The Problem

- This first assignment is about operations on sparse matrices.
- Use the existing implementation of the **`SparseMatrix`** class in the textbook as your basis. You are going to add new functionalities to this class.
- The list of functionalities to add:
  - Initialize the matrix (constructor):
    - **`SparseMatrix(int rows, int cols);`**
    - Initially **`terms`** is zero (no non-zero terms).
    - Set initial **`capacity`** to one.
  - Retrieve an item given its row/column index:
    - **`float get(int row, int col);`**
    - Return zero if the term is not found.

The Problem

- The list of functionalities to add (continued):
  - Set the value of a term:
    - `void set(int row, int col, float value);`
    - Delete the term if `value` is zero.
    - Add a new term if necessary.
    - Implement the **double-the-capacity-as-needed** method to handle the case when more capacity is needed.
  - List the triples:
    - `void list_sparse();`
    - Print out all the terms in the matrix.
    - Put each term in one line in the triple form "`<row,col,value>`"

The Problem

- The list of functionalities to add (continued):
  - List the matrix in its normal matrix form:
    - `void list_full();`
    - All the terms (non-zero and zero terms) are printed.
    - Visually align the output.
    - (The test data here will be no more than 6-column wide.)
  - Matrix addition:
    - `SparseMatrix Add(SparseMatrix b);`
    - Implement this using the same idea as the addition of sparse polynomials.
    - Return `*this + b` as a new `SparseMatrix` object.

Notes

- Always do the necessary range/size checking of the input arguments.
- Use binary search when looking for a particular term.
- Make use the `rowStart` array, if possible, in your operations. (It is more practical to pre-compute the `rowStart` array and store it as a separate member of the `SparseMatrix` class.)
- Make sure that all the terms are still in the correct order after each operation.

Additional Analysis

- For this part, you need to submit a separate file (Microsoft **Word** of **PDF** format, 1-2 pages).
- In this file, write your analysis about the following:
  - The time complexity for these operations: **`get`**, **`set`**, **`Add`**.
    - Clearly indicate your instance characteristics.
    - Explain how you derive your complexities.
  - Does the use of binary search (instead of linear search) for finding a term improve the complexities? Give your reasoning.
  - Does the use of the **`rowStart`** array improve the complexities?
  - What are the complexities of keeping the content of the **`rowStart`** array correct after each **`set`** operation?

The Guidelines (Programming Part)

- Allowed programming environment: **VS2015** only.
- For simplicity of submission, put the code of the whole C++ class, including the implementation part, in **a single header file**.
- You need to write your own `main` function to test your code. You do not need to include this `main` function in your submission.
- No usage of STL class templates allowed.
- Include documentation; this will be part of your grade.
- Demo: Only a randomly selected subset of students; the list will be announced separately after the due date.

Submission

- Use E3 only.
- For the code, submit it under "**Assignment #1 – Programming Part**". Name your code `P1_xxxxxx.h`, where `xxxxxx` is your ID. **Do not** submit your `main` function or any file that is not your code (such as the *.sln file). No compressed file (*.zip, *.rar, etc.) accepted.
- For the analysis, submit it under "**Assignment #1 – Analysis Part**". Name your file `P1_xxxxxx.docx` or `P1_xxxxxx.pdf`.
- Due date: **10/18/2016**. There's a grace period of 3 days with 10% deduction per day. (The deduction kicks in only when you have accumulated more than three days of delay during the semester.)