|  |
| --- |
|  |
| Structure de données |
| Tableaux dynamiques |
|  |
| **Lynda MEDJDOUBI** |
| **Sabrina DJENADI** |

|  |
| --- |
|  |

**Sommaire**

**Partie 1 : Repenses aux questions3**

**Question 1 :** fonction du potentiel3

**Question 2 :** coût amorti de l’opération Inserer\_Table4

**Question 3** 5

3. (a) 5

3. (b) 5

3. (c) 8

3. (d) 10

3. (e) 10

3. (f) 11

**Question 4**13

**Question 5**15

Pour alpha = 515

Pour alpha = 1.517

**Question 6**20

**Partie 2 : Conclusion22**

***Tableaux dynamiques***

**Partie 1 : Repenses aux questions**

**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.

On cherche la complexité amortie quand on effectue l’insertion dans un tableau: avec ***α*** >1

n: le nombre d’élément dans le tableau.

t: la taille du tableau.

*α: facteur pour agrandir la taille du tableau*

**Après extension:**

**Φ(i) = 2n -t** on a: 2n=t

**= t-t**

**= 0**

**Φ(i)=** *αn – t = 0 on a: αn=t*

**Avant extension:**

**Φ(i) = 2n -t**  ona: n=t

**= (2-1)-t**

**= t**

**Φ(i)=** *αn – t = (α-1)t*

l’énergie pour copier n = t

si α> 2 Φ(i)trop grand

si α< 2 Φ(i)trop petit

Cette fonction ne marche qu’avec α = 2

On pose: Φ(i)=*xn – yt*

***Après extension:***

***xn – yt = 0***

***xn – y****αn = 0 αn = t*

**Avant extension:**

***xn – yt = n***

***xn – y****n = n*

**1 = 1 + y 1 + y = *αy***

***y = 1/(α-1) x = α/(α – 1)***

***La fonction potentielle dans ce cas est :***

***=>****Φ(i)= (****α/(α – 1))n - (1/(α – 1))t***

***α = 2 =>****Φ(i)= 2n - t*

**2)**

Pour calculer notre coût amorti dans les deux cas on procède de cette façon:

On a

c = c + Φ((i) - Φ((i-1)

Et

Φ(i) = ***α/(α-1)n – 1/((α-1)t***

**1-Quand je ne dois pas étendre le tableau :**

Il n’y a pas d’extension donc la taille du tableau ne change pas, de ce fait on calcule le coût amorti comme suivant:

On a: n = n + 1

t = t - 1

Calculer le coût amorti:

c = 1((***α/(α-1))n – (1/(α-1))t) – ((α/(α-1))n – (1/(α-1))t)***

***= 1 + (α/(α-1))n – (1/(α-1))t – αn/(α-1) + α/(α-1) +t/(α-1)***

**2-Quand je dois étendre le tableau :**

Il y a une extension, c’est à dire la taille du tableau change le coût amorti de cette insertion est de:

Calculer le coût amorti:

c = c + Φ((i) – Φ((i-1) on a: n= n+1

= ***1 + α/(α-1) on a: t = t et n = t - 1***

=Φ(***α)***

***coup n élément***

***Φ(α) = Φ(1)***

= n Φ(***α)***

***= Φ(αn)***

***= Φ(n)***

=> on a ***αn constant***

***le coût amorti de l’opération Inserer-Table en fonction de α est:***

***c = 1 + α/(α-1) avec extension***

***α/(α-1) sans extension***

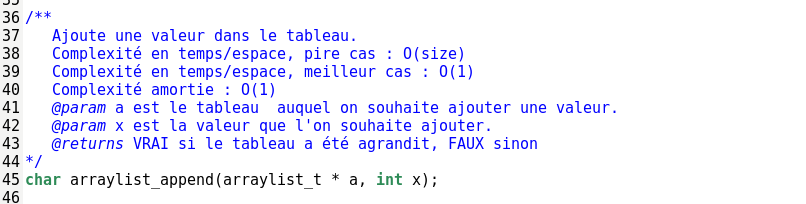
***Pour α = 1 + 1/t***

***=> c = t + 2 avec extension***

***t + 1 sans extension***

**3)**

**(a)** Pour observer le morceau de code qui prend plus de temps à s’exécuter on a regardé le code, pour notre part on a choisi le langage C, on a ainsi accéder au fichier « sda/C/arrayList.h » et on a remarqué que toutes les fonctions ont une complexité en temps/espace constante O(1) sauf la fonction « arrayList\_append » qui ajoute une valeur dans le tableau, en effet quand il y a une extension sa complexité en temps/espace est O(size), où size est la taille du tableau. C’est donc cette fonction qui prend le plus de temps à s’exécuter car l’extension du tableau et la copie de tous les éléments coûte cher.



On remarque aussi quand on lance les différents programme, l’exécution des programmes ainsi que tous les calculs sont assez rapide contrairement à la sauvegarde sur le disque dur qui elle prend beaucoup plus de temps. En effet la sauvegarde sur le disque dur prend plus de temps que la sauvegarde sur la RAM par exemple.

En plus de la complexité, il faut aussi faire le bon choix quant au stockage de nos résultats. Si on souhaite les conserver, on les stocke sur le disque dur mais si on veut que l’exécution des programmes soit plus rapide alors le mieux c’est d’écrire sur la RAM.

**(b)**

Pour observer le coût amorti en temps dans les différents langages, on a affiché cela sous forme de graphe sur **gnuplot**, on a aussi regardé le fichier suivant pour le langage C « sda/plots/dynamic\_array\_time\_c.plot » on a remarqué que le coût amorti augmente au moment de l’allocation de la mémoire**.**

On remarque bien dans l’illustration suivante que les coûts amorti de chaque langage augmente au moment de l’allocation mémoire (qui est représente par les traits verticaux jaunes)

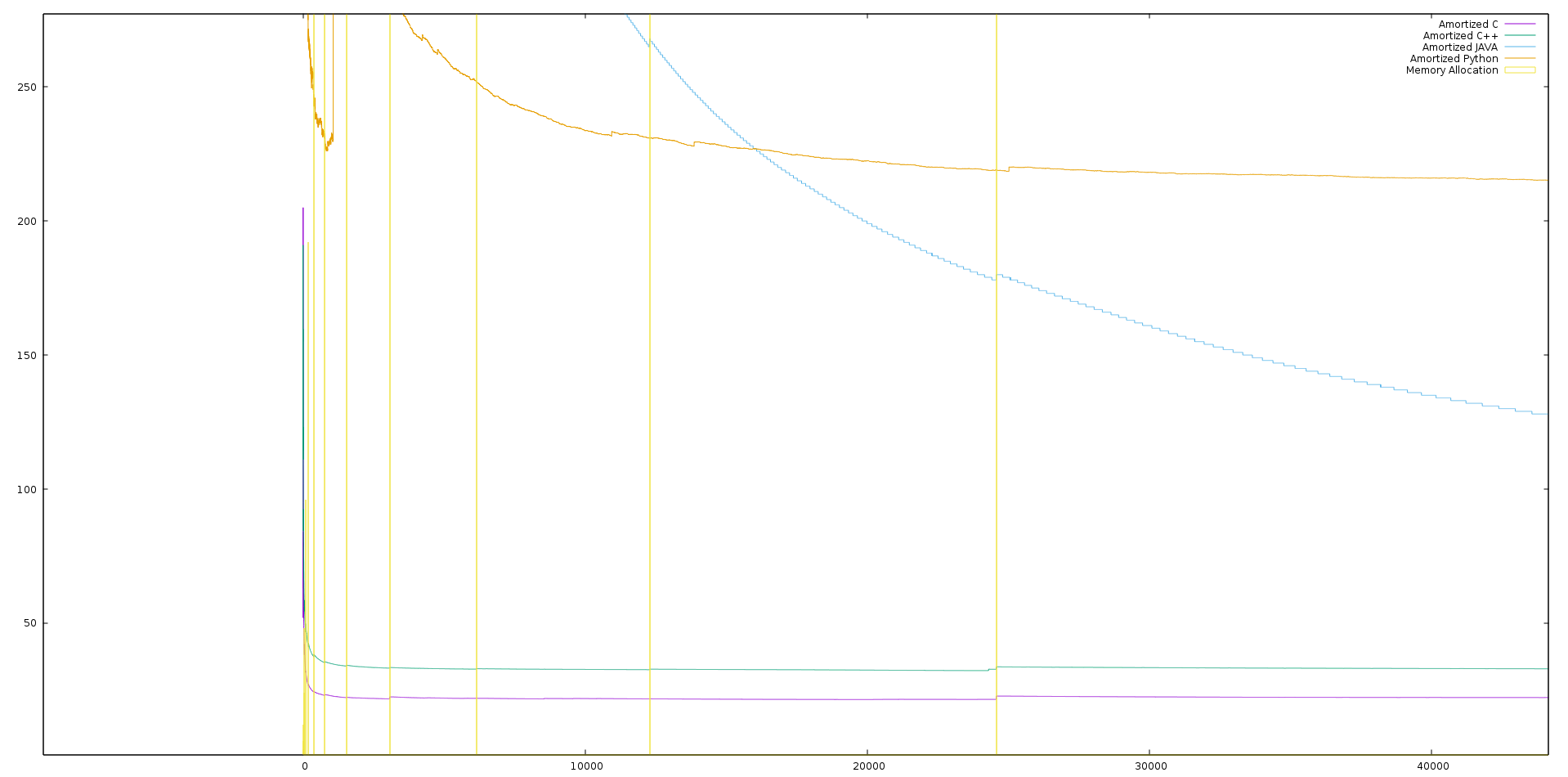
****

Figure  : Coût amorti de tous les langages

**Pour le langage C :** on voit bien que le coût amorti augmente juste au moment de l’allocation mémoire et il est constant dans d’autres cas. Le C est un langage compilé c'est-à-dire qu’il est compilé par le compilateur en un code binaire qui sera directement exécuté par le système d'exploitation qui va calculer les données de sortie, l’exécution du programme est donc plus rapide que dans les autres langages (qui ne sont pas tous compilé) c’est pour cette raison que le coût amorti en langage C est plus petits par rapport aux autres langages

