues aussi au lancement du Garbage Collector qui consiste à identifier puis à libérer les zones mémoires inutilisées ce qui ralenti l'exécution des instructions.

Java s'accélère lorsqu'il compile les bouts de code réutilisés au lieu de les interpréter.

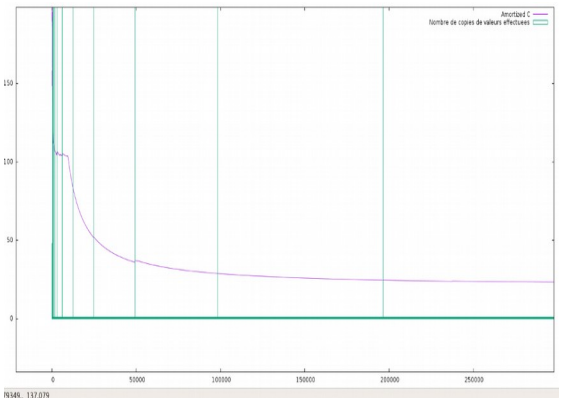
Le langage C ne possède pas ce mécanisme de ramasse-miettes, la mémoire allouée dynamiquement pour un programme doit donc être explicitement libérée grâce à la fonction free.

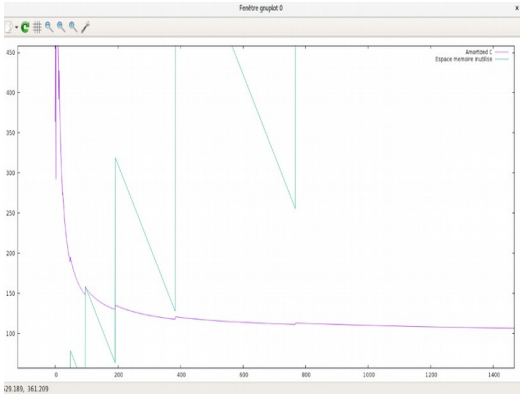
Par contre y a une indépendance de l'espace d'adressage entre les processus donc les processus ne peuvent pas accéder à la même adresse mémoire donc le C est plus performant en temps d'exécution car son coût amorti diminue.

c) Affichez le nombre de copies effectués par chaque opération (C/C++ ou Java), puis le coût amorti. Que remarque-t-on ? Quelle différence y a-t-il avec le temps réel mesuré ?

On remarque au moment où on a besoin d'élargir un tableau plein le coût amorti de chaque opération augmente car à chaque fois il faut construire un nouveau tableau plus grand que l'ancien et recopier tous ces éléments dans le nouveau.

Si on veut gagner en mémoire on fait le minimum de copies donc on minimise la taille du tableau.





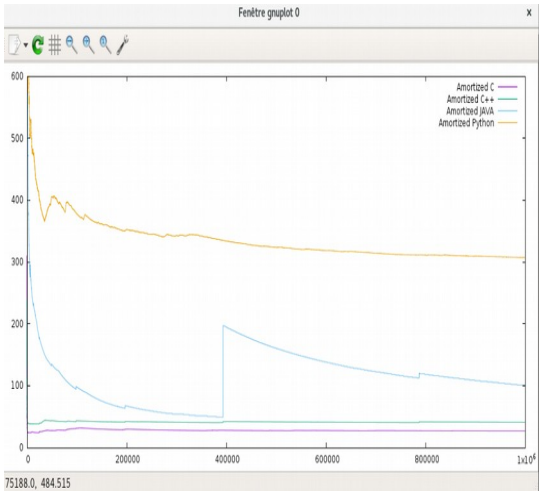
Si on veut minimiser le temps d'exécution sans regarder au gaspillage de la mémoire on réserve toute la mémoire comme ça on n'aura pas besoin de doubler la taille du tableau ni de recopier les éléments ceci est très efficace mais on perd de la mémoire en revanche.

