|  |
| --- |
|  |
| Structures de données  Tas binaires dynamiques |
|  |
|  |
| **Lynda MEDJDOUBI**  **Sabrina DJENADI** |
|  |

|  |
| --- |
|  |

**Sommaire**

**Choix d’implémentation3**

**Insertion des clés dans l’ordre croissant4**

Pour alpha = 24

Pour alpha = nombre d’or 8

Pour alpha = 2013

**Insertion des clés dans l’ordre décroissant16**

Pour alpha = 216

Pour alpha = nombre d’or19

Pour alpha = 5022

**Insertion des clés aléatoires24**

Pour alpha = 224

Pour alpha = nombre d’or27

**Insertion des clés et extraction du minimum31**

Pour alpha = 231

**Conclusion34**

**Choix d’implémentation:**

Pour l’implémentation des tas binaires dynamiques on va utiliser des tableaux dynamiques en réallouant de la mémoire à chaque fois que le tableau est plein en multipliant par un facteur alpha.

On a commencé par définir une structure pour chaque nœud, cette structure contient un pointeur vers le tas, un pointeur vers un tableau d’entiers, qui va contenir les clés des nœuds, une capacité ainsi qu’un entier size représente le nombre d’élément dans le tableau, il est utilisé pour savoir à quel moment on va réallouer de la mémoire.

Ensuite on a défini une suite de fonction dont on a besoin pour faire l’analyse de cette structure.

**1.createTas** : cette fonction alloue de la mémoire pour le pointeur du tas ainsi que pour le tableau contenant les clés.

**2.tas\_insert :** cette fonction permet d’insérer un entier x dans un tas, la seule différence avec la fonction « tas\_insert » définit pour les tas statiques c’est qu’ici on teste grâce à la fonction « do\_we\_need\_to\_enlarge\_capacity » si le tableau est plein pour pouvoir réallouer de la mémoire.

**3.extract\_min :** elle permet d’extraire la racine du tas qui est aussi l’élément le plus petit**,** elle diffère de la fonction « extract\_min » définit dans les tas statiques dans le fait qu’ici on teste en faisant appel à la fonction « do\_we\_need\_to\_reduce\_capacity » si le nombre d’éléments du tableau est plus petit qu’un quart de sa taille pour pouvoir réduire la taille en la divisant par 2.

Après avoir essayé d’analyser la structure de tas min statiques on s’est rendu compte que le fait que ces tas soient statiques ne nous arrangeait pas tellement, alors on va s’attaquer dans ce rapport à l’analyse des tas binaires dynamiques qui sont à priori une solution pour le problème que nous avons rencontré.

**Insertion des clés dans l’ordre croissant:**

**Pour alpha = 2**

On a affiché le coût réel de l’opération d’insertion des valeurs successives ordonnées de façon croissante une fois avec 1000 valeurs seulement puis on a recommencé l’expérience avec 5000 valeurs. La différence n’est pas très énorme en coût réel car on remarque qu’à chaque allocation mémoire le coût réel augmente énormément dans les deux cas. On remarque une petite différence en dehors de l’allocation mémoire le graphe d’insertion est beaucoup plus perturbé que lorsqu’on insère seulement 1000 valeurs, ce qui est logique vu qu’avec 5000 valeurs on fait plus de copies dans notre nouveau tableau après une allocation mémoire.

Comparons notre coût réel à celui du cas statique ! On voit bien que le coût a augmenté, il atteint des valeurs supérieures à 600 et cela est clair surtout au moment de l’allocation mémoire. Cela est tout à fait logique vu que dans notre cas on fait l’allocation mémoire on copie les valeurs qui existe déjà dans le tableau avant d’allouer puis on insère, ce qui nous coûte plus chère. Le fait d’insérer des valeurs dans un ordre croissant veut dire qu’on n’a pas besoin d’entasser notre tas à chaque fois, ce qui nous fait gagner en efficacité. Les figures ci-dessous nous montrent les variations du coût réel des insertions du graphe lors des insertions avec 1000 et 5000 valeurs.

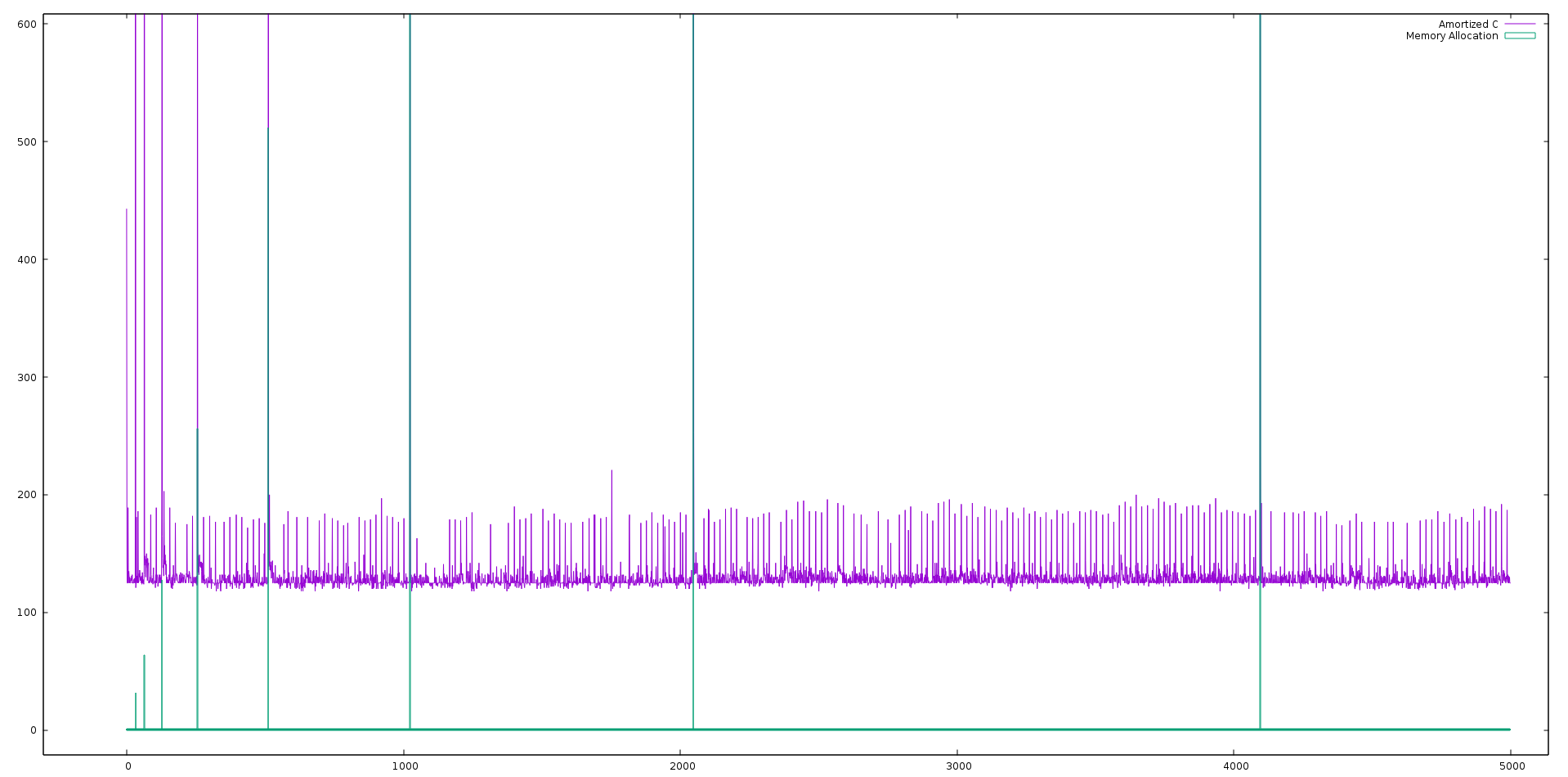


Figure  : Coût réel de l’insertion des 5000 clés dans l’ordre croissant pour alpha égal à 2

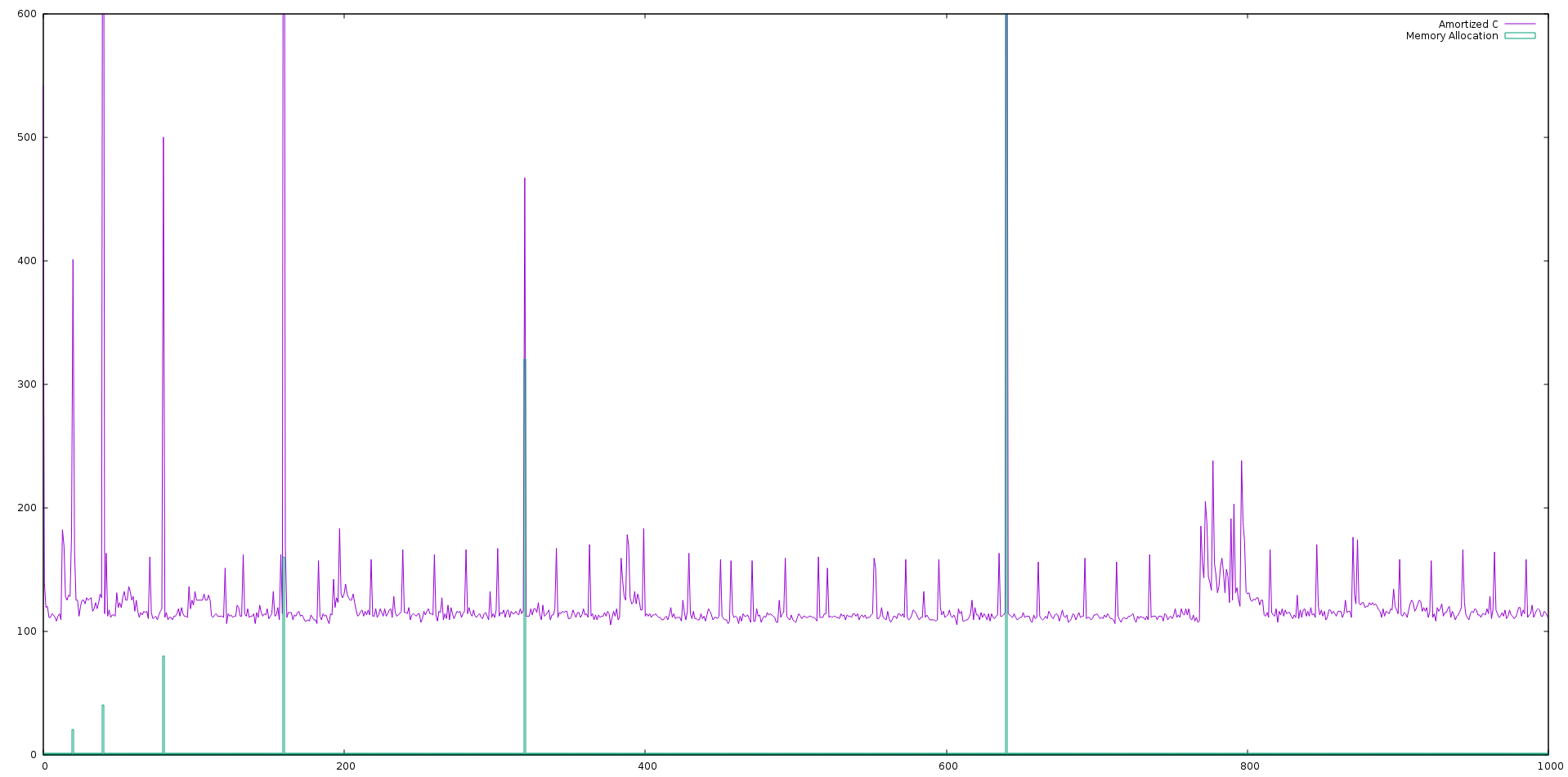


Figure  : Coût réel de l’insertion des 1000 valeurs dans l’ordre croissant pour alpha égal à 2

Ensuite on a affiché le coût amorti pour la même opération d’insertion, on remarque que le coût amorti au début est décroissant après avoir connu une valeur très élevée, puis il décroit et rencontre une petite augmentation au moment de l’allocation mémoire pour se stabiliser à une valeur entre 150 et 200, cette augmentation est normale vu qu’on fait une copie des valeurs à chaque allocation puis on insère.

Le coût amorti est plus élevé dans le cas dynamique que statique pour la même raison mentionnée dans le coût réel, c’est que dans ce cas on fait de l’allocation mémoire donc on copie les valeurs qui existe déjà dans le tableau avant d’insérer le nouvel élément, ce qui nous coûte plus chère comparé au cas statique où on ne fait pas de copies.

La figure ci-dessous nous montre les variations du coût amorti présenté sous forme de graphe.

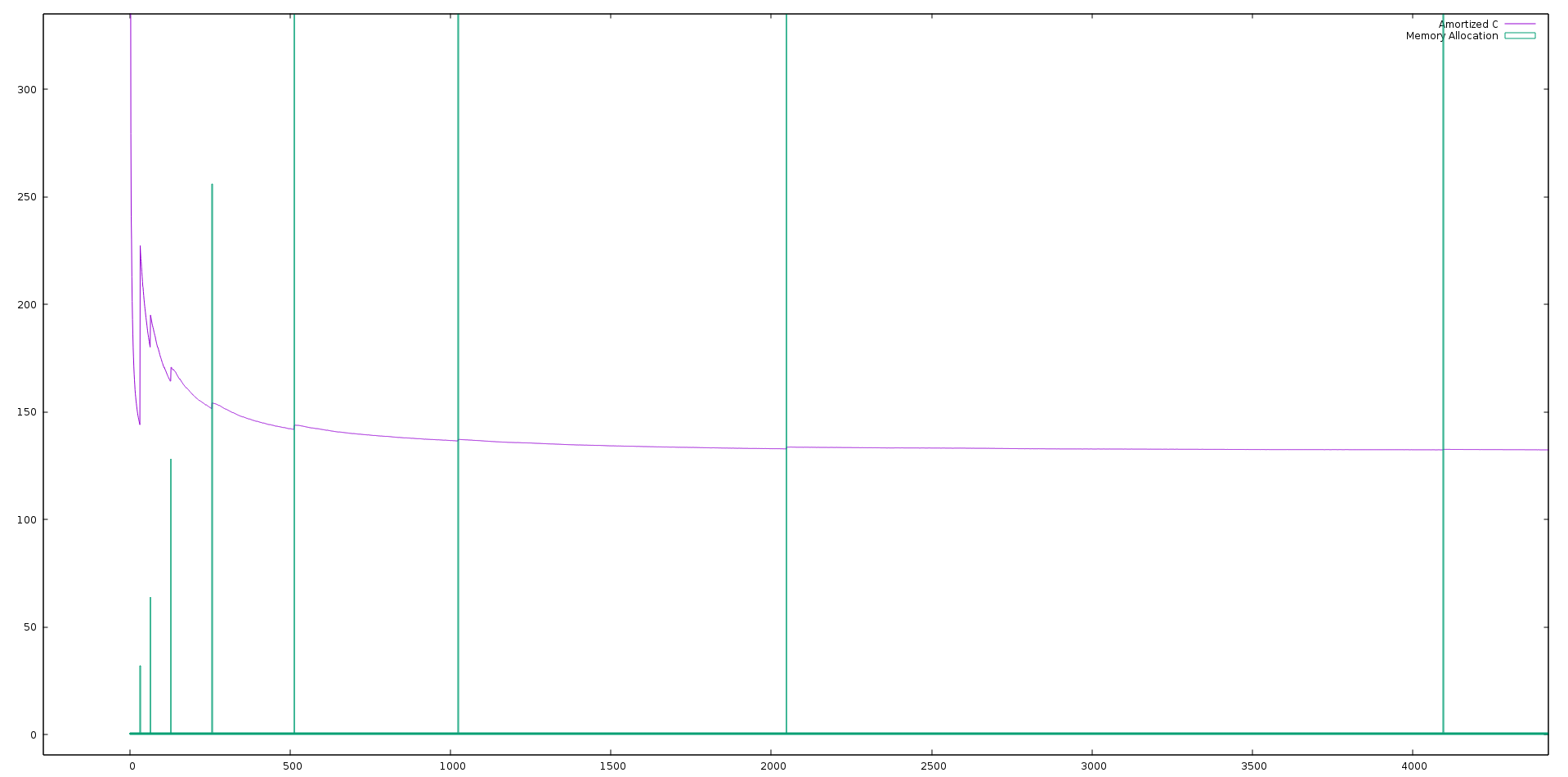


Figure  : Coût amorti de l’insertion des clés dans l’ordre croissant

Ensuite on a affiché l’espace mémoire inutilisé en fonction du temps pour l’insertion dynamique des valeurs croissantes, on remarque qu’au début on ne gaspille pas énormément de mémoire, mais en insérant plus de valeurs le tableau devient plus grand et on double la taille du tableau à chaque allocation ce qui nous fait perdre en espace mémoire. Tant le tableau est grand tant on gaspille de la mémoire, mais on voit d’après ce graphe que la mémoire gaspillée finit toujours par baisser.

Par exemple en arrivant à l’insertion des valeurs entre 4000 et 5000, on rencontre une augmentation énorme lors de l’allocation mémoire. En dehors de l’allocation mémoire notre graphe décroit rapidement vu qu’on fait des copies pour remplir le tableau.

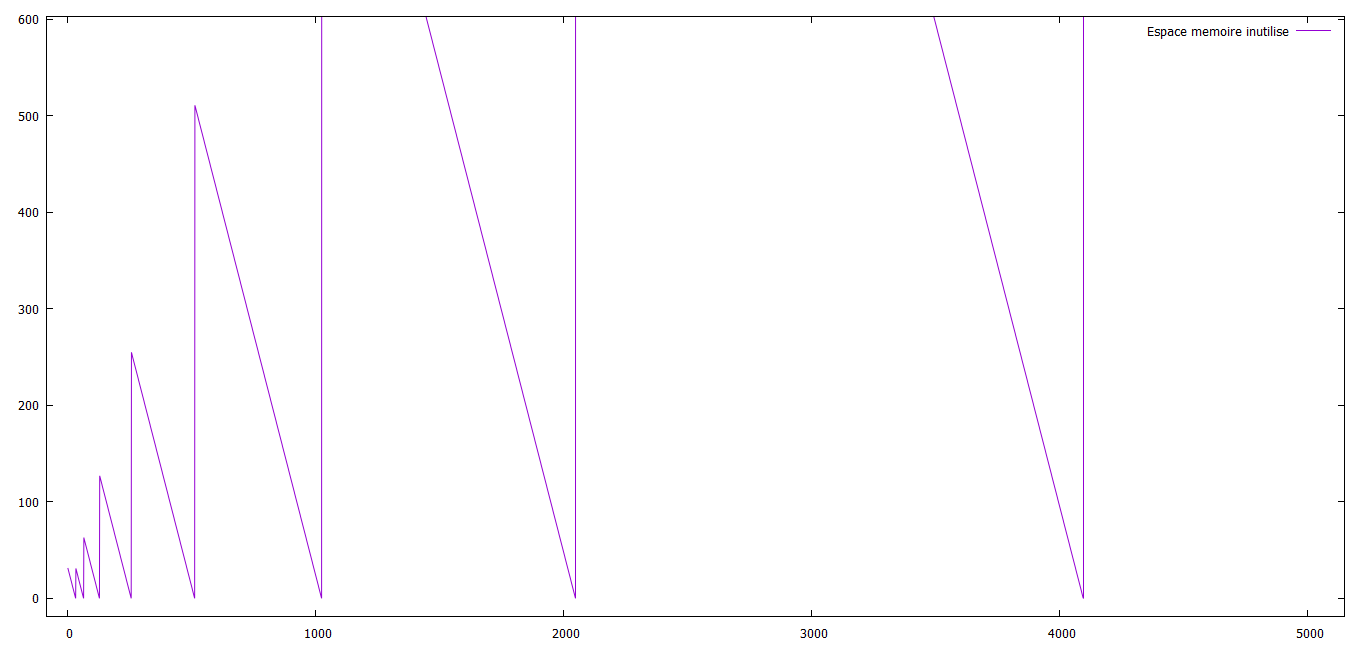


Figure  : Espace mémoire inutilisée en fonction du temps réel

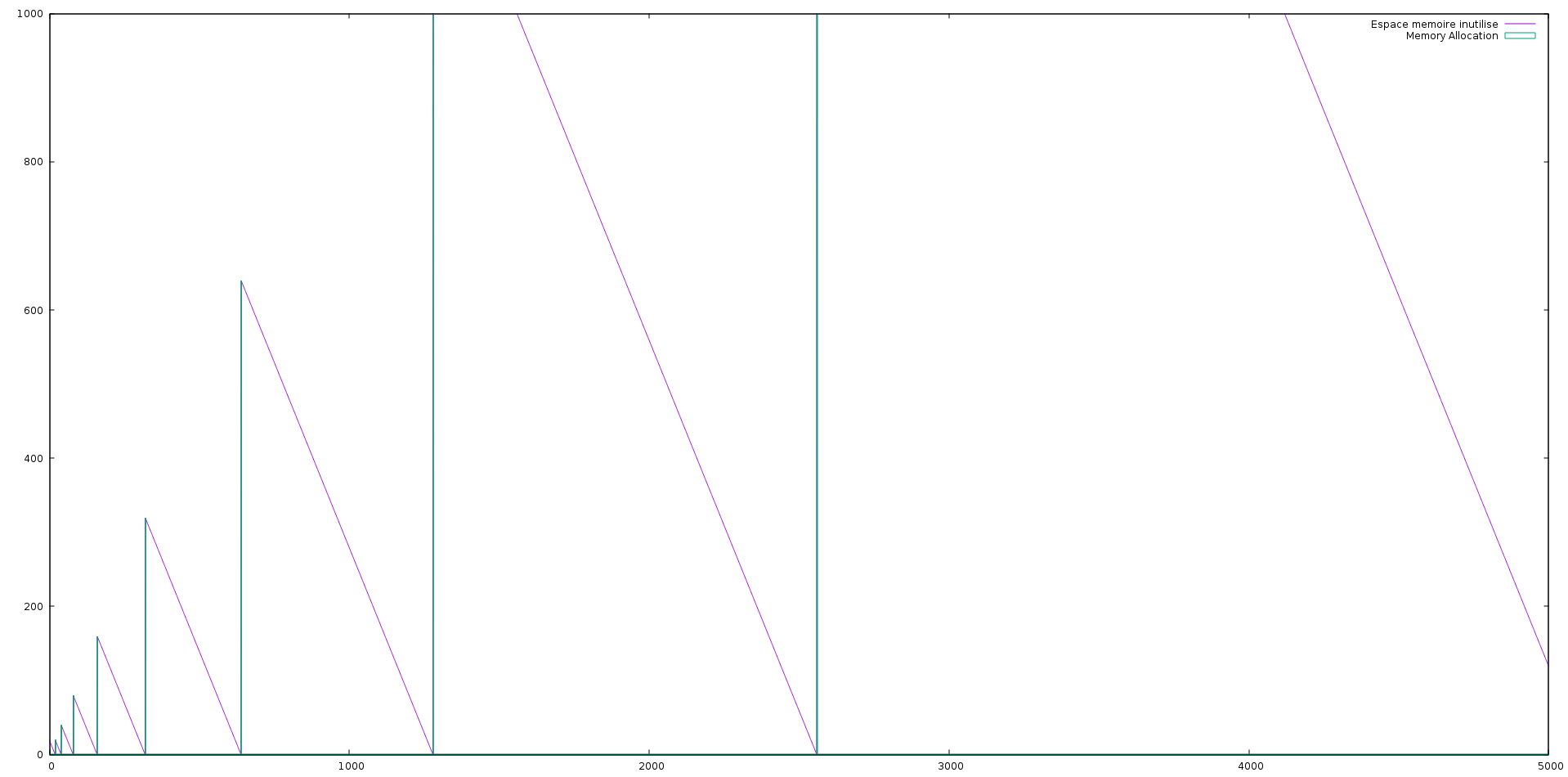


Figure  : Nombre de copies effectuées et espace mémoire inutilisé en fonction du temps réel

Puis on a affiché l’espace mémoire inutilisé en fonction du temps amorti, on remarque que au début on ne gaspille pas énormément de mémoire, mais tant que le tableau est grand tant qu’on gaspille de la mémoire en allouant c’est que le graphe nous montre clairement surtout à l’insertion des valeurs entre l’intervalle 2500 et 5000 le taux de gaspillage passe de 400 à 1000 presque ce qui est énorme.