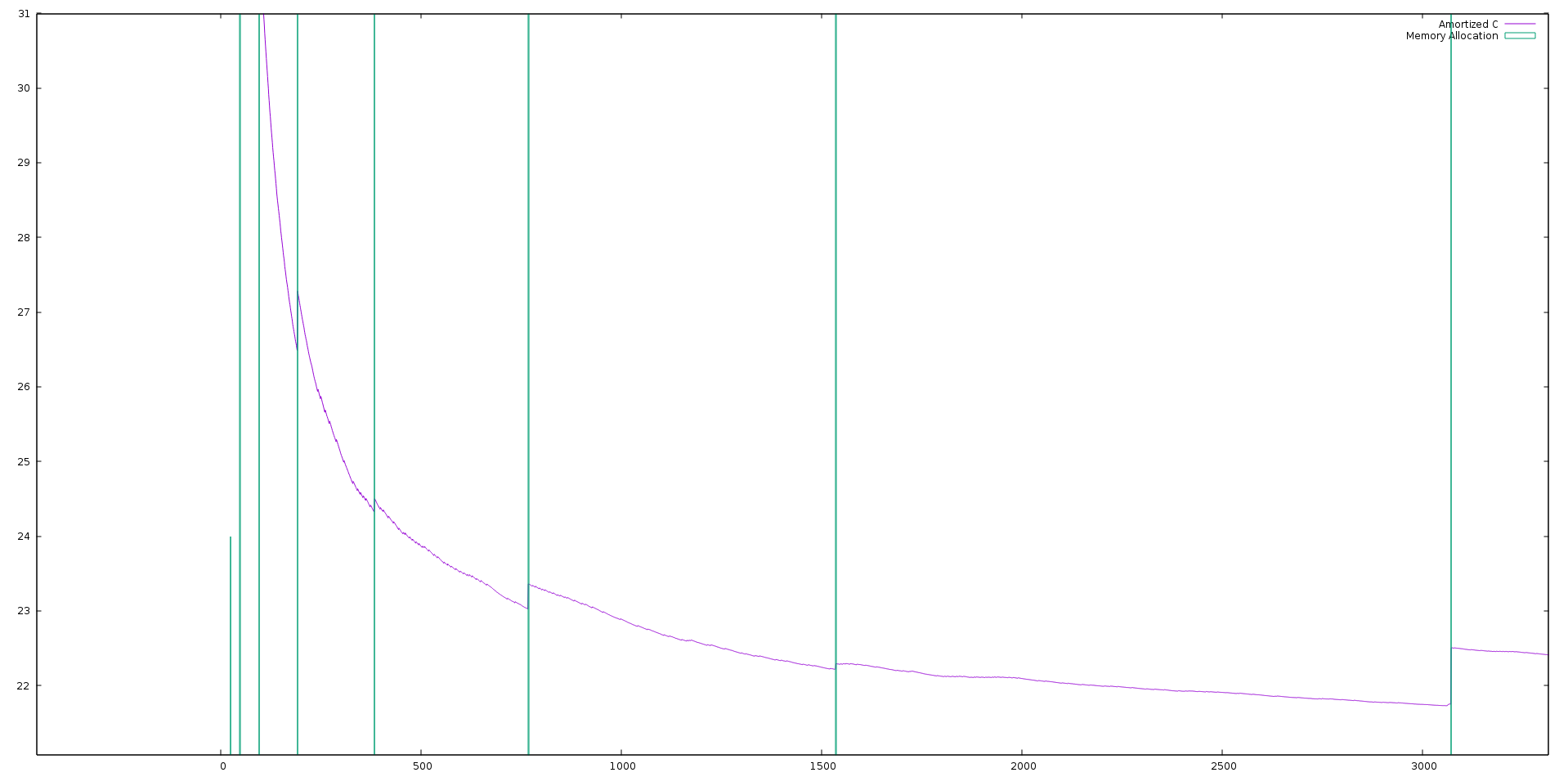
****

Figure : coût amorti pour le langage C

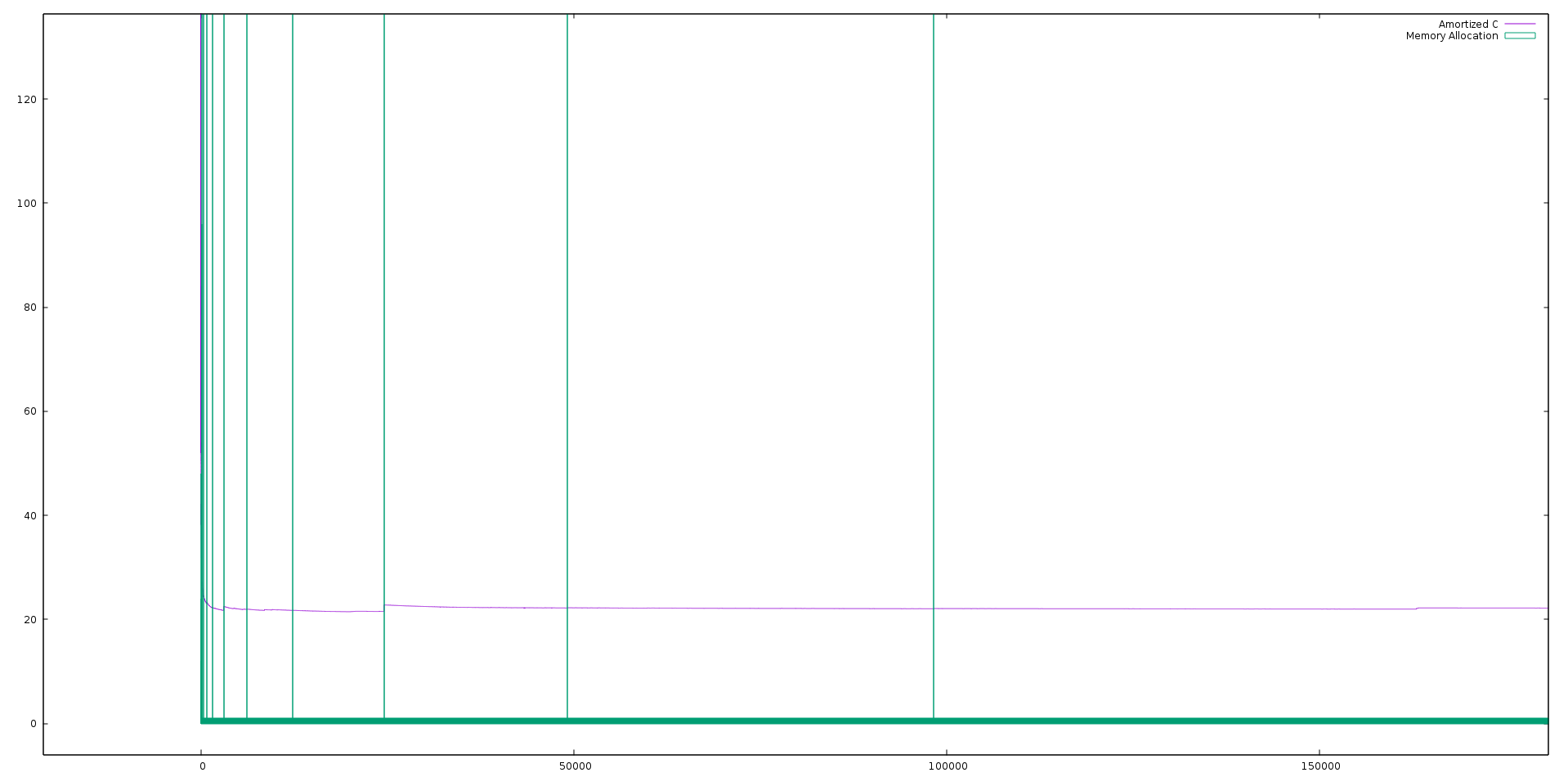
****

Figure  : coût amorti pour le langage C

**Pour le langage Java :** ici on remarque que le coût (temps) amorti augmente au moment de l’allocation mais aussi dans d’autres moments et cela est du au fait que Java soit un langage interprété c'est-à-dire l'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite, en plus du carbage collector qui s’occupe du recyclage de la mémoire préalablement allouée puis inutilisée. Tout ceci fait que le coût augmente par fois.

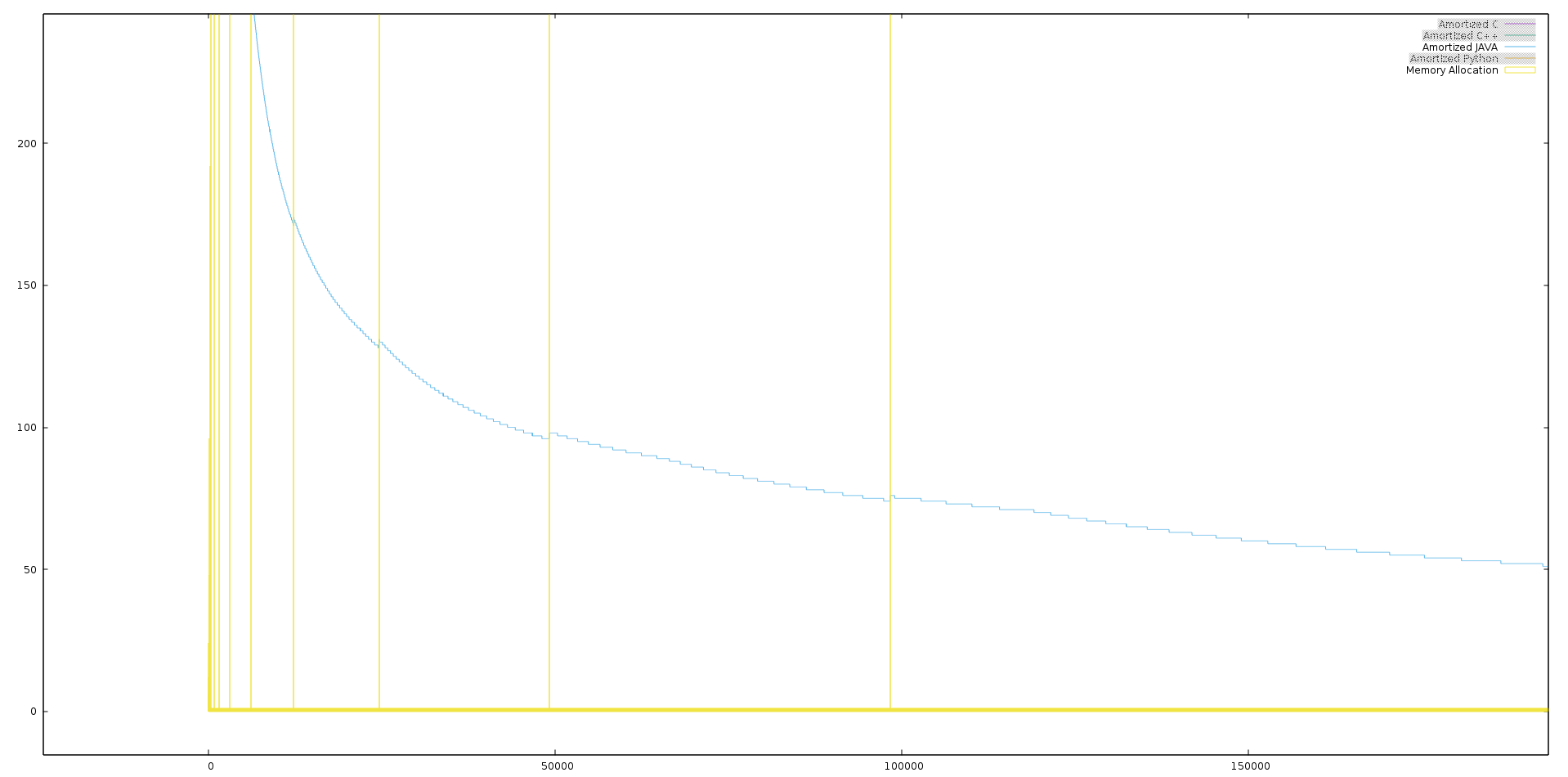
****

Figure  : coût amorti pour le langage JAVA

**Pour python :**

Comme pour le langage Java, python est aussi un langage interprété donc il peut y avoir des augmentations du coût amorti dans d’autres moments que l’allocation de la mémoire.

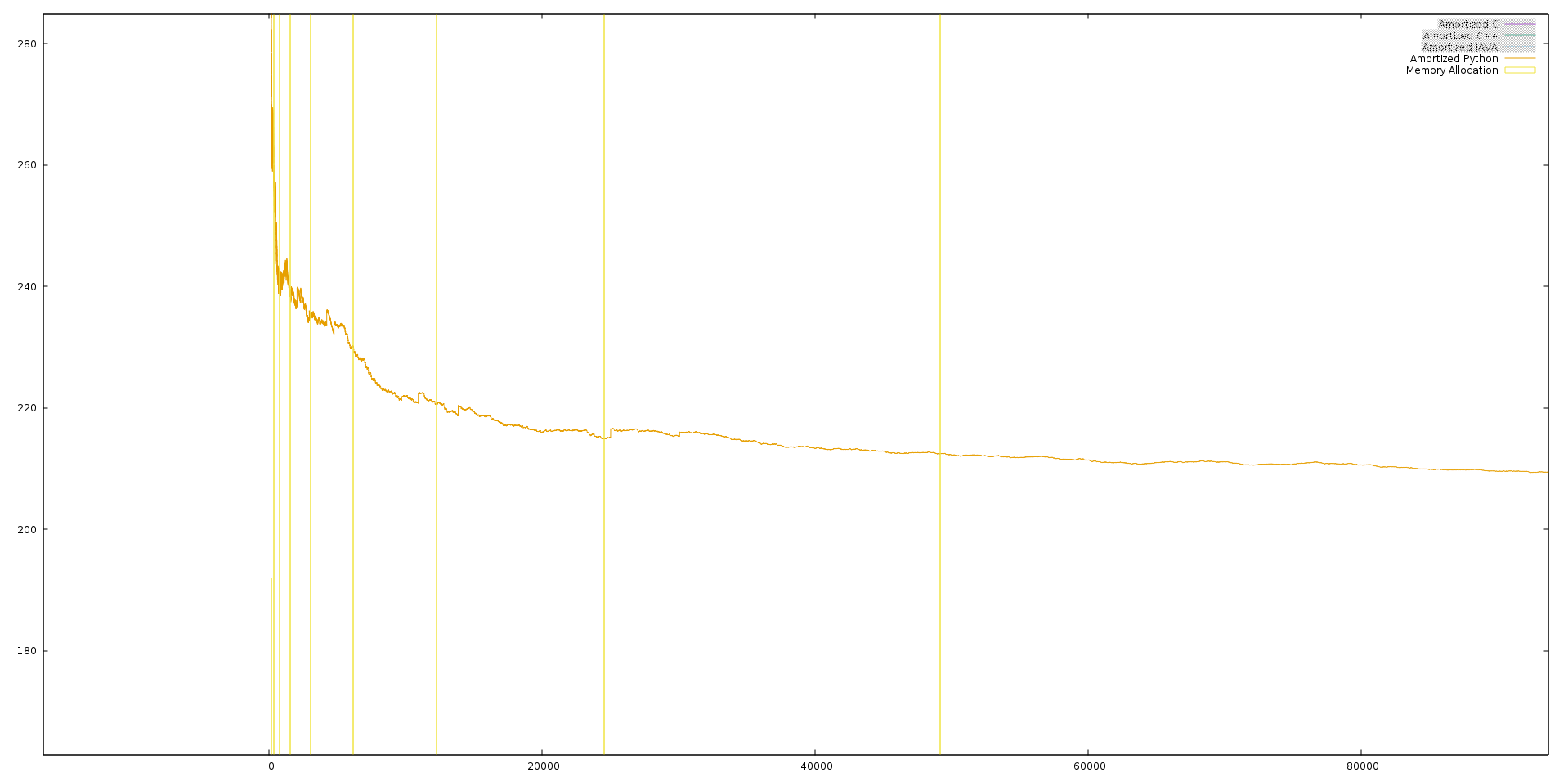


Figure  : Coût amorti pour le langage Python

**(c)**

On remarque ici que le nombre de copies et le coût amorti augmente tous les deux au moment de l’allocation mémoire. En effet quand on alloue de la mémoire on doit recopier tous les éléments du tableau ce qui provoque l’augmentation du nombre de copies et ainsi du coût amorti car cette opération coûte chère.

