Performance des types de base Entier

L'objectif d'un calcul de complexité algorithmique temporelle est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème.

Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Lors de la présentation on vous a dit, qu'il ne fallait pas aller chercher très loin les informations, dans ce premier sujet on vous disait de consulter la fiche complémentaire sur les types primitifs

Vous pouviez remarquer juste au-dessous du lien de cette fiche, une fiche complémentaire sur Accéder aux ressources Systèmes.

En l'ouvrant vous pouvez immédiatement lire dans le contenu de la page qu'il y a une rubrique sur les temps d'exécution.

Deux fonctions s'offrent alors à vous, nous avons une préférence pour la fonction nanotime, qui permet une plus grande précision.

En lisant la fiche complémentaire sur les types primitifs, on s'aperçoit qu'il existe en Java quatre types pour coder les entiers.

Et qu'ils ont une taille différente

byte 1 octet

short 2 octets

int 4 octets

long 8 octets

A priori on pourrait penser que les type les moins gros sont plus performants, aussi nous vous invitions à faire des boucles avec ces différents types.

Et précisément 100 millions d'itérations (100_000_000)

Voici la structure de base que l'on utilisera :

```
tps1 = System.nanoTime();

cpt1 = 0;
while ( cpt1 < 100_000_000 )
{
	cpt1++;
}

tps2 = System.nanoTime();
```

cpt1 sera successivement un byte un short un int et un long

Problème les byte et les short ne sont pas suffisament grand pour stocker la valeur de 100 millions

La solution consiste à faire pur les short une double boucle :

```
cpt1=0;
while ( cpt1 < 10000 )
{
	cpt2=0;
	while ( cpt2 < 10000 )
	{
		cpt2++;
	}
	cpt1++;
}
```

Et pour les bytes une quadruple boucle :

```
tps1 = System.nanoTime();

cpt1=0;
while ( cpt1 < 100 )
{
	cpt2=0;
	while ( cpt2 < 100 )
	{
		cpt3=0;
		while ( cpt3 < 100 )
		{
			cpt4=0;
			while ( cpt4 < 100 )
			{
				cpt4++;
			}
			cpt3++;
		}
		cpt2++;
	}
	cpt1++;
}

tps2 = System.nanoTime();
```

Seulement est-ce que les boucles imbriquées ont la même performance qu'une simple boucle.

Il suffit alors de le tester avec le type int.

D’autres points à prendre en compte :

Votre programme ne tourne pas seul sur votre machine, il faut donc limiter au maximum l’activité de cette dernière.

Avoir le moins d’application ouverte, fermer sa connexion internet. Malgré toutes ces précautions l’activité de la machine peut varier entre deux instants, aussi il s’avère nécessaire de ne pas lancer qu’une seule fois le programme, mais plusieurs fois, et calculer le temps moyen.

Dans vos résultats, nous souhaitions voir l’ensemble des temps d’exécution de plusieurs lancements de programme pour calculer une moyenne représentative.

Quand on donne une durée, on donne l’unité du temps, Quand on a durée de 0, c'est qu'il y a un gros problème ! Il faut faire tourner l'horloge (sans temporisation) suffisament longtemps pour qu'on ait une durée significative.

Il est également nécessaire de faire tourner les différents algorithmes sur une même machine.

En effet, selon la puissance de la machine, les durées peuvent être différentes, mais une durée est inversement proportionnelle à la puissance, c'est-à-dire qu'avec une machine deux fois plus puissante, la durée sera 2 fois plus courte. Mais si un algorithme est 3 fois plus rapide qu'un autre, cette proportion reste la même, même si on change de machine (on fait tourner les deux algorithmes dans les mêmes conditions). Ici, on est dans le cas d'algorithmes qui n'utilisent pas beaucoup de mémoire (sinon il faut prendre en compte la quantité de mémoire et la vitesse de celle-ci) ou qu'ils n'utilisent pas le fait de la présence ou non d'un GPU par exemple.

