

RAPPORT STRUCTURE DE DONNEES AVANCES

TP1 TABLES DYNAMIQUES

Réalisé par :

LE Minh Hao

KEITA Aïssata

Table des matières

1. Quand la table est pleine, on en multiplie la taille par un facteur $\alpha \geq 1$	3
2. Le coût amorti est $\hat{C}i = Ci + \phi i - \phi i$	3
Cas 1 : La i ème opération ne déclenche pas une extension, alors on a :.....	3
Cas 2 : La i ème opération déclenche une extension, alors on a :.....	4
3. Compilez et exécutez les programmes dans les 4 langages, puis observez les résultats expérimentaux dans gnuplot:.....	4
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 ?	5
b. Observez maintenant le coût amorti en temps dans les différents langages. À quel moment le coût amorti augmente-t-il ? Pourquoi ? (La réponse dépend du langage).....	6
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 ?.....	7
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 ?.....	9
e. Tentez d'expliquer pourquoi certains langages sont plus rapides que d'autres dans cette expérience	9
4. Modifiez la fonction « do_we_need_to_enlarge_capacity » pour ne se déclencher que lorsque le tableau est plein. Relancer une expérience et commentez les résultats.	11
a) On utilise la classe ArrayListProxy.....	11
b) On utilise la implémentation simple de ArrayList.....	14
5. Dans la fonction enlarge_capacity, faites varier le facteur multiplicatif α . Que se passe-t-il ? Donner une règle, décrivant le rapport entre le coût en temps et le coût en espace.	15
6. Dans la fonction enlarge_capacity, faites varier la capacité n vers une capacité $n + \alpha n$. Que se passe-t-il ?.....	20

1. Quand la table est pleine, on en multiplie la taille par un facteur $\alpha \geq 1$.

Donc, on a la fonction potentielle d'ième : $\phi(i) = \alpha n_i - t_i$.

Avant l'extension : $\phi(i) = \alpha n_i - t_i = (\alpha - 1) t_i$

Après l'extension : $\phi(i) = \alpha n_i - t_i = t_i - t_i = 0$

On a un problème avec cette fonction potentielle ci-dessous :

Si $\alpha > 2$, $\phi(i)$ est trop grand

Si $\alpha < 2$, $\phi(i)$ est trop petit

Cette fonction potentielle ne marche que avec $\alpha = 2$. Donc le numéro 2 est le facteur d'agrandissement du tableau.

On peut utiliser cette fonction potentielle ci-dessous afin de généraliser des **facteurs** appropriés de **nombre** et de **taille** : $\phi(i) = x n_i - y t_i$

Avant l'extension :

$$\phi(i) = x n_i - y t_i = 0$$

$$\phi(i) = x n_i - y \alpha n_i = 0$$

$$\rightarrow x = y \alpha$$

Après l'extension :

$$\phi(i) = x n_i - y t_i = n_i$$

$$\phi(i) = x n_i - y n_i = n_i$$

$$\rightarrow x = 1 + y$$

$$\rightarrow 1 + y = y \alpha$$

$$\rightarrow y = \frac{1}{\alpha - 1} \text{ et } x = \frac{\alpha}{\alpha - 1}$$

$$\rightarrow \phi(i) = \frac{\alpha n_i - t_i}{\alpha - 1}$$

Enfin, on va choisir cette fonction potentielle $\phi(n) = \frac{\alpha n - t}{\alpha - 1}$.

2. Le coût amorti est $\hat{C}_i = C_i + \phi_i - \phi_i$

Cas 1 : La ième opération **ne déclenche pas** une extension, alors on a :

$$t_i = t_{i-1}$$

$$C_i = 1$$

$$n_i = n_{i-1} + 1$$

Et le coût amorti :

$$\hat{C}_i = C_i + \phi_i - \phi_i$$

$$= 1 + \frac{\alpha n_i - t_i}{\alpha - 1} - \frac{\alpha n_{i-1} - t_{i-1}}{\alpha - 1}$$

$$= 1 + \frac{\alpha n_i - t_i - \alpha(n_{i-1} - 1) + t_i}{\alpha - 1}$$

$$= 1 + \frac{\alpha n_i - \alpha n_{i-1} + \alpha}{\alpha - 1}$$

$$= 1 + \frac{\alpha}{\alpha-1}$$

Cas 2 : La ième opération **déclenche** une extension, alors on a :

$$t_i = \alpha t_{i-1}$$

$$C_i = n_i$$

$$n_i = n_{i-1} + 1 = t_{i-1} + 1$$

$$n_{i-1} = t_{i-1}$$

Et le coût amorti :

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$

$$= n_i + \frac{\alpha n_i - t_i}{\alpha - 1} - \frac{\alpha n_{i-1} - t_{i-1}}{\alpha - 1}$$

$$= \frac{(\alpha - 1)n_i + \alpha(n_{i-1} + 1) - \alpha(t_{i-1})}{\alpha - 1} - \frac{\alpha n_{i-1} - t_{i-1}}{\alpha - 1}$$

$$= \frac{\alpha n_{i-1} + \alpha - n_{i-1} - 1 + \alpha n_{i-1} + \alpha - \alpha n_{i-1} - \alpha n_{i-1} + n_{i-1}}{\alpha - 1}$$

$$= \frac{2\alpha - 1}{\alpha - 1}$$

3. Compilez et exécutez les programmes dans les 4 langages, puis observez les résultats expérimentaux dans gnuplot:

Ce sont les coûts totaux et les coûts moyens des programmes dans les 4 langages.

Langage	Total cost	Average cost
C	22844032.0	22.844
C++	35950900.0	35.950
Java	30888744	30.888
Python	218849897.3	218.849