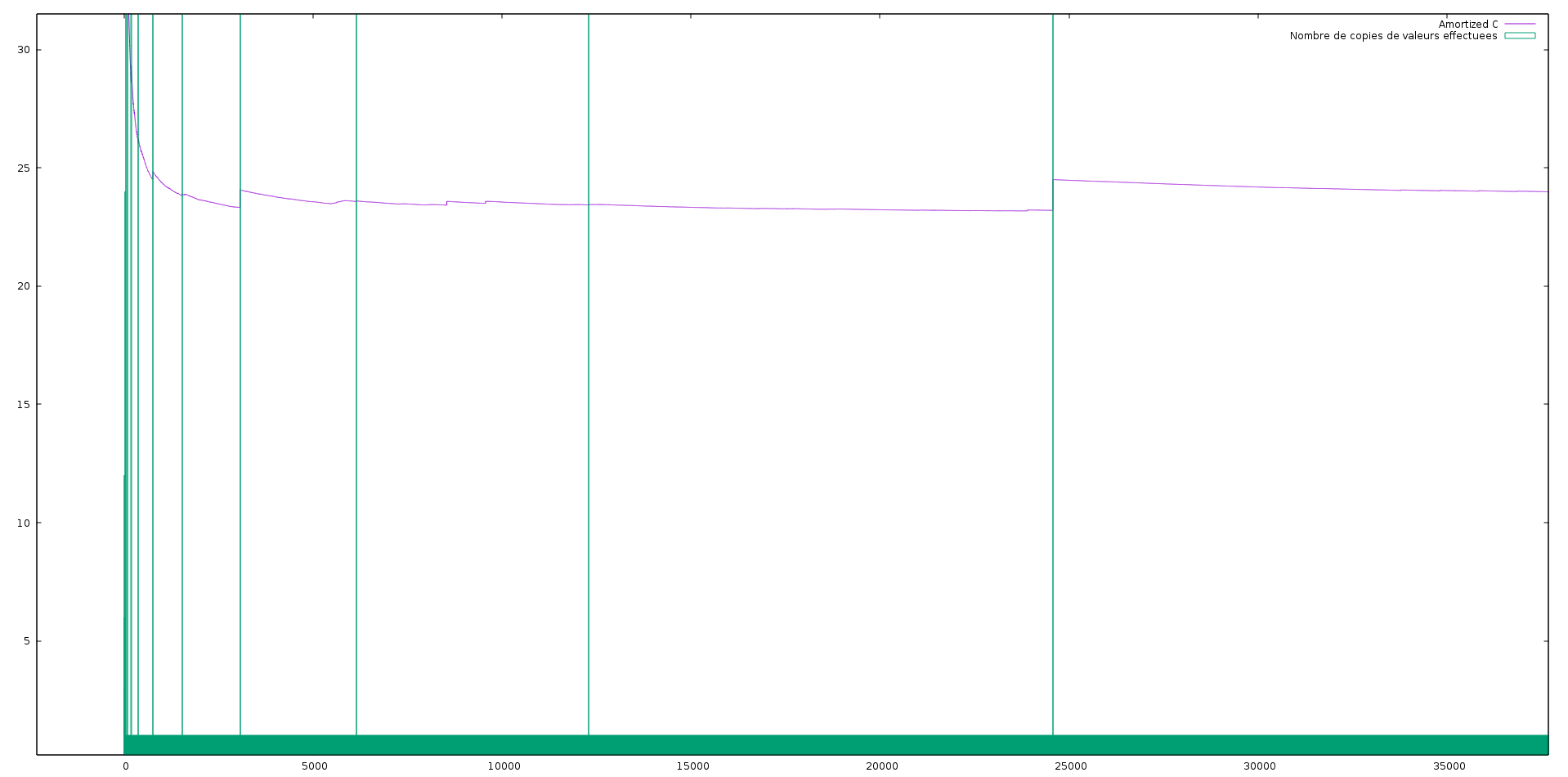


Figure  : Rapport nombre de copies et coût (temps) amorti

**Rapport entre gaspillage mémoire et nombre de copies (efficacité) :**

Si on choisit de faire beaucoup de copies alors on gagne en gaspillage mémoire car plus on fait de copies moins on gaspille de la mémoire (car la taille du tableau est petite). C’est d’ailleurs ce qu’on constate dans le graphe suivant :

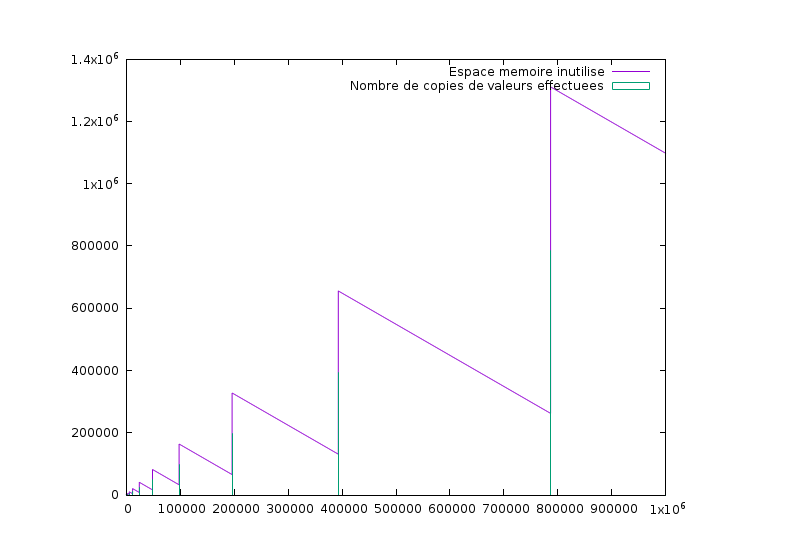


Figure  : Rapport mémoire gaspillée et nombre de copies effectuées

**Rapport gaspillage mémoire et temps d’exécution (temps réel) :**

Plus on fait de copies

Au moment de l’agrandissement de la taille du tableau on gaspille du temps en recopiant tous les éléments du tableau.

Pour la question C on doit regarder le nombre de copie par rapport au cout réel 1:2 (on parle aussi de temps réel) c’est ce qu’on a fait mais aussi le faire pour le cout amorti 1:3 (on peut parler aussi de temps amorti)

**Différence entre le coût amorti et le temps réel mesuré :**

En comparant les deux graphes on remarque que le coût réel est très perturbé même en dehors de l’allocation mémoire, cela est du au fait qu’il dépend de l’état de la structure et donc il change plus souvent et d’une opération à une autre, le coût amorti quant à lui est théoriquement constant car on accumule de l’énergie potentielle à chaque opération d’insertion pour la dépenser dans les opération couteuses comme quand on réalloue de l’espace mémoire et qu’on recopie tous les éléments du tableau.

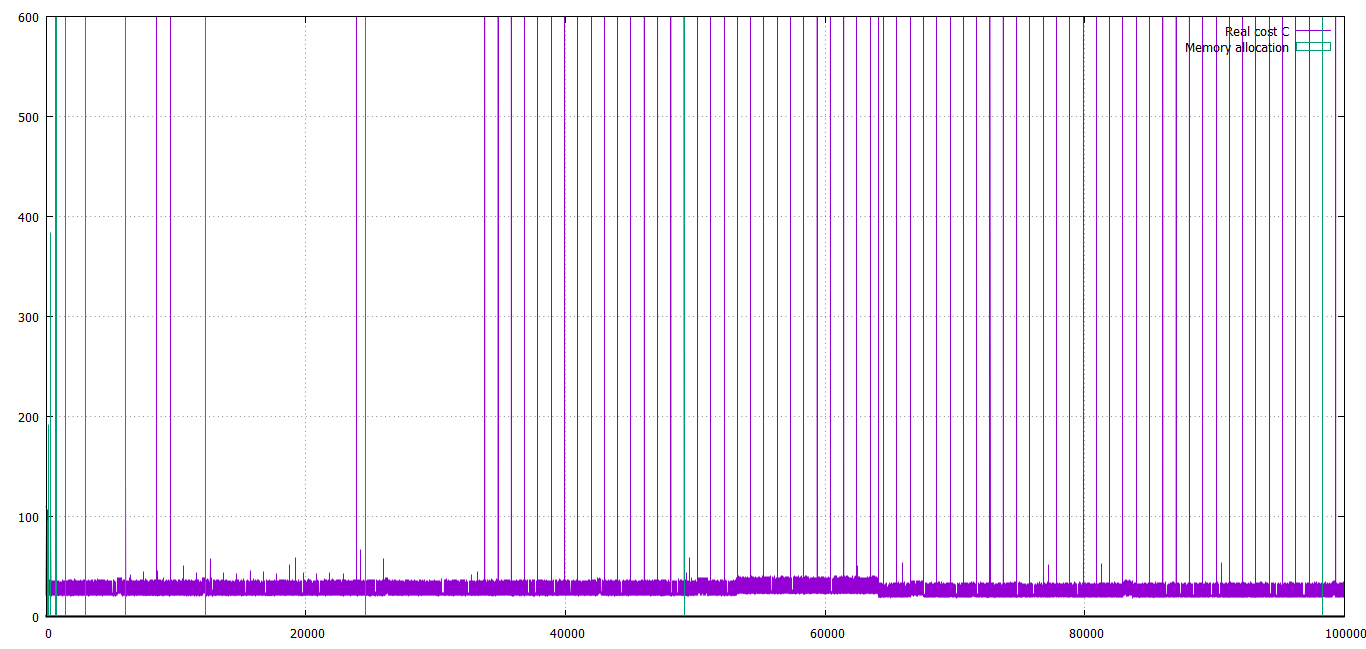
****

Figure  : coût réel et allocation mémoire

**(d)**

En recommençant plusieurs fois cette expérience on remarque que l’augmentation du coût amorti au moment de l’allocation mémoire ne change jamais, par contre il y a des moments où le coût amorti augmente sans raison et cela est dû au fait que plusieurs processus tournent sur l’ordinateur. Mais ce n’est pas tout, car si on regarde la courbe de chaque langage on trouve des choses différentes.

Pour JAVA par exemple on remarque que la courbe augmente par fois même quand il n y a pas d’allocation mémoire et cela est du au fait que c’est un langage interprété en plus du garbage collector, on voit aussi qu’il y a de grosses chutes dans la courbe (des accélérations) et ceci se produit car JAVA compile des bouts de codes qui sont souvent réutilisés au lieu de les interpréter.

**(e)**

JAVA et Python sont plus lents que les langages C et C++ dans cette expérience et on pense que c’est du au fait que les langages compilés sont plus rapide que les langages interprétés.

Sur ce graphe on voit bien que la courbe de python est la plus élevée même par rapport à JAVA qui aussi un langage interprété et cela revient au fait que python soit un langage interprété via un Bytecode et que java soit compilé à la volé en code natif, donc il accélère parfois contrairement à Python.

Et ces deux derniers sont plus élevés par rapport à la courbe de C et C++.

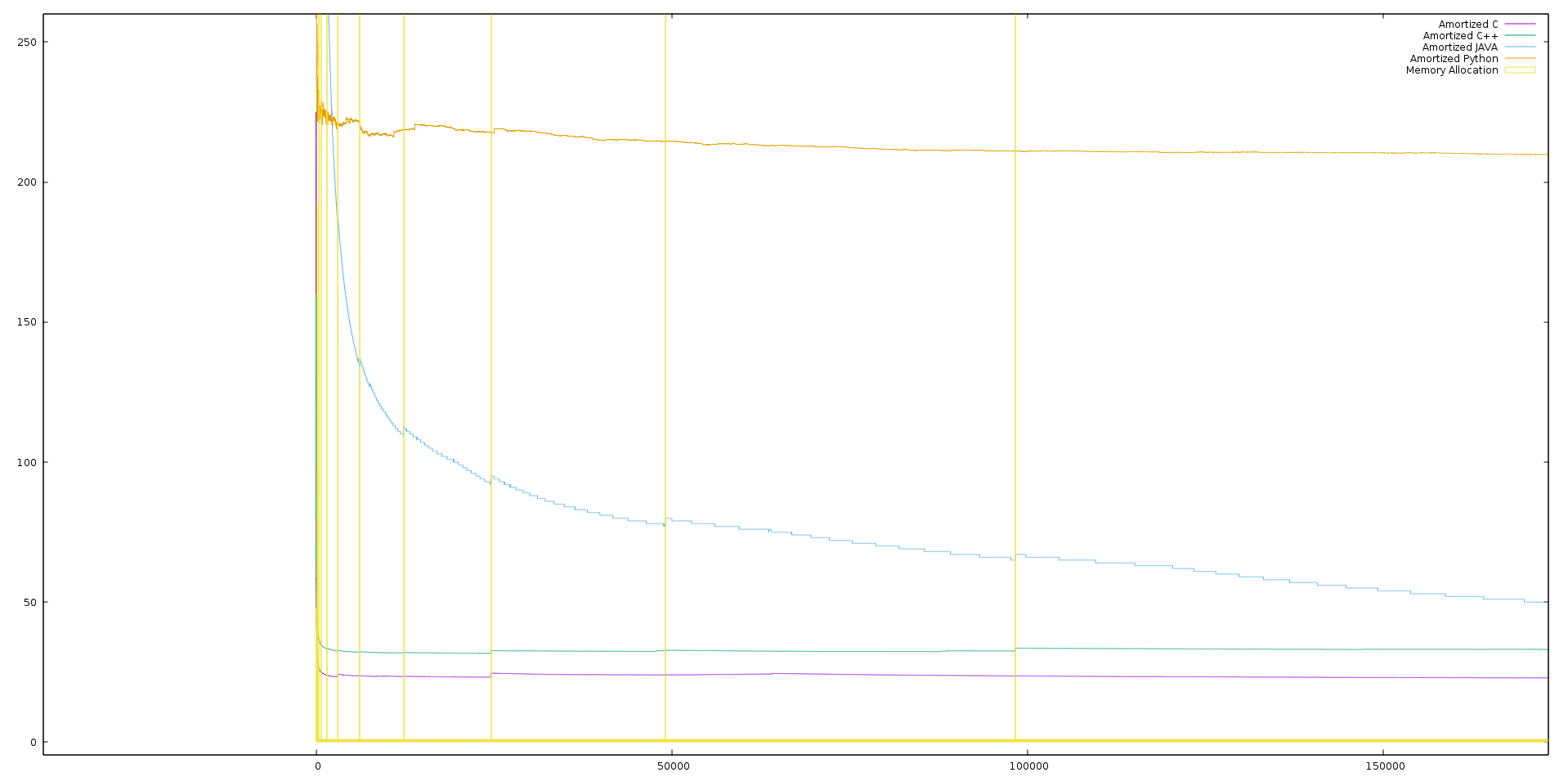


Figure  : Coût amorti pour tous les langages

**(f)**

En observant l’espace mémoire inutilisé on remarque que celui-ci ne cesse d’augmenter surtout au moment de l’allocation, cette augmentation se produit car dans cette expérience on multiplie la taille du tableau par 2 à chaque réallocation ce qui fait que le tableau devient énorme et que la mémoire gaspillée augmente.