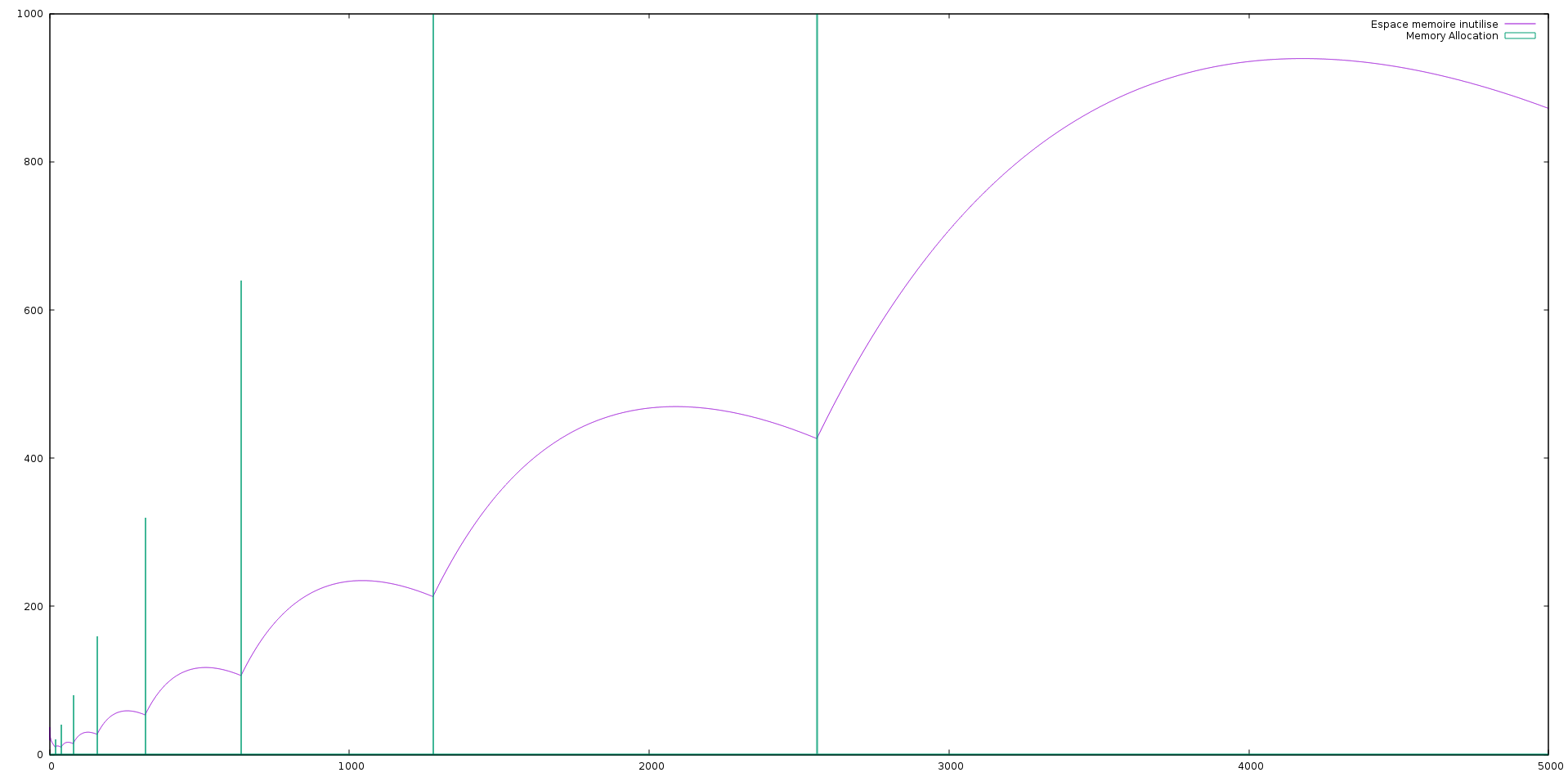


Figure  : Nombre de copies effectuées et espace mémoire inutilisé en fonction du temps amorti

Ensuite on a affiché le nombre de copies, on remarque que le nombre de copies n’augmente qu’au moment de l’allocation mémoire vu que c’est à ce moment là qu’on recopie les valeurs qui existent déjà dans le tableau rempli pour les mettre dans le nouveau tableau. Et le nombre de copies est de plus en plus grand après insertion de plusieurs valeurs, c’est là qu’on fait plus de copies, prenant exemple si on a un tableau plein de 4000 valeurs, on veut insérer une autre valeur, on doit allouer d’abord de la mémoire puis recopier les 4000 valeurs et insérer la 4001 nième.

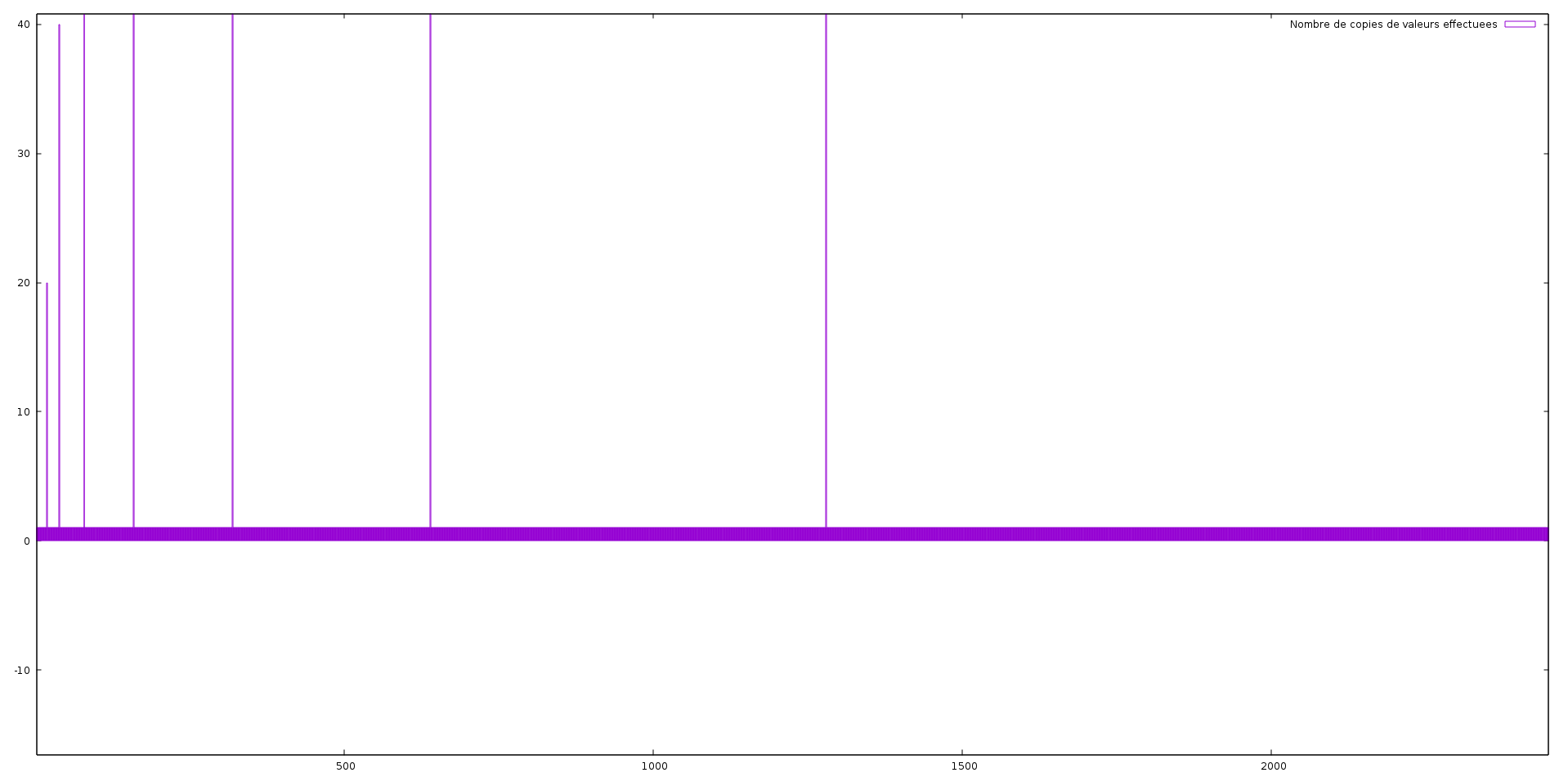


Figure  : Nombre de copies effectuées en fonction du coût amorti pour l’insertion dans l’ordre croissant

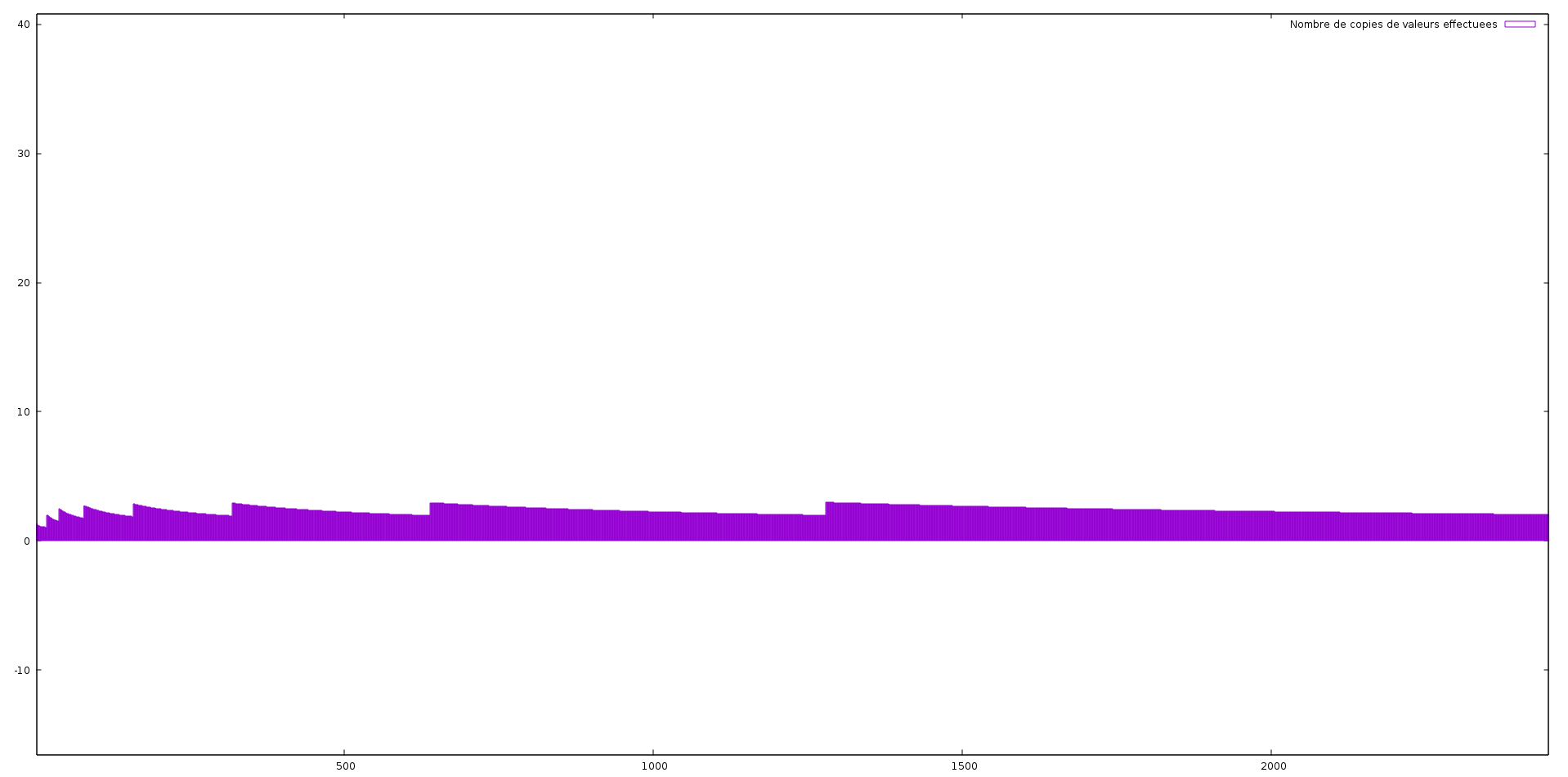


Figure  : Nombre de copies effectuées en fonction du temps réel pour l’insertion des clés dans l’ordre croissant

**Pour alpha = (Nombre d’or)**

On a affiché le coût réel de l’opération d’insertion des valeurs successives ordonnées de façon croissante.

On remarque qu’à chaque allocation mémoire le coût réel augmente énormément et cela est plus clair au début de l’insertion vu que c’est là qu’on fait plus d’allocation mémoire. Comparons notre coût réel à celui du cas statique ! On voit bien que le coût a augmenté il atteint des valeurs supérieures à 600 au moment de l’allocation mémoire, moment où les valeurs du tableau sont copiées.

En comparant avec le coût réel obtenu lors de l’insertion dynamique des valeurs croissantes avec alpha égal à 2, on remarque qu’avec l’alpha qui est égale au nombre d’or, le coût réel est légèrement plus élevé vu qu’on a plus d’allocation mémoire dans notre cas. Le fait que les valeurs sont insérées dans un ordre croissant fait qu’on n’a pas besoin d’entasser notre tas à chaque fois ce qui nous fait gagner en efficacité.

La figure ci-dessous nous montre les variations du coût réel.

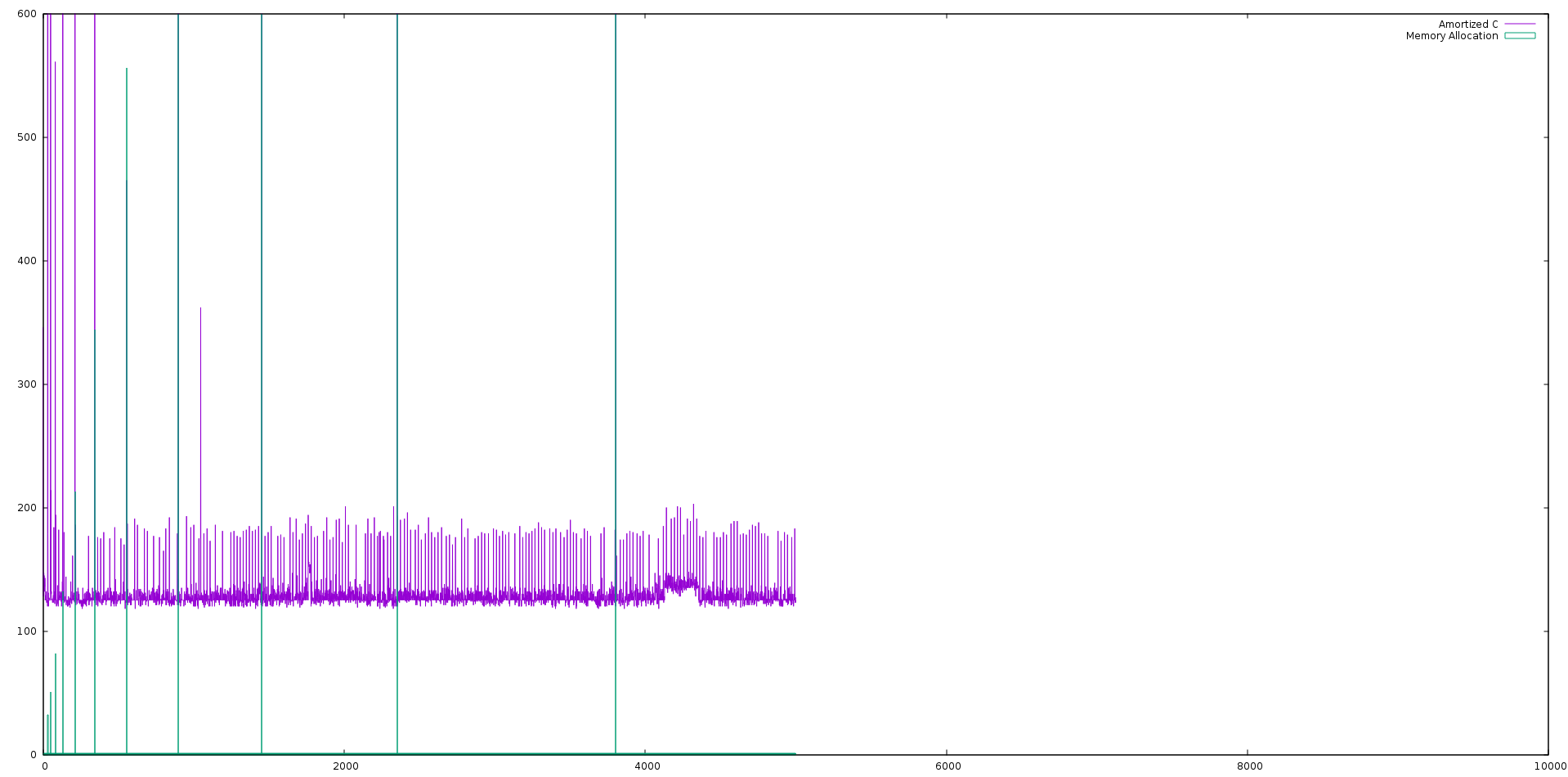


Figure  : Coût réel de l’insertion des clés dans l’ordre croissant avec alpha égal au nombre d’or

Ensuite on a affiché le coût amorti pour la même opération d’insertion, on remarque que le coût amorti au début est décroissant après avoir connu une valeur très élevée, puis il décroit et rencontre une petite augmentation au moment de l’allocation mémoire pour se stabiliser à une valeur entre 150 et 200. Ce qui n’est pas très différent du coût amorti de l’insertions des valeurs croissantes avec alpha égal à 2.

On remarque que le coût amorti se stabilise après plusieurs insertions n’empêche qu’il augmente légèrement lors de l’allocation mémoire et cela dans un intervalle très petite entre 150 et 180 à peu près.

Le coût amorti est plus élevé dans le cas de dynamique que statique pour la même raison précédemment mentionnée dans le coût réel.

La figure ci-dessous nous montre les variations du cout amorti présenté sous forme de graphe.

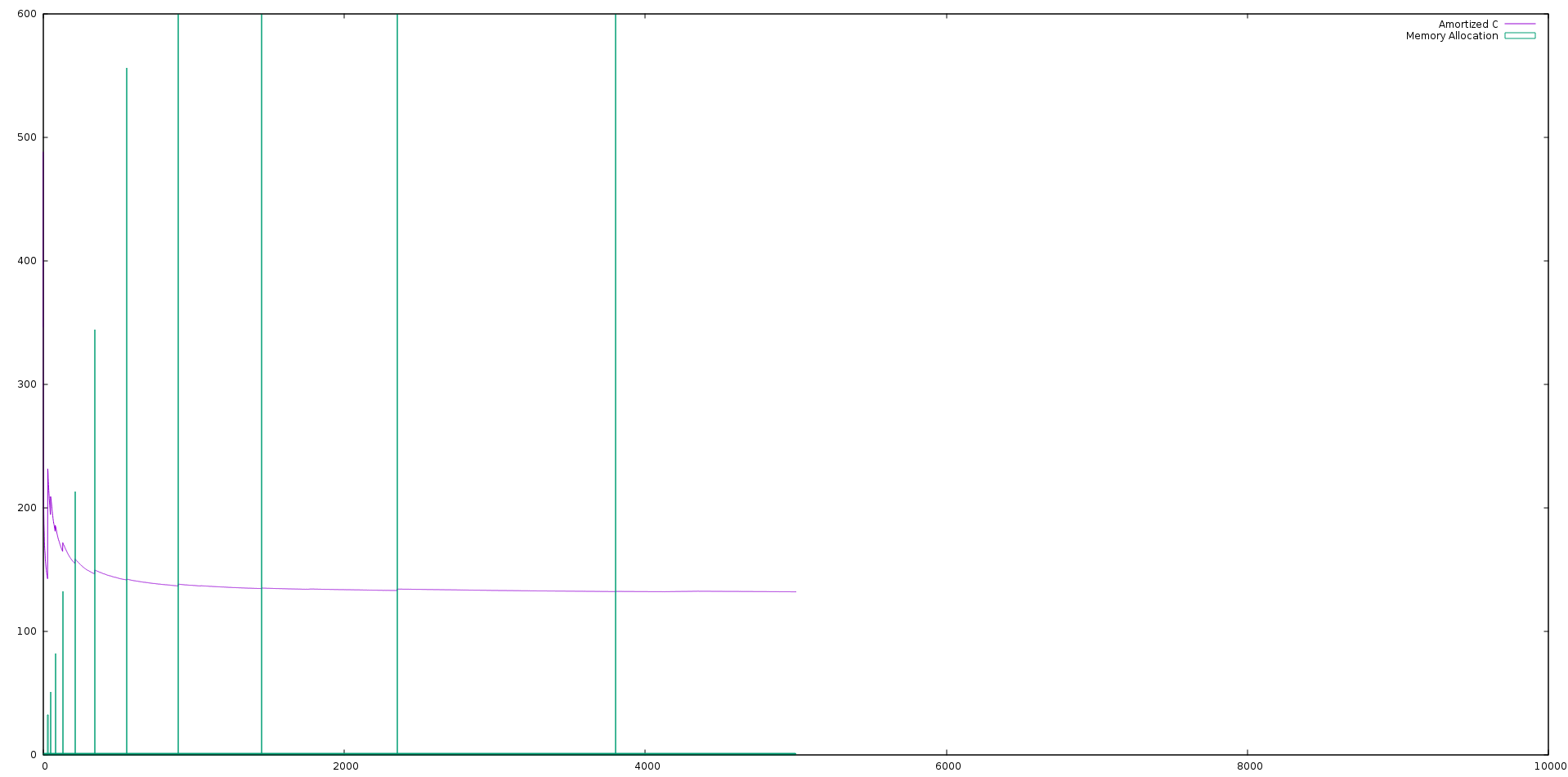


Figure  : coût amorti de l’insertion des clés dans l’ordre croissant pour alpha égal au nombre d’or

Ensuite on a affiché l’espace mémoire inutilisé en fonction du temps réel, on remarque qu’au début on ne gaspille pas énormément de mémoire, mais cela augmente au fur et à mesure qu’on insère les clés. Comparé au cas où on multiplie la taille du tableau par 2, ici on gaspille beaucoup moins en espace mémoire vu qu’on multiplie notre tableau par un nombre inférieur à 2.