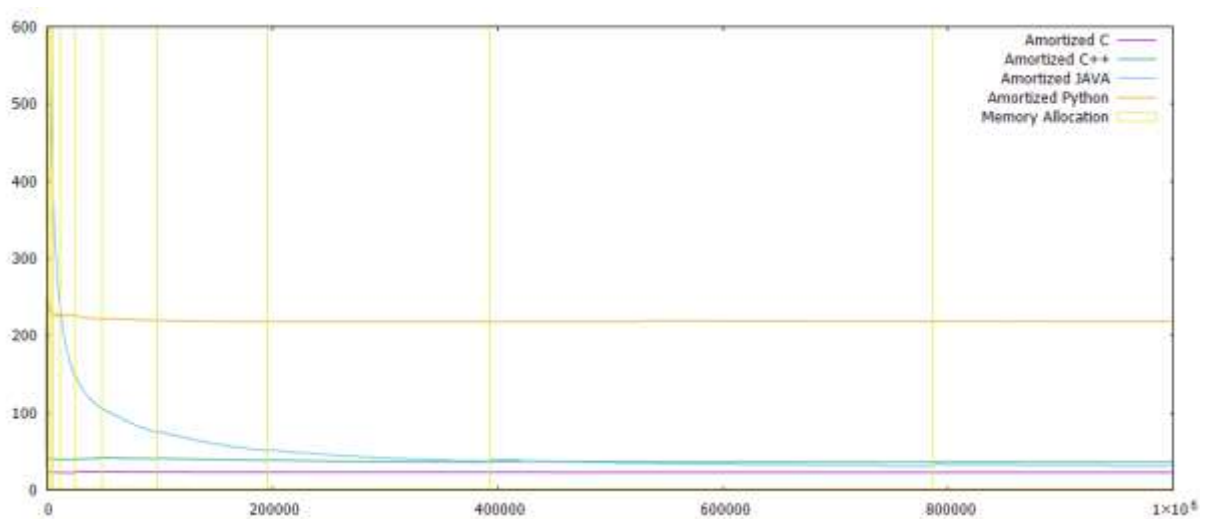


Figure 1. Les coûts amortis des programmes et L'allocation de mémoire

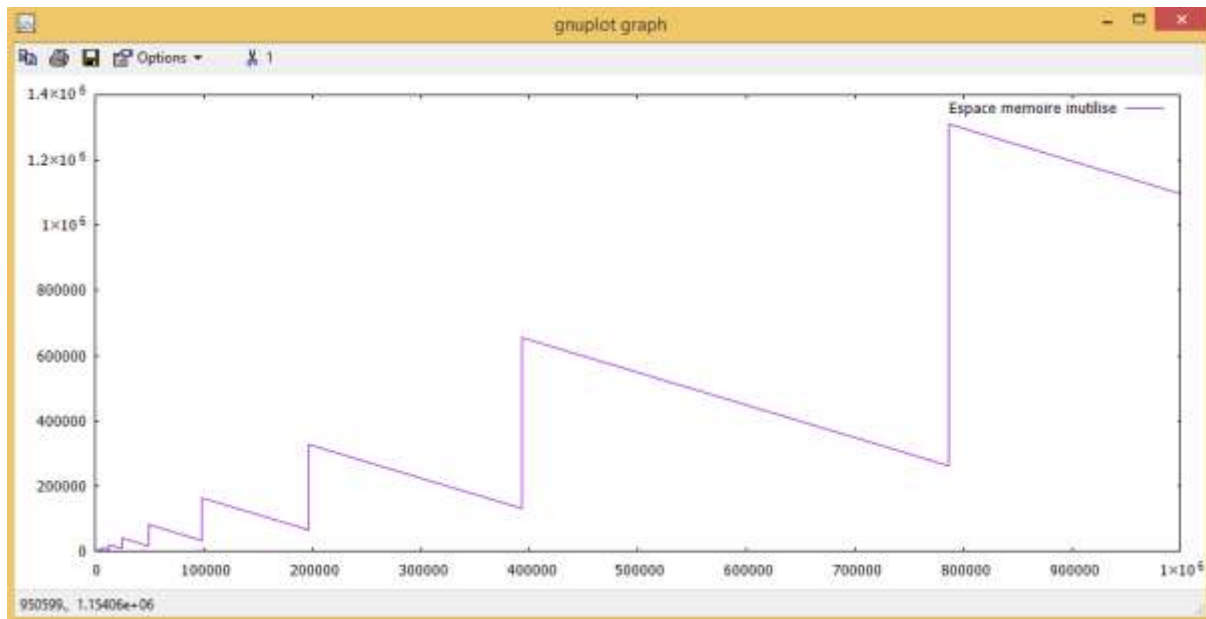


Figure 2. Espace mémoire inutilisé dans C

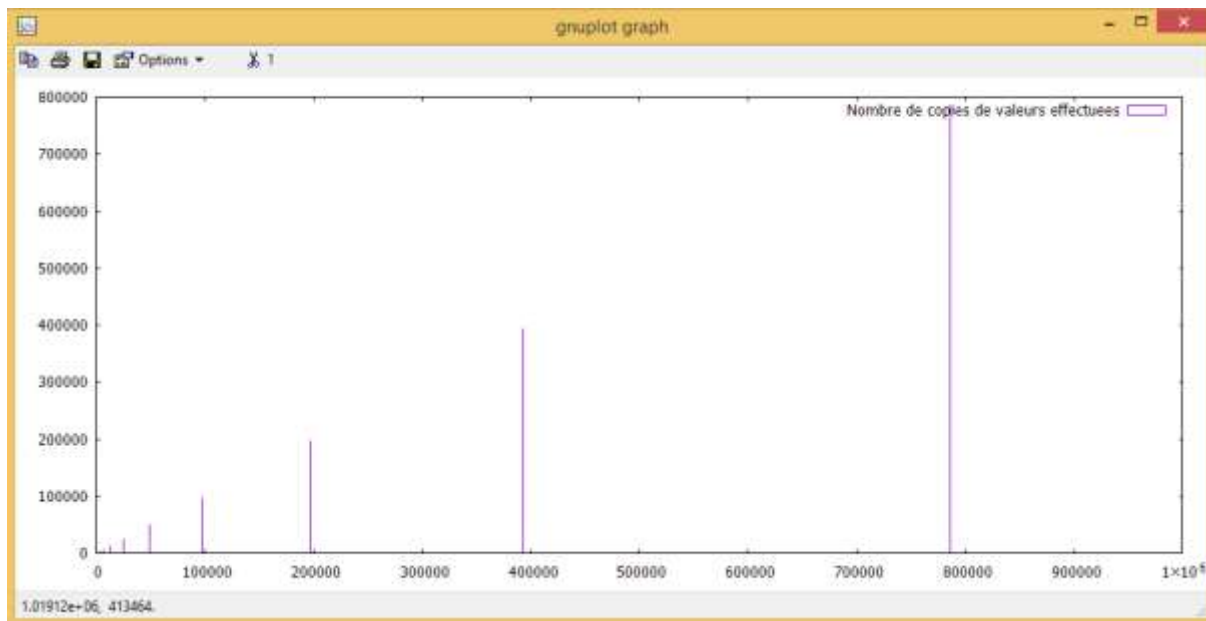


Figure 3. Nombre de copies de valeurs effectuées dans C

- 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 ?

Lors de l'exécution des programmes, le morceau de code qui prend le plus de temps à s'exécuter est la boucle d'insérer. Puisque c'est le morceau principal des programmes.

Parce que la boucle a une complexité $O(n)$ et dans la boucle, la méthode « **append** » de Dynamic Table a la complexité $O(n)$ dans le pire cas et $O(1)$ dans le meilleurs cas. Alors, la complexité de ce morceau est $O(n^2)$ dans le pire cas (où on multiplie la taille et $O(n)$ dans le meilleurs cas.

Ce morceau de code est plus lent que le reste puisque il y a seulement deux actions : insertion et écriture le fichier. L'écriture ne prend pas plus de temps que l'insertion (inclus l'analyse du coût)

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

Au moment où la taille du tableau est agrandie, le coût amorti est augmenté un peu puis que l'on doit allouer un nouveau tableau vide et copier tous les anciens éléments au nouveau tableau.

Dans java, l'interprétation du *bytecode* qui est l'implémentation standard de la machine virtuelle Java (JVM) ralentit l'exécution des programmes. Pour améliorer les performances, les compilateurs JIT interagissent avec la JVM au moment de l'exécution et compilent les séquences de *bytecode* appropriées en code machine natif.

D'abord, le programme Java est exécuté par JVM qui a besoin d'un peu d'interprétation et après, le compilateur JIT réalise la boucle et il compile le bytecode des éléments dans cette boucle en code machine natif au lieu de l'interpréter en plusieurs temps.

Dans python, on insérait seulement les nouveaux éléments et Python va gérer la taille du tableau.

```
...
# Ajoute l'element x au tableau
# Complexite en temps/espace, pire cas : O(data.size)
# Complexite en temps/espace, meilleur cas : O(1)
# Complexite amortie : O(1)
def append(self, x):
    self.data.append(x)|
...
```

Figure 4. La méthode *append* dans Python

C'est la stratégie de Python sur l'extension de Dynamic Table.

```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
new_allocated += newsize;

/* This over-allocates proportional to the list size, making room
 * for additional growth. The over-allocation is mild, but is
 * enough to give linear-time amortized behavior over a long
 * sequence of appends() in the presence of a poorly-performing
 * system realloc().
 * The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
 */
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}
```

Figure 5. Partie de la méthode *resize* dans Python

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 ?

D'abord, On change la méthode « **get_amortized_cost** » dans classe Analyser pour tenir **deux décimales**. Le but est à facilement comparer entre les exécutions et à regarder le changement de cout amorti.

```
BigDecimal get_amortized_cost(int pos){
    return (pos > 0)? cumulative_cost.get(pos).divide(new BigDecimal(pos), 2, RoundingMode.HALF_UP) : cumulative_cost.get(pos);
}
```

Figure 6. La méthode `get_amortized_cost` dans Java

0	1.0	1
1	1.0	2.00
2	1.0	1.50
3	3.0	2.00
4	1.0	1.75
5	1.0	1.60
6	6.0	2.33
7	1.0	2.14
8	1.0	2.00
9	1.0	1.89
10	1.0	1.80
11	1.0	1.73
12	12.0	2.58

Figure 7. Les résultats de cout amorti après le changement

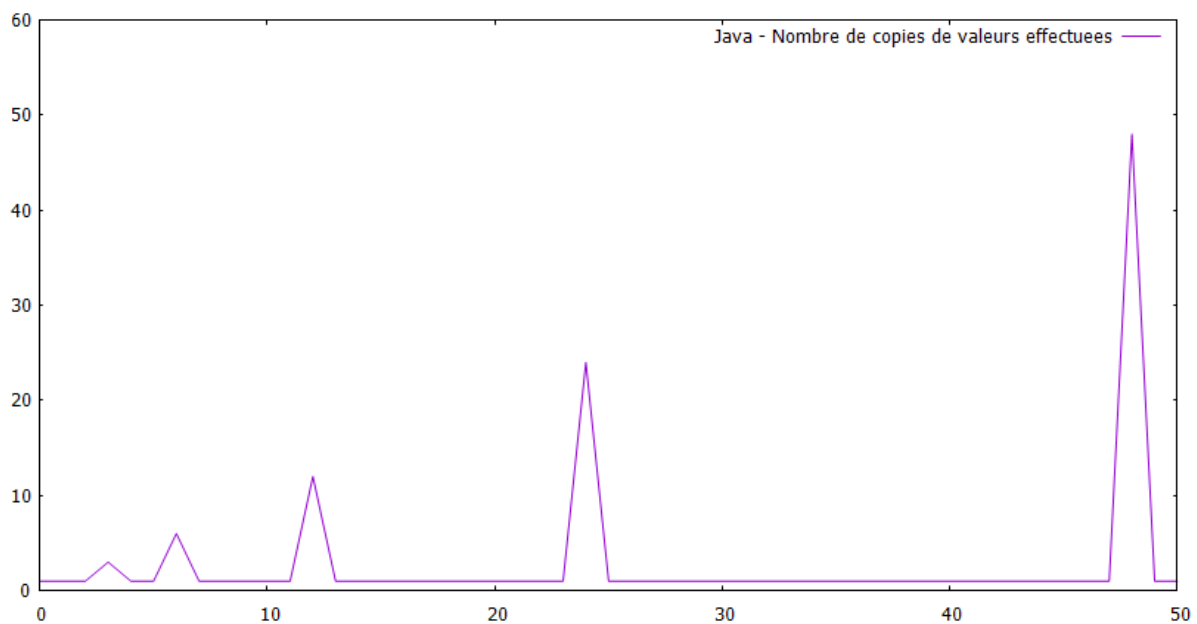


Figure 8. plot [0:50][0:60] 'dynamic_array_copy_java.plot' using 1:2 w lines title "Java - Nombre de copies de valeurs effectuées"

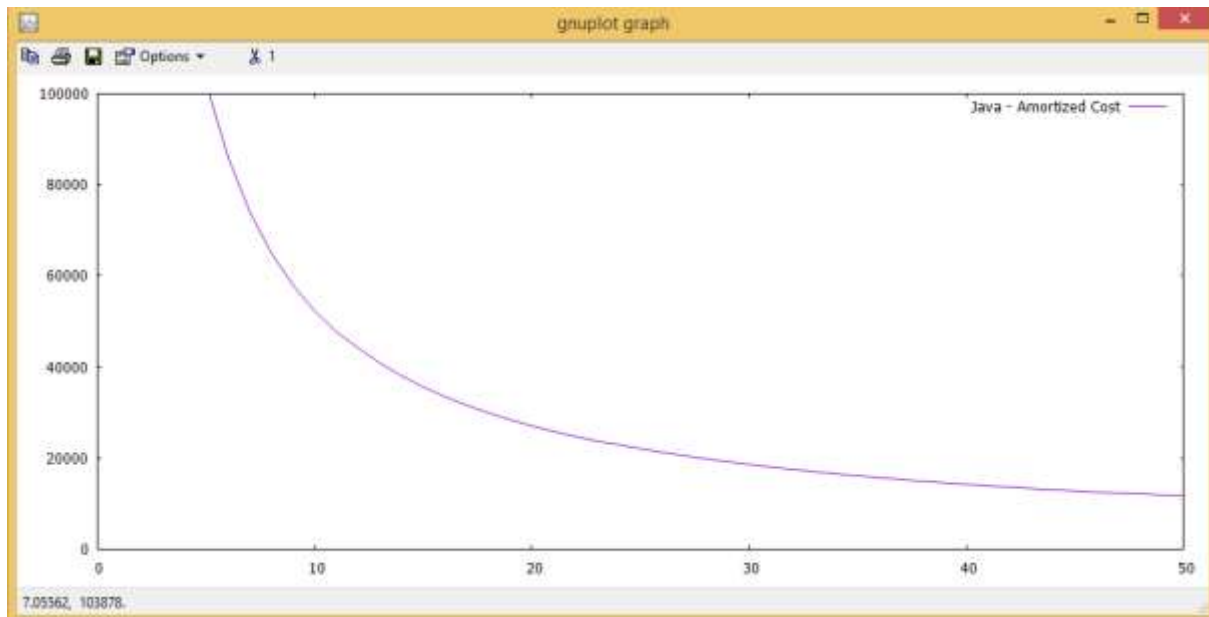


Figure 9. plot [0:50][0:100000] 'dynamic_array_time_java.plot' using 1:3 w lines title "Java - Amortized Cost"

Dans le TP1, on utilise la **méthode de l'agrégat**. Dans cette méthode on montre que, pour tout n , une suite de n opérations prend le **temps total** $T(n)$ dans le cas le plus défavorable. Dans le cas le plus défavorable le **coût moyen**, ou **coût amorti**, par opération est donc $T(n)/n$.

