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
Pour la *première* structure, la matrice est de 100x100, elle occupe donc 80 000 octets (100*100*8).
Pour la *deuxième* structure, elle occupe toujours 336 octets (6*56), car aucun nouvel élément différent de 0 n'a été introduit.
Pour la *troisième* structure, il y a 100 lignes donc 100 têtes et 100 queues fictives plus les 4 éléments différent de 0, elle occupe donc 11 424 octets (204*56).

Question 1.5
Pour la *première* structure, la matrice est de 1 000x1 000, elle occupe donc 8 000 000 octets (1 000*1 000*8).
Pour la *deuxième* structure, elle occupe toujours 336 octets (6*56), car aucun nouvel élément différent de 0 n'a été introduit.
Pour la *troisième* structure, il y a 1 000 lignes donc 1 000 têtes et 1 000 queues fictives plus les 4 éléments différent de 0, elle occupe donc 112 224 octets (2 004*56).

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.

Question 2.4
Nous devons stocker la donnée membre **`_size`** dans la classe **`cmatrix2`** car sans elle, il nous serait impossible de connaitre la taille de la matrice d'origine, car nous ne stockons que les valeurs différentes de zéro et non la matrice entière.

Question 3.1
Voir **`cmatrix3`** dans code source inclus.

Question 3.2
Voir **`cmatrix3`** dans code source inclus.

Question 3.3
Voir **`cmatrix3`** dans code source inclus.

Question 4.1
Sachant que la complexité sur un accès aléatoire dans un **`std::vector`** est de *O(1)* :
C'est la première structure de données dans tous les cas.

Question 4.2
Car elle fait le moins d'opérations, il suffit juste de retourner la case a l'indice souhaité de **`matrix`**, case qui est une ligne de la matrice.

Question 4.3
C'est la première structure de données dans tous les cas.

Question 4.4
Car nous avons juste à, pour chaque ligne, copier la valeur a la colonne souhaitée dans un vecteur de retour, au même index que le numéro de ligne courant.
Les autres structures de données doivent premièrement initialiser le vecteur de retour à 0 pour n lignes, puis :

- Vérifier le numéro de colonne pour chaque node de la liste de **`cmatrix2`**, et le copier au bon endroit dans le vecteur de retour.
- Vérifier le numéro de colonne de chaque node de chaque liste du vecteur de **`cmatrix3`**, et le copier au bon endroit dans le vecteur de retour.

Question 4.5
C'est la première structure de données dans tous les cas.

Question 4.6
Car nous avons juste à retourner la valeur a [ligne][colonne] de la matrice.
Les autres structures de données doivent :

- Vérifier le numéro de colonne et de ligne pour chaque node de la liste de **`cmatrix2`**, et retourner la valeur associée si ces conditions sont vérifiées.
- Vérifier le numéro de colonne de chaque node dans la liste associée à la ligne spécifiée, du vecteur de **`cmatrix3`**, et retourner la valeur associée si ces conditions sont vérifiées.
- Retourner 0 si la recherche n'a rien donnée dans ces deux dernières structures.

Question 4.7
On aurait pu utiliser la structure suivante : **`CList<std::pair<unsigned, CList<pairColVal>>`**

Question 4.8
~~Parce que c'était écrit en bas de page 2.~~
Car nous avons bien nos trois valeurs symbolisant le numéro de ligne (première valeur de la première paire), le numéro de colonne (première valeur de la seconde paire) et la valeur a cet endroit (seconde valeur de la seconde paire).
Nous avons aussi dans la première liste toutes nos lignes, avec une seconde liste associée a chaque ligne contenant les colonnes et leur valeur.