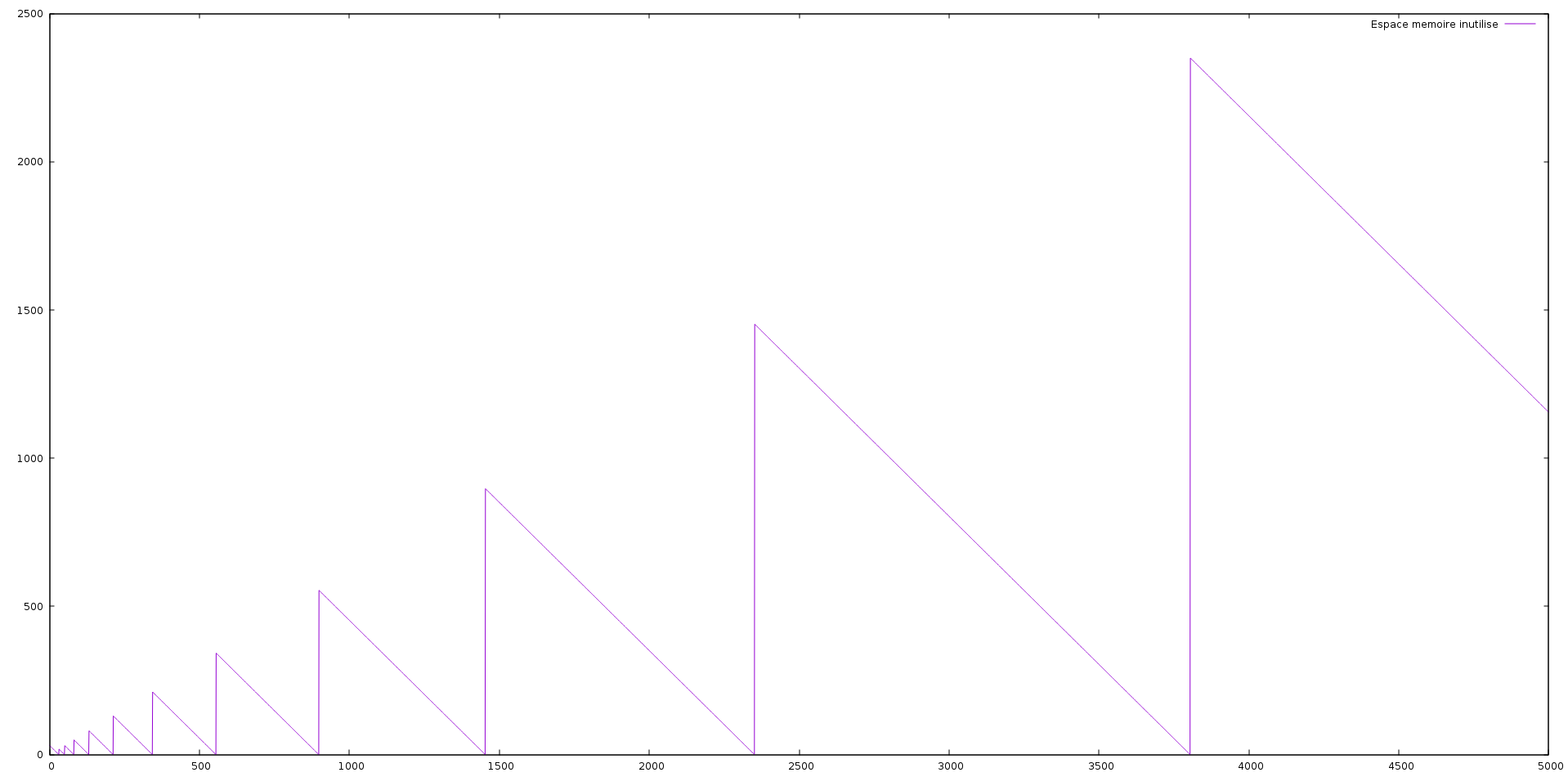


Figure  : Espace mémoire inutilisé en fonction du temps réel pour alpha égal au nombre d’or

Puis on a affiché l’espace mémoire inutilisé en fonction du temps amorti, on remarque que l’espace mémoire gaspillé à énormément baissé comparé au cas où alpha est égal à 2. Le maximum de la valeur de l’espace inutilisé arrive à 500 ce qui est prèsque la moitié dans le cas ou alpha est égale à deux.

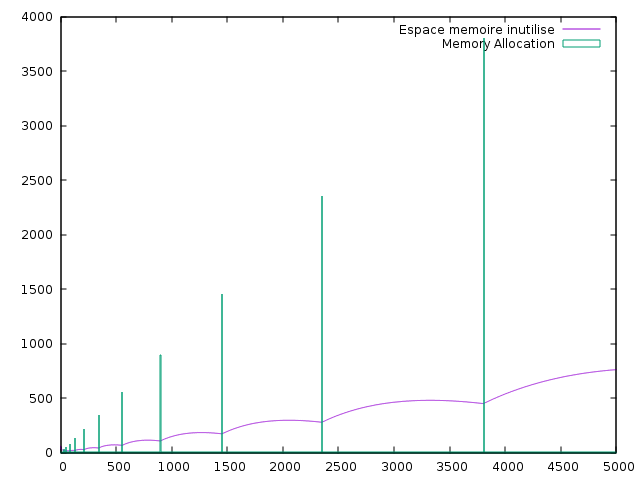


Figure  : Espace mémoire inutilisé en fonction du coût amorti pour alpha égal au nombre d’or

Ensuite on a affiché le nombre de copies effectuées.

On remarque que le nombre de copies n’augmente qu’au moment de l’allocation mémoire ce qui est logique, vu que c’est à ce moment là qu’on recopie les valeurs qui existe déjà dans le tableau rempli pour les mettre dans le nouveau tableau.

Et le nombre de copies quand alpha est égale au nombre d’or est plus grand que celui de notre cas précédant quand alpha est égale à 2, et cela est du au fait qu’ici on fait plus d’allocation mémoire.

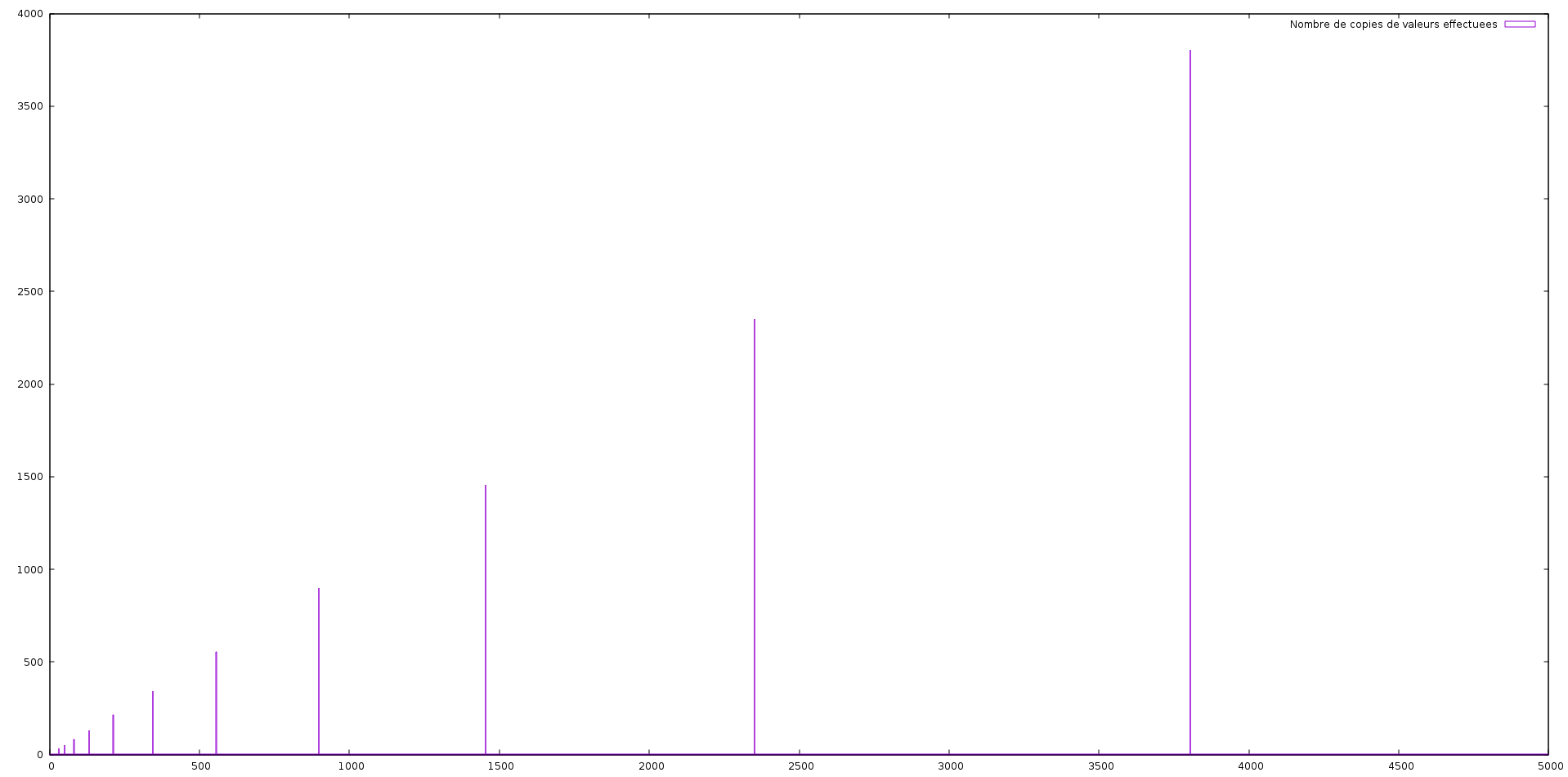


Figure  : Nombre de copies effectuées en fonction du temps réel pour alpha égal au nombre d’or

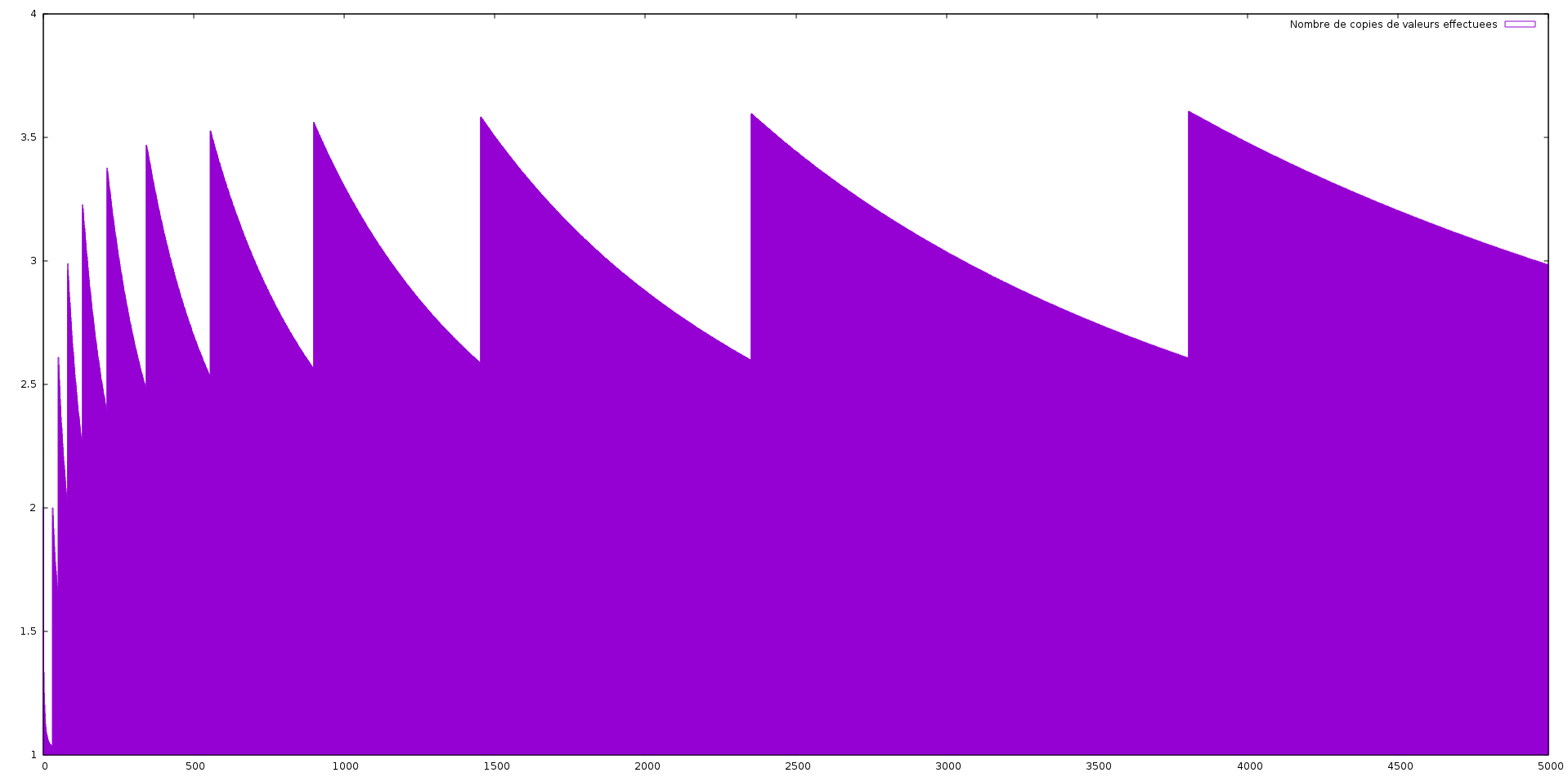


Figure  : Nombre de copies effectuées en fonction du temps amorti pour alpha égal au nombre d’or

**Pour alpha = 20**

On a affiché le cout réel de l’opération d’insertion des valeurs successives ordonnées de façon croissante.

On remarque que le coût réel augmente plus au moment de l’allocation mémoire car c’est à ce moment là qu’on recopie les éléments du tableau. On voit aussi que le coût réel a légèrement baissé comparé aux 2 expériences précédentes, cela est du au fait qu’ici on ne fait pas beaucoup d’allocation mémoire donc on copie moins d’éléments.

Le fait d’insérer les clés dans un ordre croissant fait qu’on n’a pas besoin d’entasser notre tas à chaque fois ce qui nous fait gagner en efficacité.

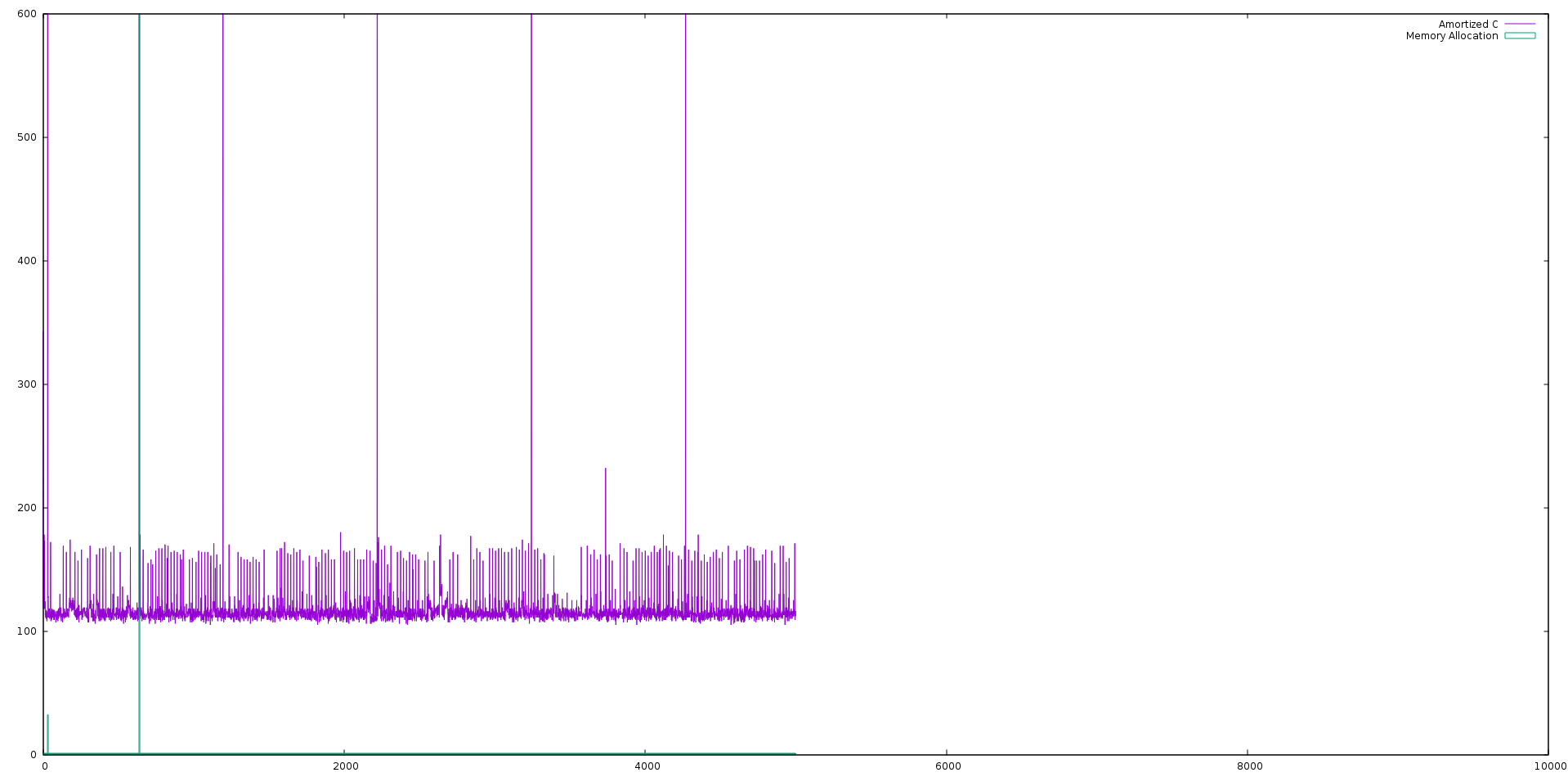


Figure  : Coût réel de l’insertion des clés dans l’ordre croissant pour alpha égal à 20

Ensuite on a affiché le coût amorti pour la même opération d’insertion, on remarque que le coût amorti a légèrement baissé comparé au cas où on multiplie la taille du tableau par 2 ou par le nombre d’or, cela s’explique par le fait qu’on ne copie pas beaucoup d’éléments dans ce cas. Le seul moment où le coût amorti rencontre une grande augmentation c’est quand on fait une allocation mémoire.

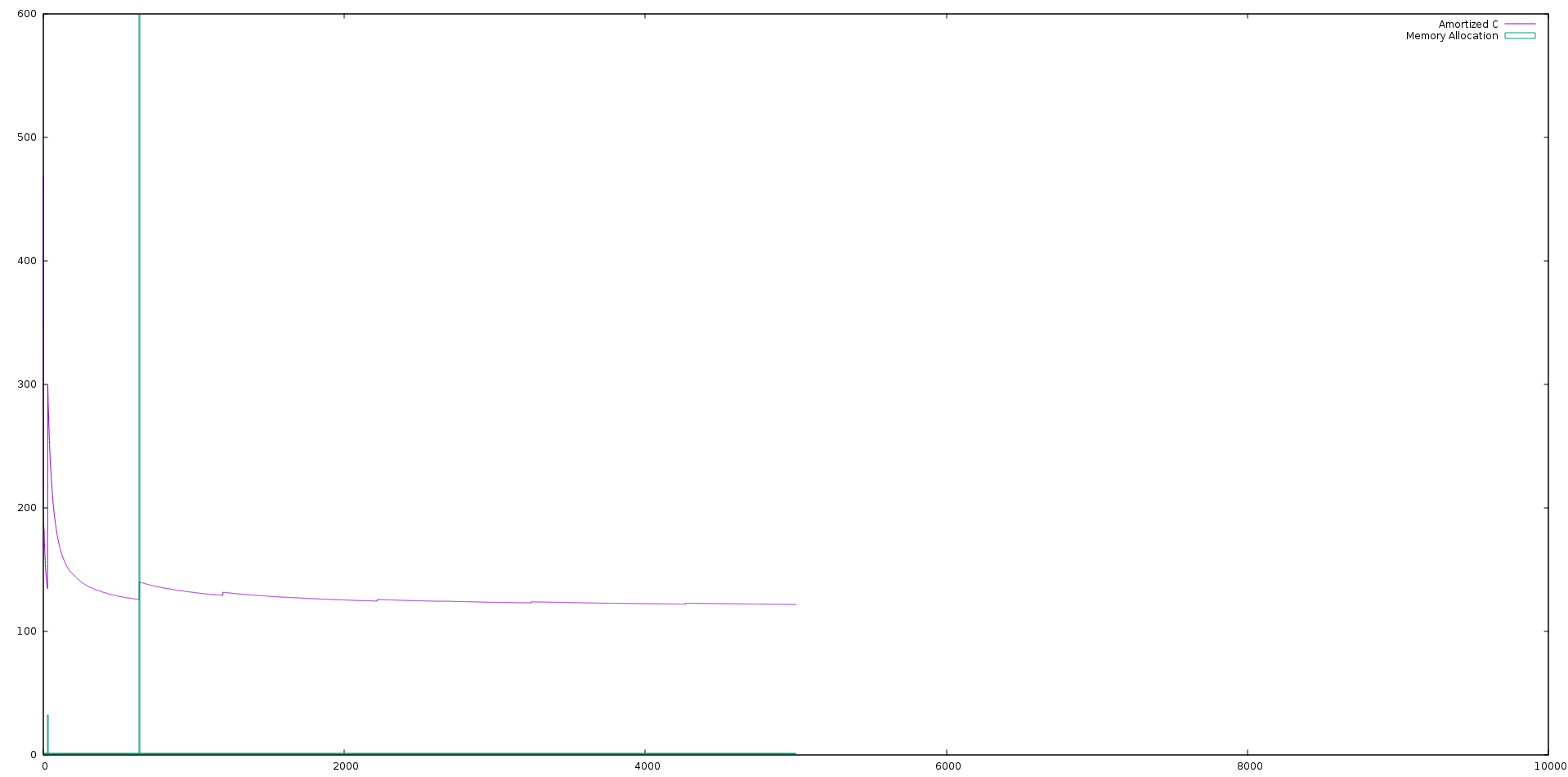


Figure  : Coût amorti de l’insertion des clés dans l’ordre croissant pour alpha égal à 20

On a maintenant affiché l’espace mémoire inutilisé en fonction du temps réel, on remarque que l’espace mémoire gaspillé à considérablement augmenté par rapport au deux cas précédents, ceci est normal car on multiplie la taille du tableau par 20.

Prenons exemple en arrivant à l’insertion des valeurs entre 4000 et 5000 la valeur de l’espace inutilisé augmente jusqu’à des valeurs qu’on n’arrive pas à observer dans le graphe avec gnuplot les valeurs sont tellement grandes.

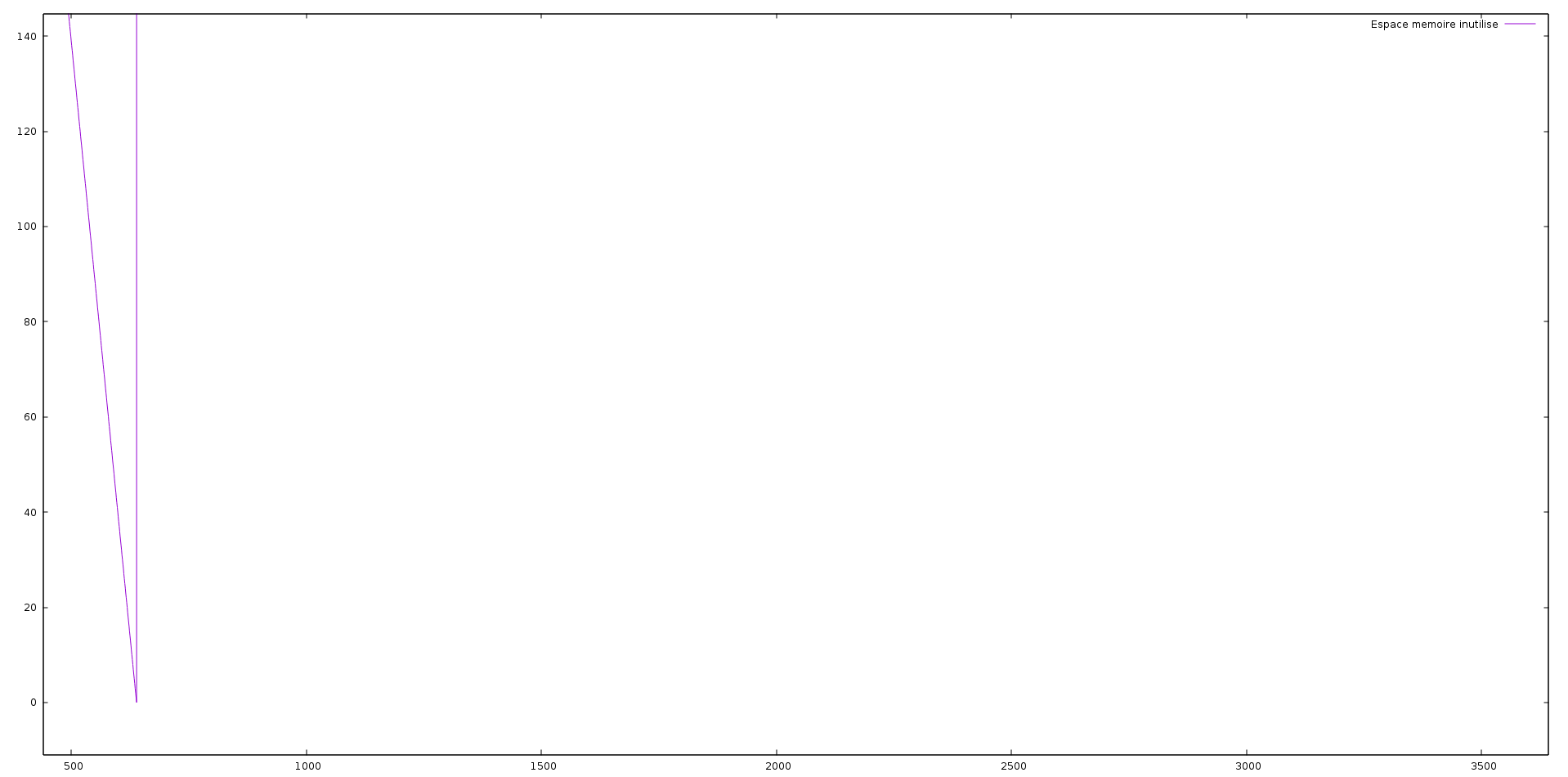


Figure  : Espace mémoire inutilisé en fonction du coût réel pour alpha égal à 20

Ensuite on a affiché le nombre de copies effectuées lors de l’insertion, on remarque que le nombre de copies n’augmente qu’au moment de l’allocation mémoire car c’est à ce moment là qu’on recopie tous les éléments du tableau.