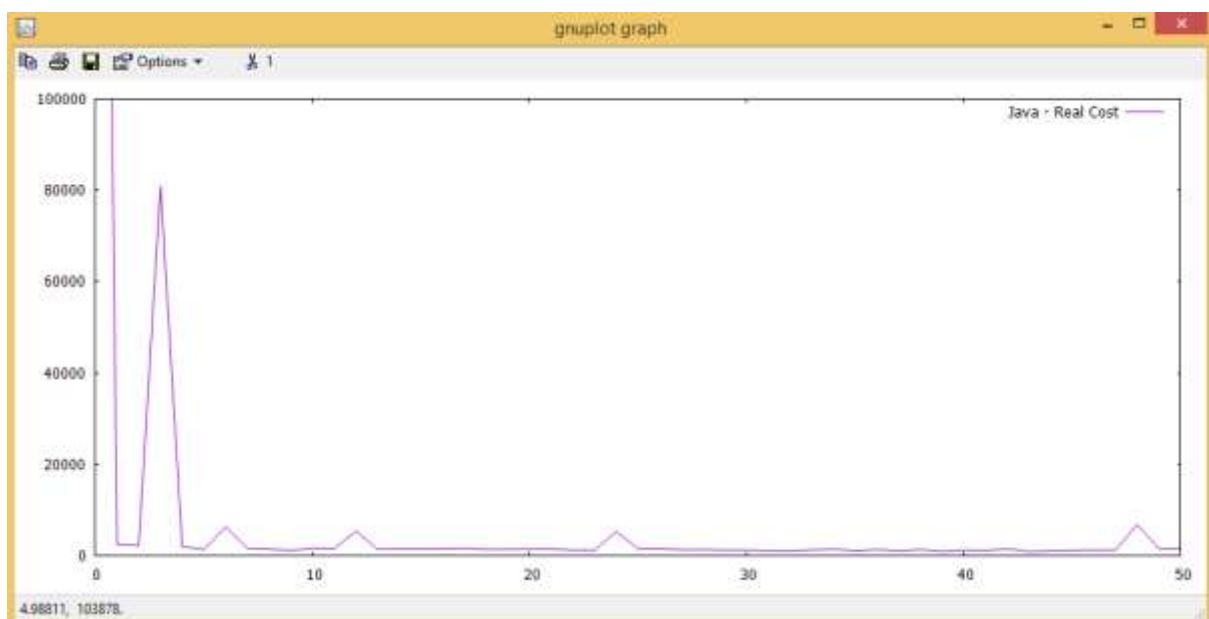


Figure 10. plot [0:50][0:100000] 'dynamic_array_time_java.plot' using 1:2 w lines title "Java - Coût du temps"

Quand on agrandit la taille, le coût réel est augmenté **approximativement** avec le nombre de copies. Alors, le coût amorti est augmenté un peu en ce moment. En plus, le coût amorti est diminué au fur et à mesure jusqu'à la prochaine extension.

Par exemple, la taille courante est égale à 32 et le moment où on a 24 éléments, la taille est doublée et le coût amorti est augmenté. Après cet agrandissement, le coût amorti est diminué au fur et à mesure jusqu'à la 48^{ème} opération (la taille sera 64 et le nombre d'éléments est 48).

21	1.0	1.90
22	1.0	1.86
23	1.0	1.83
24	24.0	2.75
25	1.0	2.68
26	1.0	2.62
27	1.0	2.56
28	1.0	2.50
29	1.0	2.45
30	1.0	2.40
31	1.0	2.35
32	1.0	2.31
33	1.0	2.27
34	1.0	2.24
35	1.0	2.20
36	1.0	2.17
37	1.0	2.14
38	1.0	2.11
39	1.0	2.08
40	1.0	2.05
41	1.0	2.02
42	1.0	2.00
43	1.0	1.98
44	1.0	1.95
45	1.0	1.93
46	1.0	1.91
47	1.0	1.89
48	48.0	2.85
49	1.0	2.82
50	1.0	2.78
51	1.0	2.75

Figure 11. Partie de nombre de copies dans Java

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 ?

e. Tentez d'expliquer pourquoi certains langages sont plus rapides que d'autres dans cette expérience

En général, C et C++ ont la vitesse d'exécuter plus vite que Java. Trois langages sont aussi **Interpreted Language**, mais Java s'exécute ses bytecodes sur Java Virtual Machine (**JVM**) parce le point essentiel de Java est « Write one, Run anywhere ». JVM convertira les bytecodes au code machine natif avec les stratégies comme: compiling, interpreting, Just In Time (JIT). Donc, Java s'exécute plus lentement que C ou C++ mais la distance est progressivement raccourcie par le nouveau JRE. Même dans certains cas, Java s'exécute plus rapidement que C ou C++.

Lors que les programmes Java sont compilés directement en JVM, Python est interprété ce qui ralentit les programmes Python pendant l'exécution. La détermination du type de variable qui survient pendant l'exécution augmente la charge de travail de l'interpréteur. De plus, le fait de se souvenir du type d'objet récupéré à partir d'objets de conteneur contribue à l'utilisation de la mémoire.

f. Observez l'espace mémoire inutilisé au fur et à mesure du programme. Qu'en pensez-vous ? Imaginez un scénario dans lequel cela pourrait poser problème.

On compare l'espace mémoire inutilisé entre C, C++ et Java dans la graphie et les résultats statistiques suivant:

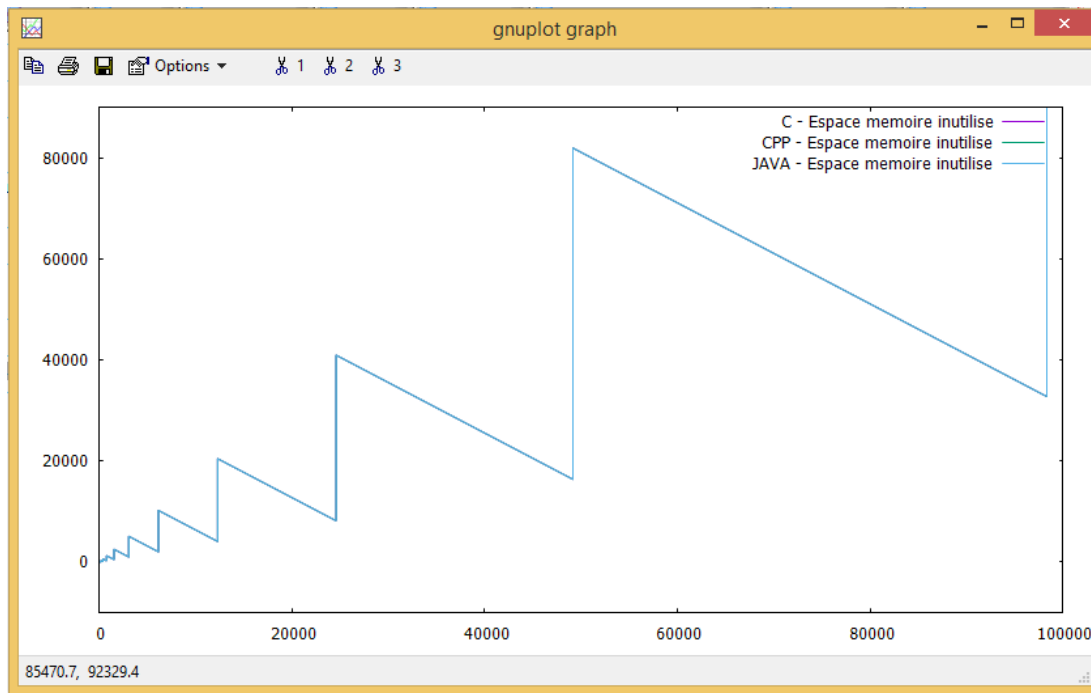


Figure 12. Espace mémoire inutilisé de C, de C++, de Java

```
0 1.0 1
1 1.0 2.00
2 1.0 1.50
3 3.0 2.00
4 1.0 1.75
5 1.0 1.60
6 6.0 2.33
7 1.0 2.14
8 1.0 2.00
9 1.0 1.89
10 1.0 1.80
11 1.0 1.73
12 12.0 2.58
```

Java

```
0 1.000000 1.000000
1 1.000000 2.000000
2 1.000000 1.500000
3 3.000000 2.000000
4 1.000000 1.750000
5 1.000000 1.600000
6 6.000000 2.333333
7 1.000000 2.142857
8 1.000000 2.000000
9 1.000000 1.888889
10 1.000000 1.800000
11 1.000000 1.727273
12 12.000000 2.583333
```

C

```
0 1 1
1 1 2
2 1 1.5
3 3 2
4 1 1.75
5 1 1.6
6 6 2.33333
7 1 2.14286
8 1 2
9 1 1.88889
10 1 1.8
11 1 1.72727
12 12 2.58333
```

C++

On réalise que : L'espace mémoire inutilisé est **pareil** entre Java, C et C++. Puisque les codages utilisent un même facteur multiplicatif et un même facteur de charge (a load factor). De plus, l'espace mémoire inutilisé fortement augmente avec le temps.

Dans le cas où le nombre d'insertion est stable avec le temps alors l'espace mémoire inutilisé sera fortement augmenté. Par exemple, si on ajoute 10 records par heure au dynamique tableau. Quand le nombre d'élément est 900000/1200000, la taille sera 2400000 et l'espace mémoire inutilisé sera 1500000. C'est le gaspillage parce que l'espace mémoire est inutilisé dans le longtems lorsque le nombre d'insertion par heure n'est pas changé.