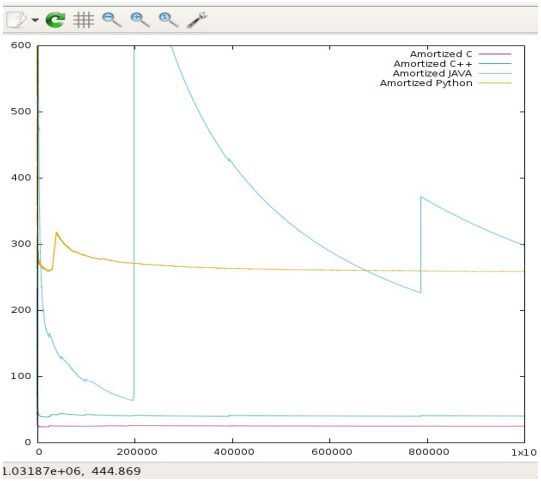
Et si on veut on gagner en mémoire alors on fait beaucoup de copies mais on perd en temps. Pour cela on doit trouver un meilleur alpha qui fera en sorte de ne pas perdre beaucoup de temps ou beaucoup de mémoire.

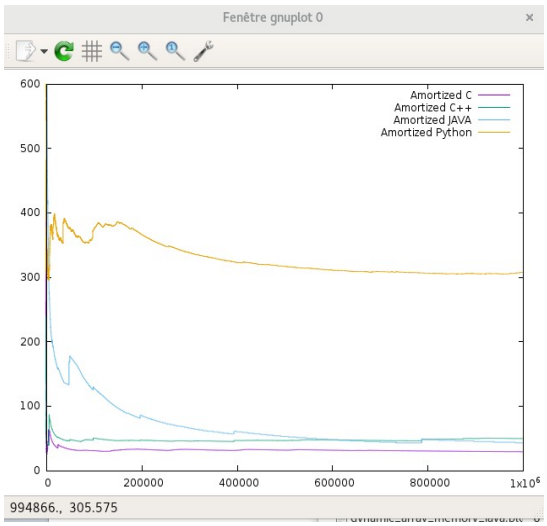
En java par exemple si on multiplie notre tableau par un alpha =1,5 qui est inférieur au nombre d'or (1,6) cela ne va pas influencer sue la complexité amortie mais ça garanti que le processeur n'a pas besoin de réserver de nouvelles pages mémoires car on peut utiliser l'espace mémoire déjà réservé. Donc on maximise l'utilisation de la cache des processeurs et donc et cela n'a pas de relation avec la RAM qui permet d'utiliser la cache d'une manière efficace, dans ce cas-là la pire valeur va être 2 ce qui fait pour chaque nouvelle réservation le processeur doit allouer de nouvelles pages mémoires.

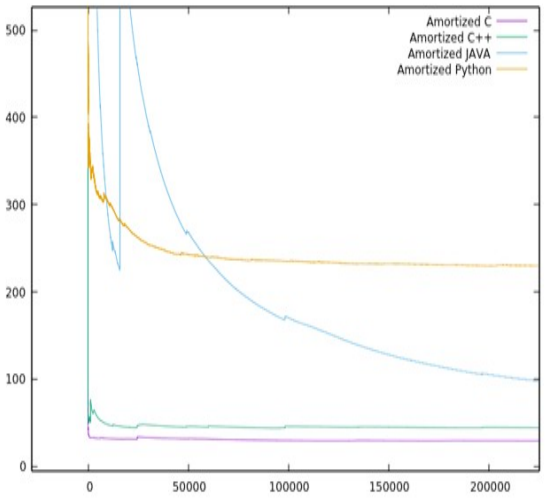
Quand on utilise des puissances de 2 lors de la réservation mémoire on peut réutiliser les données dans la cache.

d) Recommencez plusieurs fois l'expérience avec les différents langages. Qu'est-ce qui change d'une expérience à une autre ? Qu'est ce qui ne change pas ?









On remarque que le temps d'exécution change d'une expérience à une autre parce qu'un processus n'est pas le seul à s'exécuter sur la machine, ce processus peut accéder à une ressource directement lorsqu'il la demande comme il peut attendre que les autres processus finissent leurs **tâches**.

L'avantage qu'offre le langage C est de permette au programmeur de contrôler la gestion de la mémoire donc c'est à l'utilisateur de gérer la mémoire.

En Java et CPP c'est la machine qui s'occupe de la gestion de la mémoire donc l'utilisateur ne peut pas modifier la mémoire.