Cependant on remarque aussi que cet espace mémoire finit toujours par baisser au bout d’un moment, ce qui pourrai nous intéresser ou nous embête n’est pas la perte de la mémoire dans laps de temps petit mais plutôt le fait

La figure ci-dessous représente l’espace mémoire inutilisé en fonction du temps réel, on remarque que la courbe baisse au moment de l’allocation mémoire mais elle n’atteint pas l’axe des abscisses (la mémoire gaspillé ne vaut jamais 0) car à chaque fois on réalloue de la mémoire quand le tableau est rempli à trois quarts du coup au moment de l’agrandissement il y a encore de l’espace dans notre tableau ce qui fait que la mémoire inutilisé ne vaut jamais 0.

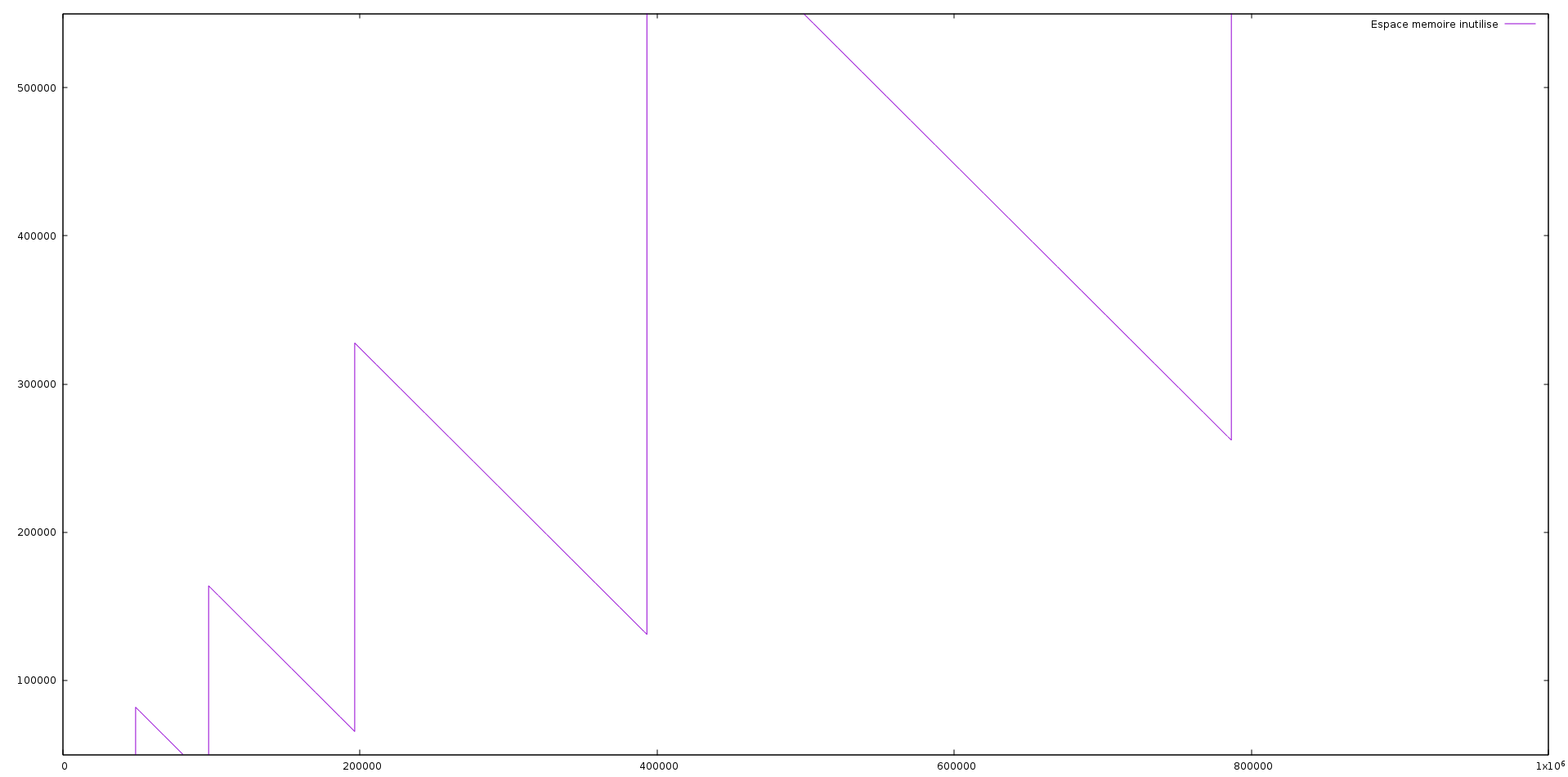


Figure  : Espace mémoire inutilisé par rapport au temps réel

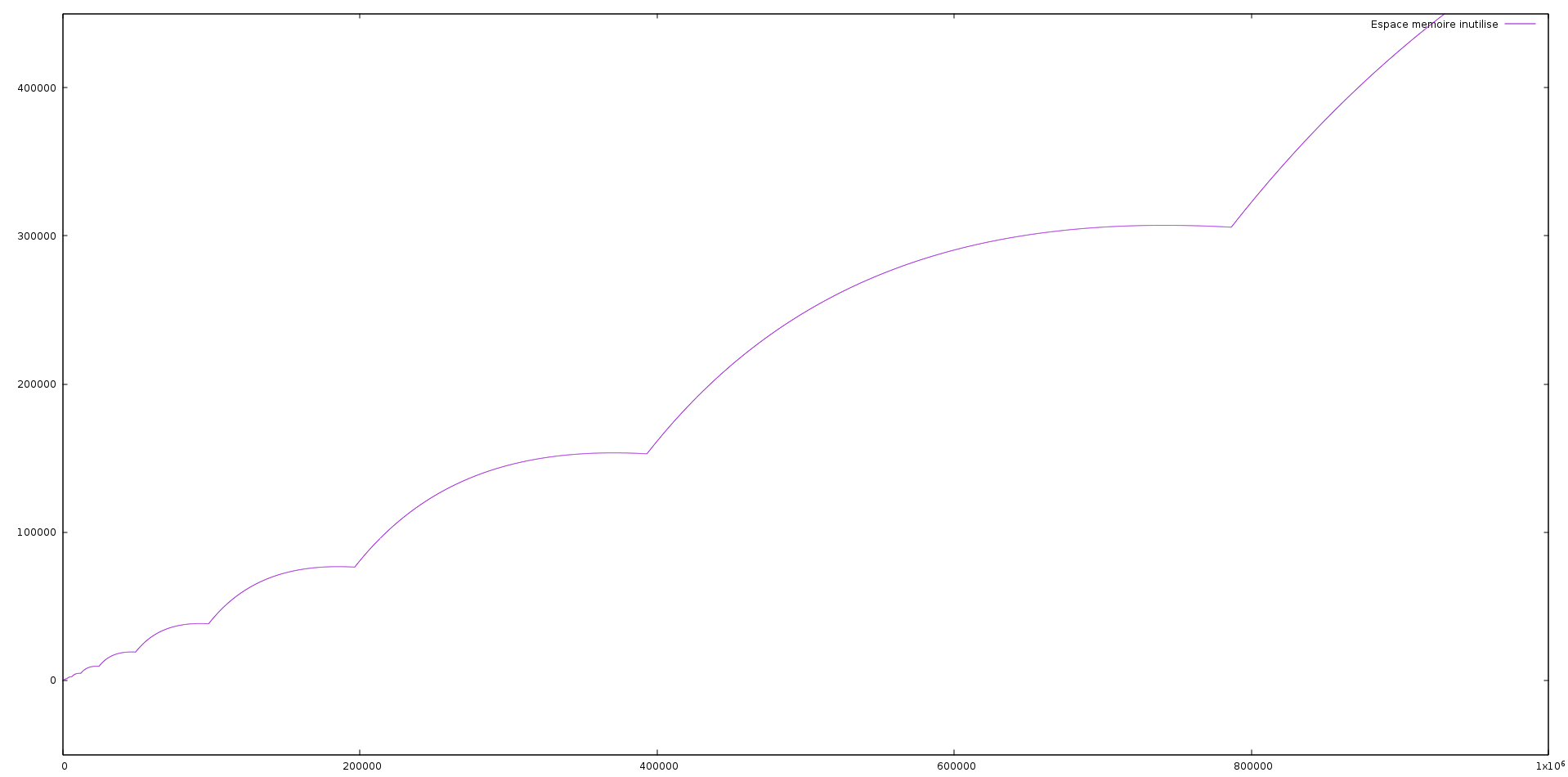


Figure  : mémoire inutilisé par rapport au temps amorti

Imaginons un scénario : considérons qu’on a un tableau de taille ‘n’, un nombre très grand, toutes les cases de notre tableau sont remplies .Toutefois on veut insérer un seul élément, on va procéder de la manière suivante :

-Allouer de la mémoire c’est-à-dire multiplié la taille du tableau par deux, on obtiendra notre nouvelle taille 2n.

-Insérer notre élément à la case n+1.

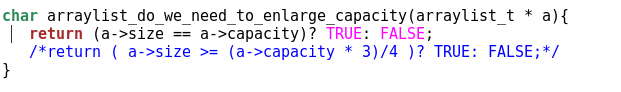
Après cette insertion on aura n -1 cases qui seront vides du coup on a gaspillé énormément de mémoire pour l’insertion d’un seul élément et le problème se pose là.

**(4)**

Le fait de réallouer de la mémoire quand le tableau est rempli à 3 quarts est complètement débile car on va faire trop de réallocation alors qu’on en a pas forcément besoin.

On a choisit le langage C :

On a modifié la fonction **« do\_we\_need\_to\_enlarge\_capacity »** pour qu’elle se déclenche lorsque le tableau est plein :



Après la modification on a relancé l’expérience et affiché la courbe de la mémoire inutilisée, en l’analysant on voit que la mémoire gaspillée augmente beaucoup au moment de la réallocation de la mémoire comme pour les questions précédentes mais il y a une différence entre les deux car ici on remarque que la courbe de la mémoire gaspillée atteint l’axe des abscisse à chaque fois qu’il y a un agrandissement du tableau et cela est logique car dans cette question on agrandi le tableau quand celui-ci est rempli donc à ce moment là il n y a pas de mémoire libre et donc la mémoire inutilisée vaut 0.

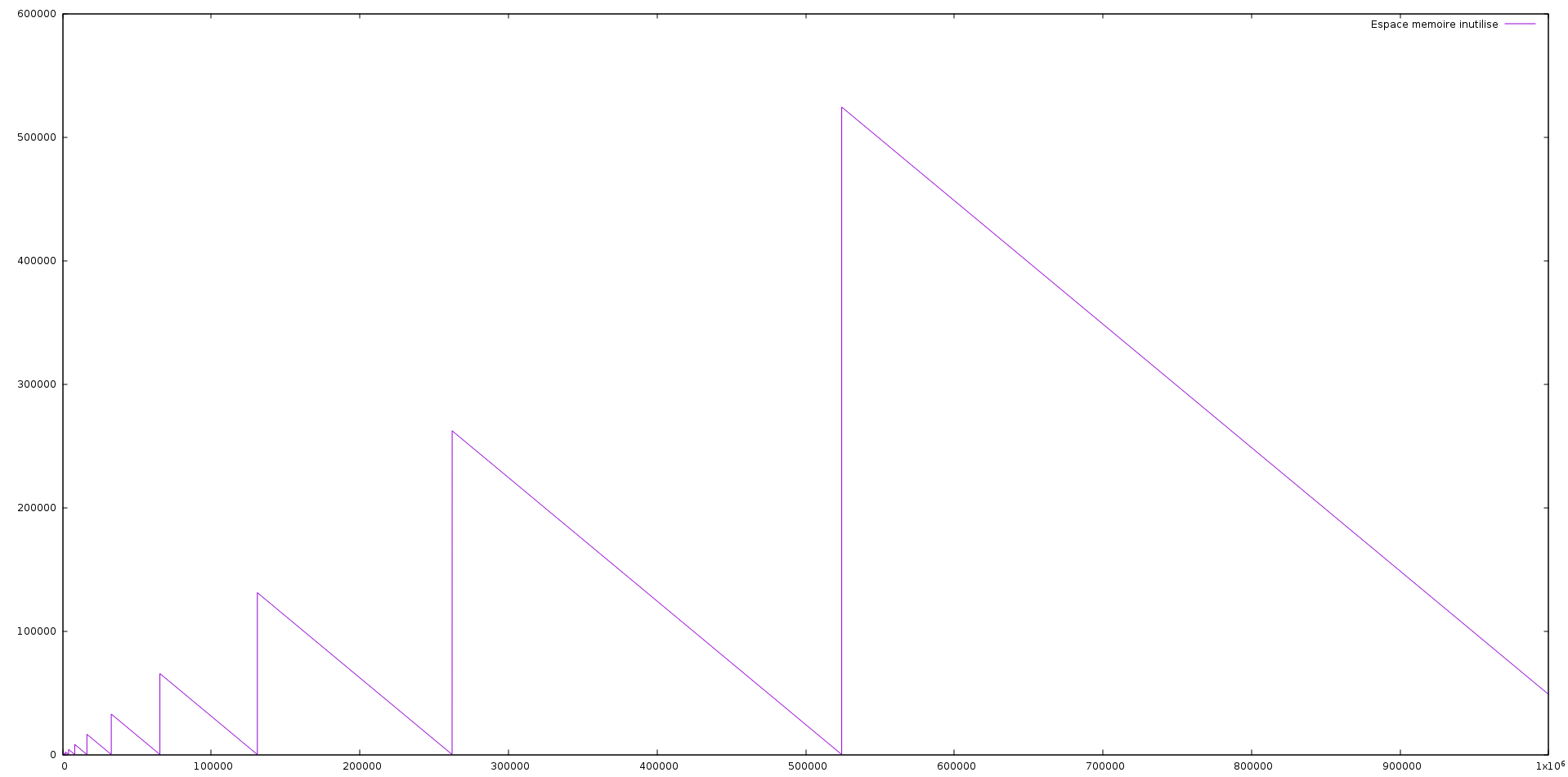


Figure  : mémoire inutilisée (agrandissement quand capacity = size)

On a aussi affiché le coût amorti et on l’a comparé avec le coût amorti (sans modification de la fonction do\_we\_need\_to\_enlarge\_capacity ) on remarque qu’il n y a pas beaucoup de différences.