Si on n'agit pas bien l'espace mémoire inutilisé, on peut avoir l'exception « **OutOfMemoryError** » qui est une erreur populaire dans la programmation, surtout dans Java et Android (la ressource de smartphone est limitée).

On peut dépendre des propriétés d'opération ou la capacité de RAM, de HardDisk, la capacité de processus pour décider à choisir le facteur multiplicatif et le facteur de charge. C'est-à-dire, équilibre entre la capacité de mémoire et le prix de temps est le plus important.

4. Modifiez la fonction « do_we_need_to_enlarge_capacity » pour ne se déclencher que lorsque le tableau est plein. Relancer une expérience et commentez les résultats.

a) On utilise la classe ArrayListProxy

D'abord, on change la fonction « do_we_need_to_enlarge_capacity » pour renvoyer toujours faux.

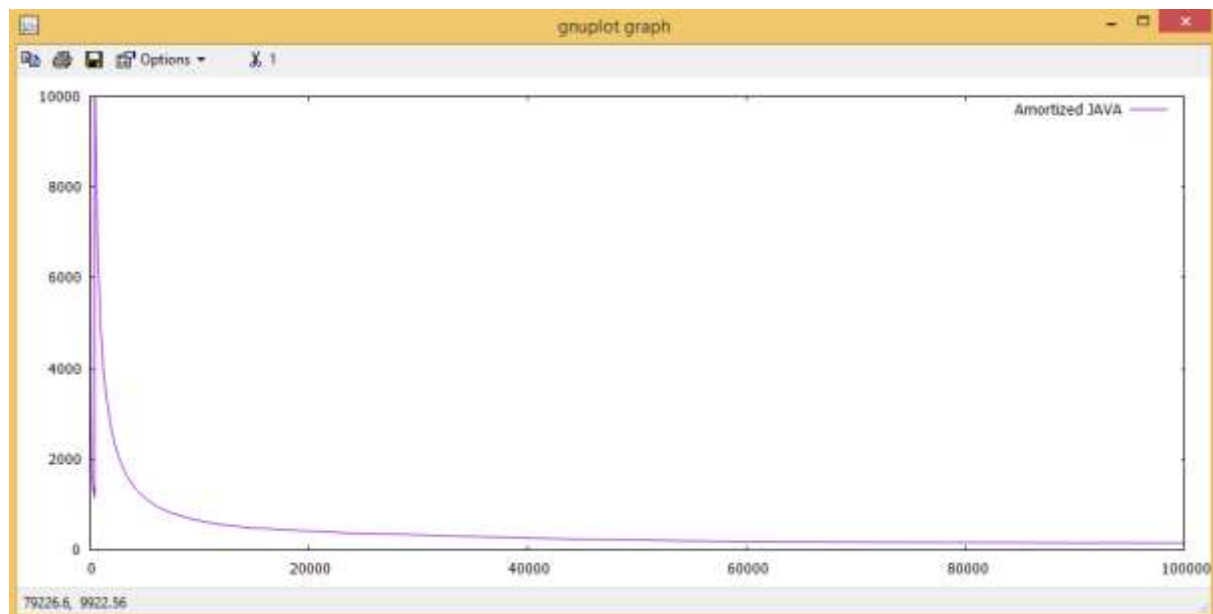
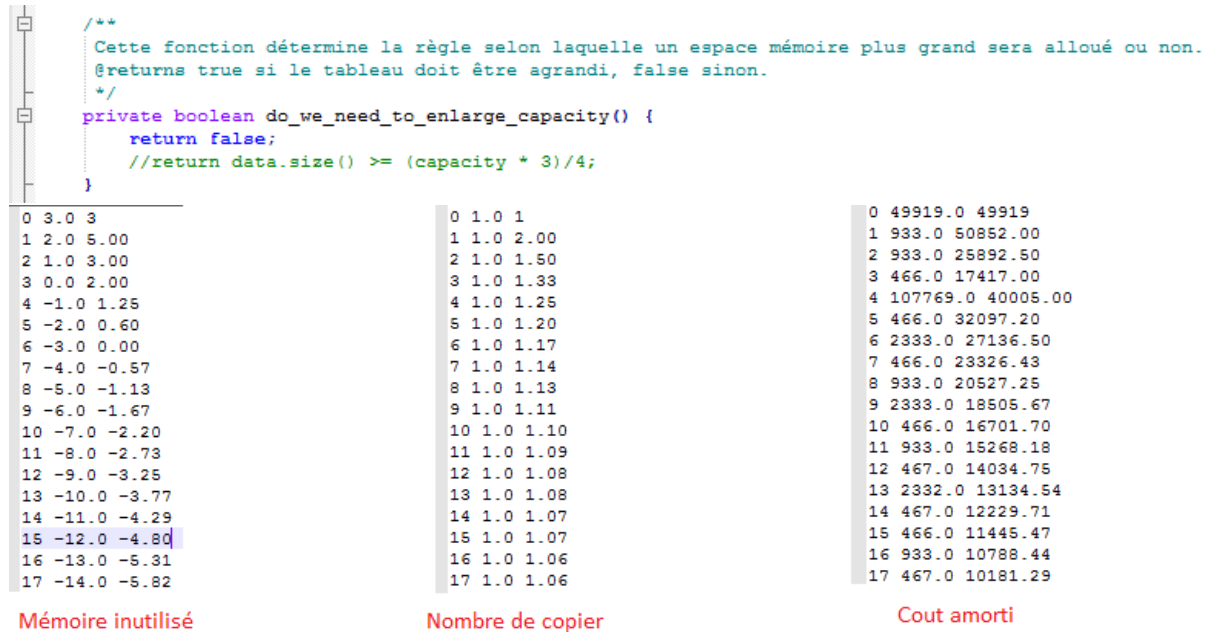


Figure 13. Le coût amortisé de Java avec plot [0:100000][0:10000]

On reconnaît que la classe ArrayListProxy augmente automatiquement la capacité du tableau puisqu'il est basé sur la classe ArrayList de Java.

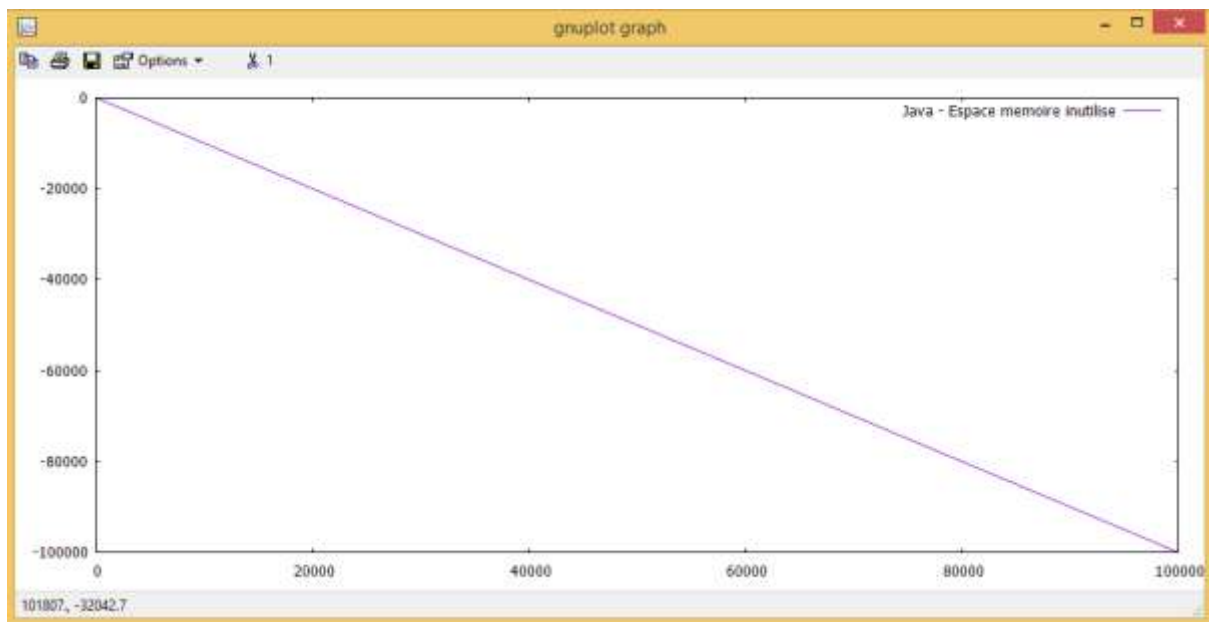


Figure 14. Espace mémoire inutilisé est négatif de Java

L'espace memoire inutilisé est négatif puisque cette méthode « **enlarge_capacity** » n'est pas appelle pour actualiser la taille courant du tableau.

```
/**
 * Cette fonction augmente la capacité du tableau.
 */
private void enlarge_capacity(){
    capacity *= 2;
    data.ensureCapacity( capacity );
}
```