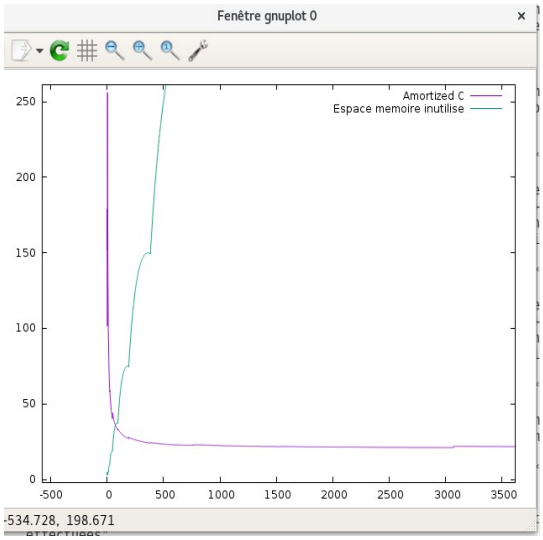
e) La rapidité de certains langages par rapport aux autres :

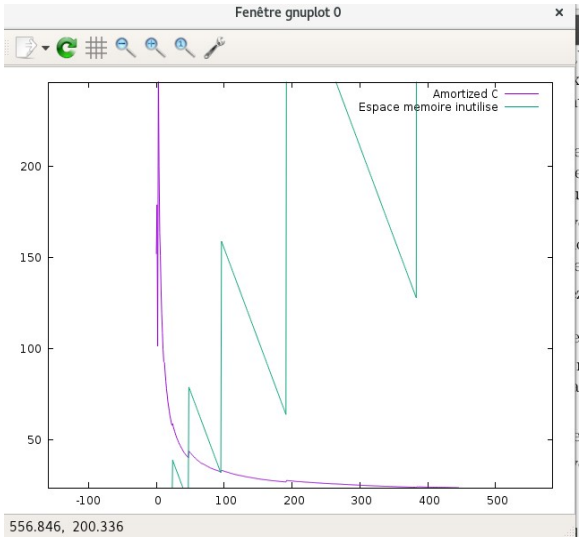
Certains langages sont plus rapides que d'autres dans les expériences réalisées ci-dessus est dû notamment, aux multiples différences qui existent entre ces langages.

Les langages C et CPP sont des langages totalement compilés.

Les langages Java et Python sont des langages interprétés et ont un Garbage collector c'est pour cette raison Java accélère parfois.

f) Observez l'espace mémoire inutilisé au fur et à mesure du programme.





On remarque que l'espace mémoire non-utilisé évolue énormément, il s'élève trop lors de l'agrandissement de la table puis diminue lors de la copie. Mais, cela reste identique pour toutes expériences, ce qui peut poser Problème dans le cas ou notre machine ne dispose pas d'une mémoire suffisante.

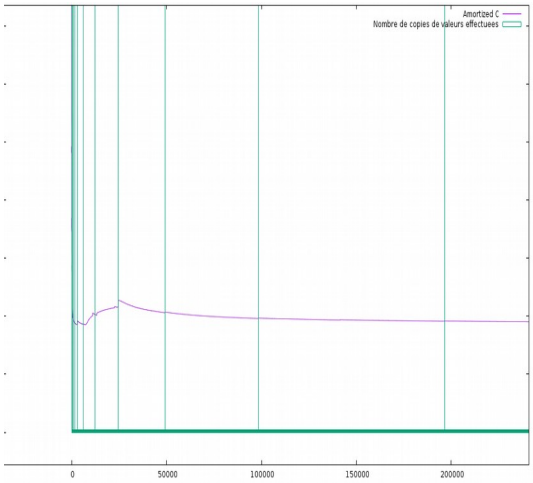
(4)Modifiez la fonction do_we_need_to_enlarge_capacity pour ne se déclencher que lorsque le tableau est plein.