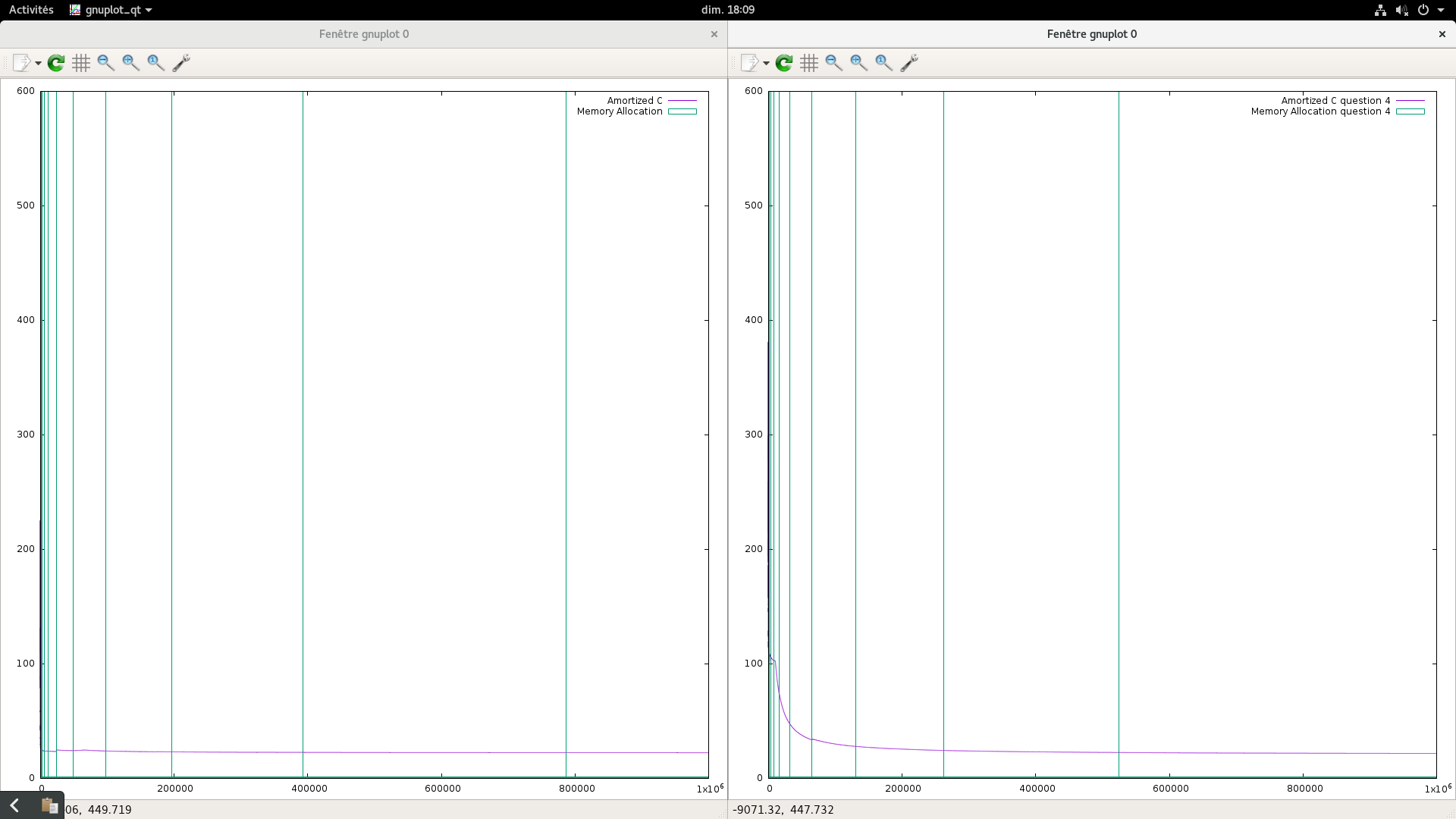


Figure  : différence coût amorti avant et après modification fonction

On a aussi comparé le nombre de copies avant et après la modification de la fonction, on a remarqué qu’il n ya pas de différence.

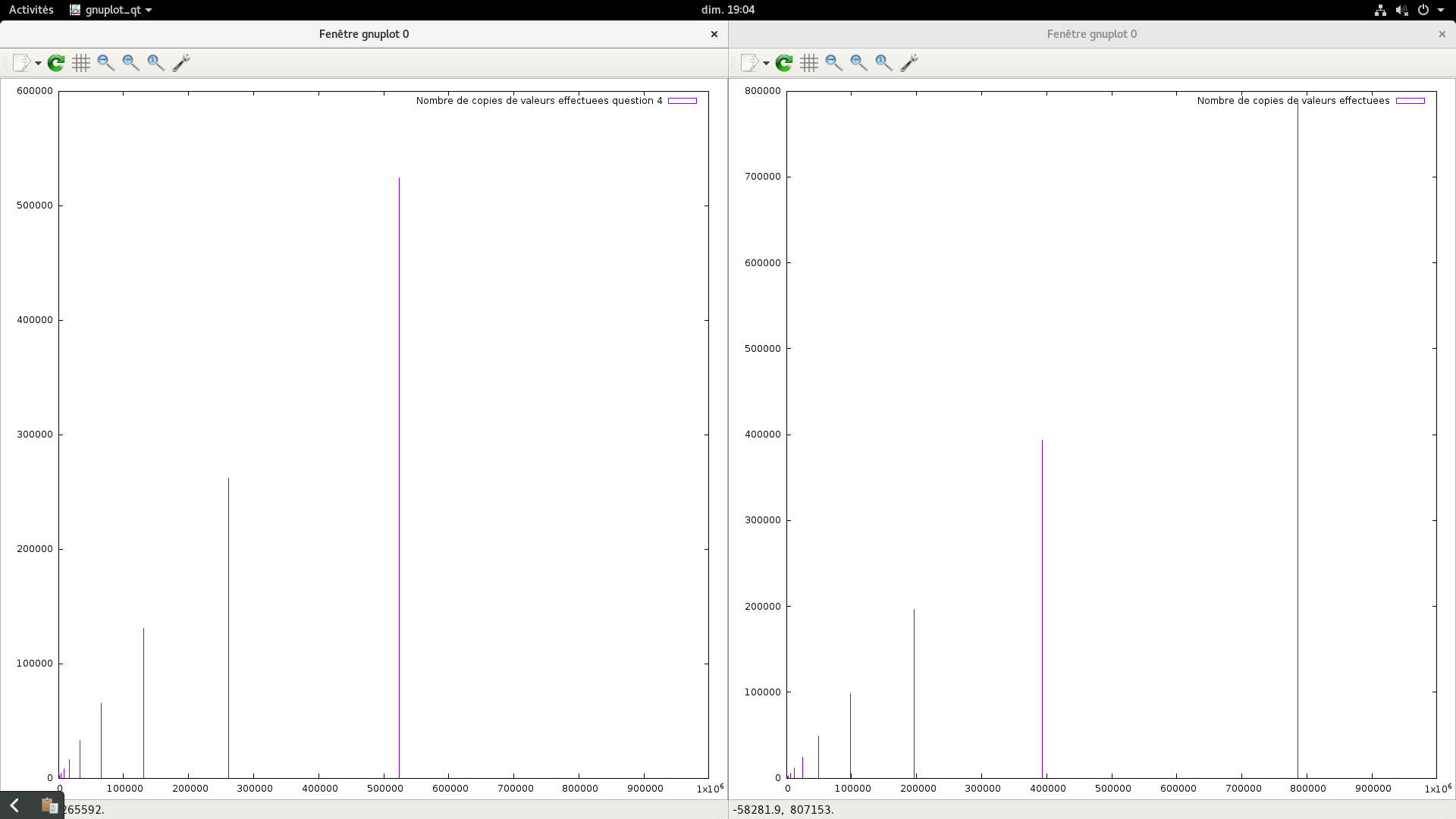


Figure  : différence nombre de copies avant et après modification de la fonction

**(5)**

**Pour α=5 :**

Pour commencer on a mit **« α=5 »** et on a obtenu les résultats suivants :

On a commencé par affiché le coût amorti des opérations, on remarque qu’il a légèrement baissé comparé au cas où alpha est égal à 2 et cela est du au fait qu’ici on fait moins de copies