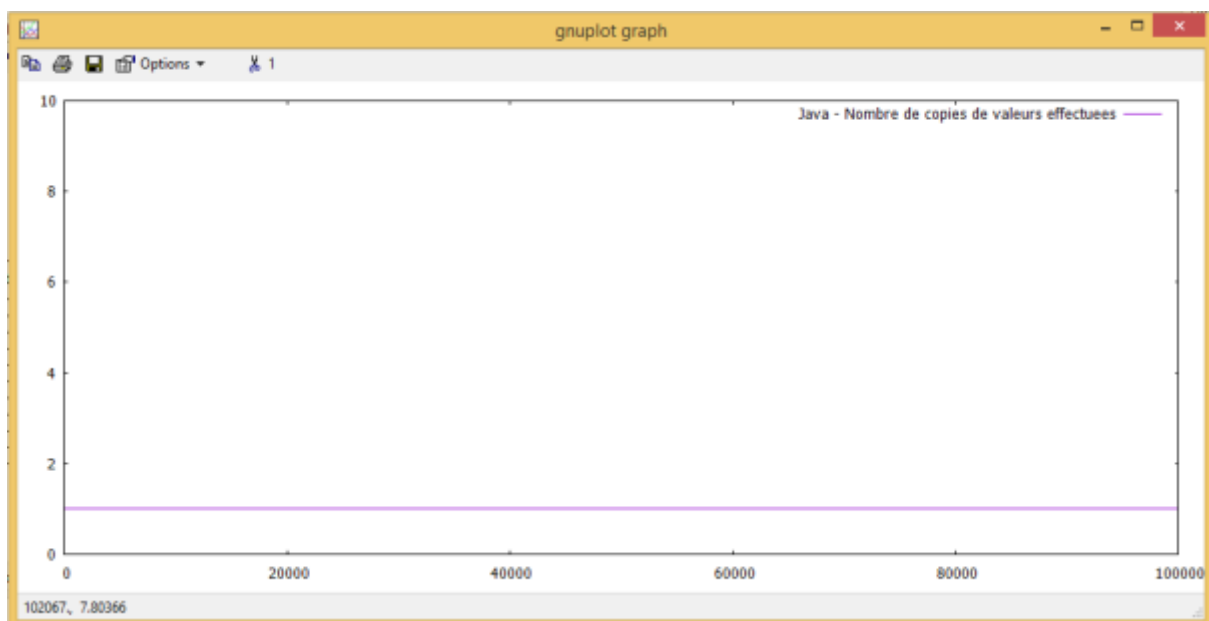


Figure 15. Nombre de copies est toujours égale 1

Nombre de copies est toujours égale à 1 puisque dans la classe Main, la variable « *memory_allocation* » est toujours égale à 1.

```
// Enregistrement du temps pris par l'opération
time_analysis.append(after - before);
// Enregistrement du nombre de copies effectuées par l'opération.
// S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
copy_analysis.append( (memory_allocation == true)? 1: 0);
```

Quand on insère un nouveau élément, ArrayList appelle d'abord la méthode « **ensureCapacityInternal (currentLength + 1)** » pour valider la taille courante.

```
436    /**
437     * Appends the specified element to the end of this list.
438     *
439     * @param e element to be appended to this list
440     * @return <tt>true</tt> (as specified by {@link Collection#add})
441     */
442    public boolean add(E e) {
443        ensureCapacityInternal(size + 1); // Increments modCount!!
444        elementData[size++] = e;
445        return true;
446    }
```

Figure 16. Source code de la méthode add de ArrayList

Si le tableau est plein, il va augmenter la taille de ce tableau en appelant la méthode « **grow(minCapacity)** »

```
207    private void ensureCapacityInternal(int minCapacity) {
208        if (elementData == EMPTY_ELEMENTDATA) {
209            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
210        }
211
212        ensureExplicitCapacity(minCapacity);
213    }
214
215    private void ensureExplicitCapacity(int minCapacity) {
216        modCount++;
217
218        // overflow-conscious code
219        if (minCapacity - elementData.length > 0)
220            grow(minCapacity);
221    }
```

La nouvelle taille est plus 1.5 fois l'ancienne taille dans la classe ArrayList de Java.

```
231  /**
232   * Increases the capacity to ensure that it can hold at least the
233   * number of elements specified by the minimum capacity argument.
234   *
235   * @param minCapacity the desired minimum capacity
236   */
237  private void grow(int minCapacity) {
238      // overflow-conscious code
239      int oldCapacity = elementData.length;
240      int newCapacity = oldCapacity + (oldCapacity >> 1);
241      if (newCapacity - minCapacity < 0)
242          newCapacity = minCapacity;
243      if (newCapacity - MAX_ARRAY_SIZE > 0)
244          newCapacity = hugeCapacity(minCapacity);
245      // minCapacity is usually close to size, so this is a win:
246      elementData = Arrays.copyOf(elementData, newCapacity);
247  }
```

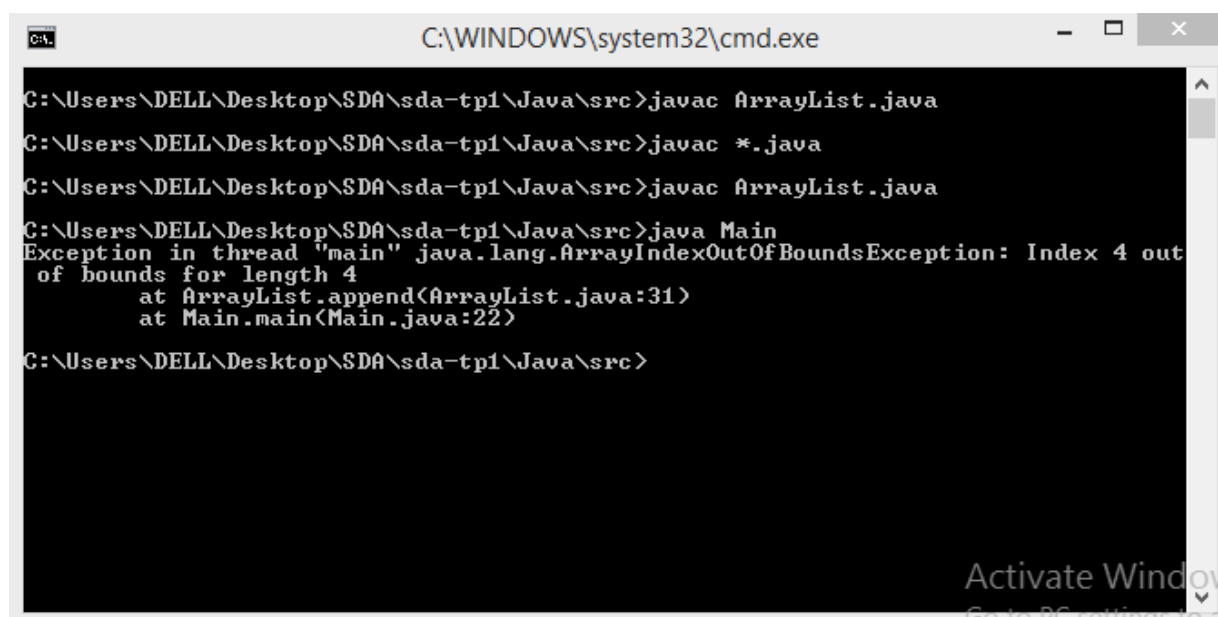
Figure 17. La méthode grow de ArrayList avec le facteur multiplicatif 1.5

b) On utilise la implémentation simple de ArrayList

D'abord, on change la fonction « do_we_need_to_enlarge_capacity » pour renvoyer toujours faux.

```
/**
 * 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;
    return false;
}
```

Après, on exécute la classe Main et le résultat est une exception **ArrayIndexOutOfBoundsException** de Java comme ci-dessous :



```
C:\WINDOWS\system32\cmd.exe

C:\Users\DELL\Desktop\SDA\sda-tp1\Java\src>javac ArrayList.java
C:\Users\DELL\Desktop\SDA\sda-tp1\Java\src>javac *.java
C:\Users\DELL\Desktop\SDA\sda-tp1\Java\src>javac ArrayList.java
C:\Users\DELL\Desktop\SDA\sda-tp1\Java\src>java Main
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out
of bounds for length 4
    at ArrayList.append(ArrayList.java:31)
    at Main.main(Main.java:22)
C:\Users\DELL\Desktop\SDA\sda-tp1\Java\src>
```

5. Dans la fonction `enlarge_capacity`, faites varier le facteur multiplicatif `_`. Qu'est-ce qui se passe-t-il ? Donner une règle, décrivant le rapport entre le coût en temps et le coût en espace.

En raison de l'influence d'autres processus, le coût de temps n'est pas exact et il a une grande variance. Alors on regarde le nombre de copier comme une unité indépendante qui n'est pas effectuée par l'autre processus. Ça n'est exactement ni la méthode de l'agrégat ni la méthode de potentiel mais on croit que le nombre de copier est plus exact que le coût de temps. C'est-à-dire, dans un même environnement, un même langage et ces opérations d'insérer, le nombre de copier est la somme approximative d'exécutions de programme.

Pour le facteur multiplicatif $\alpha = 1.5$, on a les résultats ci-dessous :

- Total cost : 95649593
- Average cost : 95.649593
- Nombre moyen de copies dans 100000 opérations : 3.07
- De plus, le nombre moyen de copies est entre 3 et 4
- L'espace mémoire inutilisé est petit