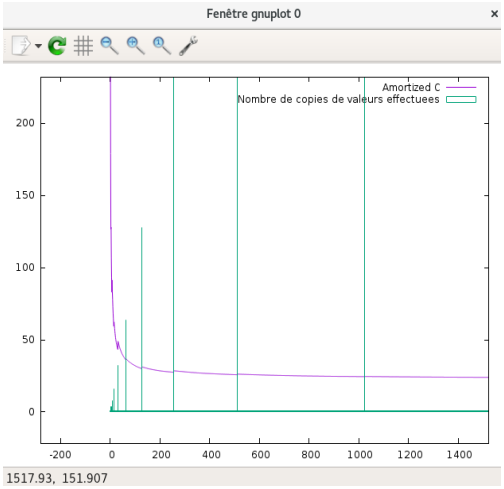
J'ai choisi de programmer en langage.

Les résultats obtenus avant la modification de la fonction **do_we_need to_enelarge_capacity**

Le tableau est donc rempli à 3/4 et l'extension se fait quand il est plain à ¾:

```
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
 return ( a->size >= (a->capacity * 3) /4)? TRUE: FALSE;
}
```

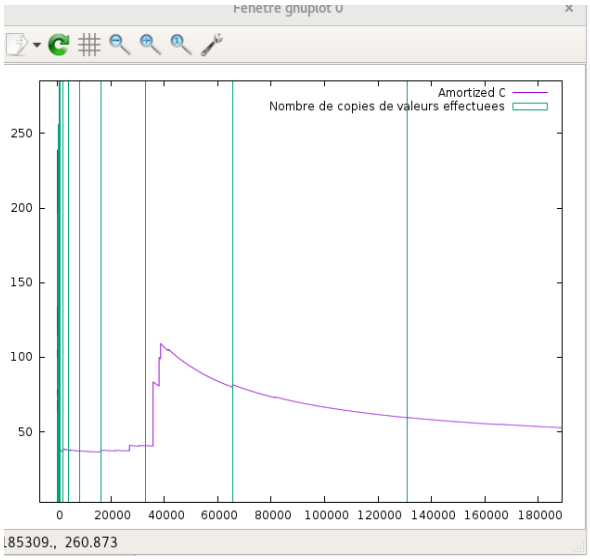


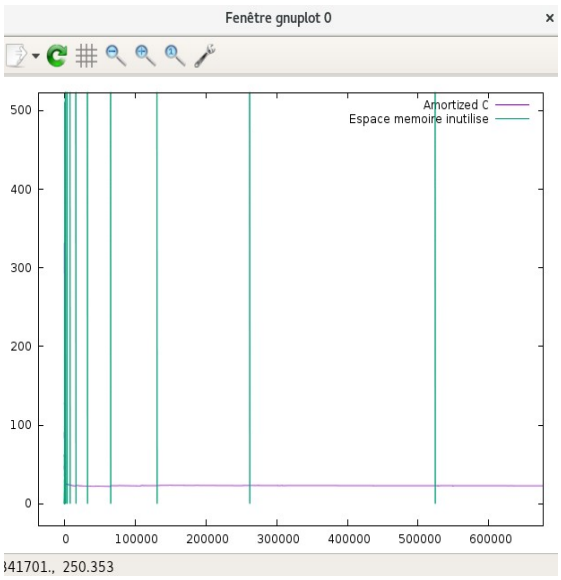


les résultats obtenus après la modification de la fonction **do_we_need to_enelarge_capacity :**

le tableau s'élargit quand il est plein.

```
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
  return ( a->size >= (a->capacity) )? TRUE: FALSE;
}
```