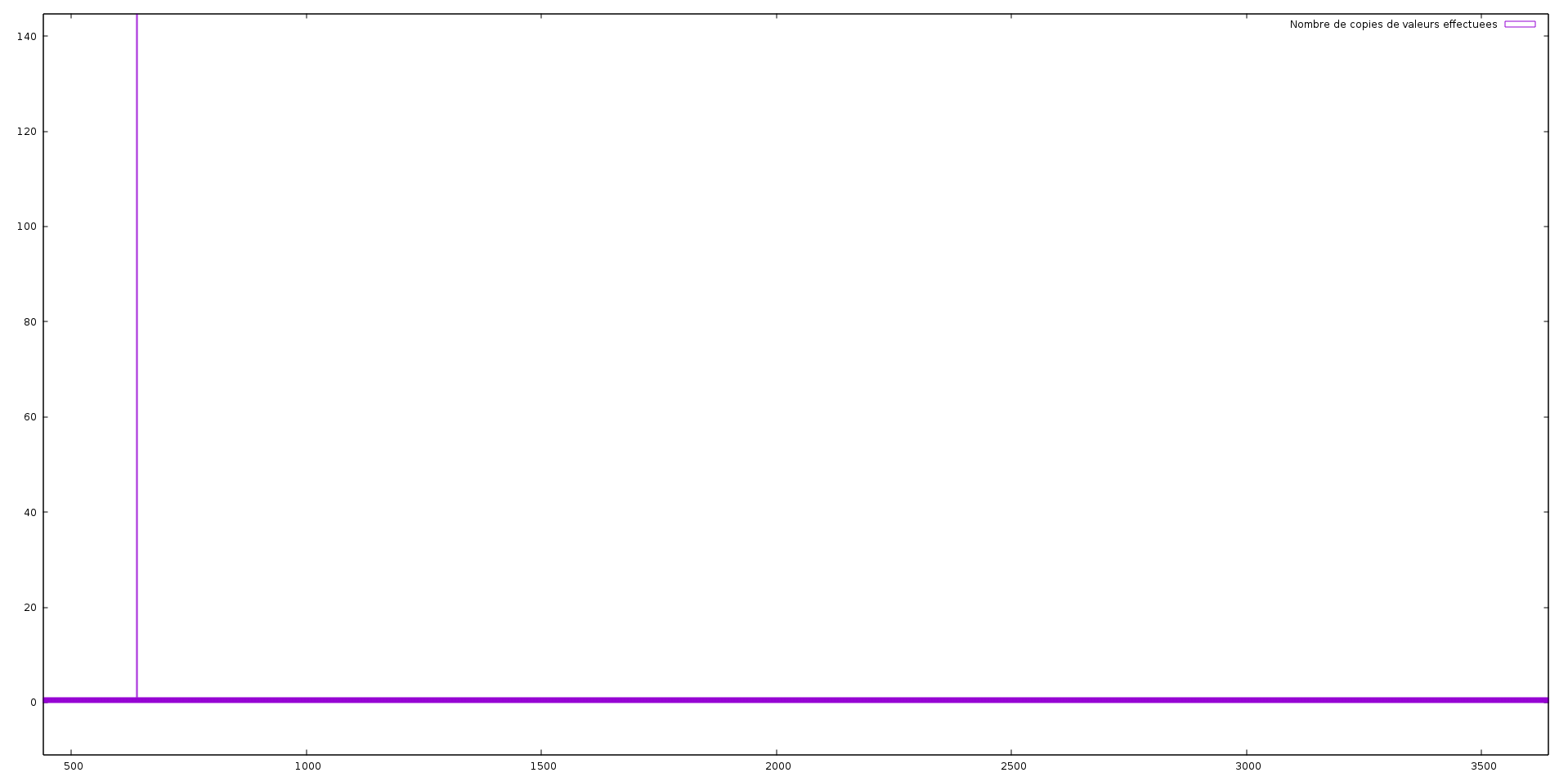


Figure  : Nombre de copies effectuées pour l’insertion croissante pour alpha égal à 20

**Résumé :**

On déduit de ces expériences que l’espace mémoire inutilisé change selon le alpha choisit, il augment dès que alpha est grand et baisse sinon, par contre le coût amorti ou réel quant à eux, ils ne varient pas beaucoup, on voit qu’il baisse dès que alpha est grand mais cette baisse est presque pas remarquable. Entre ces trois expériences la plus intéressante est celle où on multiplie par le nombre d’or car elle nous permet de gagner en espa=ce sans perdre en efficacité.

**Insertion des clés dans l’ordre décroissant :**

**Pour alpha = 2**

Nous avons effectué des insertions de clés dans l’ordre décroissant tout en doublant la taille du tableau à chaque fois que celui-ci est plein.

Nous avons ensuite commencé notre analyse par l’affichage du coût amorti et de l’allocation mémoire de ces opérations sur gnuplot.

On remarque d’après le graphe suivant que le coût amorti des opérations augmente à chaque allocation mémoire car tous les éléments du tableau sont recopiés à chaque fois que le tableau est doublé, ce qui coute cher en temps amorti. C’est ce que nous avons vu en détails dans le rapport sur les tableaux dynamiques.

La copie des éléments n’est pas la seule raison pour laquelle le temps d’exécution est élevé, le fait d’insérer des valeurs décroissantes fait aussi que le coût augmente et c’est ce qu’on a expliqué dans le rapport sur les tas binaires statiques.

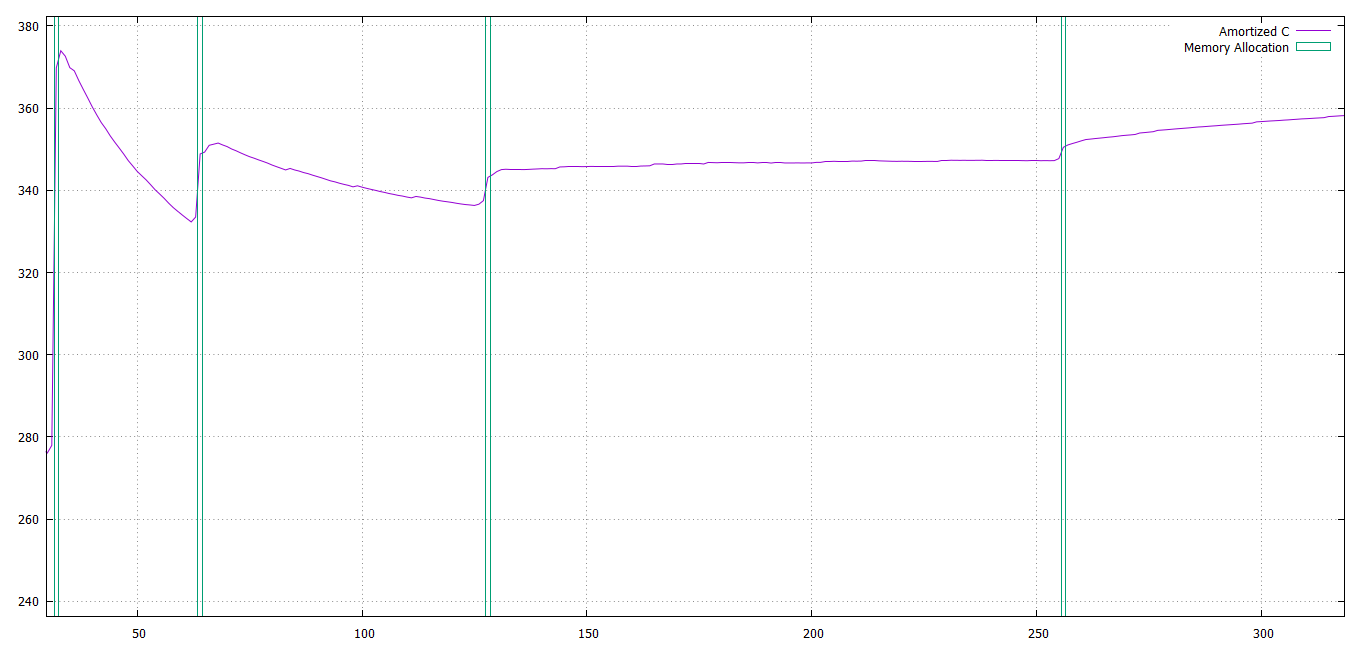


Figure  : Coût amorti de l’insertion des clés dans l’ordre croissant.

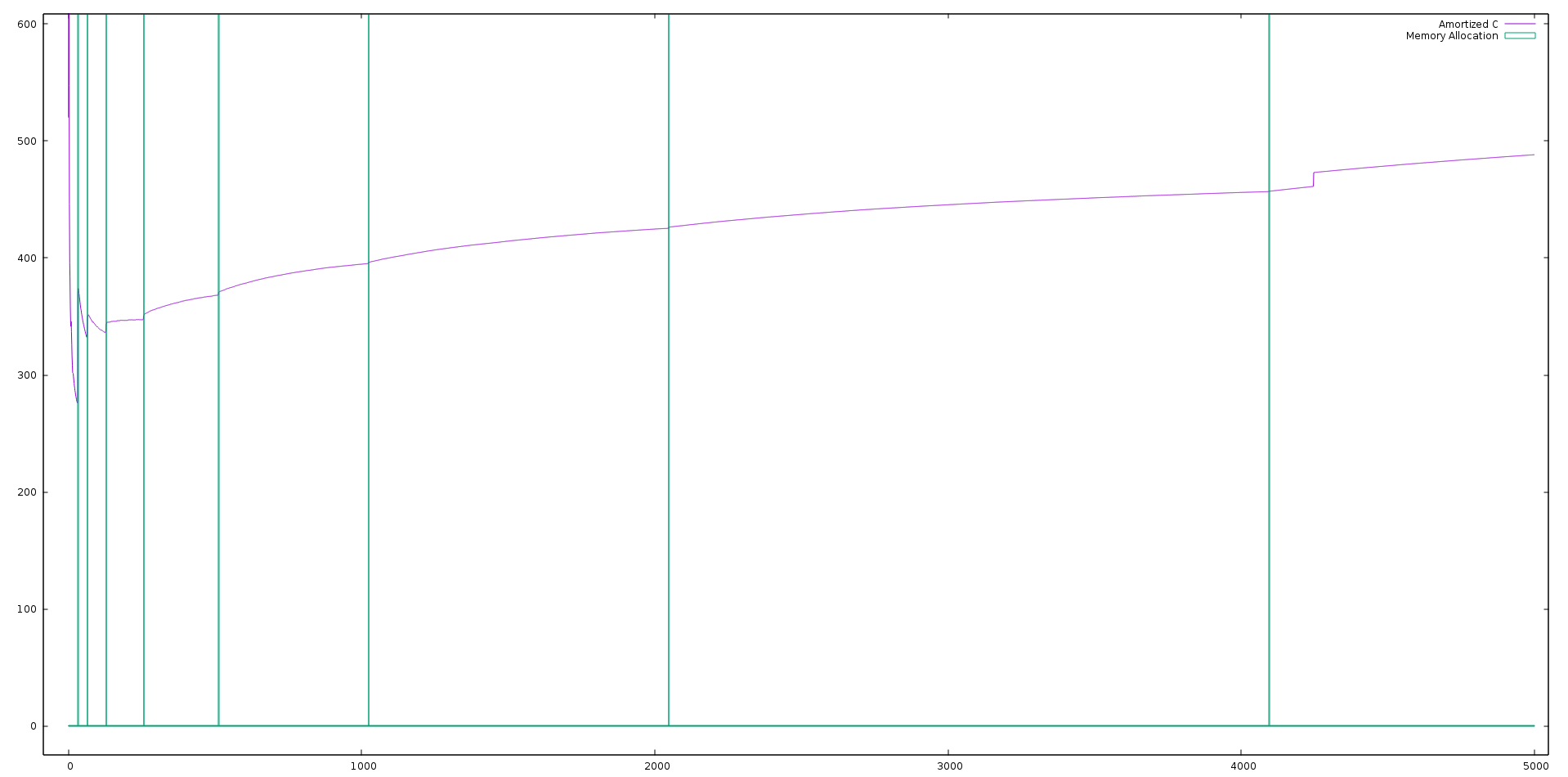


Figure  : Coût amorti de l’insertion des clés dans l’ordre décroissant

Pour voir ce qui se passe lors de chaque insertion nous avons affiché le coût réel des opérations car à l’inverse du coût amorti, qui lui est en théorie constant, le coût réel vari beaucoup et dépend de la structure de donnée.

Ce graphe obtenu nous confirme cela, car on voit que le coût réel augment à chaque insertion car on doit faire appel à la fonction qui entasse vers le haut vu que l’élément à insérer est plus petit que le précédent. Ce n’est pas tout, on a dit précédemment que le coût réel dépend de la structure de données, si on regarde bien ce graphe on voit que c’est tout à fait le cas, car à chaque allocation mémoire, c'est-à-dire à chaque fois que la structure du tableau change, le temps réel d’exécution lui aussi change et augmente de plus en plus

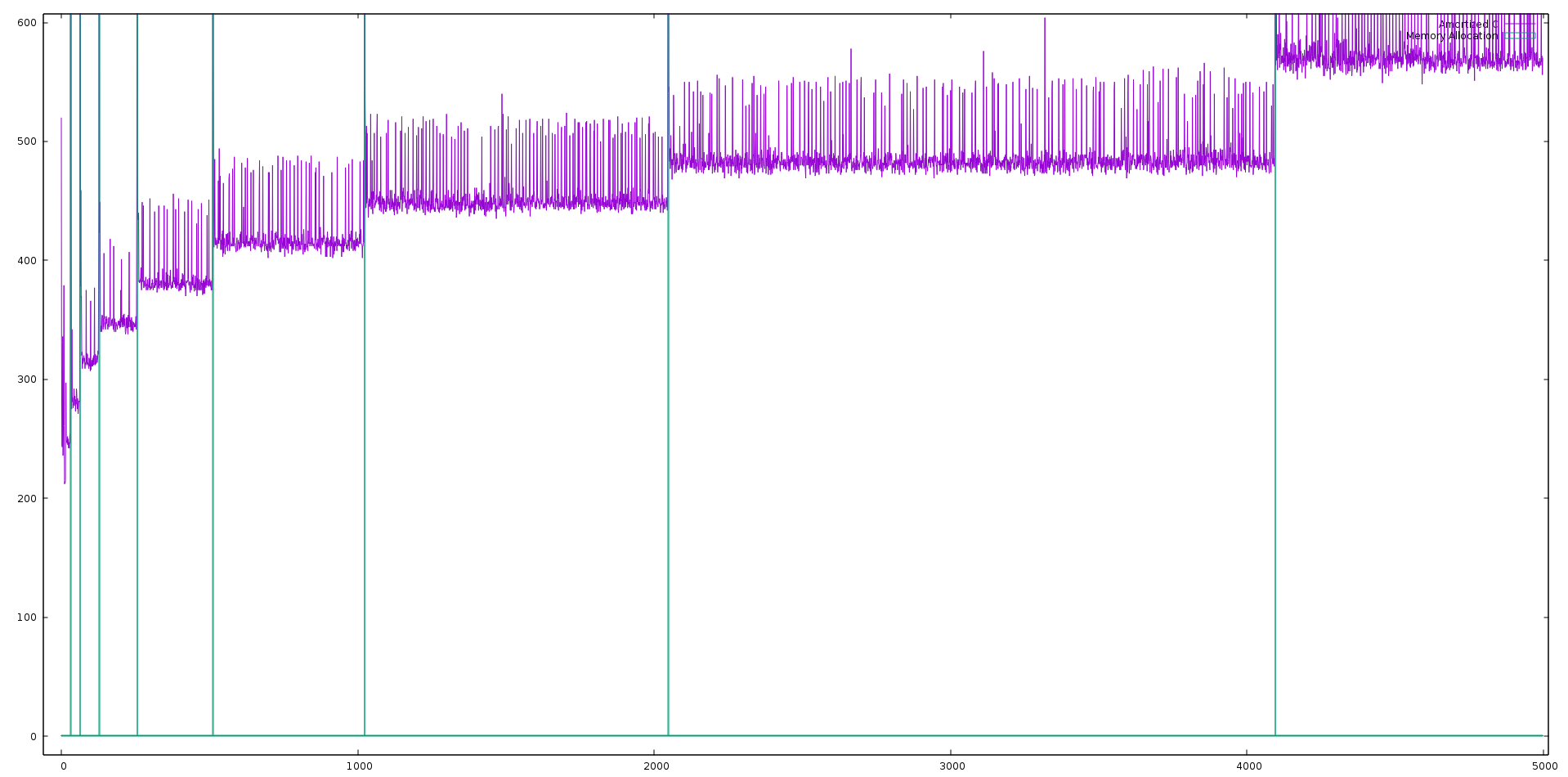


Figure  : Coût réel de l’insertion des clés dans l’ordre décroissant avec réallocation de la mémoire

Nous avons regardé aussi le gaspillage de l’espace mémoire. On remarque d’après ce graphe que l’espace mémoire inutilisé augmente au fur et à mesure qu’on insère de nouvelles clés dans le tas et cela est causé par le fait qu’on double la taille le tableau à chaque fois que celui-ci est plein.

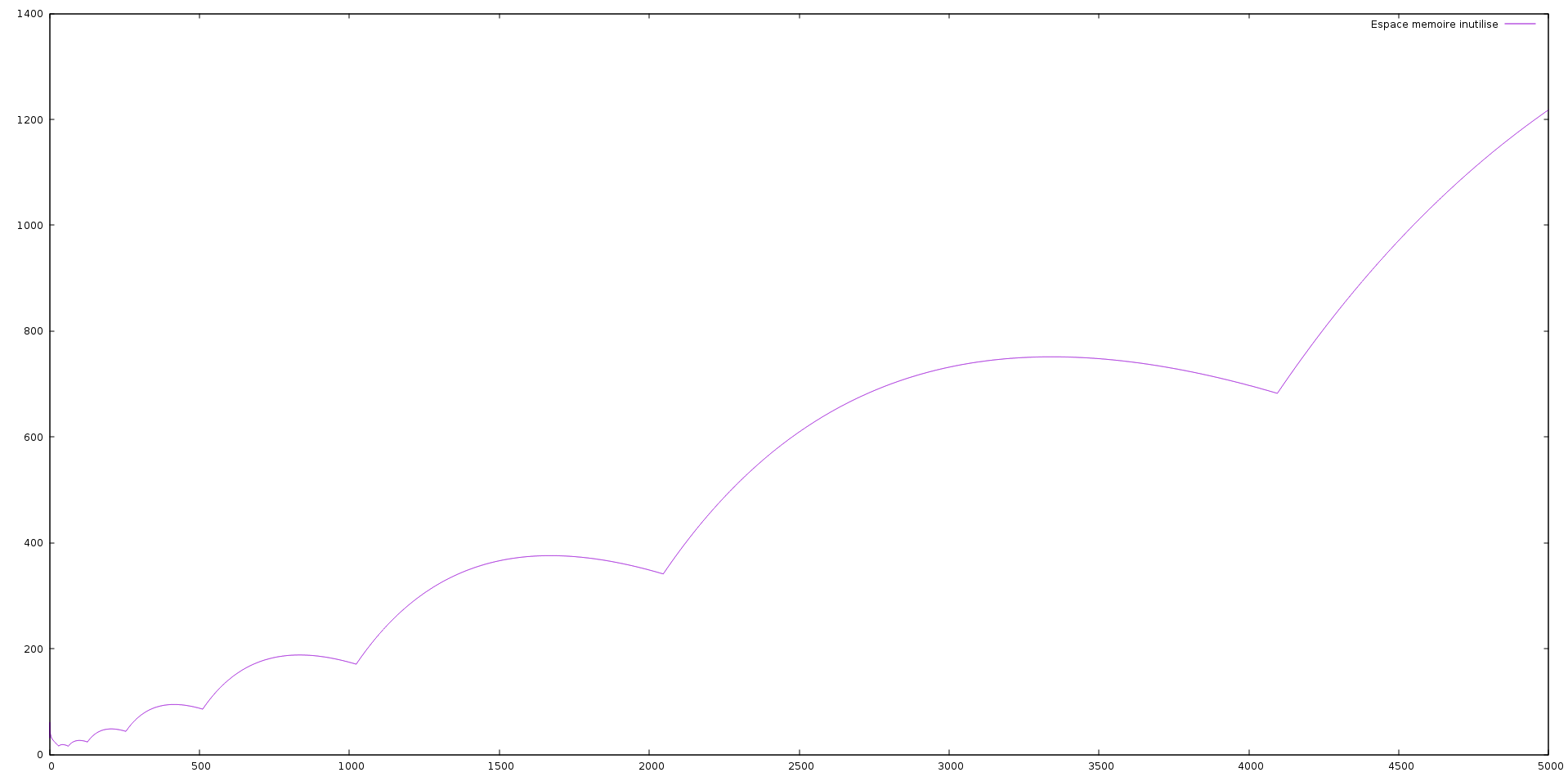


Figure  : Espace mémoire gaspillé, insertion des clés dans l’ordre décroissant

Pour comparer le temps d’exécution amorti (nombre de copies effectuées) avec l’espace mémoire inutilisé nous avons affiché le graphe suivant.

On remarque que l’espace mémoire gaspillé augmente très rapidement et surtout au moment où on double la taille du tableau

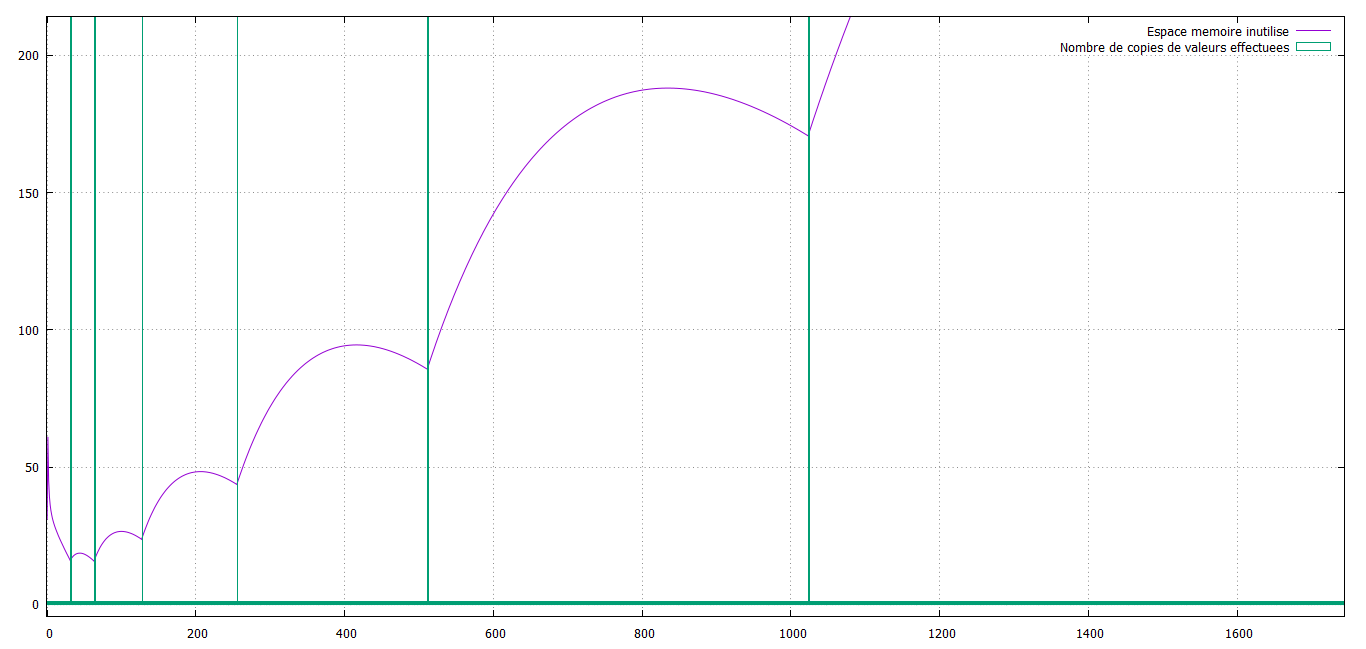


Figure  : Nombre de copies effectuées et espace mémoire gaspillé en fonction du temps amorti.