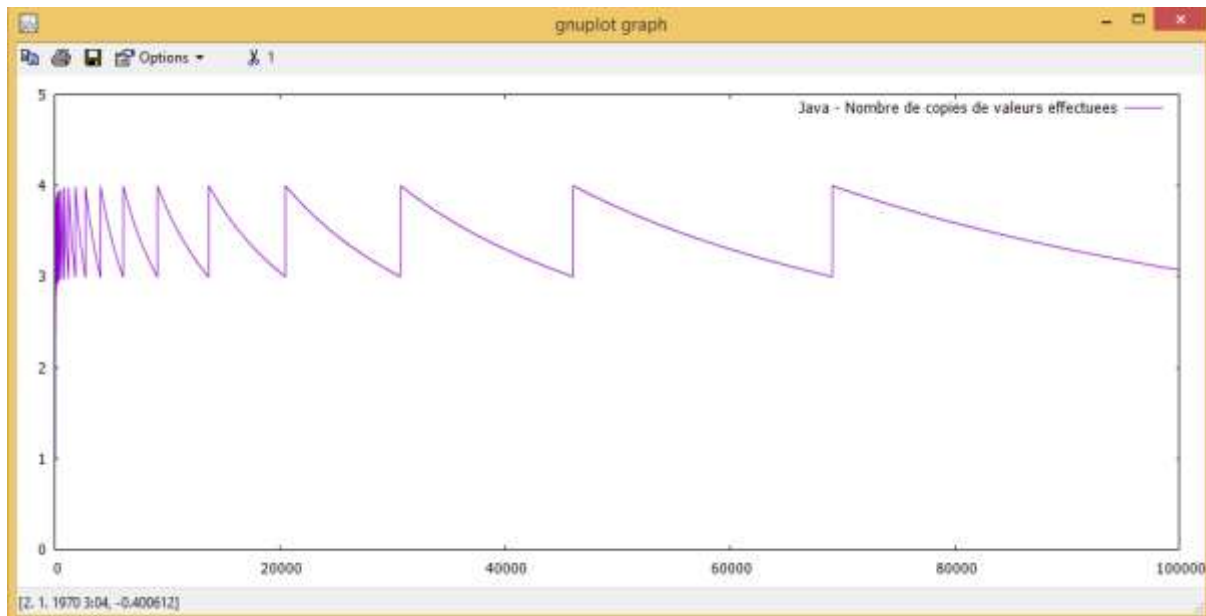


Figure 18. Le nombre moyen de copies pour le facteur multiplicatif 1.5

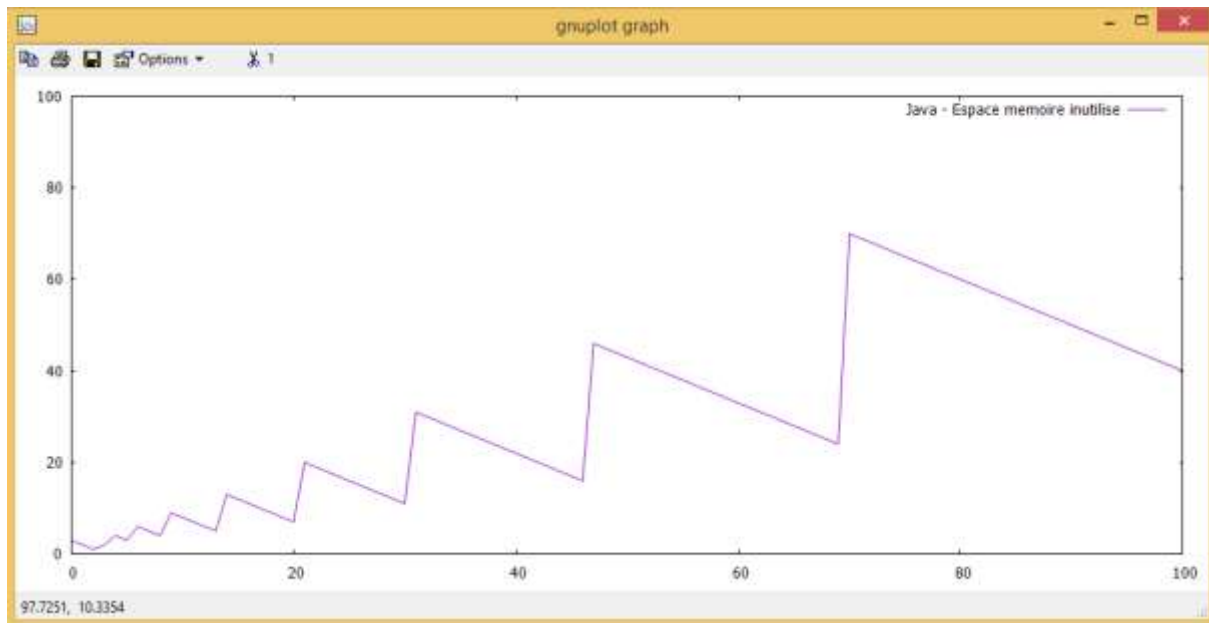


Figure 19. L'espace mémoire inutile pour le facteur multiplicatif 1.5

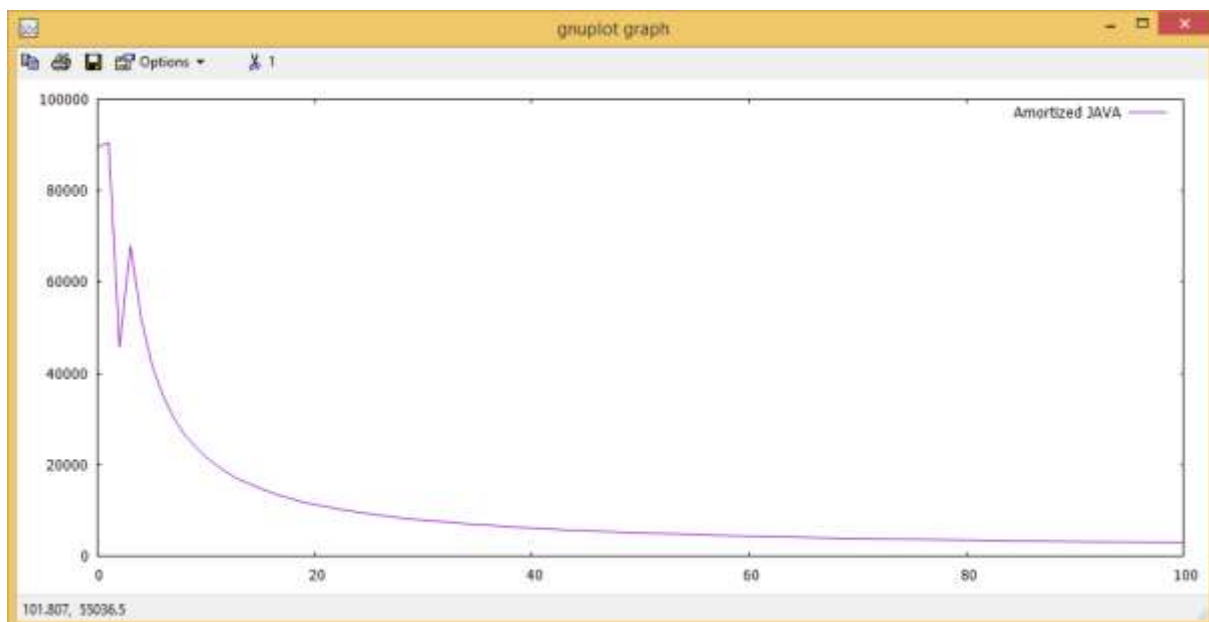


Figure 20. Le cout amorti pour le facteur multiplicatif 1.5

Pour le facteur multiplicatif $\alpha = 2$, on a les résultats ci-dessous :

- Total cost : 148816385
- Average cost : 148.816385
- Nombre moyen de copies dans 100000 opérations : 2.97
- De plus, le nombre de copies est entre 3 et 4
- L'espace mémoire inutilisé est moyen