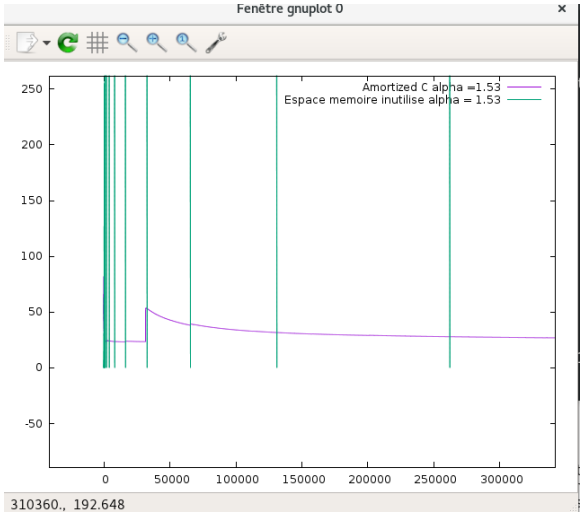


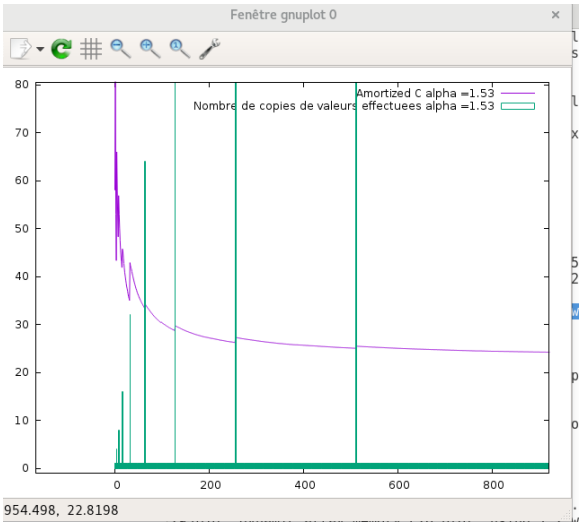
Explication des résultats :

On remarque que le coût amorti augmente plus dans le cas où on élargit notre tableau lorsque ce dernier est plein par rapport à l'élargissement lorsqu'il est rempli à 3/4 uniquement car lorsque le tableau est plein on doit recopier tous ses éléments dans le nouveau tableau ce qui nous permettra du gagner en mémoire.

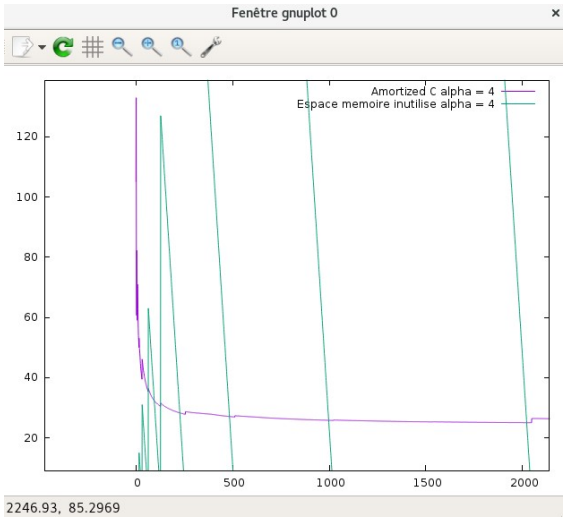
(5) Dans la fonction enlarge_capacity, faites varier le facteur multiplicatif α. Que se passe-t-il ?

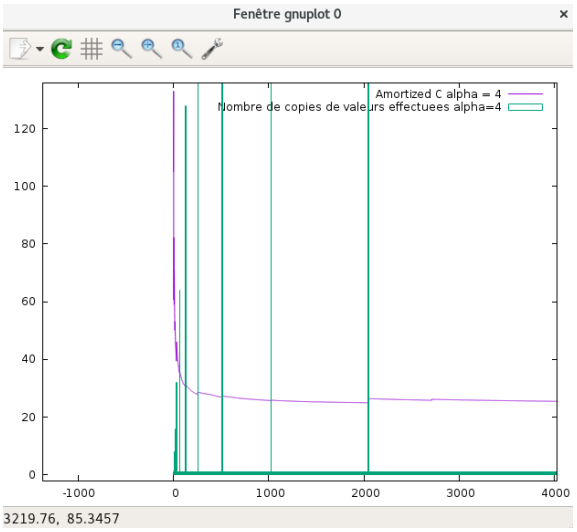
Le cout amorti pour la valeur alpha = 1.53