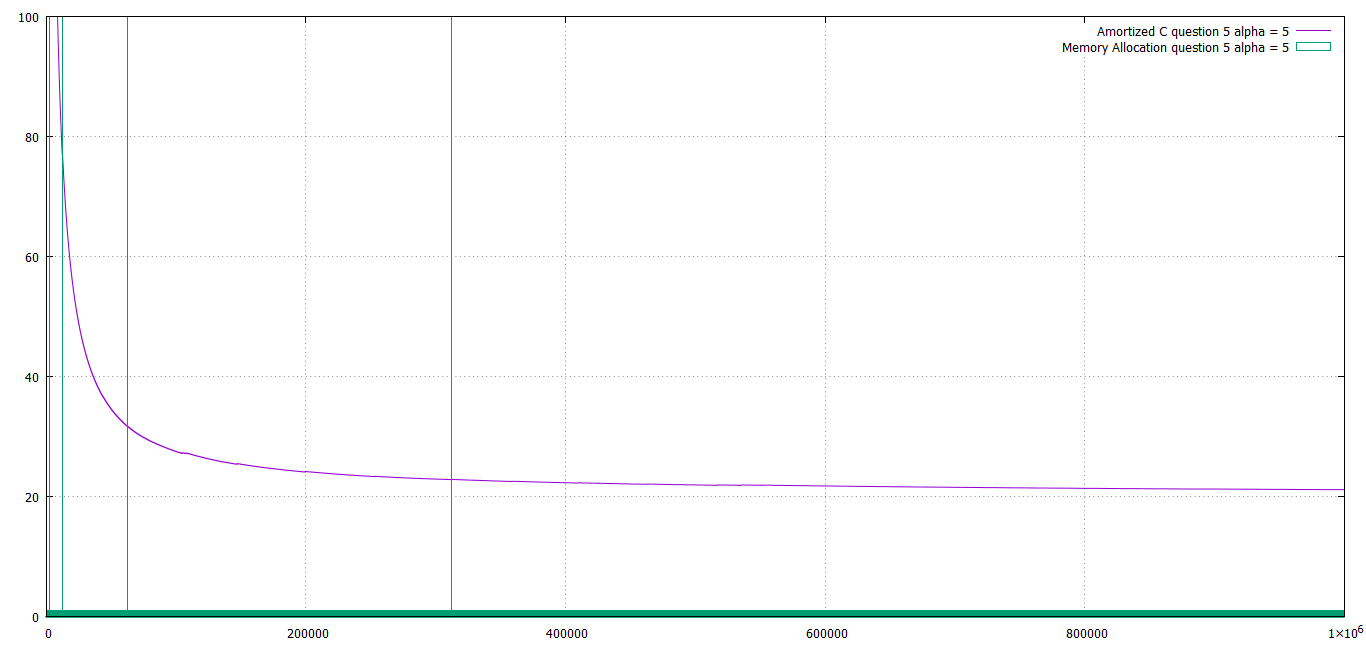


Figure  : Coût amorti et allocation mémoire quand alpha = 5

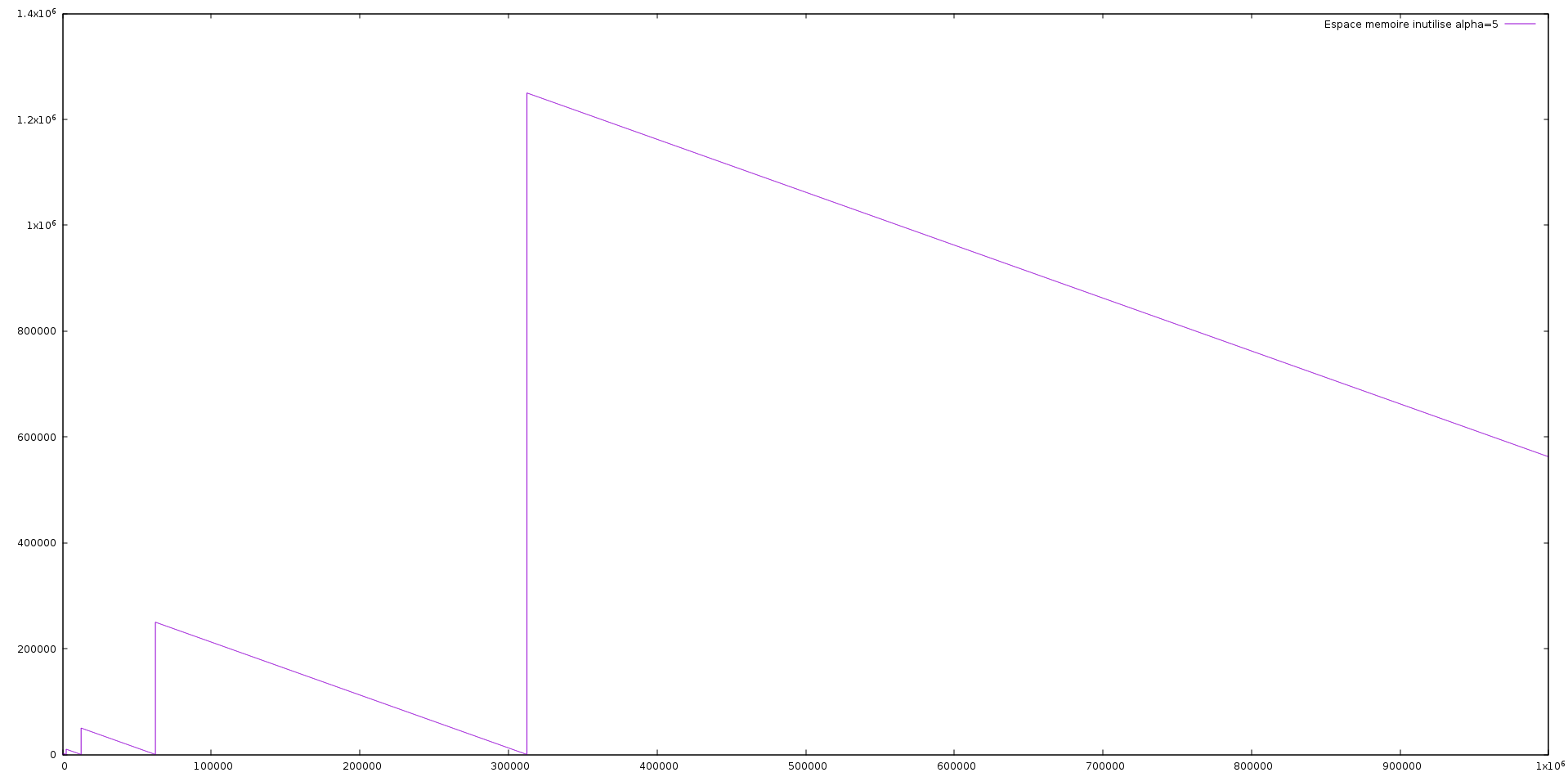
la mémoire inutilisée on se rend compte qu’elle augmente beaucoup plus qu’avant car la taille du tableau est multipliée par 5 à chaque fois, ce qui fait qu’il y a des cases mémoire vides qu’on n’a pas encore remplis autrement dit beaucoup d’espace mémoire gaspillée.   

Figure  : mémoire gaspillé quand α=5

En revanche pour le nombre de copies effectuées on remarque qu’on en a beaucoup moins qu’avant (quand α=2) car on multiplie la taille du tableau par 5 à chaque agrandissement, celui-ci devient alors de plus en plus grand, ainsi on aura de moins en moins de réallocation mémoire et donc moins de copies à effectuer.

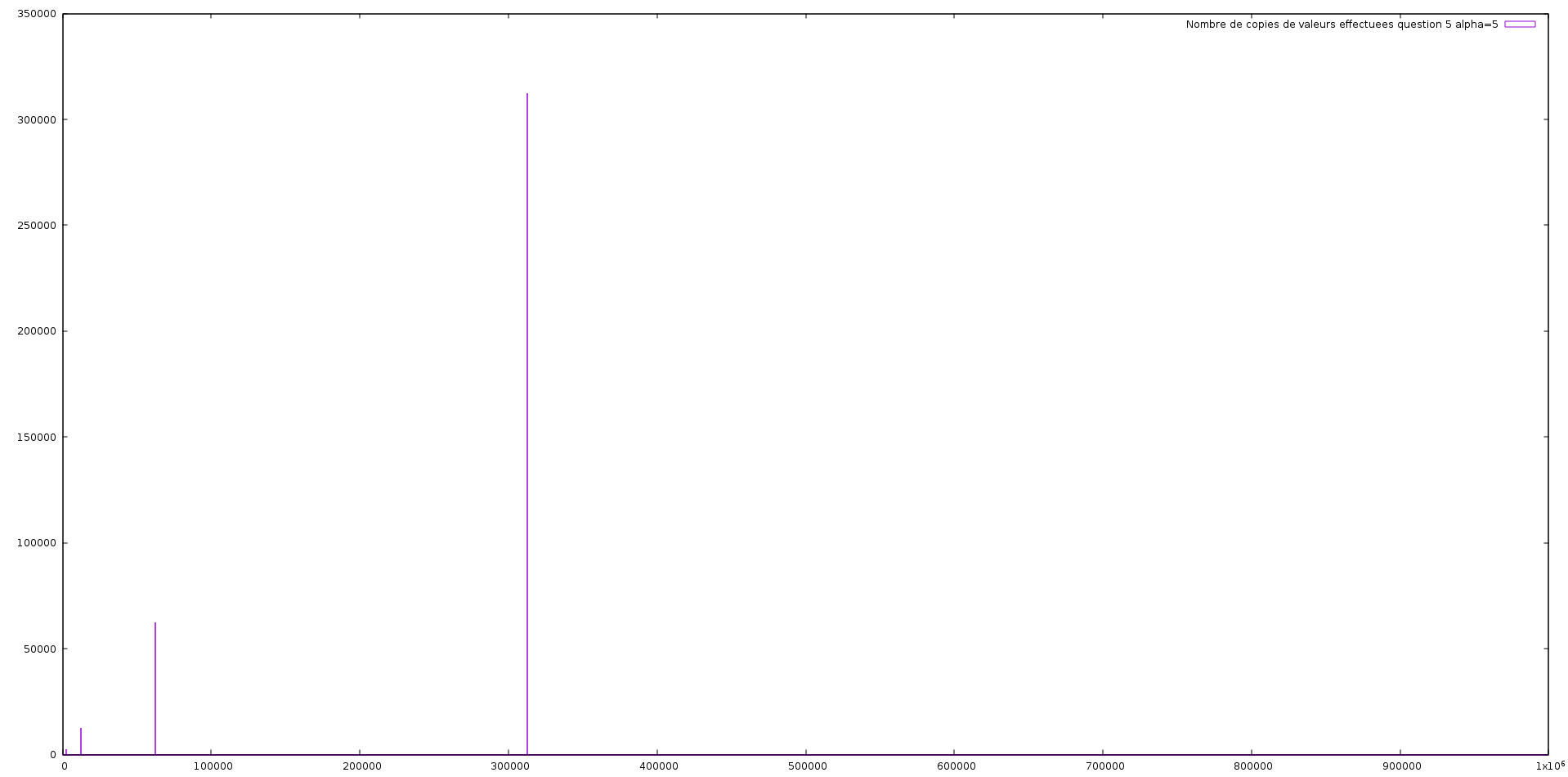
****

Figure  : Nombre de copies effectuées quand α=5

En comparant le graphe qui représente la mémoire inutilisé et celui qui représente le nombre de copies effectuées on remarque que : les deux courbes augmentent toutes les deux au moment de l’allocation mémoire. Mais les valeurs de la mémoire gaspillée sont beaucoup plus grandes.

Pourquoi ? On a choisitα=5 ce qui veut dire que la taille du tableau va être multiplié à chaque fois par 5, elle croît donc très vite, le nombre de cases mémoires est très grand, arrivé à un certain moment on n’aura plus besoin de réallouer de la mémoire, le nombre de copies est donc beaucoup plus petits, d’où les valeurs très petites du coût amorti.

Si le nombre de copies n’est pas élevé alors le coût en temps ne l’est pas aussi car on n’a pas à copier tout le contenu du tableau. Donc on peut dire qu’on est efficace en termes de temps d’exécution.

Conclusion : en choisissant α=5 (assez grand) on a gagné en efficacité temps mais en revanche on a gaspillé énormément de mémoire et on est sur que cette mémoire ne sera jamais utilisée par la suite. Donc cette stratégie n’est pas très efficace.

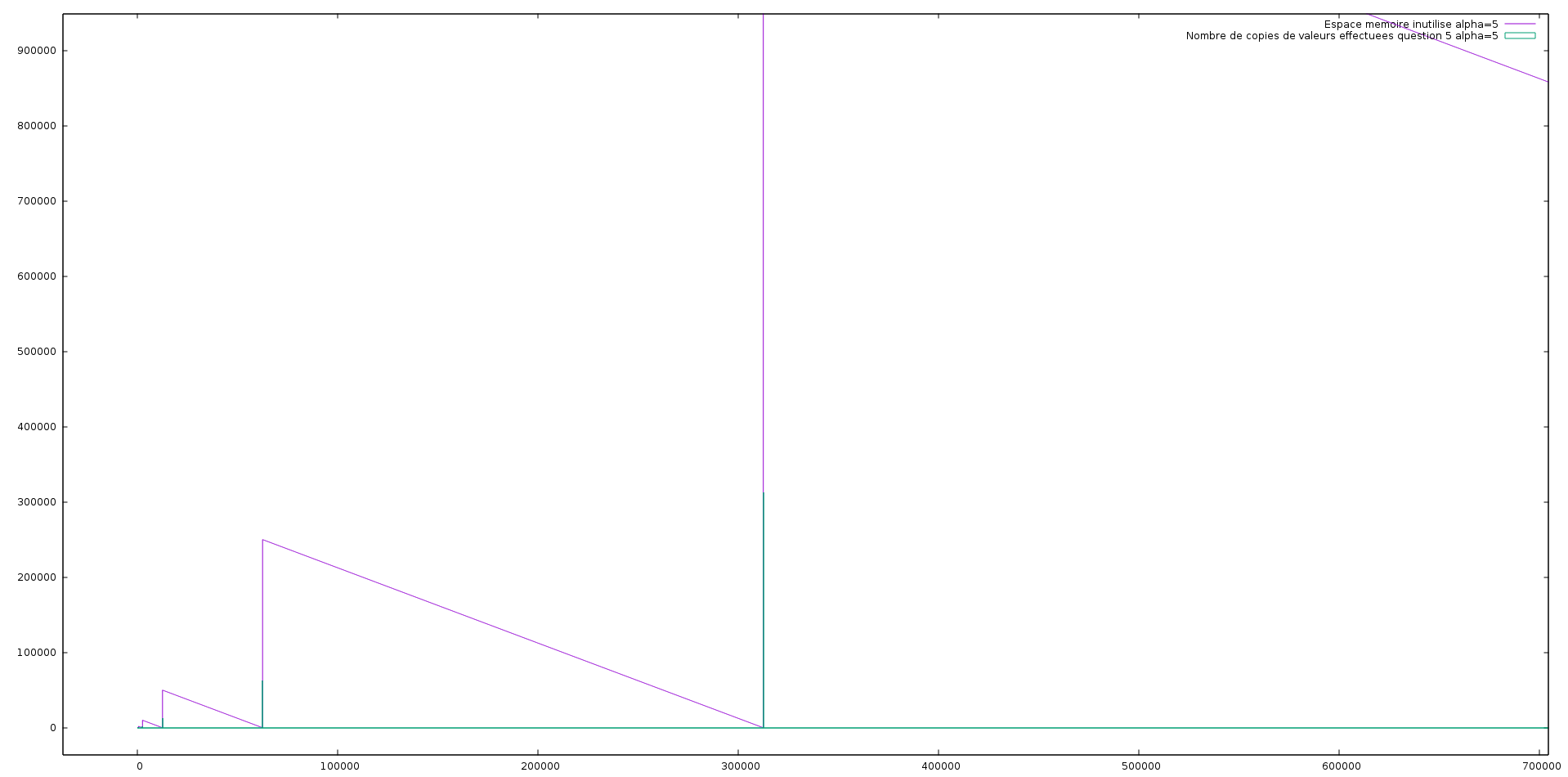
****

Figure  : Rapport nombre de copies effectuées et mémoire inutilisée quand α=5

**Pour α=1.5**

On a obtenu les résultats suivants :

Ce graphe montre que la mémoire gaspillé augmente plus lentement que dans le cas oùα=1.5, et ces augmentations sont plus fréquentes car on à chaque fois que le tableau est plein on multiplie sa taille par 1.5, ce qui est assez petit par rapport à 5 et 2 (qu’on a analysé dans les questions précédentes) , ainsi la taille du tableau ne croît pas vite, on doit réallouer de la mémoire plus souvent.