Voici une présentation possible des résultats :

byte	short	int	long
70 431 700 ns	62 629 000 ns	2 043 500 ns	32 043 500 ns
70 194 500 ns	60 987 400 ns	1 844 400 ns	32 257 200 ns
69 601 400 ns	62 784 000 ns	1 897 100 ns	45 581 700 ns
67 570 900 ns	61 210 500 ns	1 821 200 ns	32 104 400 ns
67 471 100 ns	61 342 600 ns	1 972 000 ns	32 972 400 ns
68 959 700 ns	60 922 500 ns	2 396 600 ns	31 860 900 ns
69 448 500 ns	61 332 400 ns	2 121 700 ns	31 914 500 ns
68 858 700 ns	63 677 000 ns	1 953 900 ns	38 160 400 ns
72 296 500 ns	61 536 100 ns	1 897 400 ns	32 461 800 ns
69 613 700 ns	62 524 600 ns	2 566 400 ns	32 314 200 ns
69 444 670 ns	61 894 610 ns	2 051 420 ns	34 167 100 ns

Certes un byte prend 4 fois moins d’espaces mémoire qu’un Int mais sa manipulation est 33x plus lente. Il est donc nécessaire de savoir si on souhaite privilégié l’espace mémoire ou la vitesse.

Horloge

Dans le cas de notre horloge il fallait être capable d'avoir à tout instant la partie heure minute et seconde.

Mais attention à l'affichage qui peut biaiser totalement vos résultats.

Partons des deux solutions suivantes :

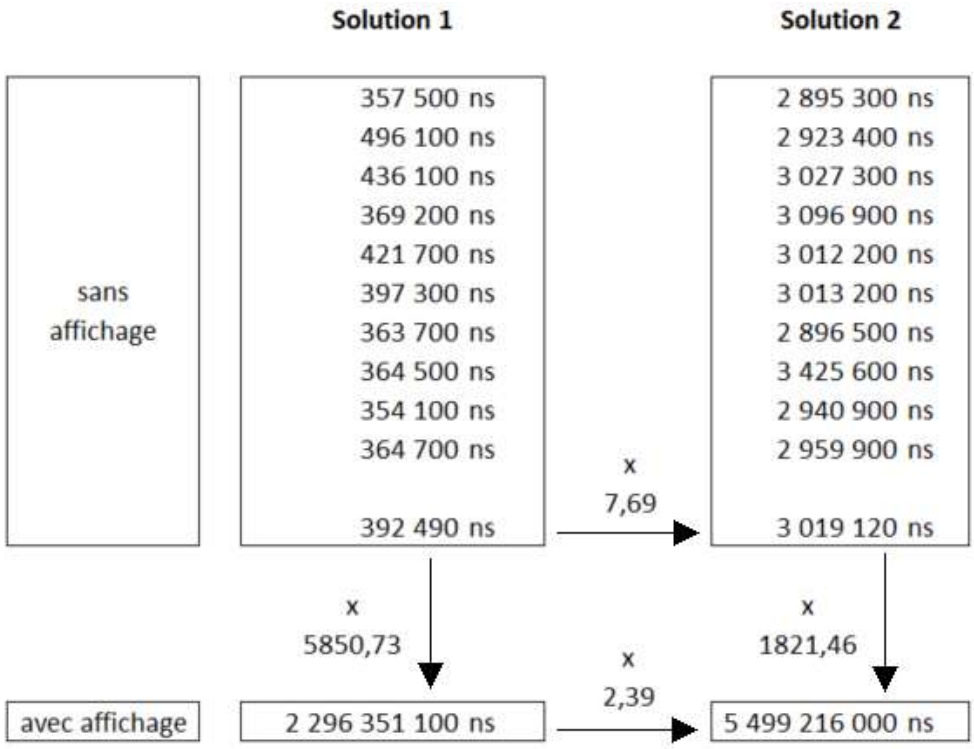
Solution 1

```
h = m = s = 0;

for (int cpt=1; cpt< 120_000; cpt++)
{
   s++;

   if ( s == 60 ){   s = 0; m++;   }
   if ( m == 60 ){   m = 0; h++;   }
   if ( h == 24 ){   h = 0;        }

   // System.out.println
   //        ( h + ":" + m + ":" + s );

}
```

Solution 2

```
totalS = 0;

for (int cpt=1; cpt< 120_000; cpt++)
{

   totalS++;

   h = (totalS / 3600) % 24;
   m = (totalS % 3600) / 60;
   s = totalS % 60;

   // System.out.println
   //        ( h + ":" + m + ":" + s );

}
```

Dont voici les temps d'exécution



Pour les temps avec affichage les deux Sop ont été décommentés.

Certes il demeure toujours un écart favorable à la première solution, mais on peut remarquer que l'affichage ralentit considérablement le temps d'exécution.

Donc nous retenons une solution sans affichage.