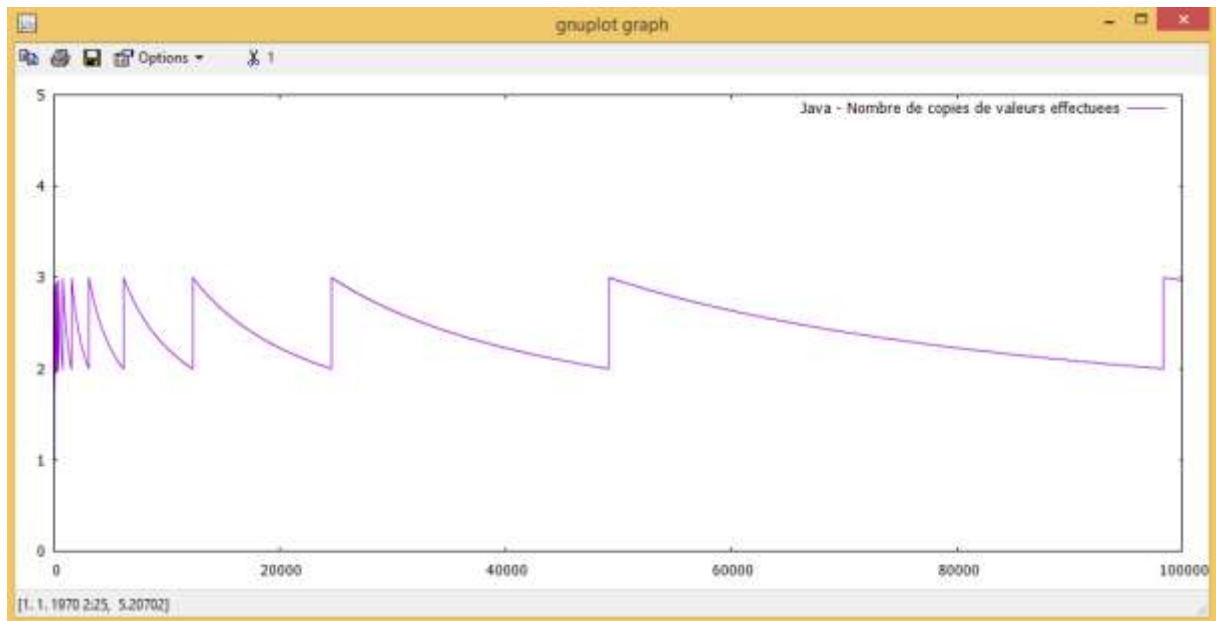


Figure 21. Le nombre moyen de copies pour le facteur multiplicatif 2.0

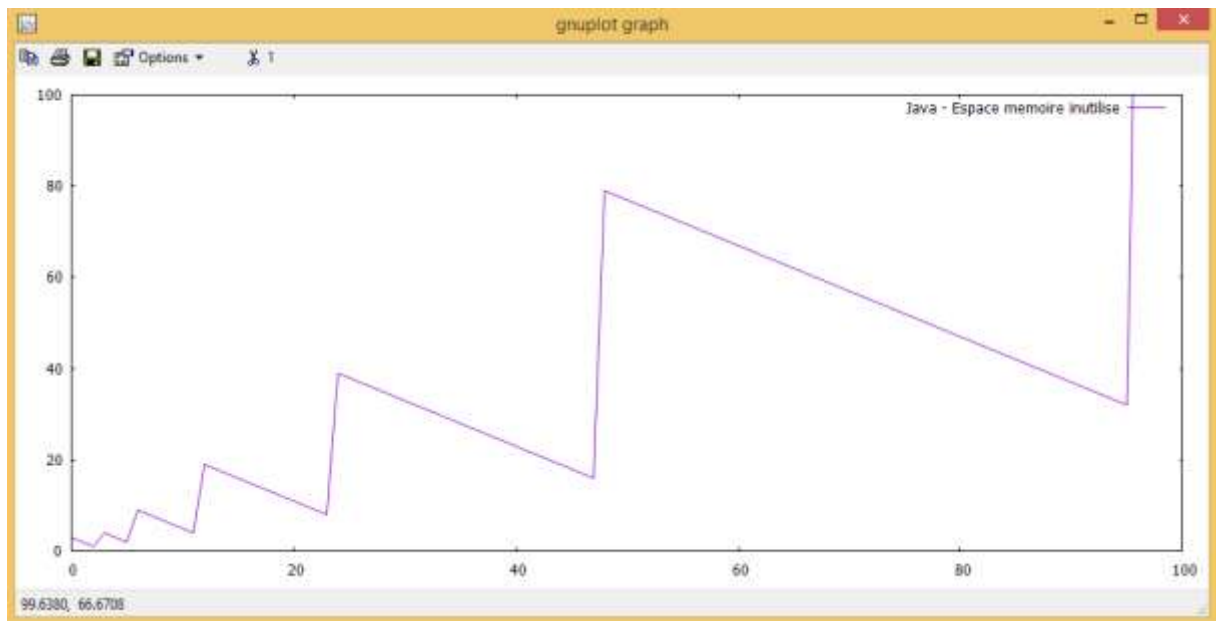


Figure 22. L'espace mémoire inutile pour le facteur multiplicatif 2.0

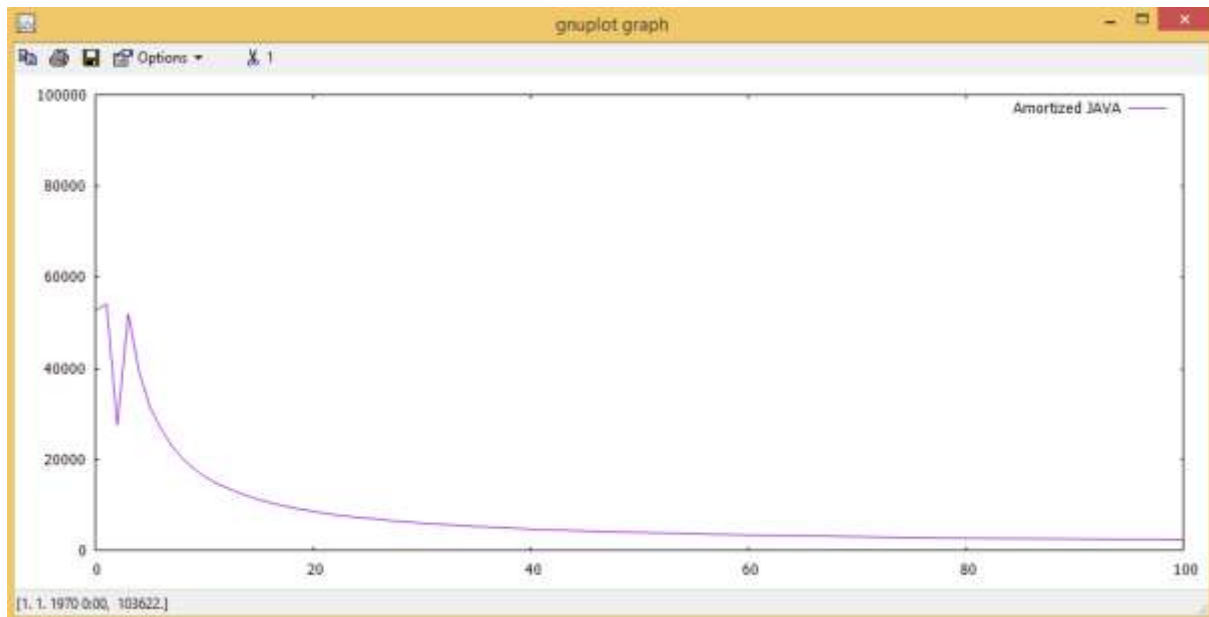


Figure 23. Le cout amorti pour le facteur multiplicatif 2.0

Pour le facteur multiplicatif $\alpha = 2.5$, on a les résultats ci-dessous :

- Total cost : 123901122
- Average cost : 123.901122
- Nombre moyen de copies dans 100000 opérations : 2.18
- De plus, le nombre de copies est entre 2.5 et 3.5
- L'espace mémoire inutilisé est très grand

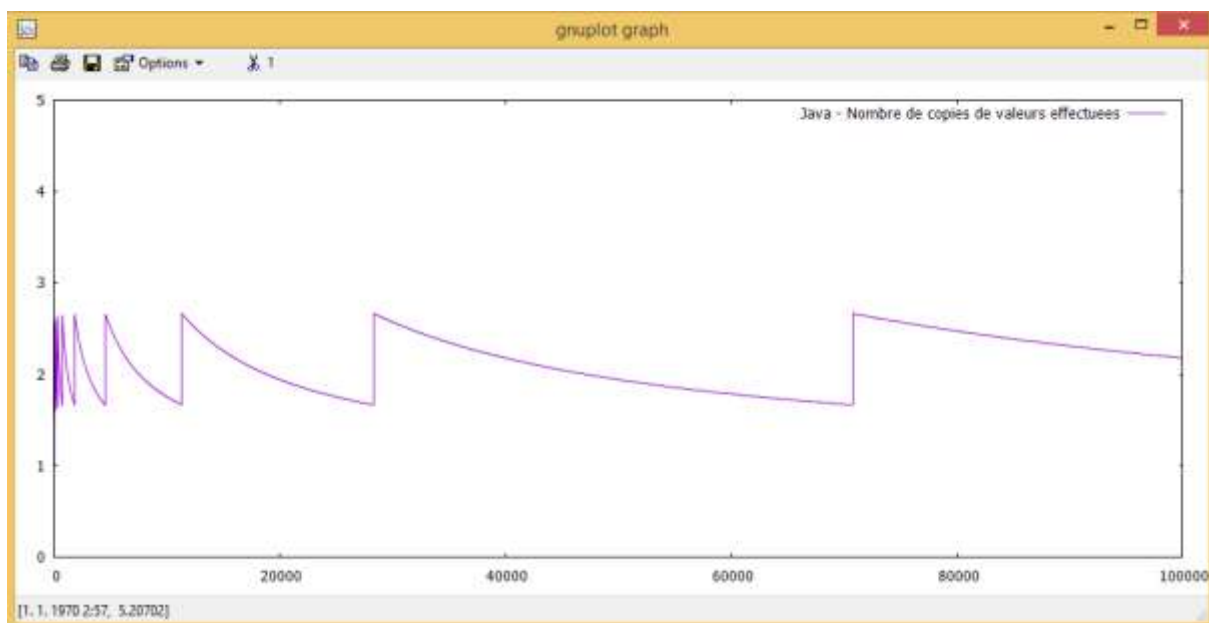


Figure 24. Le nombre moyen de copies pour le facteur multiplicatif 2.5

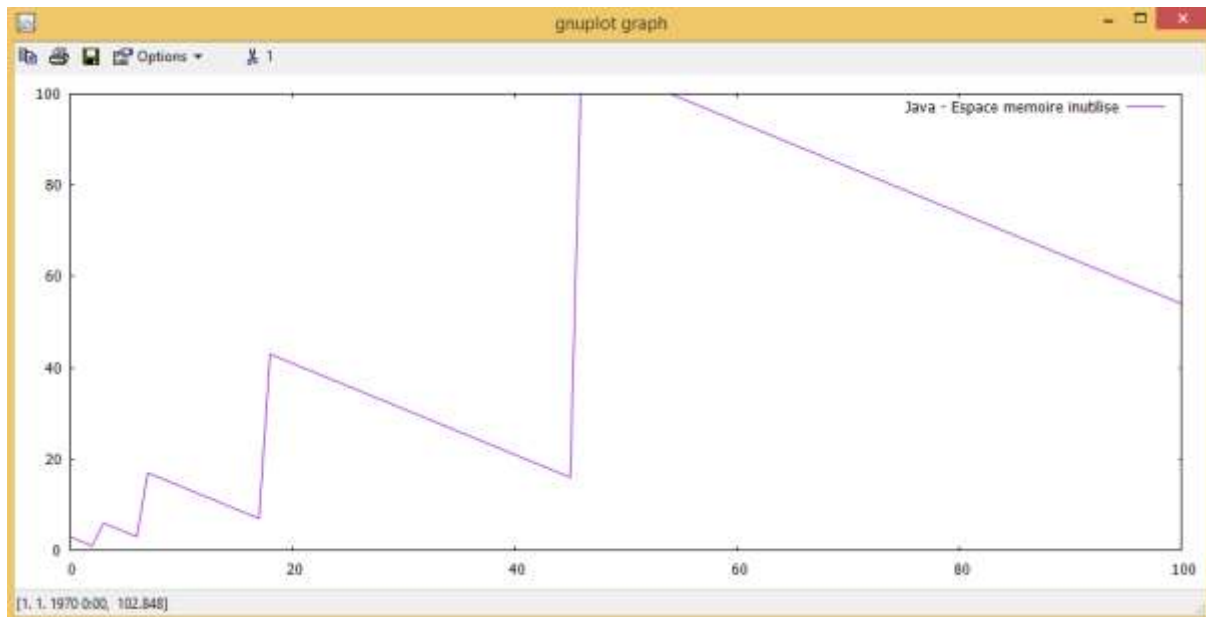


Figure 25. L'espace mémoire inutile pour le facteur multiplicatif 2.5

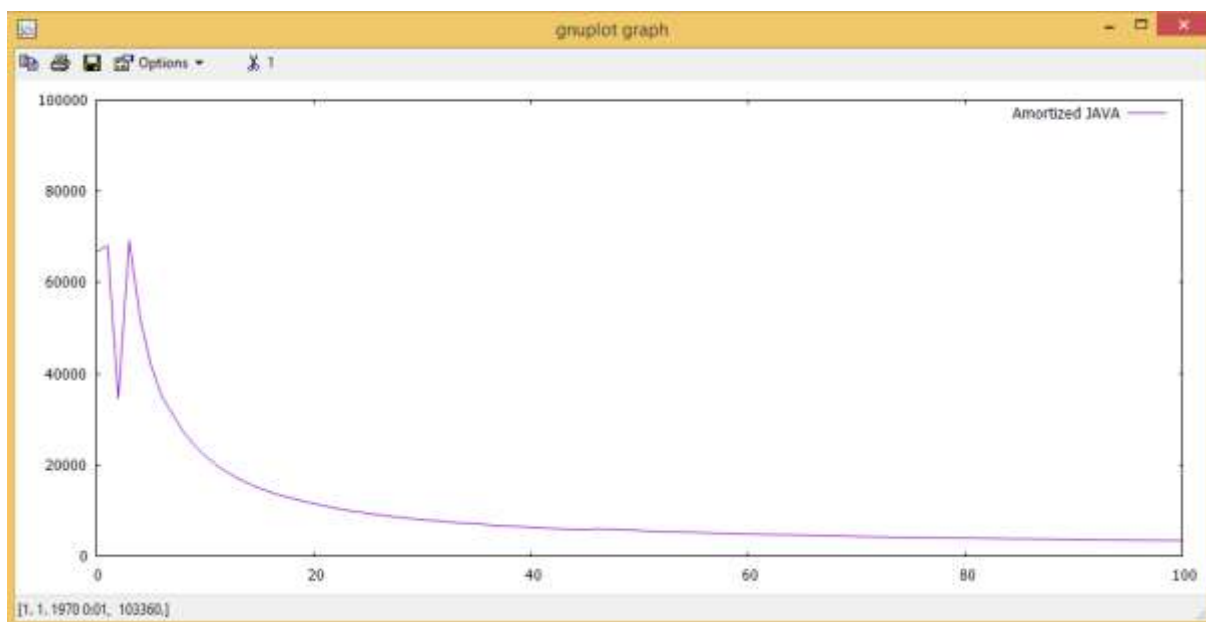


Figure 26. Le cout amorti pour le facteur multiplicatif 2.5

Si l'alpha est augmenté, le coût du temps sera réduit en conséquence. Parce que le programme prendra moins de temps pour allouer une nouvelle mémoire pour la taille de la table. Bien sûr, nous avons plus de mémoire inutilisée.