En affichant le nombre de copies effectuées et l’espace mémoire inutilisé en fonction du temps réel on remarque la meme chose que dans le graphe précédent sauf qu’ici on voit aussi que la mémoire gaspillée baisse à un moment donné avant d’augmenter à nouveau et cela se passe après l’allocation mémoire.

On pense donc que meme si la mémoire inutilisée augmente, il y a toujours un moment où elle baisse.

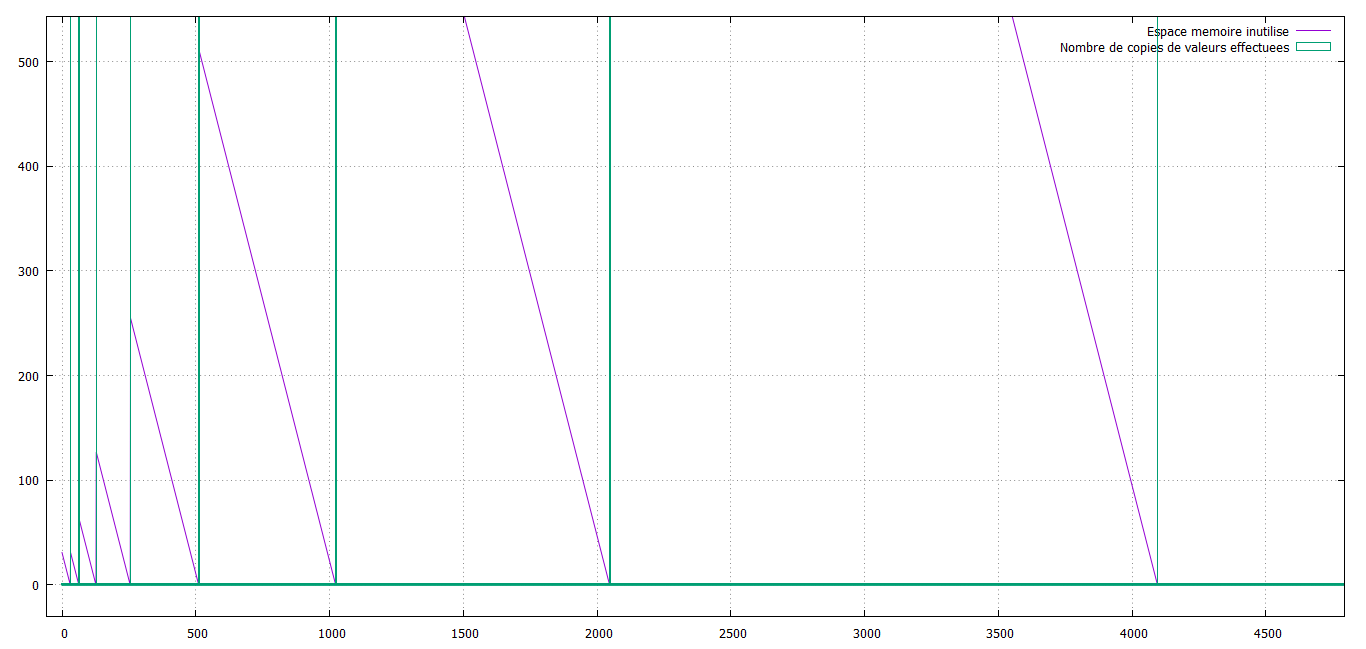


Figure  : Nombre de copies effectuées et espace mémoire gaspillé en fonction du temps réel.

**Pour alpha = (Nombre d’or)**

Comme nous travaillons avec des tableaux dynamiques pour l’implémentation de notre structure de tas alors on multiplie la taille du tableau par un nombre alpha à chaque réallocation, dans l’expérience précédente on a utilisé alpha=2 et on s’est rendu compte qu’on ne faisait pas beaucoup de réallocation par contre on perdait un peu en mémoire.

On va donc dans cette deuxième expérience tester l’efficacité de la structure avec alpha égal au nombre d’or.

Pour cela on commence par afficher le coût amorti.

En comparant ces résultats à ceux obtenus dans le cas où alpha = 2 on remarque qu’il n’y a pas une grande différence.

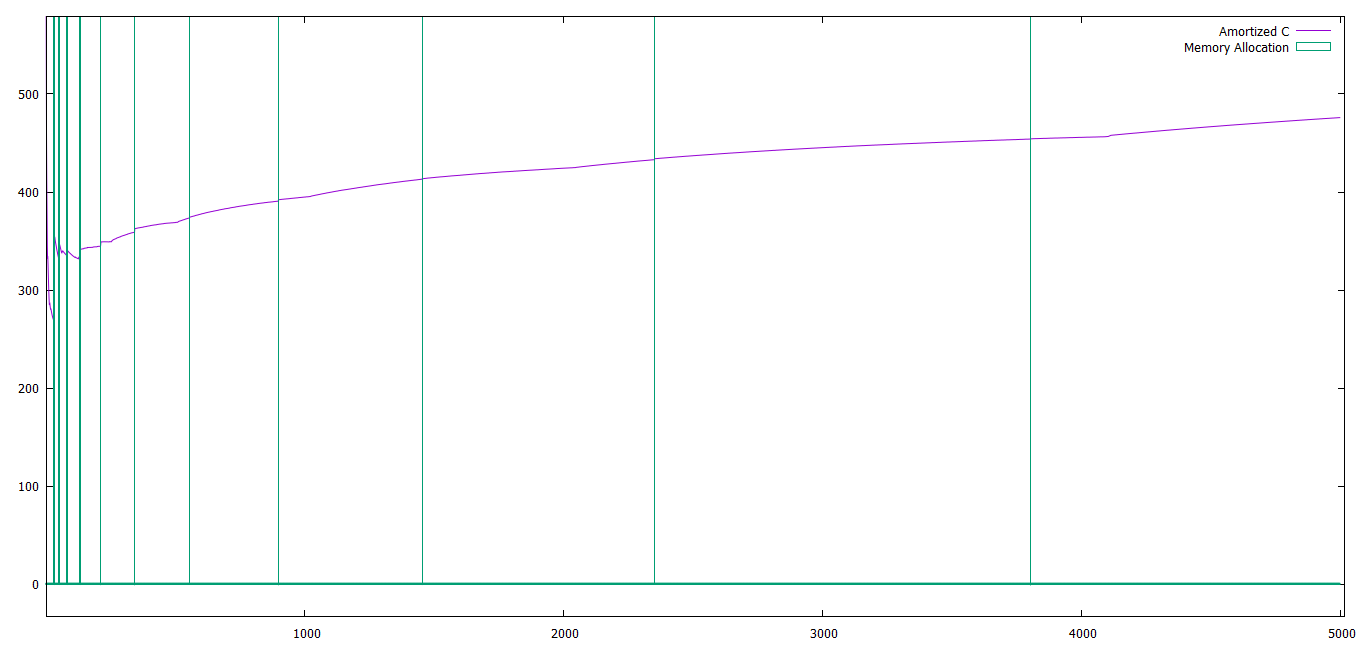


Figure  : coût amorti de l’insertion des clés dans l’ordre décroissant pour alpha égal au nombre d’or

On a aussi affiché le coût réel des insertions.

On remarque que le coût réel est légèrement plus élevé que dans le cas ou alpha = 2 et cela est du au fait que l’on multiplie la taille du tableau par un nombre plus petit que 2, ce qui veut dire qu’on fait plus de copies, d’où le fait que le coût réel soit un peu plus haut.

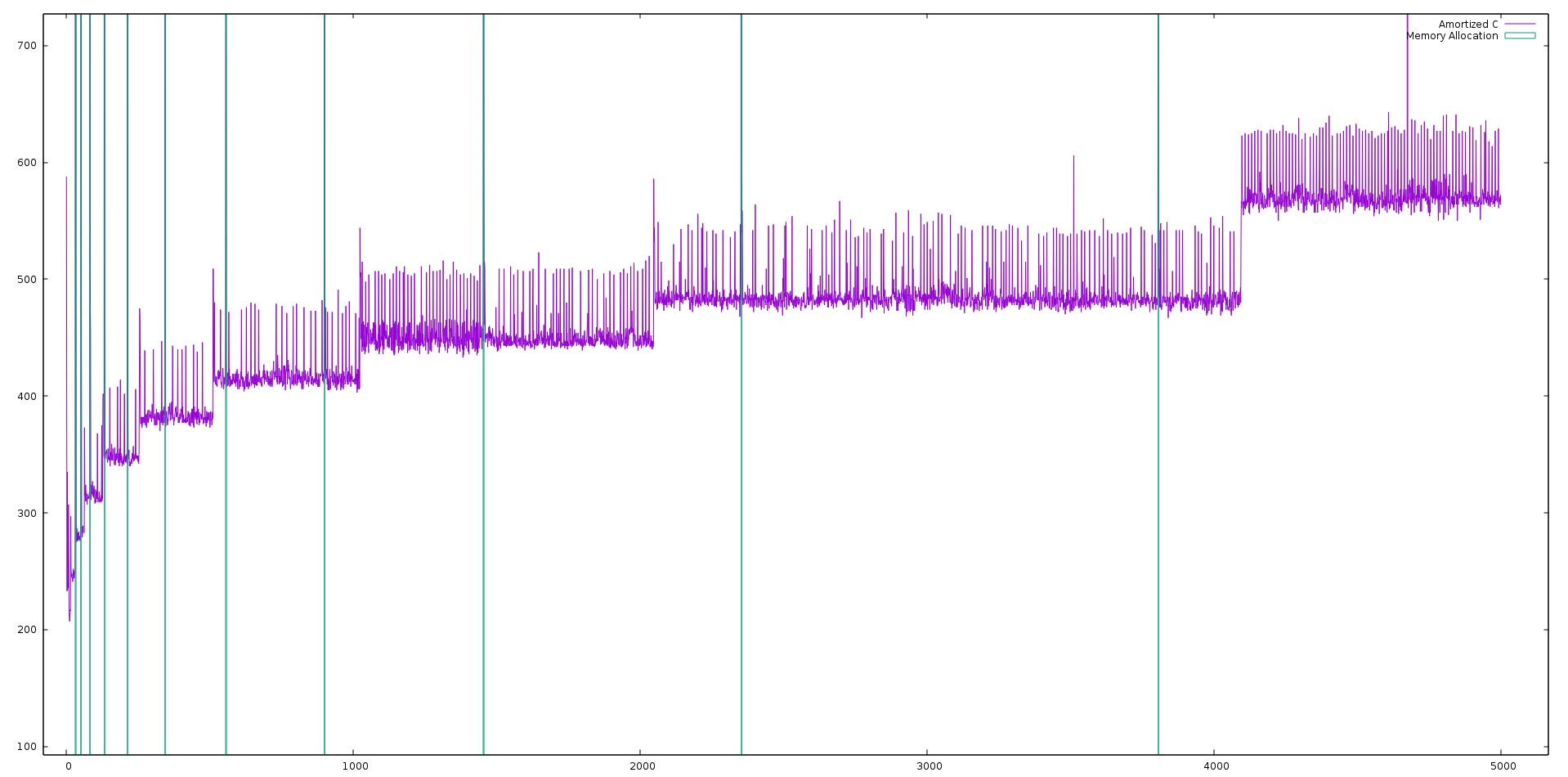


Figure  : Coût réel de l’insertion des clés dans l’ordre décroissant pour alpha égal au nombre d’or

On va maintenant s’intéresser à l’espace mémoire inutilisé.

On remarque ici que la mémoire gaspillée a baissé comparé au cas où on multiplie le tableau par 2, cela s’explique par le fait que l’on multiplie la taille du tableau par un nombre plus petit, ce qui veut dire qu’on gaspille moins de mémoire.

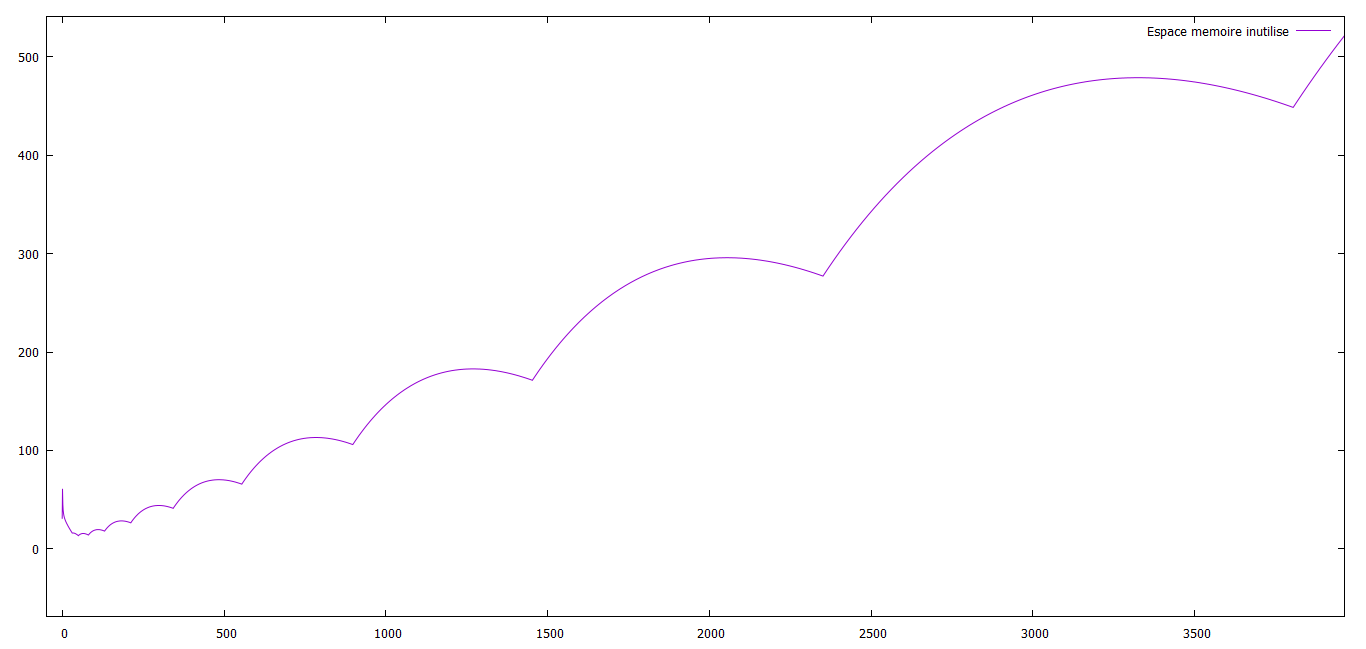
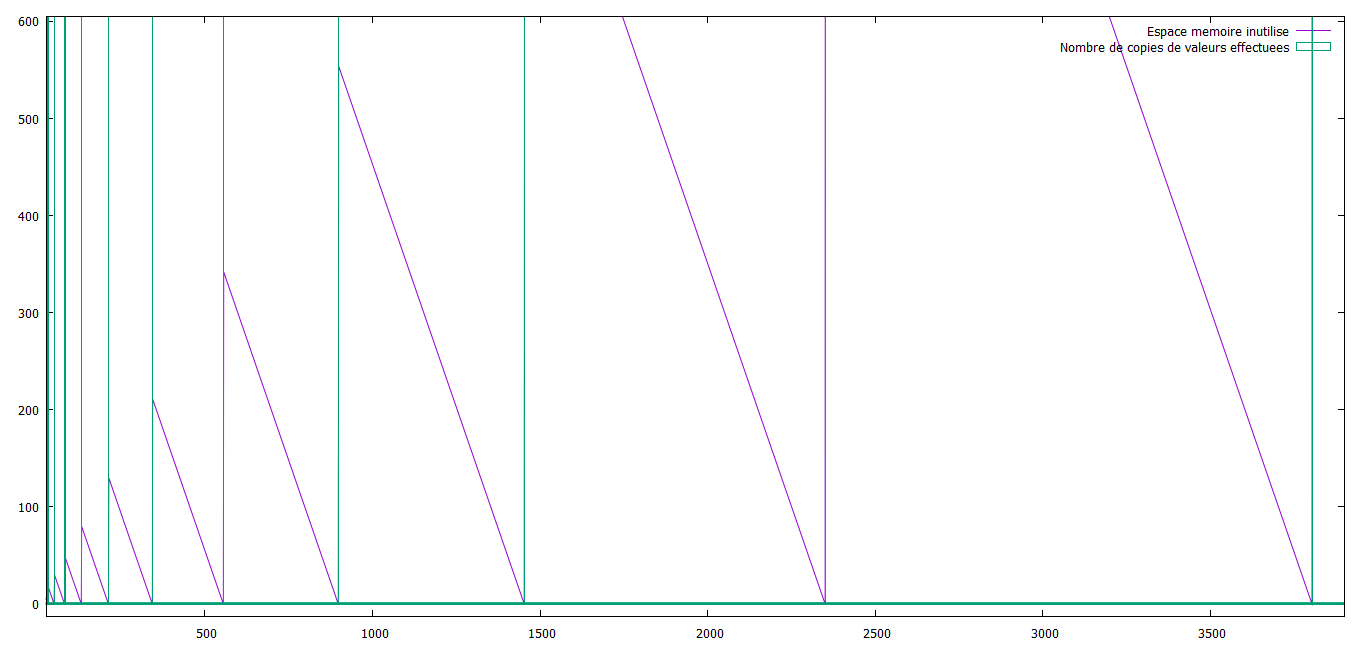


Figure  : Espace mémoire gaspillé pour l’insertion de clés dans l’ordre décroissant pour alpha égal au nombre d’or

On a aussi affiché le nombre de copies effectuées et l’espace mémoire inutilisé en fonction du temps réel, on remarque qu’il ya un peu plus d’allocation mémoire (nombre de fois qu’on effectue des copies) mais aussi le l’espace mémoire inutilisé en temps réel a considérablement baissé, on peut donc constater qu’on a gagné en mémoire sans trop perdre en temps d’exécution.



**Pour alpha = 50**

Après avoir choisi des valeurs petites pour alpha, on veut voir ce qui se passe quand alpha est très grand.

On remarque que pour l’insertion des 5000 clés il y a eu une seule allocation mémoire uniquement ce qui est plutôt logique car la taille du tableau a été multiplier par 50 ce qui donne un tableau très grand.

Le coût amorti a baissé comparé au cas où on multiplie par 2 ou par le nombre d’or, ses valeurs sont passées de « entre 400 et 500 » à « entre 300 et 500 ». En effet lorsqu’on multiplie par un alpha très grand on n’a pas besoin de réallouer de la mémoire à plusieurs reprises et ainsi on n’aura pas à recopier les éléments du tableau, ce qui fait que le coût amorti baisse.

Le coût amorti a baissé certes mais pas comme on l’aura imaginé, car en passant de alpha égal à 2 à alpha égal à 50 le coût amorti a seulement baissé de 100 unités, il y a donc quelque chose qui le maintient haut. En effet on est toujours dans le cas d’une insertion de clés dans l’ordre décroissant donc ces valeurs sont plutôt cohérentes.

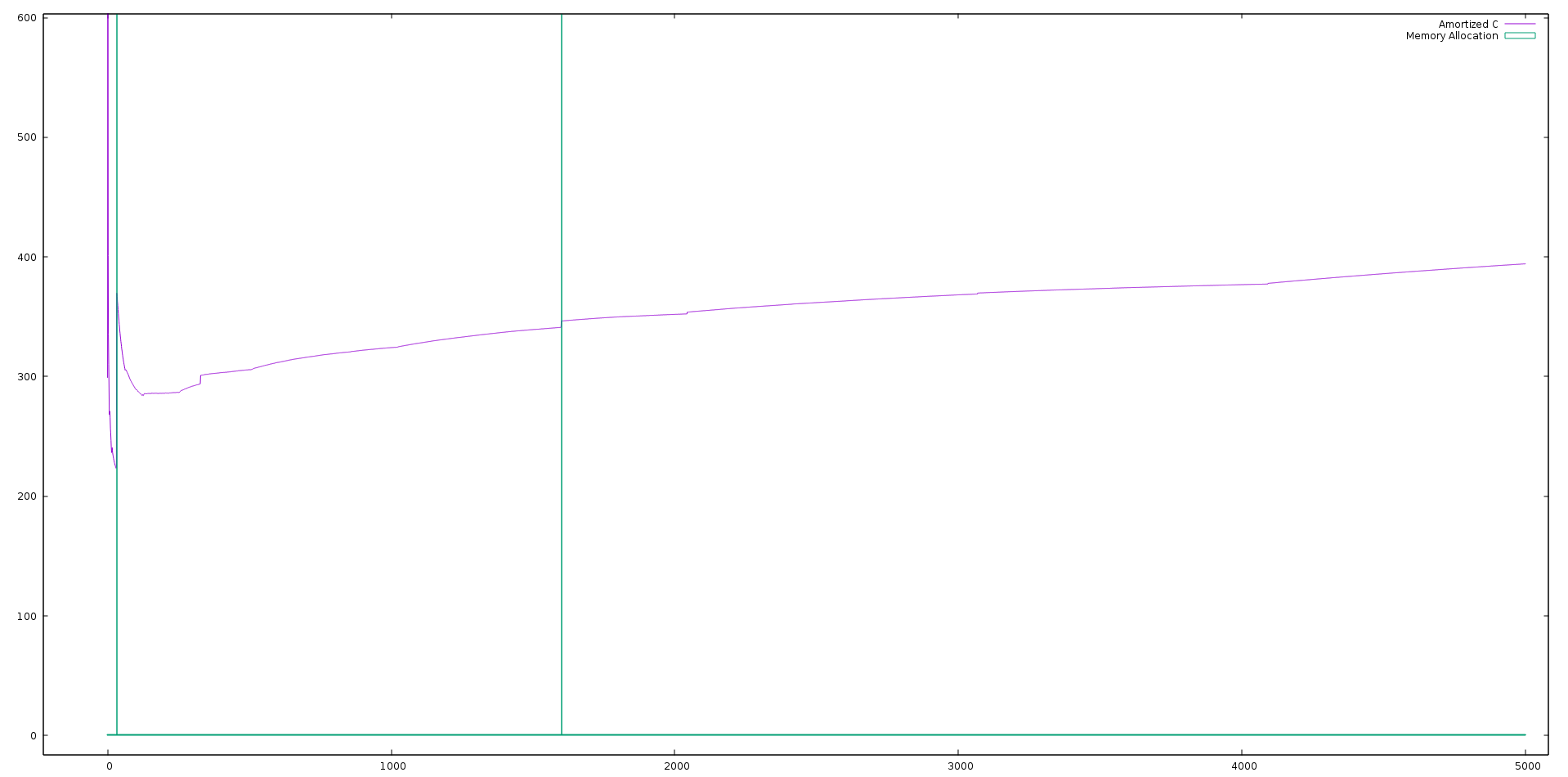


Figure  : coût amorti de l’insertion des clés dans l’ordre décroissant pour alpha égal à 50

Pour ce qui est du coût réel, on voit qu’il est toujours aussi perturbé qu’avant lors de chaque insertion, néanmoins on remarque que ses valeurs ont un peu baissé aussi tout comme le coût amorti et les raisons sont les mêmes bien évidemment.