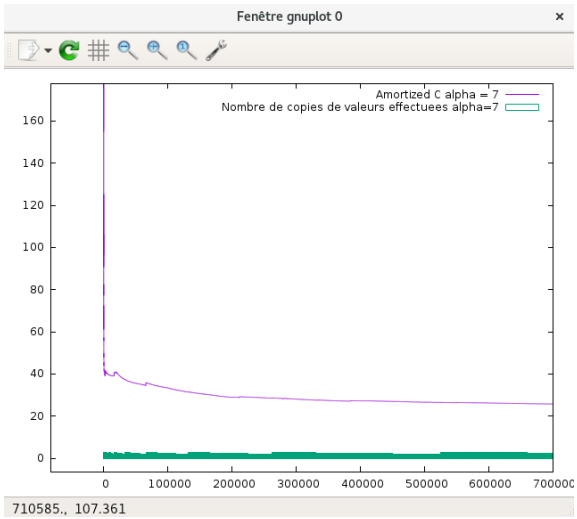


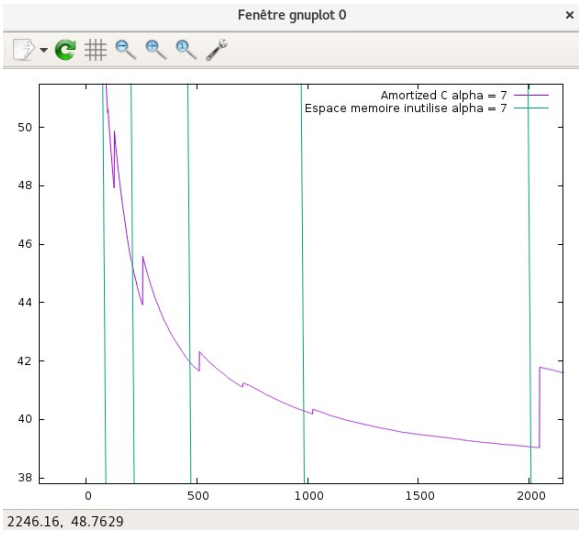
Le cout amorti pour la valeur alpha = 4



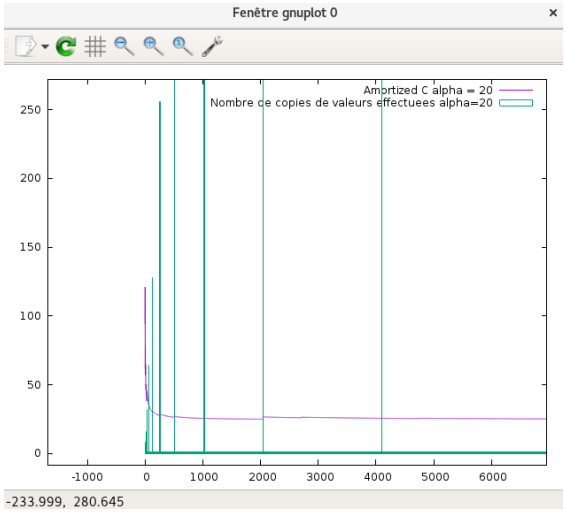


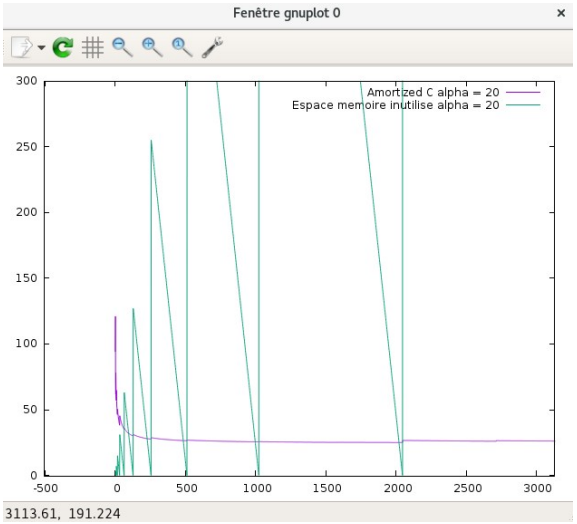
Le cout amorti pour la valeur alpha = 7





Le cout amorti pour la valeur alpha = 20





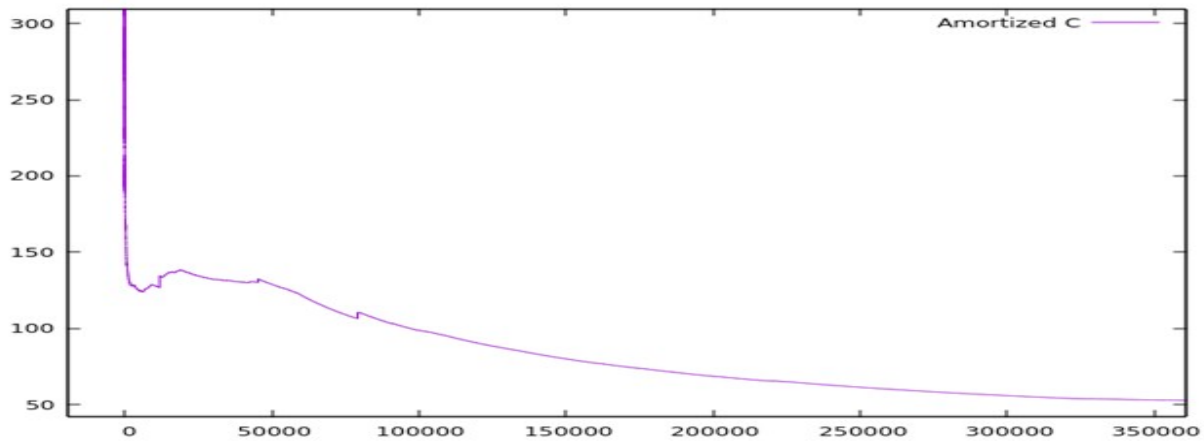
On remarque que lorsque le facteur de remplissage alpha est petit on fait beaucoup de copies et on gagne en mémoire mais ceci coûte en temps d'exécution car il faut recopier à chaque fois les éléments du tableau.

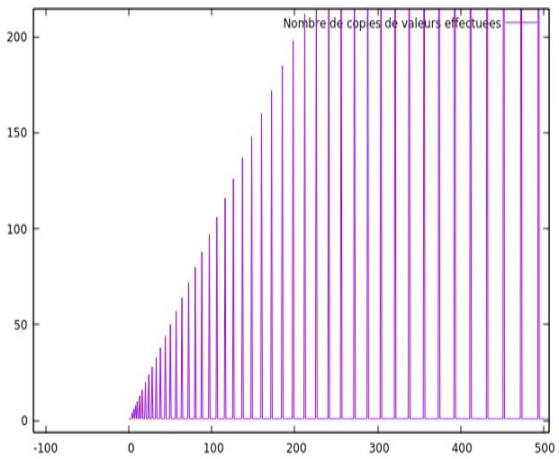
Lorsque le facteur de remplissage alpha est grand on ne fait pas beaucoup de copies donc on perd en termes d'espace mémoire mais on devient efficace en temps d'exécution.

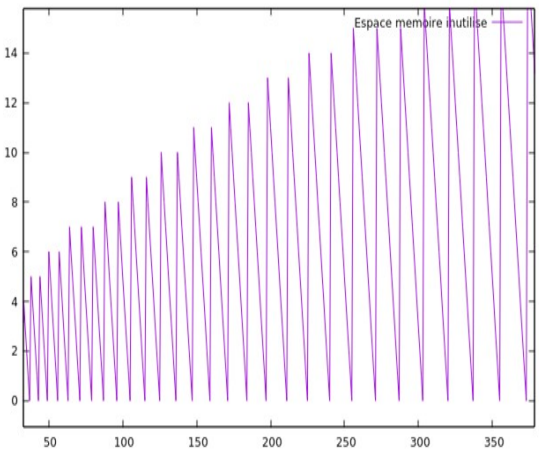
6) Dans la fonction enlarge_capacity, faites varier la capacité n vers une capacité n + √ n. Que se passe-t-il ?

```
Void Arraylist_enlarge_capacity(arraylist * a){
a->capacity *2;
a-> capacity += sqrt(a-> capacity);
a->data = (int *) realloc(a->data, sizeof (int ) * a-> capacity);
}
```

On a modifié la capacité du tableau en n+sqrt (n) et on va observer les résultats obtenus :







On remarque, que l’espace mémoire inutilisé varie très rapidement et atteint des valeurs très élevées dans des intervalles de temps courts, c’est un gain en terme de mémoire.

On remarque aussi, une variation du coût amorti qui augmente notamment, au moment ou l’on fait des copies.

Cette stratégie est intéressante dans le cas où l’on fait uniquement des insertions En revanche son atout est que l’espace mémoire inutilisé augmente dans des durées courtes.

Mais maintenant on a pas vraiment des problèmes de mémoire.