C'est une chose normale dans les technologies de l'information, nous devons trouver un équilibre entre la mémoire et le temps d'exécution. Si nous avons moins de mémoire, nous devons appliquer l'algorithme plus compliqué pour résoudre le problème.

6. Dans la fonction `enlarge_capacity`, faites varier la capacité n vers une capacité $n + \sqrt{n}$. Que se passe-t-il ?

D'abord, on change le fonction « `enlarge_capacity` » comme suivant :

```
/**
 * Cette fonction augmente la capacité du tableau.
 */
private void enlarge_capacity(){
    capacity = capacity + (int) Math.sqrt(capacity);
    data.ensureCapacity( capacity );
}
```

Pour le facteur multiplicatif $\alpha = 2.5$, on a les résultats ci-dessous :

- Cout en temps est le plus grand : 4697073360
- Cout moyen en temps est le plus grand : 4697.07336
- Nombre moyen de copies dans 100000 opérations : 16787.972
- De plus, le nombre de copies est très grand.
- L'espace mémoire inutilisé est petit

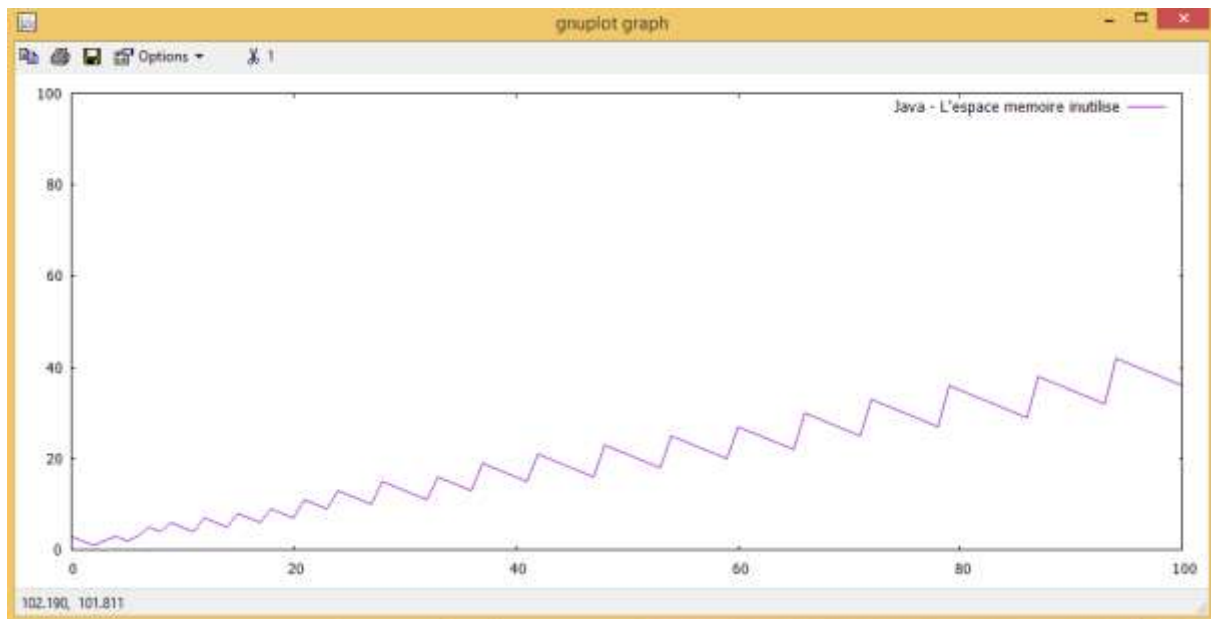


Figure 27. L'espace mémoire inutile pour le facteur multiplicatif $n + \sqrt{n}$

Le nombre moyen de copier est plus augmenté que les précédents alphas.

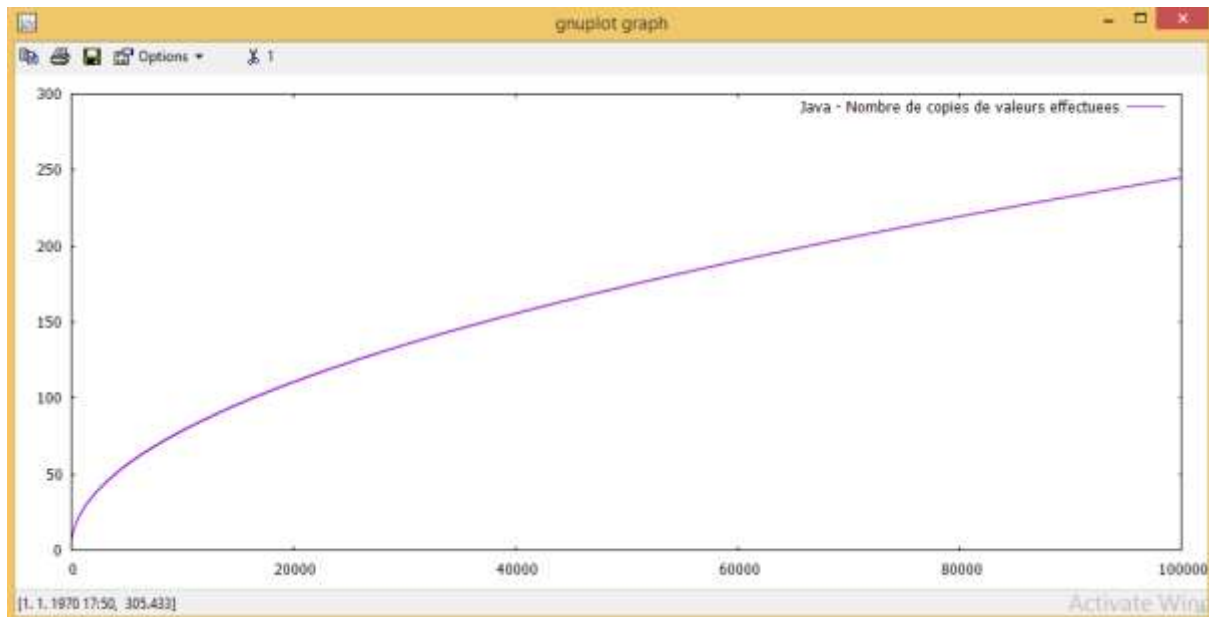


Figure 28. Le nombre **moyen** de copies pour le facteur multiplicatif $n + \sqrt{n}$

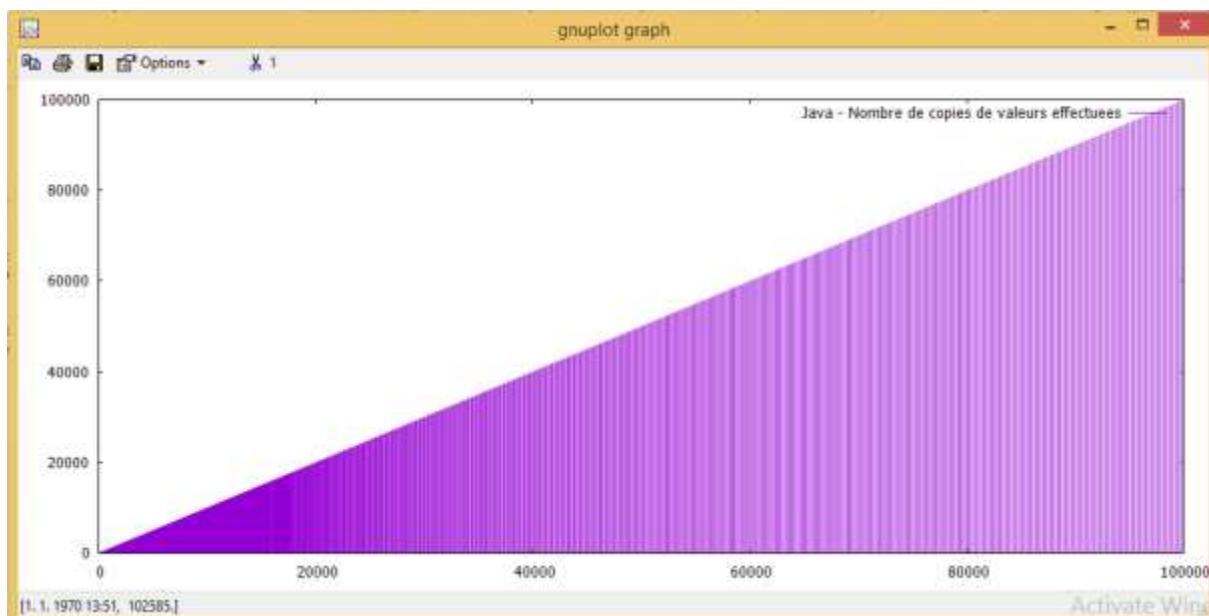


Figure 29. Le nombre de copies pour le facteur multiplicatif $n + \sqrt{n}$

Parce que l'on doit allouer un nouveau tableau plusieurs fois et on prend plus de effort pour copier l'ancien contenu au nouveau tableau surtout le grand tableau.

Par exemple : quand la taille est égale 100000, \sqrt{n} est 100. C'est-à-dire, on doit allouer un nouveau tableau et copier environ 100000 éléments chaque 100 insertions. **Cette stratégie est très onéreuse pour la très grande taille du tableau.**