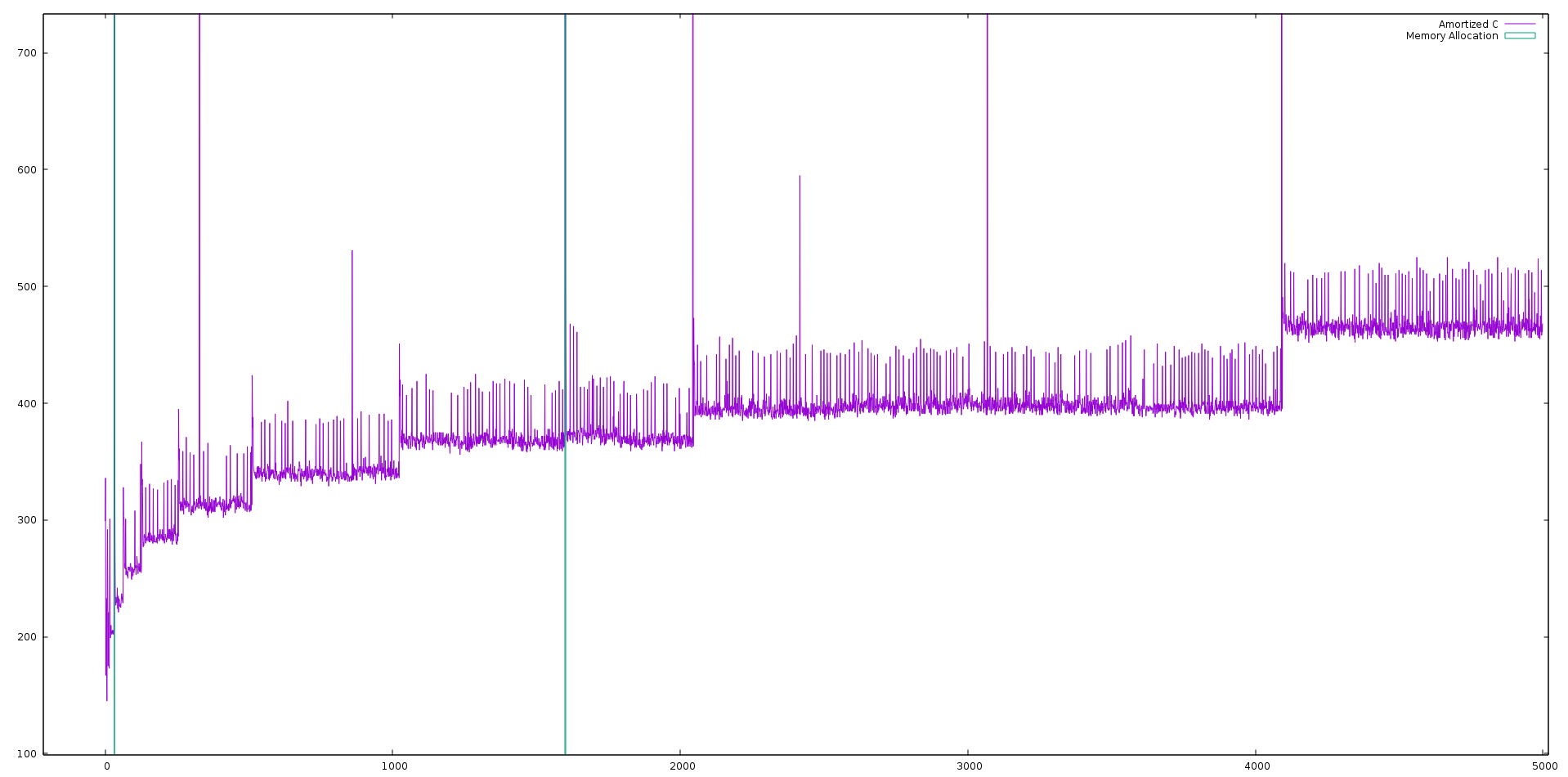


Figure  : Coût réel de l’insertion des clés dans l’ordre décroissant pour alpha égal à 50

En ce qui concerne l’espace mémoire inutilisé on voit très bien qu’il atteint des valeurs énormes. Ce n’est donc pas une bonne idée de choisir un alpha aussi grand.

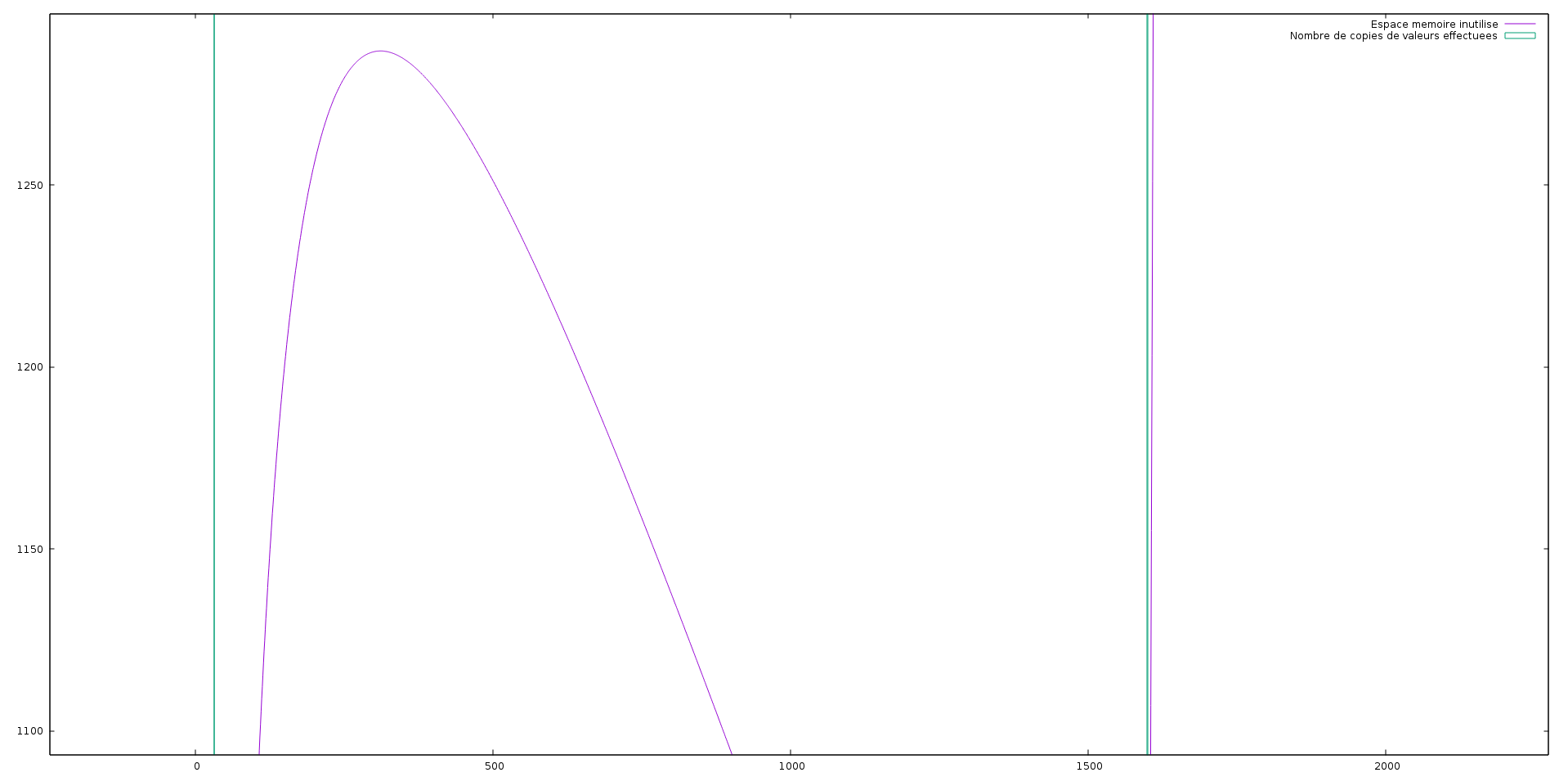


Figure  : Nombre de copies effectuées et espace mémoire inutilisé en fonction du coût amorti

**Résumé :**

Après avoir fait toutes ces expériences on remarque que le nombre de copies effectuées et l’espace mémoire inutilisé varie beaucoup selon l’alpha qu’on choisit, plus celui-ci est grand plus on perd en espace.

Par contre si on analyse bien le coût amorti ou le coût réel on remarque qu’il ne varie pas autant que l’espace mémoire inutilisé, c’est vrai qu’il baisse un peu quand le alpha est plus grand mais sa valeur reste grande, on conclut donc que la valeur de alpha n’est pas le seul facteur qui fait augmenter le coût amorti. En effet le fait d’insérer des clés dans l’ordre décroissant joue aussi un rôle très grand car à chaque insertion on doit entasser vers le haut l’élément inséré, d’où le fait que le coût amorti et réel restent grands.

Pour résumé on peut dire que l’espace mémoire gaspillé dépend beaucoup de la valeur d’alpha, mais le coût réel et amorti quant à eux dépendent beaucoup plus de la manière dont on insère les clés.

**Insertion des clés aléatoires:**

**Pour alpha = 2**

Pour effectuer cette expérience on a modifié notre main du tas binaire en utilisant randum pour en insérer des valeurs aléatoires. Comme dans le cas statique.

On remarque que le cout réel est très perturbé il augmente énormément, cette augmentation est due au fait qu’ici on insère des valeurs aléatoires ce qui veut dire que certaines sont plus petites que les précédentes, la fonction entasser est alors appelé pour réorganiser le tas, ainsi le coût réel devient plus grand en faisant la permutation de plusieurs valeurs.

Après avoir fait plusieurs insertions la hauteur du tas devient plus grande et les permutations deviennent plus couteuses pour réorganiser notre tas, c’est ce qui fait que la valeur du cout réel après plusieurs insertions atteint une valeur qui dépasse 600. Cette méthode n’est pas efficace comparé à l’insertion effectuée dans le cas précédent.

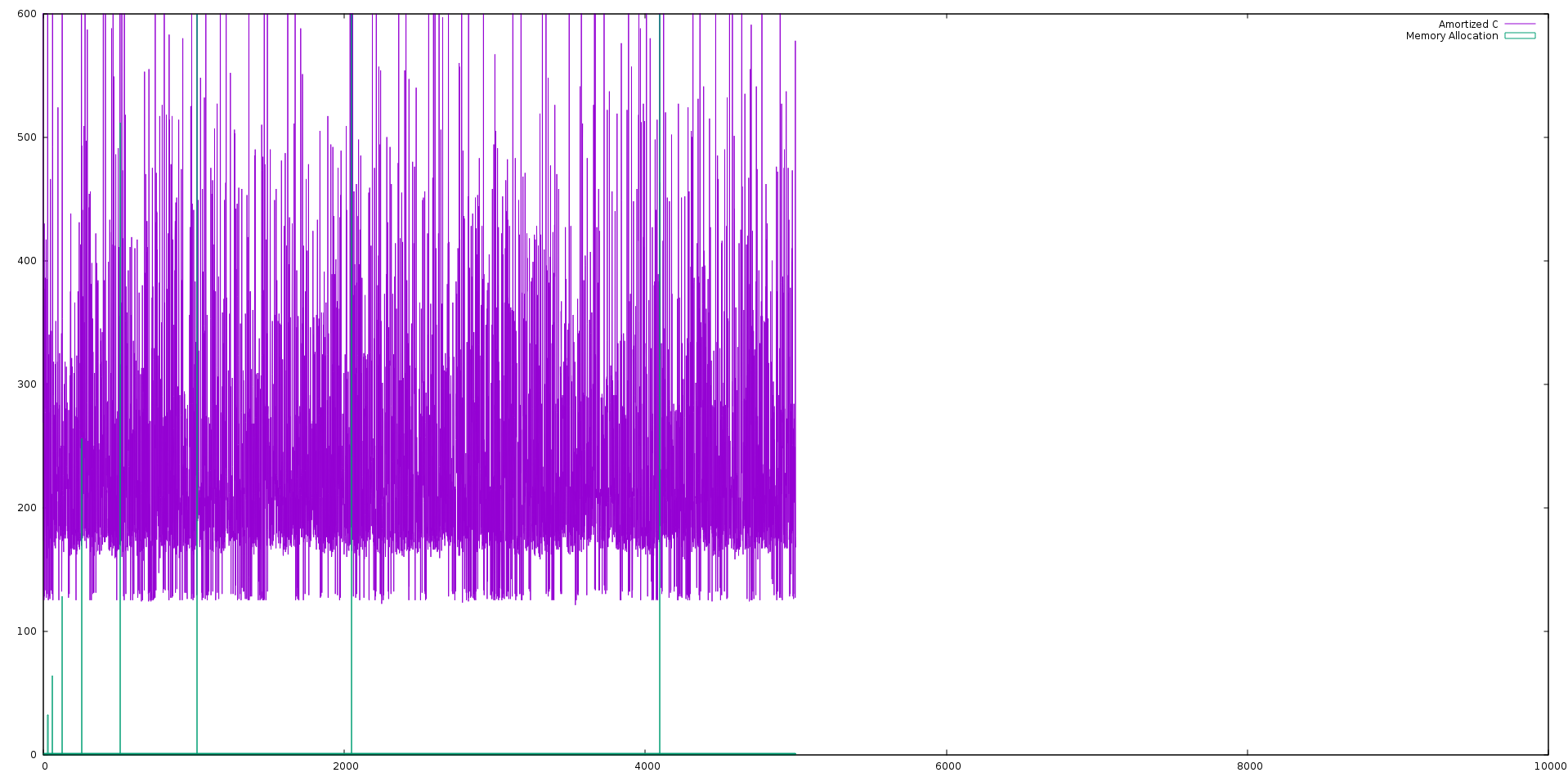


Figure  : Coût réel de l’insertion des clés aléatoires pour alpha égal à 2

De la même manière on a analysé le coût amorti de l’insertion, on a remarqué la même chose que pour le coût réel.

La valeur du coût amorti a augmenté comparé au cas de l’insertion des valeurs croissantes et cette fois ci elle varie entre 250 et 350. On justifie cette différence par le fait que l’insertion des valeurs est aléatoire, à chaque fois qu’on insère y a possibilité d’appeler entasser pour réorganiser le tas ce qui nous coûte plus chère. Ainsi Après avoir fait plusieurs insertions la hauteur du tas devient plus grande et les permutations deviennent plus couteuses pour réorganiser notre tas, c’est ce qui fait que la valeur du coût amorti après plusieurs insertions augmente comparé à notre précédente expérience de l’insertion des valeurs croissantes.

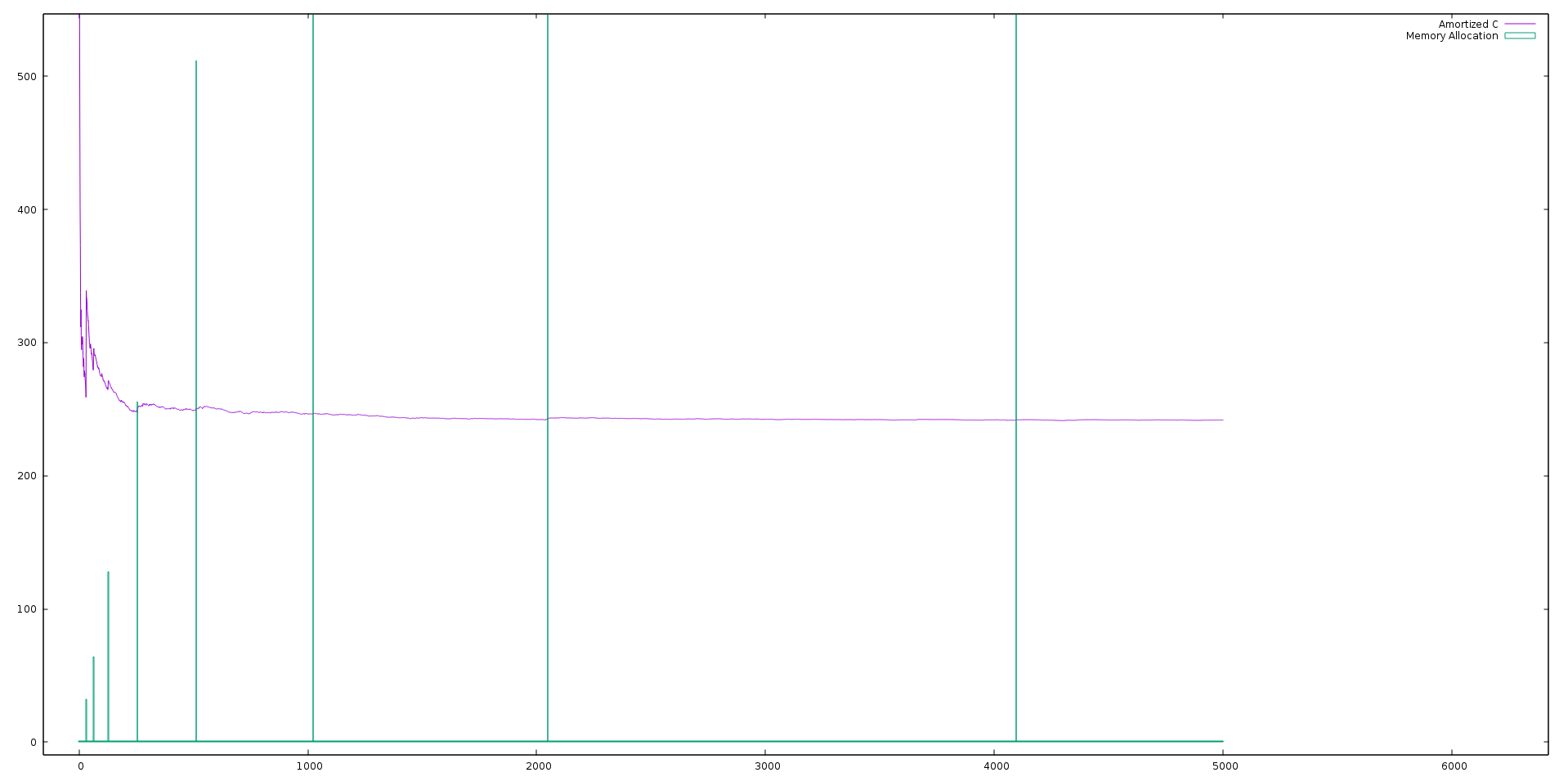


Figure  : Coût amorti de l’insertions des clés aléatoires pour alpha égal à 2

Ensuite on a affiché l’espace mémoire inutilisé en fonction du coût réel, on remarque qu’au début le gaspillage mémoire est assez petit, mais en insérant plus de valeurs on voit qu’il augmente de plus en plus.

On remarque aussi que c’est le même graphe obtenu dans le cas de l’insertion dynamique des valeurs croissantes vu que en mémoire ca ne change rien c’est le même espace mémoire qui est utilisé. Et c’est pareil pour le cout amorti de l’insertion dynamique des valeurs croissantes on obtient le meme grahe que le précédent les deux figures si dessous obtenus de notre éxpérience nous montre qu’il n ya aucune différence :

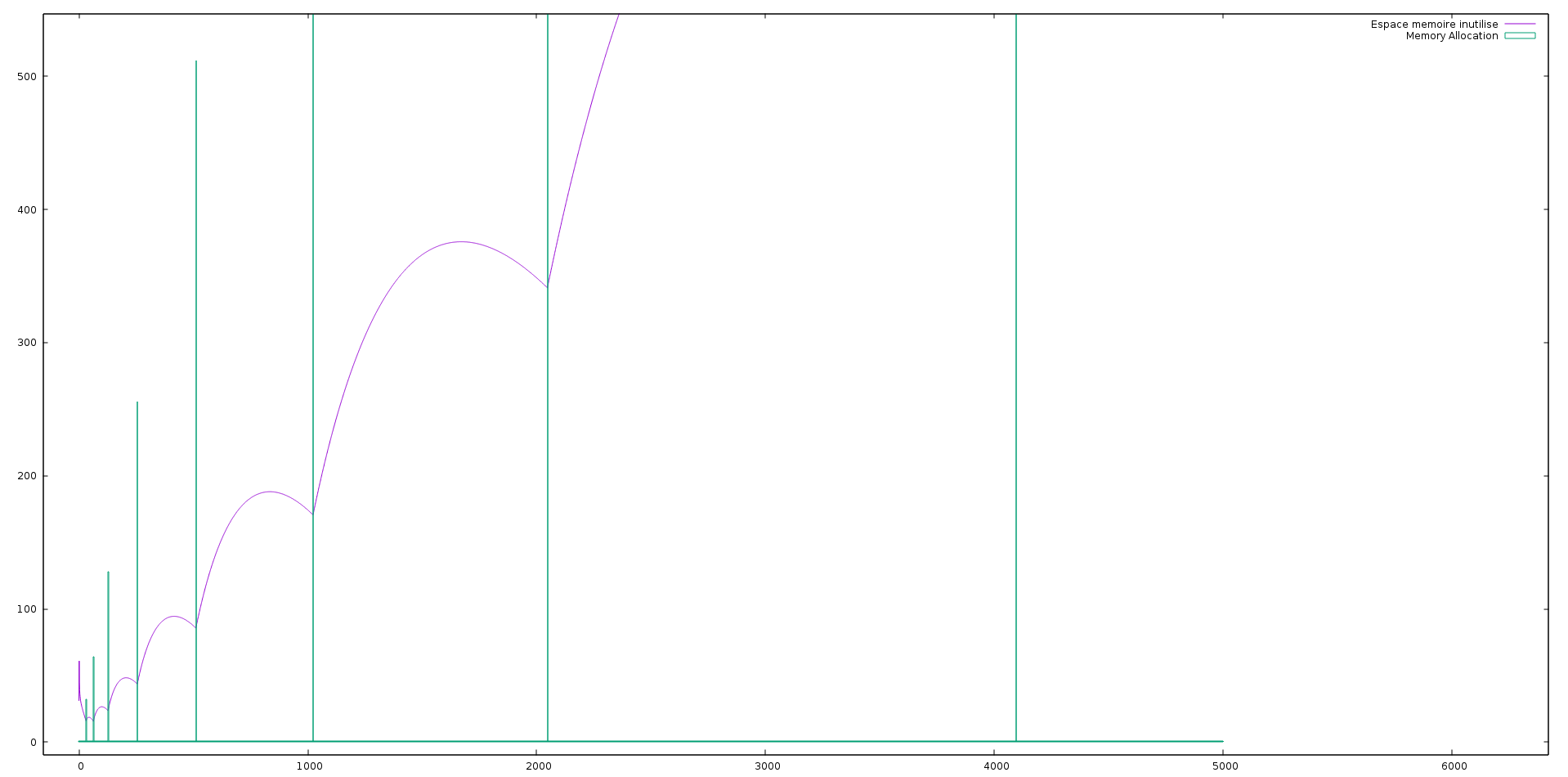


Figure  : Espace mémoire inutilisé pour l’insertion des clés aléatoires pour alpha égal à 2

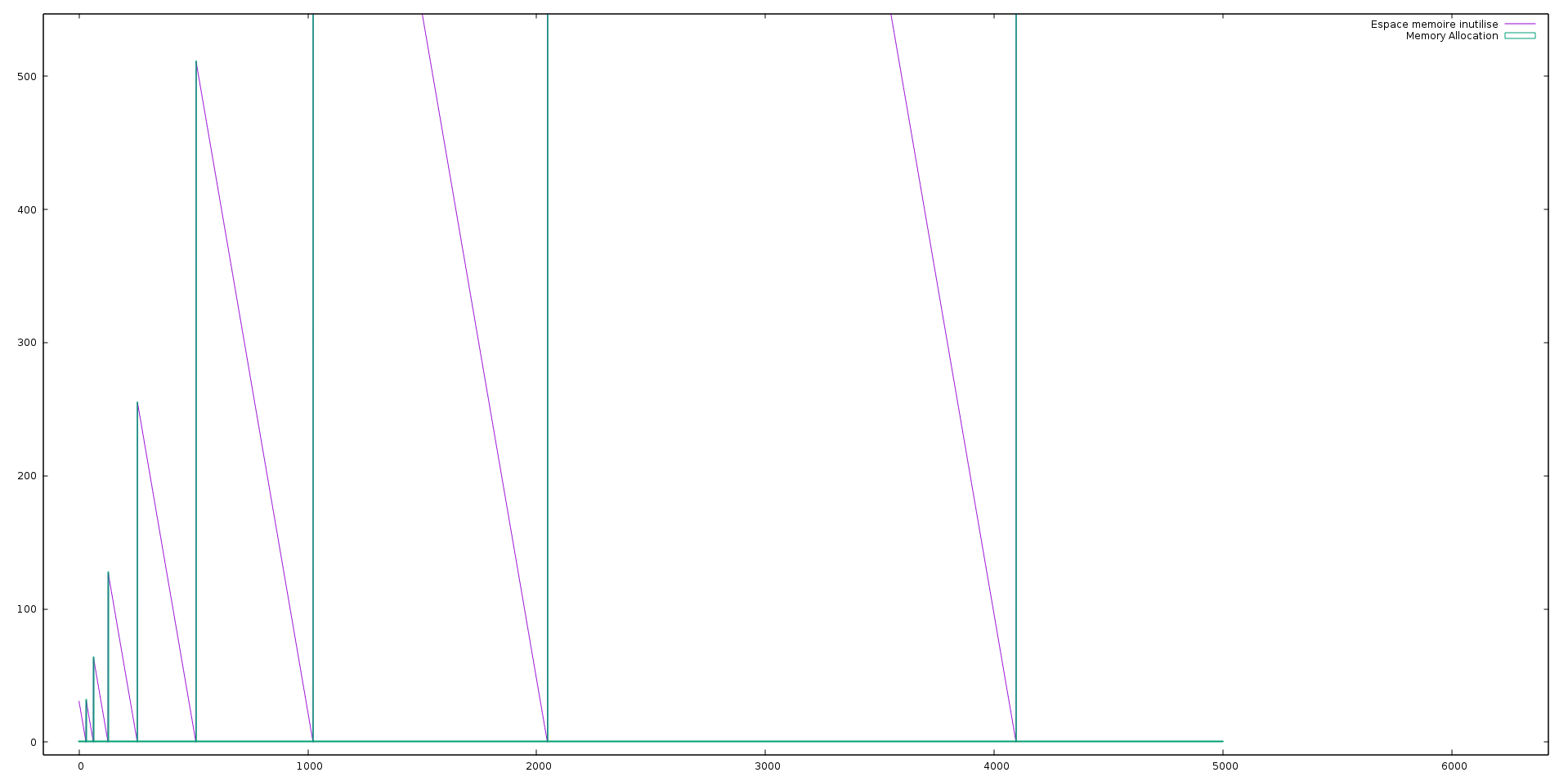


Figure  : Espace mémoire inutilisé en fonction du coût réel pour l’insertion des clés aléatoires pour alpha égal à 2

Ensuite on a affiché le nombre de copies effectuées, on remarque que le nombre de copies n’augmente qu’au moment de l’allocation mémoire. On remarque aussi que c’est le même graphe obtenue pour l’insertion dynamique des valeurs croissantes car le nombre d’allocation est le même.