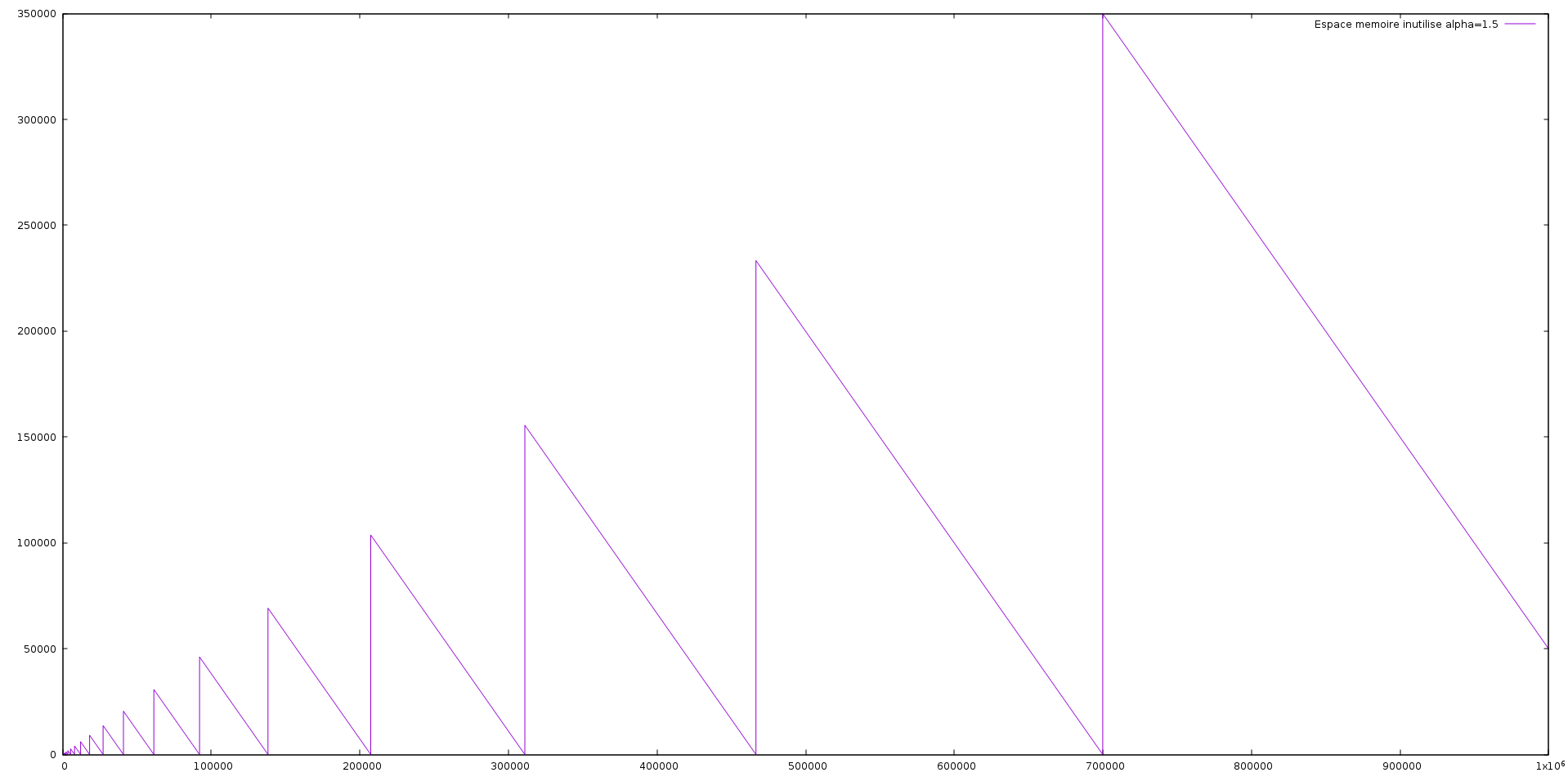
****

Figure  : mémoire gaspillé quand α=1.5

En ce qui concerne le nombre de copies effectuées, on remarque qu’il a augmenté comparé à l’expérience précédente. En effet on multiplie le tableau par 1.5 à chaque agrandissement (où on doit copier tous ses éléments), mais comme notre α est petit alors la taille du tableau ne devient pas trop grande ce qui entraine des réallocations de la mémoire plus fréquentes et le nombre de copies qui est devenu plus grand, car on doit copier les éléments à chacune de ces réallocations.

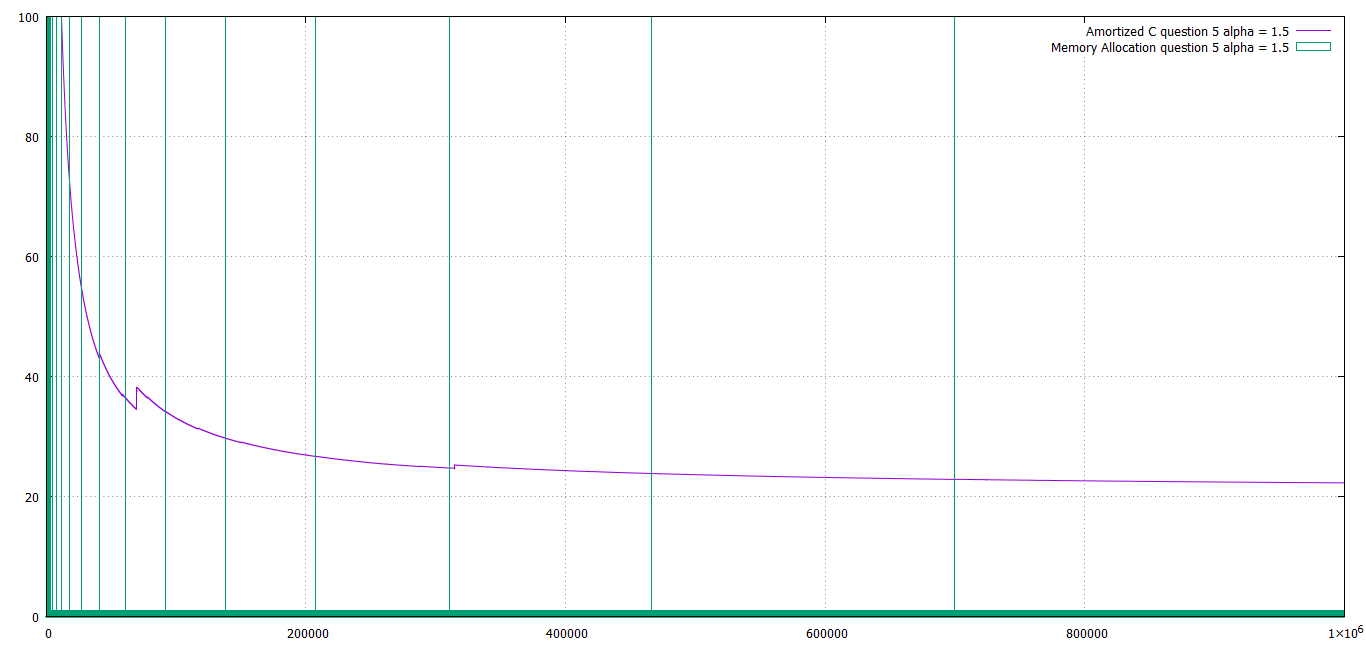


Figure  : coût amorti et allocation mémoire pour alpha = 1.5

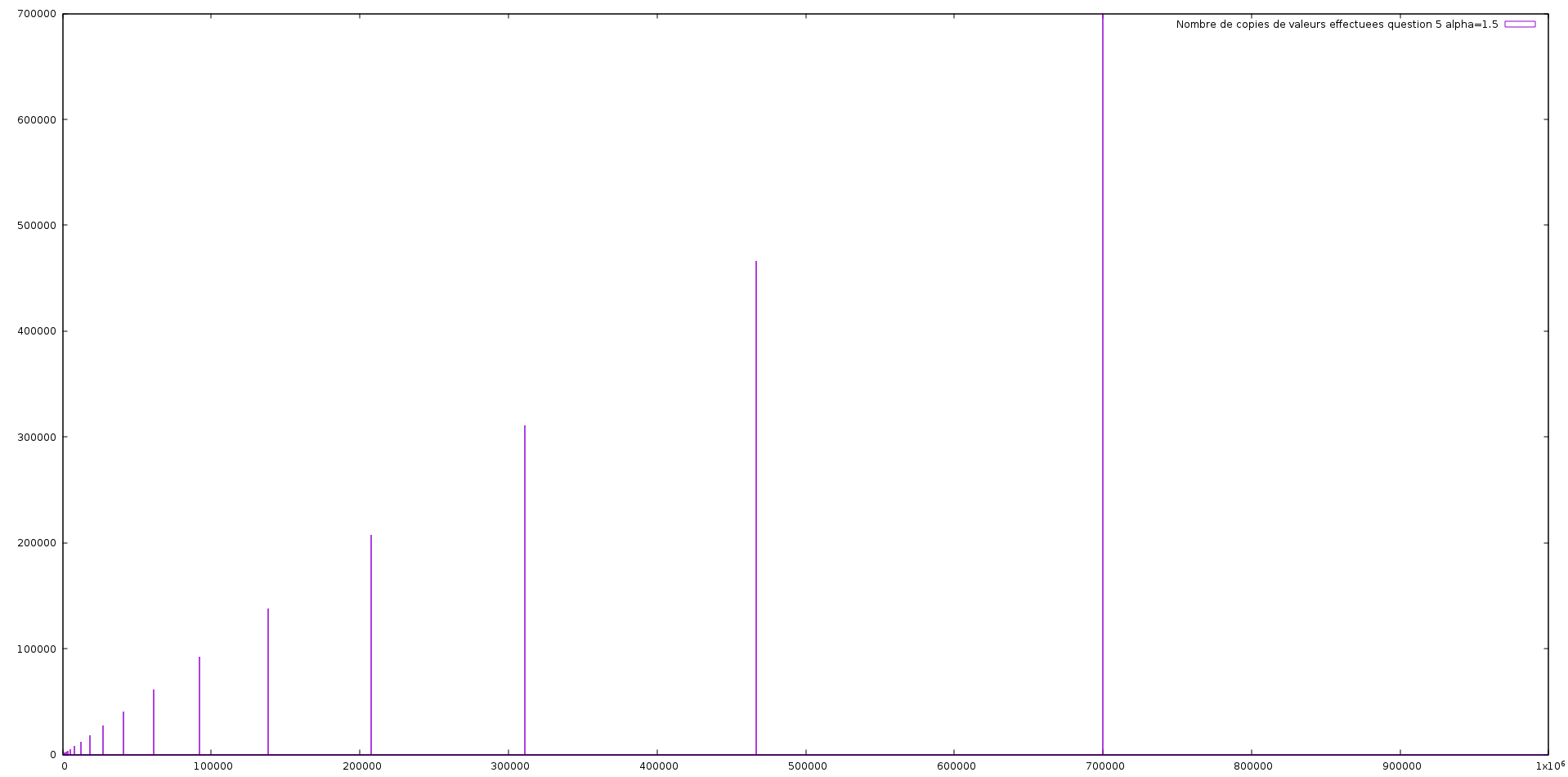
****

Figure  : Nombre de copies effectuées quand α=1.5

En comparant le graphe qui représente la mémoire inutilisé et celui qui représente le nombre de copies effectuées on remarque que : les deux courbes augmentent toutes les deux au moment de l’allocation mémoire. Mais les valeurs du nombre de copies effectuées sont beaucoup plus grandes.

On a choisi ici un α petit, ce qui fait que la taille du tableau n’augmente pas beaucoup, autrement dit il n y a pas beaucoup de cases mémoire inutilisées donc la mémoire gaspillé n’est pas très élevée, grâce à cela le coût en espace n’est pas très grand, on gagne alors en efficacité espace.

Par contre le fait que α soit petit n’arrange pas les choses pour ce qui est du nombre de copies effectuées, car le fait que le tableau n’est pas grand provoque une réallocation de la mémoire assez fréquemment, ainsi le nombre de copies est grand et coûte chère pour en termes de coût en temps.

Dans ce cas on a été efficace en termes d’espace car on a minimisé les pertes de mémoire, mais on a perdu en temps car le fait de copier tous les éléments à chaque fois coûte chère, mais on pense que ces pertes en temps d’exécution restent raisonnables.

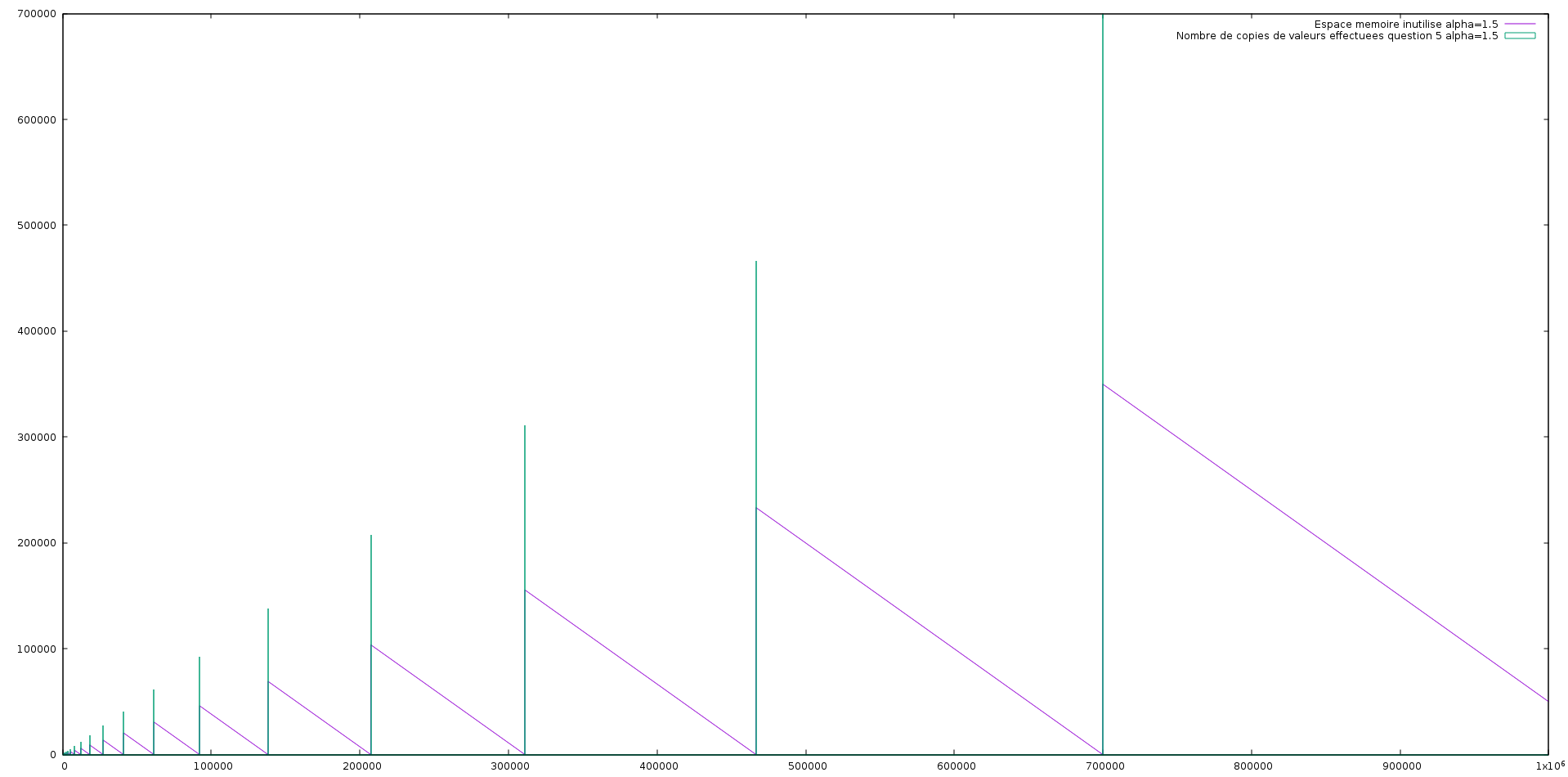
****

Figure  : : Rapport nombre de copies effectuées et mémoire inutilisée quand α=1.5

**Rapport entre le coût en temps et le coût en espace :**

Après avoir effectué plusieurs analyses on a constaté que quand α est petit on a plus de copies et moins de gaspillage mémoire autrement dit le coût en espace est plus élevé que le coût en temps. Et quand α est grand c’est l’inverse, le coût en temps est plus élevé que le temps en espace.

