**Question 1.1**

Voir **Q1.1.png**.

**Question 1.2**

Voir **Q1.2.png**.

**Question 1.3**

On sait qu’un int pèse 8 octets en mémoire :

Pour la *première* structure, sachant que la matrice de l’exemple est une matrice 10x10, elle occupe donc 800 octets (10\*10\*8).

On sait qu’une size\_t pèse 16 octets en mémoire :

Calculons la taille d’une struct triplet : il possède 2 size\_t (2\*16 octets) et 1 int (8 octets), soit une taille totale de 40 octets.

Calculons ensuite la taille d’une CNode<triplet> : il possède 1 struct triplet (40 octets) et 2 shared\_ptr<CNode> (2\*8 octets), soit une taille totale de 56 octets.

Pour la *deuxième* structure, sachant que la CList<triplet> matrix contient 6 éléments de type CNode<triplet> (tête fictive, queue fictive et 4 éléments de la matrice ≠ 0), elle occupe donc 336 octets (6\*56).

Pour la *troisième* structure, sachant que le std::vector<CList<pairColVal>> matrix contient 10 têtes fictives et 10 queues fictives (car comme la matrice d’exemple possède 10 lignes, le vecteur possède 10 CList<pairColVal> qui possèdent donc automatiquement une tête et queue fictives chacun). De plus, il existe 4 éléments ≠ 0 dans la matrice, ce qui fait au total 24 éléments de stocké, elle occupe donc 1344 octets (24\*56).

Ces calculs n’ont pas pris en compte l’overhead introduit par des éléments « externes », tel que la taille du vecteur lui-même ou encore l’alignement en mémoire de la structure.

**Question 1.4**

TBA

**Question 1.5**

TBA

**Question 2.1**

Voir **cmatrix2** dans code source inclus.

**Question 2.2**

Voir **cmatrix2** dans code source inclus.

**Question 2.3**

Voir **cmatrix2** dans code source inclus.