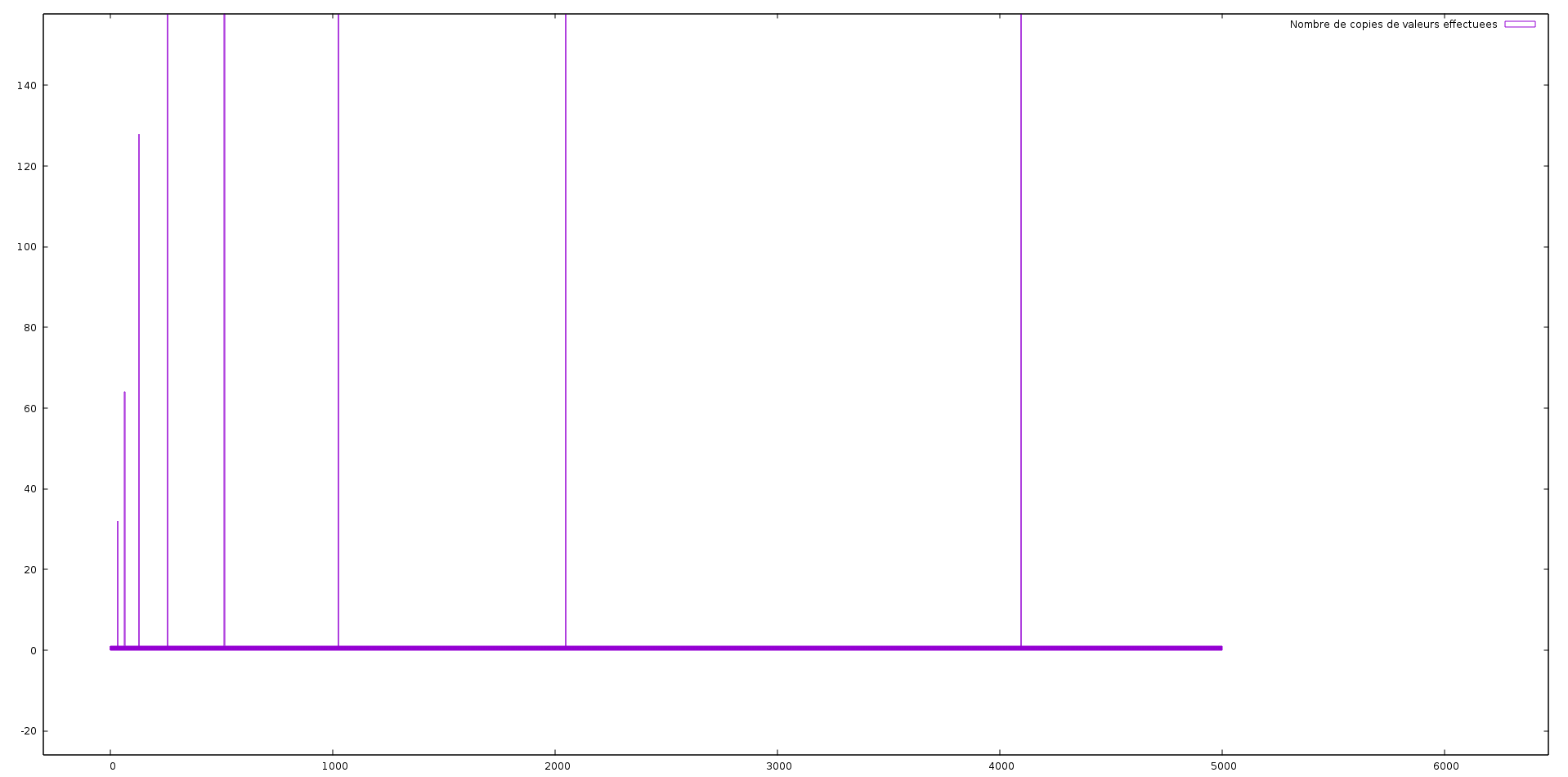


Figure  : Nombre de copies effectuées en fonction du temps réel pour l’insertion des clés aléatoires (alpha = 2)

**Pour alpha = (Nombre d’or)**

On a affiché le coût réel de l’opération d’insertion des valeurs aléatoires, on remarque qu’à chaque allocation mémoire le coût réel augmente énormément.

Comparons notre cout réel à celui du cas où alpha est égale à deux et que l’insertion est celle des valeurs aléatoires on remarque qu’avec l’alpha qui est égale au nombre d’or, le coût réel est légèrement plus élevé (presque pareil) vu qu’on a plus d’allocation mémoire dans notre cas.

Par contre si on compare ces résultats à ceux obtenus lors de l’insertion croissante, on voit que le coût réel a énormément augmenté, ses valeurs étaient entre 100 et 200, maintenant on dépasse 600. Cette augmentation s’explique par le fait que certaines clés insérées sont plus petites que les précédentes, ainsi on devra parcourir toute la hauteur du tas (dans le pire des cas) pour entasser ces clés.

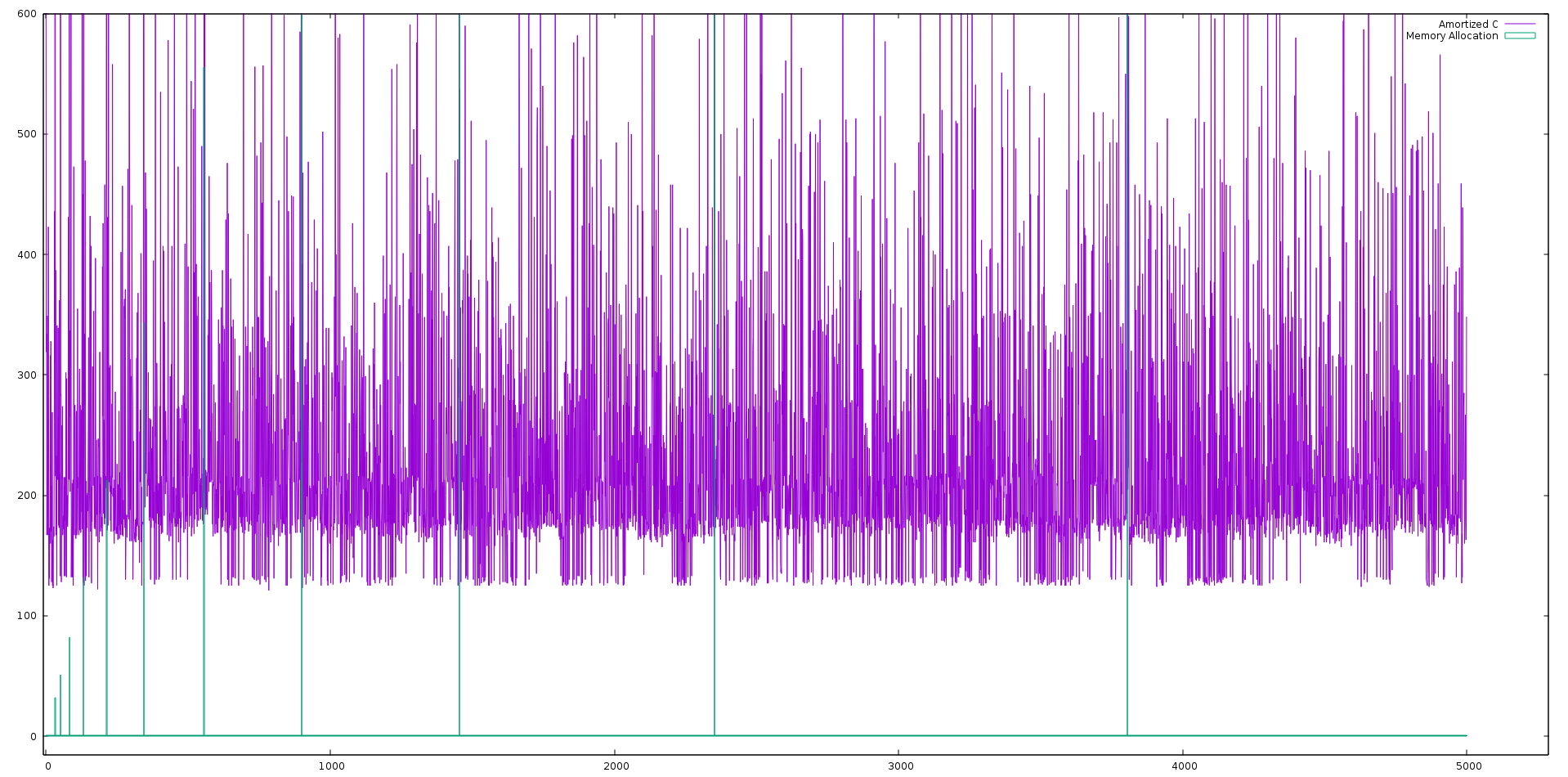


Figure  : coût réel de l’insertion des clés aléatoires pour alpha égal au nombre d’or

Ensuite on a affiché le cout amorti pour la même opération d’insertion, on remarque que le cout amorti au début est décroissant après avoir connu une valeur très élevée, puis il décroit et rencontre une augmentation au moment de l’allocation mémoire pour se stabiliser à une valeur entre 250 et 370. Ce qui n’est pas très différent du coût amorti de l’insertion des valeurs aléatoires avec une allocation mémoire qui double le tableau.

Par contre il est bien plus élevé que le coût amorti lors des insertions croissantes.

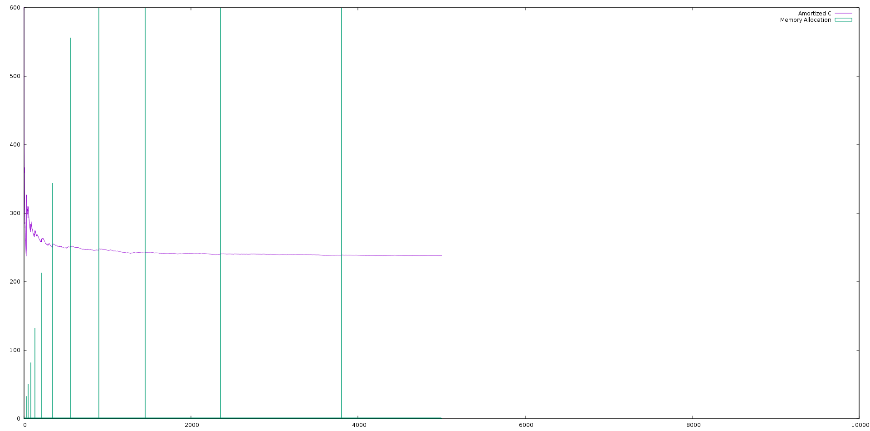


Figure  : Coût amorti de l’insertion des clés aléatoires pour alpha égal au nombre d’or

Ensuite on a affiché l’espace mémoire inutilisé en fonction du coût réel

Comparé à notre cas précédant on gaspille moins en éspace mémoire vu qu’on multiplie notre tableau par le nombre qui est inférieur à 2. Comparé au cas où on fait les insertions des valeurs croissantes avec alpha égale au nombre d’or c’est le même graphe vu qu’ils utilisent le meme espace mémoire pour stocker le meme nombre de valeurs ce qui diffère c’est l’ordre des veleurs insérées seulement.

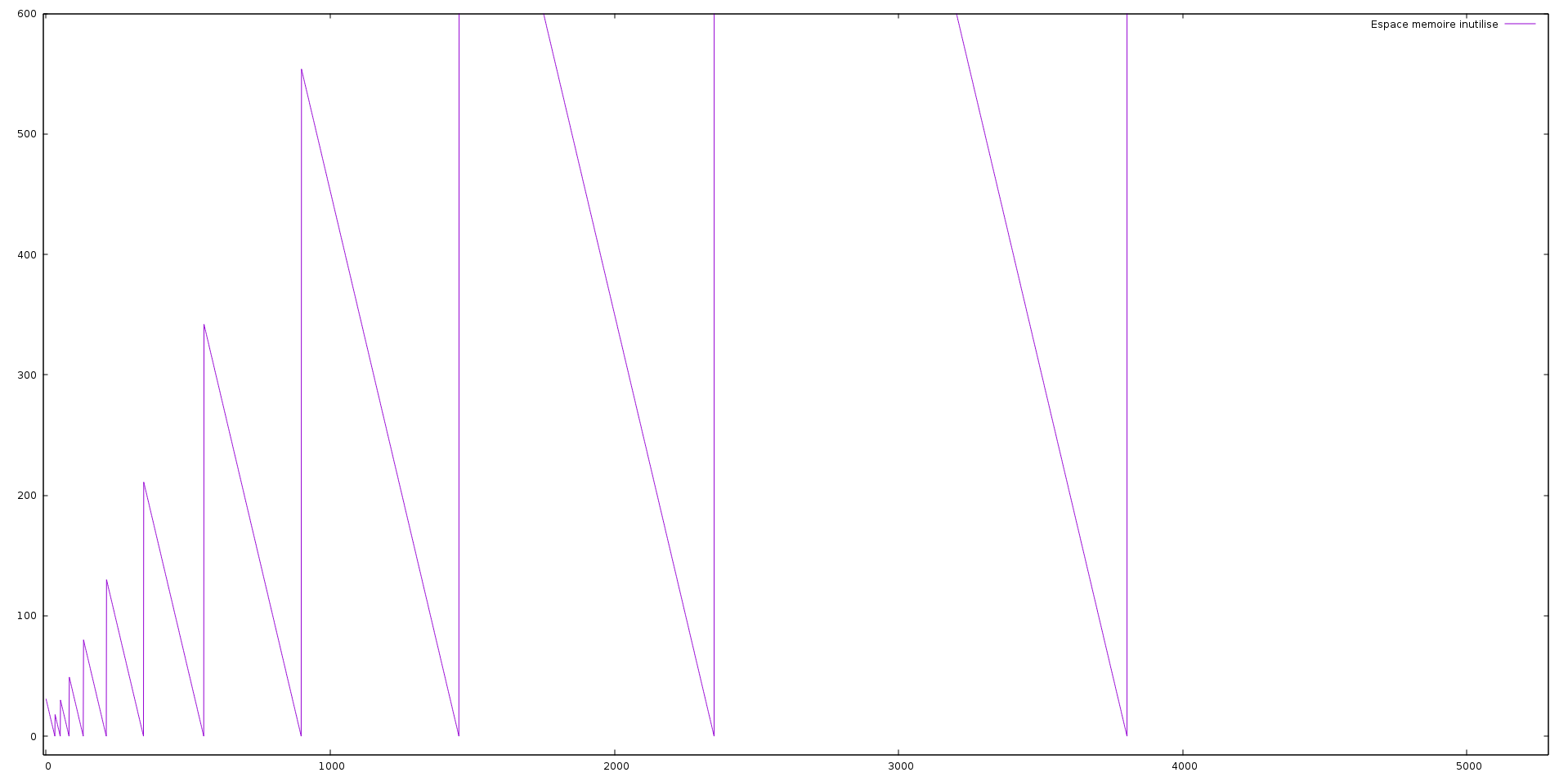


Figure  : Espace mémoire inutilisé en fonction du temps réel pour l’insertion des clés aléatoires et alpha égal au nombre d’or

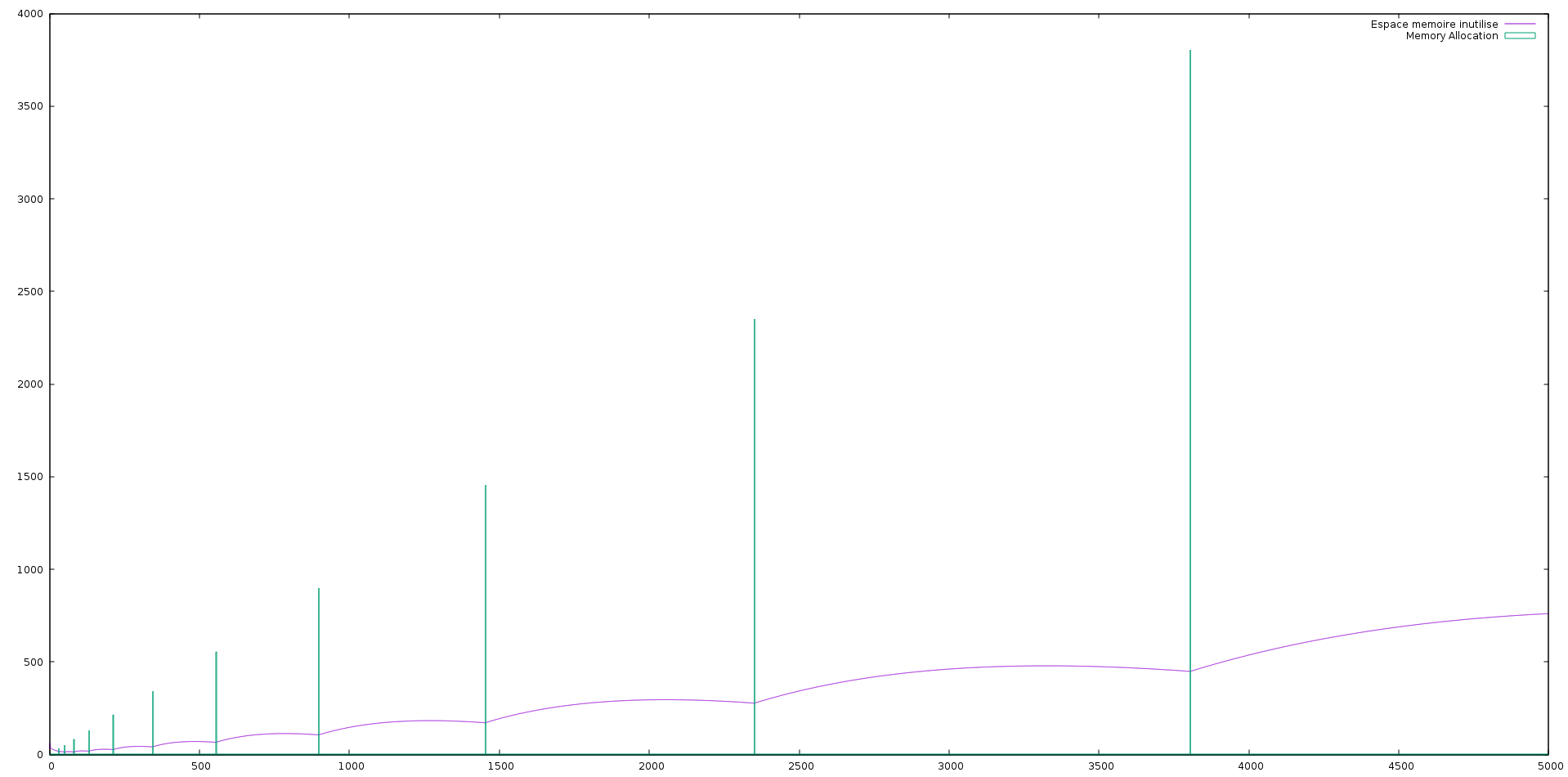


Figure  : Espace mémoire inutilisé en fonction du temps amorti pour l’insertion des clés aléatoires et alpha égal au nombre d’or

Ensuite on a affiché le nombre de copies effectuées, on remarque que le nombre de copies n’augmente qu’au moment de l’allocation mémoire. On remarque que c’est le même graphe obtenu lors de l’insertion dynamique des valeurs croissantes avec alpha égale au nombre d’or.