**Conclusion :**

On pense que les bons rapports sont ceux qui sont plus petit que 2 car ils nous permettent de gagner en espace mémoire et d’avoir un coût amorti constant, certes un peu plus élevé que quand alpha est grand mais pas tellement.

**(6)**

Dans cette question nous avons varié la capacité n vers **n+sqrt(n)**

On a commencé par afficher le coût amorti et l’allocation mémoire, on remarque d’après ce graphe qu’il y a énormément de copies comparé aux expériences précédentes.

On remarque aussi qu’il ne se passe pas beaucoup de temps entre une allocation et une autres et cela peut s’expliquer par le fait qu’on multiplie la taille du tableau par un facteur très petit ainsi le tableau est vite rempli.

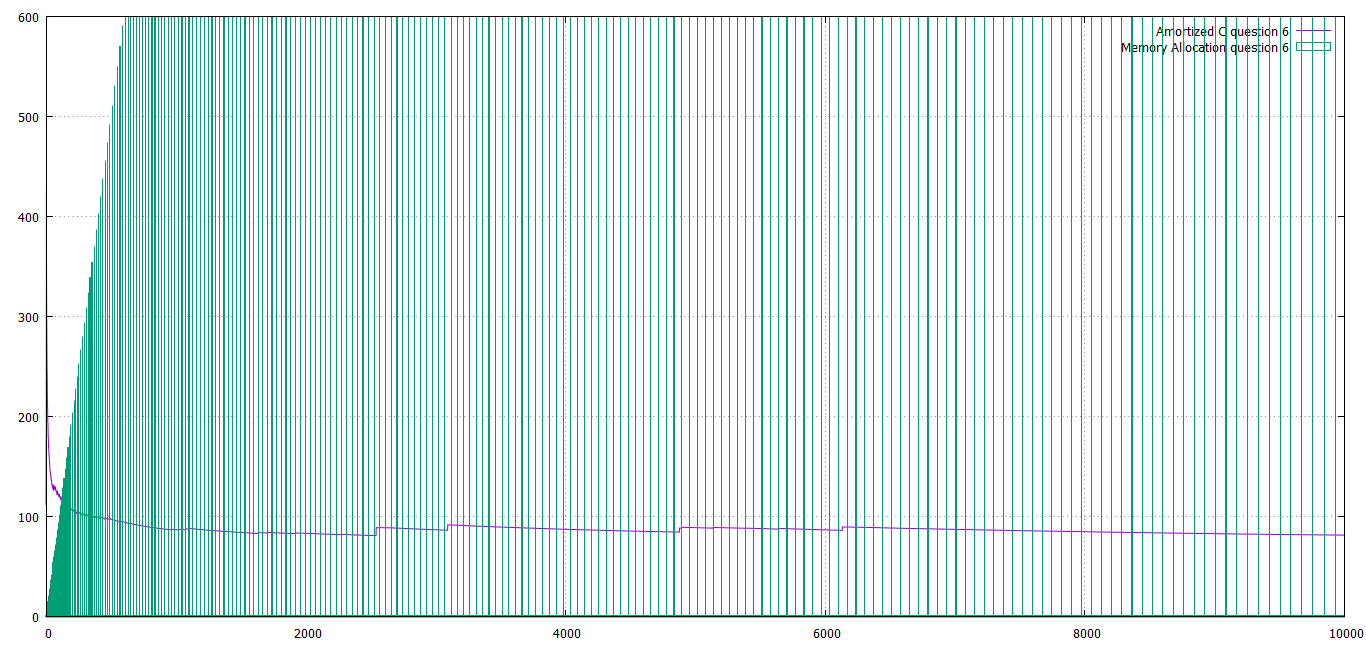


Figure  : Coût amorti et allocation mémoire pour alpha en fonction de la capacité

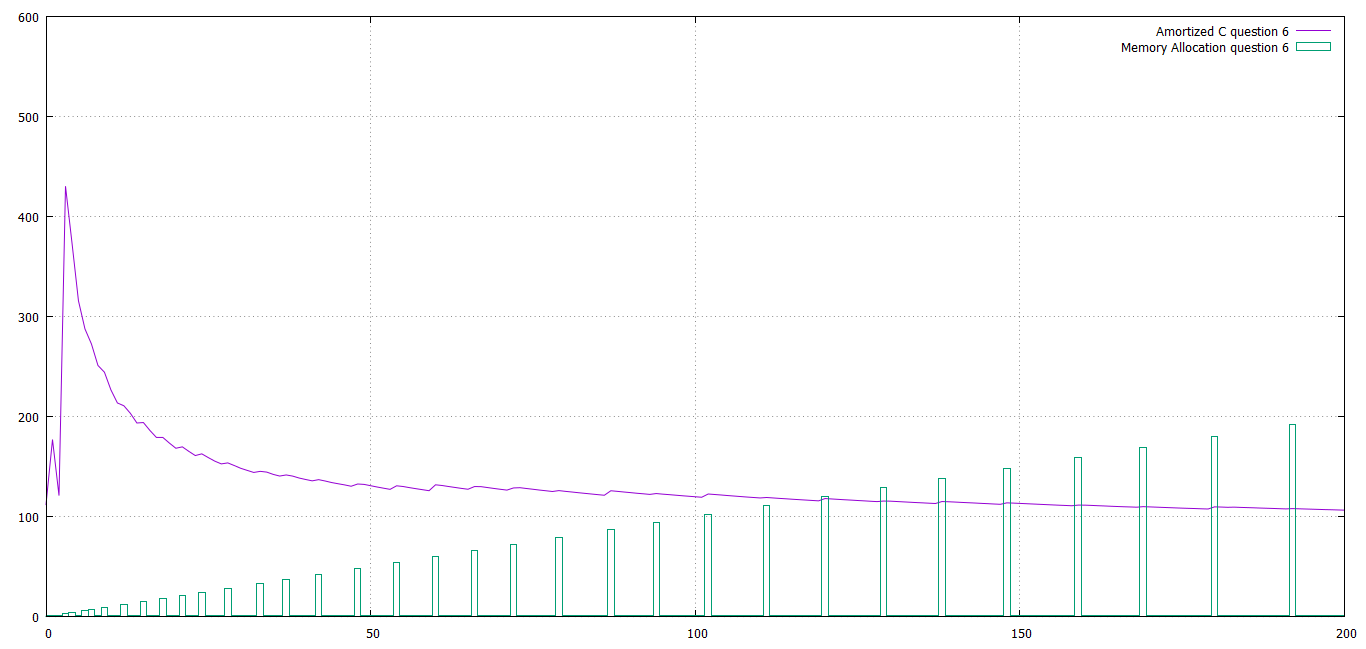


Figure  : Coût amorti et allocation mémoire pour alpha en fonction de la capacité

En regardant l’espace mémoire inutilisé on remarque qu’il augmente et baisse souvent et cela se passe au moment de l’allocation mémoire, les augmentations ne durent pas beaucoup car, comme dit précédemment, le tableau est vite rempli.

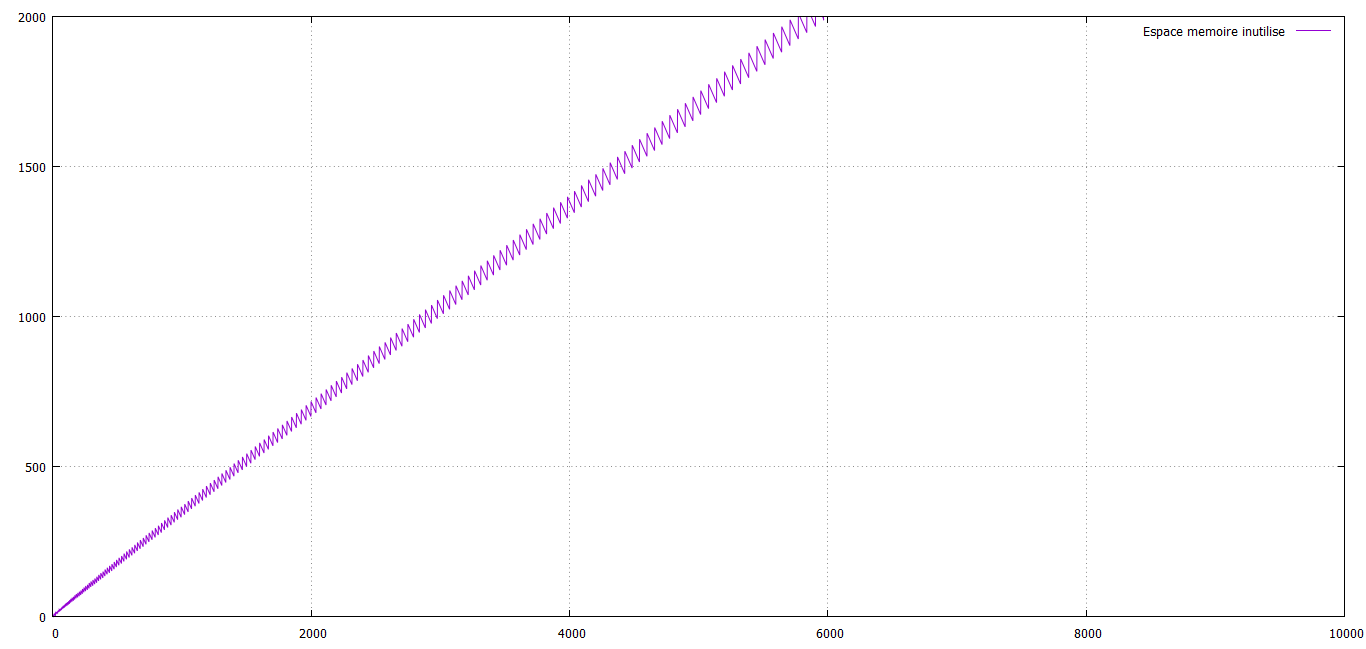


Figure  : Espace mémoire inutilisé pour alpha en fonction de la capacité

**Conclusion :**

On pense que cette méthode de réallocation ne convient pas du tout pour le cas des insertions successives car on a beaucoup d’allocation mémoire, ce qui ne nous arrange pas car cela nous fait perdre énormément en temps d’exécution.

Nous pensons cependant qu’elle pourrait être utile en cas d’insertions et suppressions successives car l’idée serait de réallouer un peu de mémoire à chaque fois que le tableau est plein, car on va effectuer des suppressions, qui vont libérer des cases dans le tableau, qui ne sera donc pas souvent rempli, et pas trop vide non plus car sa taille va rester assez proche de son nombre d’éléments, donc on ne va pas faire beaucoup d’allocation ni de contractions.

Ce n’est qu’une intuition pour le moment, nous allons essayer de la confirmer en faisant l’analyse du TP2.

**Partie 2 : Conclusion**

Après avoir répondu aux questions du sujet du tp1 dans la première partie, on va dans cette partie, conclure par ce que nous pensons de cette structure, qui est les tableaux dynamiques et aussi par ce que nous avons appris là-dessus.

Un tableau dynamique est une structure qui permet de stocker des données et qui est utilisé quand on veut construire un programme mais qu’on ne sait pas à l’avance combien d’espace mémoire on va utiliser, on utilise donc un tableau dynamique pour pouvoir allouer de la mémoire dès qu’on en a besoin et que ce dernier est plein.

Prenons l’exemple des files de priorité dans lequel on aura besoin d’un tableau dynamique, les files à priorités sont des structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité, un exemple simple ou ces files sont utilisées est l’ordonnanceur dans les systèmes d’exploitation, qui gère l’ordre d’exécution des processus. Pour implémenter ces files on pourra donc utiliser un tableau dynamique, si on choisit de pendre le premier élément du tableau comme élément de priorité forte alors on pourra le récupérer en temps constant, mais dans ce cas là on devra décaler tous les éléments du tableau à chaque extraction ce qui n’est pas pratique car on obtient un temps d’exécution qui coûte cher et qui augmente à chaque fois que le nombre d’élément augmente, il a une complexité de O(N) (N taille du tableau) rajouté à cela le nombre de copies effectuées à chaque réallocation. Je pense donc que dans cet exemple ce n’est pas trop pratique d’utiliser des tableaux dynamiques car le but d’une file c’est d’extraire l’élément le plus prioritaire efficacement, or ici il n’est pas du tout efficace .

Selon nous, la structure la plus adapté serait les tas, l’élément de haute priorité serait la racine, l’extraction et l’ajout se font en O(Log N).