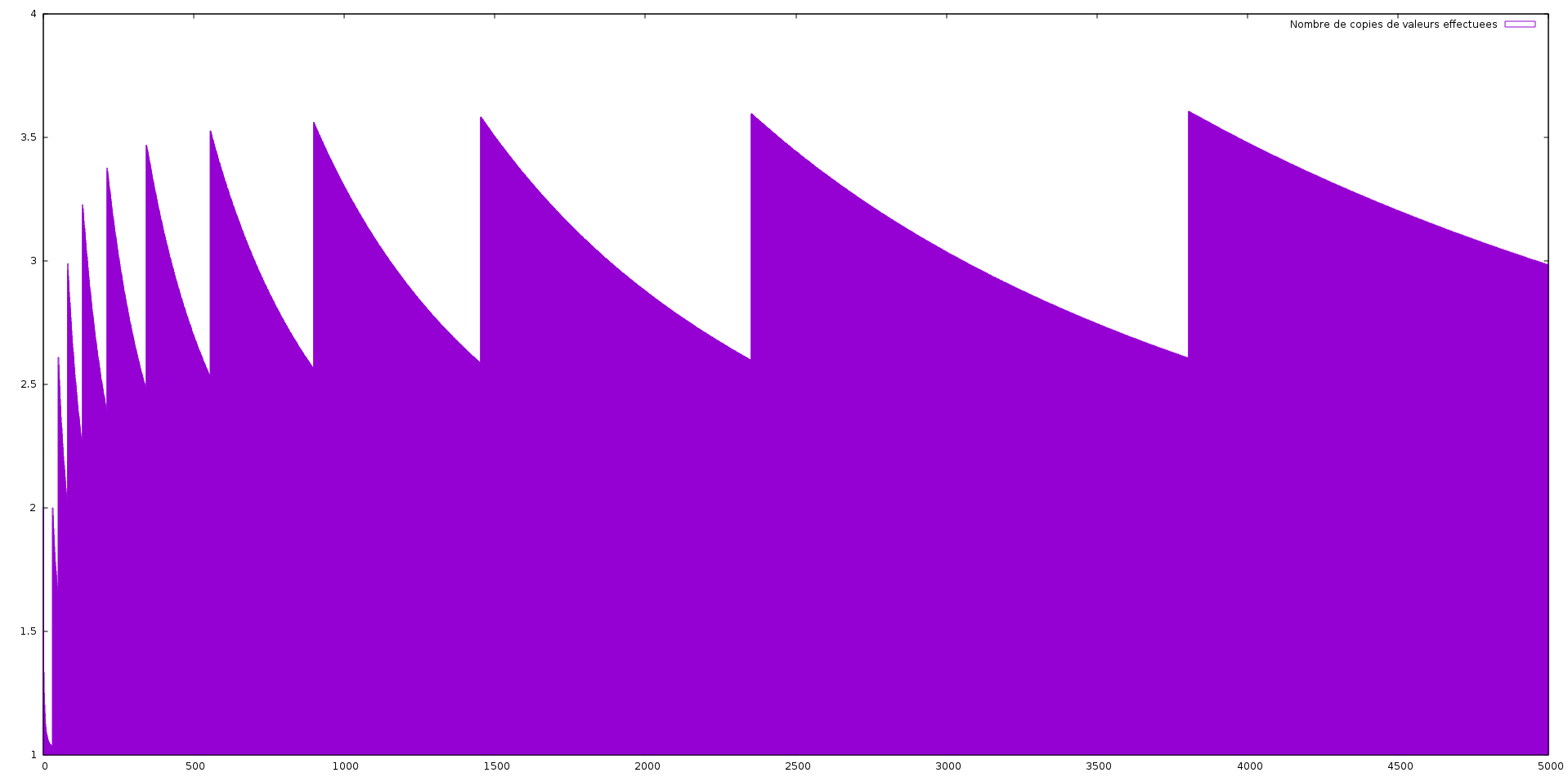


Figure  : Nombre de copies effectuées en fonction du temps amorti pour alpha égal au nombre d’or

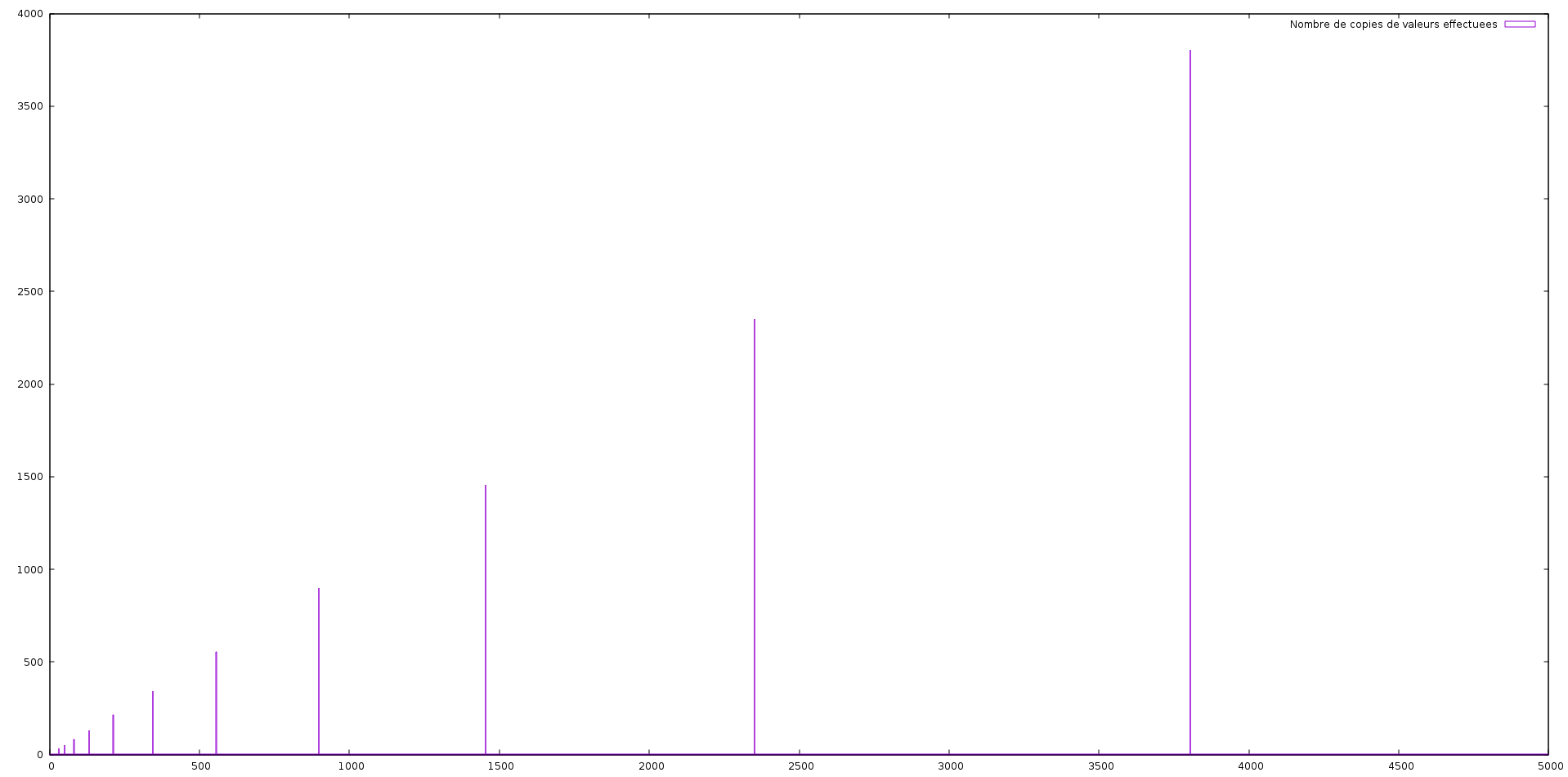


Figure  : Nombre de copies effectuées en fonction du temps réel pour alpha égal au nombre d’or

**Résumé :**

Après avoir effectué ces expériences on déduit que le coût amorti et réel sont moins élevés quand on fait une insertion de clés dans l’ordre croissant, de plus en variant la valeurs de alpha entre deux et le nombre d’or on a remarqué une différence au niveau de l’espace mémoire gaspiller qui est plus élevé dans le cas où le alpha est égale à 2. Par contre le coût amorti et réel ne diffère pas trop quand alpha égal à 2 et quand il est égal au nombre d’or. Il y a aussi une petite différence dans le nombre de copies effectuées.

Donc le cas le plus optimal pour nous est celui où on effectue l’insertion dynamique des valeurs croissante et on multiplie la taille du tableau par le nombre d’or lors de la location mémoire, comme on l’a déjà déduit auparavant.

**Insertion des clés et extraction du minimum :**

**Pour alpha = 2**

Après avoir analysé notre structure de tas pour l’insertion des clés dans l’ordre croissant, on va maintenant tester le coût amorti dans le cas où on fait des insertions et suppression.

On remarque d’après ce graphe que le coût amorti à des valeurs ente 200 et 300, il a donc baissé par rapport à l’expérience précédente où ses valeurs étaient entre 400 et 500. Cette baisse est due au fait qu’ici on insère des valeurs croissantes mais aussi on extrait le minimum par fois. Si on compare ses résultats au cas où on ne fait qu’insérer des valeurs croissantes on voit que le coût amorti a augmenté. En effet quand on extrait le minimum on le remplace par la dernière valeur du tableau, ensuite on doit l’entasser vers le bas à chaque extraction, d’où l’augmentation du coût amorti.

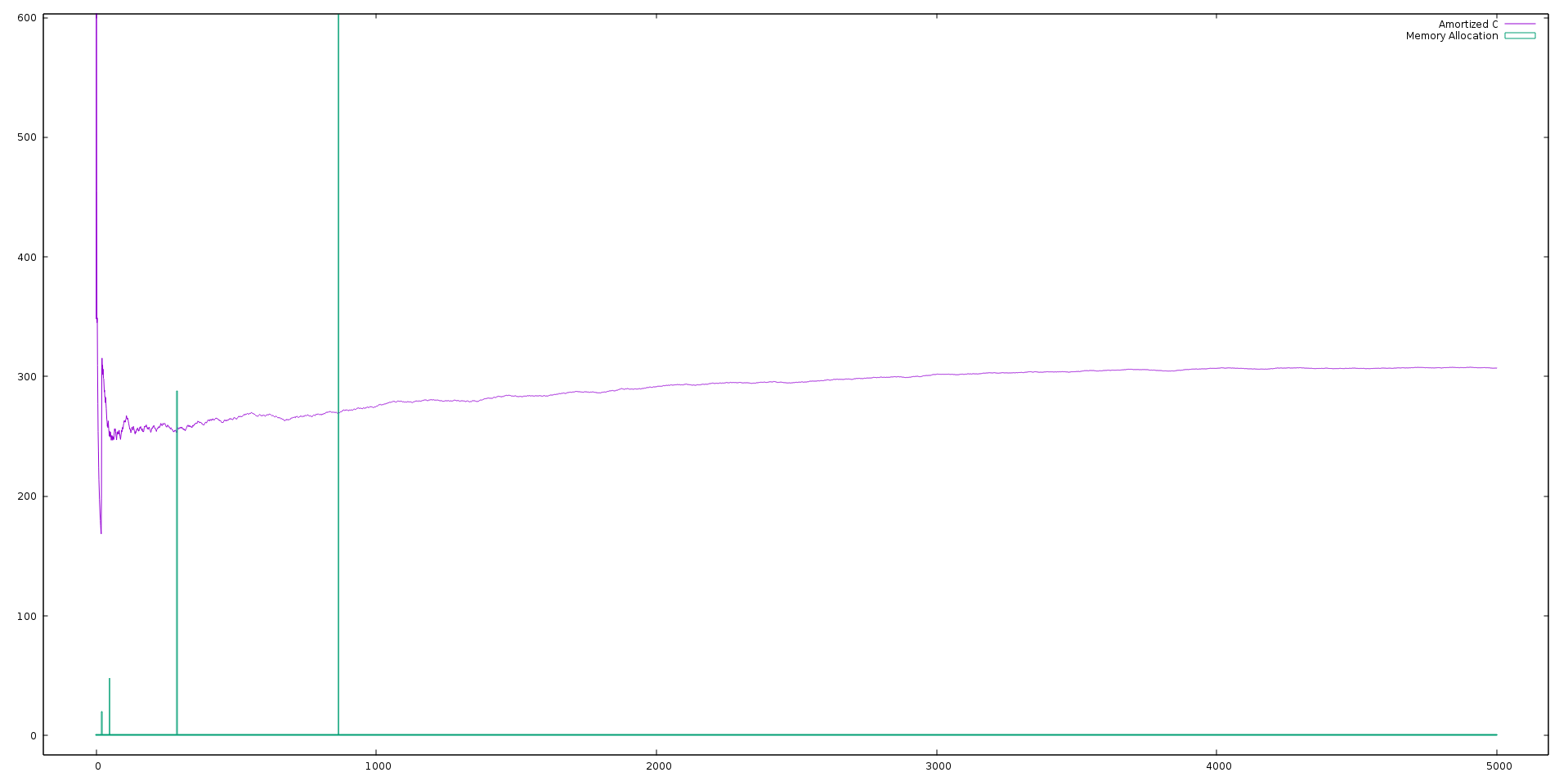
****

Figure  : Coût amorti de l’insertion est extraction

En affichant le coût réel des ces insertions et extraction on remarque qu’il est très perturbé et qu’il augmente beaucoup par fois et on pense que c’est au moment de l’extraction.

En comparant ces résultats au cas où ne fait que des insertions croissantes on voit que le coût réel ici est beaucoup plus grand et cela est bien sur du au fait qu’on fait des extractions par fois donc on doit entasser vers le bas l’élément par lequel on a remplacé la racine.

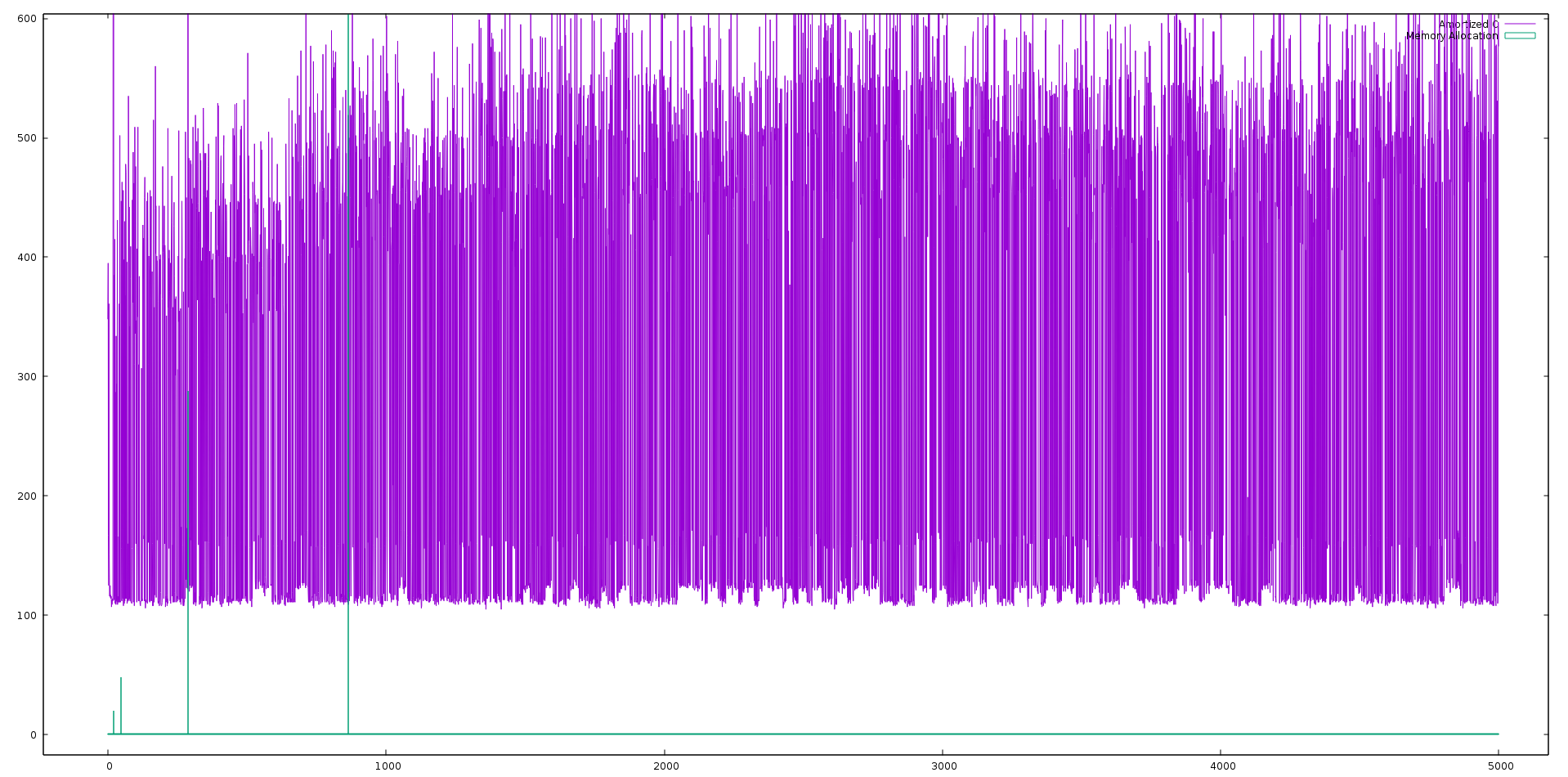


Figure  : coût réel de l’insertion des clés et extraction du minimum

On va passer maintenant à l’analyse de l’espace mémoire gaspillé et du nombre de copies effectuées.

On remarque que l’espace mémoire inutilisé a beaucoup baissé comparé à l’expérience où on faisait des insertions croissantes uniquement, ses valeurs étaient entre 100 et 1000 et maintenant elles ne dépassent même pas la centaine, on peut expliquer cela par le fait que beaucoup d’extraction du min ont été faites donc le tableau ne se rempli pas vite, on n’a donc pas besoin de réallouer de la mémoire plusieurs fois, d’ailleurs on remarque que sur 5000 opération on n’a eu que 4 allocation mémoire, d’où le fait qu’on n’a pas beaucoup de mémoire gaspillé, ce qui nous rend assez content !

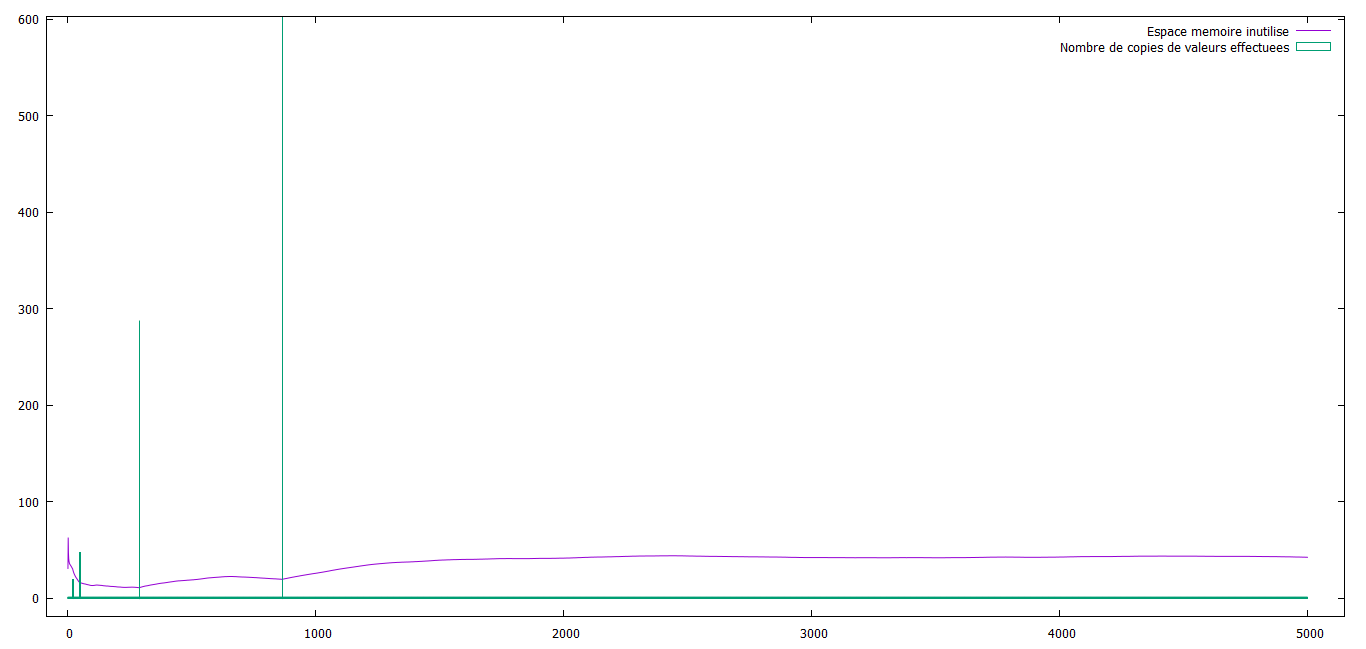


Figure  : Nombre de copies effectuées et espace mémoire inutilisé en fonction du coût amorti

Pour plus de détails on a affiché l’espace mémoire inutilisé en fonction du temps réel.

On remarque la même chose que dans le graphe précédent, on voit aussi que l’espace mémoire inutilisé augmente au moment de l’allocation mémoire car c’est à ce moment là qu’on copie les éléments, mais il finit toujours par baisser, d’ailleurs on voit qu’à la dernière opération (5000) sa valeur vaut 0 on n’a donc aucune mémoire gaspillée.

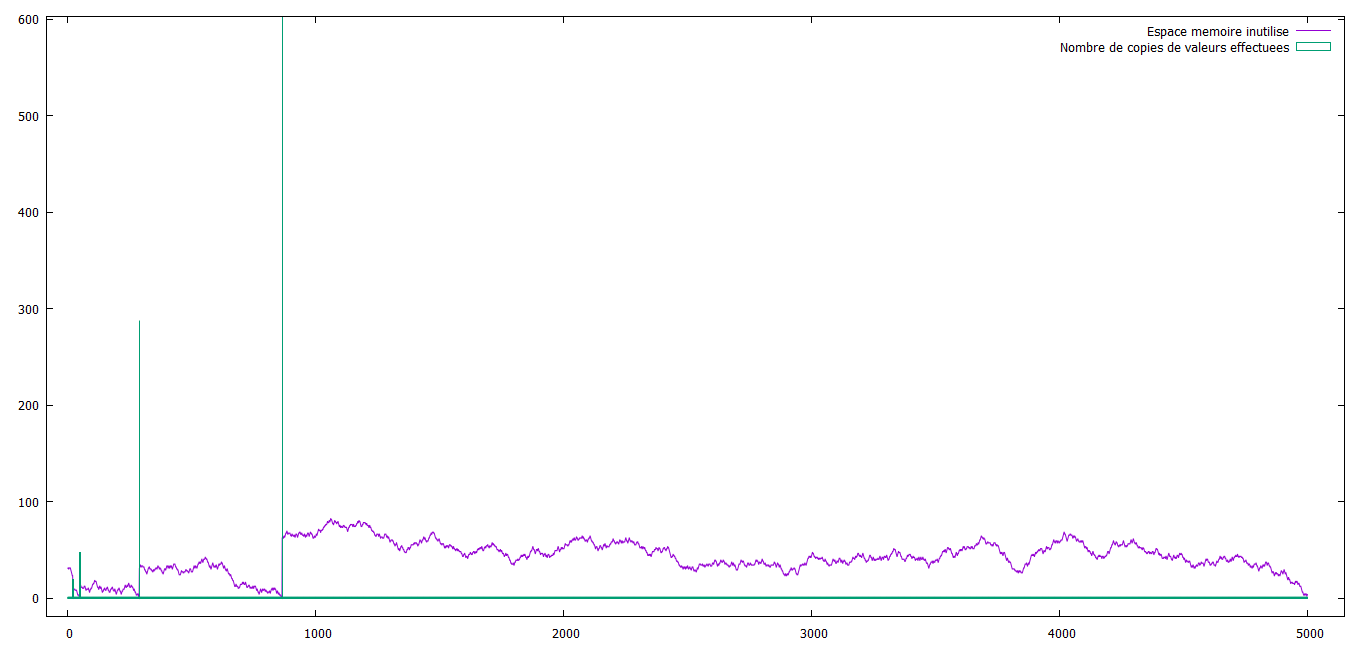


Figure  : Nombre de copies effectuées et espace mémoire inutilisé en fonction du temps réel

**Conclusion :**

Faisant maintenant le point sur l’ensemble des expériences :

Pour l’insertion croissante des clés, on a vue que le coût amorti n’est pas très élevé et ne varie pas tellement lorsqu’on change la valeur de alpha, en revanche l’espace mémoire inutilisé quant à lui, change beaucoup selon le alpha choisit.

Pour l’insertion décroissante des clés, on a remarqué que le coût amorti était très élevé et cela à cause de l’entassement des éléments à chaque insertion. En modifiant l’alpha on voit que seul l’espace mémoire gaspillé change beaucoup.

Pour l’insertion des clés aléatoires, on a vu que le coût amorti était plus élevé que dans la première expérience car on doit entasser certains éléments, par contre il n’est pas aussi grand que celui des insertions décroissantes car on ne fait pas appel à la fonction entasser pour chaque insertion.

Pour l’insertion des clés et extraction du min, on a remarqué qu’il est plus grand que celui de la première expérience car en faisant l’extraction du minimum on fait appel à la fonction entasser vers le bas, par contre il est moins élevé que le coût amorti de la deuxième expérience car même si on entasse en cas d’extraction du min, le fait d’insérer des valeurs croissantes nous fais gagner en efficacité.

Pour résumer on peut donc dire que les tas sont un peu différent des tableaux dynamiques, car le coût amorti des opérations sur les tas ne dépend pas beaucoup de la valeur d’alpha mais de la manière dont les éléments sont insérés.

Néanmoins on pense que le meilleure alpha à choisir c’est le nombre d’or car il permet de gagner en espace mémoire sans perdre en temps d’exécution et c’est un nombre connu pour ses propriétés mathématiques très intéressantes.

Après avoir finit ce rapport on peut dire qu’on a trouvé une solution au problème rencontré dans le rapport sur les tas statiques. En effet on peut utiliser un tableau dynamique pour l’implémentation d’un tas min en choisissant alpha égal au nombre d’or.

Tout comme les tas binaires statiques, les tas dynamiques sont aussi utilisés pour l’implémentation de files de priorités, qui sont utilisées pour l’ordonnancement des taches d’un processeur par exemple.

Les tas binaires sont inefficaces pour l’opération RECHERCHER ; trouver un nœud contenant une clé donnée peut prendre beaucoup de temps. Pour cette raison, que ces tas sont généralement utilisés pour rechercher ou renvoyer un élément en particulier, soit le min si on parle tas min, soit le max si on parle de tas max.