Instructions Follow instructions *carefully*, failure to do so may result in points being deducted. Hand in all your source code files through webhandin and make sure your programs compile and run by using the webgrader interface. You can grade yourself and re-handin as many times as you wish up until the due date. Print a hardcopy of the rubric for this assignment and hand it in by the due date.

Partner Policy You may work in pairs for this assignment if you chose. If you do work in any groups or pairs, you must follow these guidelines:

1. You must work on *all* problems *together*. You may not simply partition the work between you.
2. You should not discuss problem details with other groups or individuals beyond general questions.
3. Hand in only one hard copy (and one soft copy) under the first author's name/cse login. Be sure to include both names.

Programs

1. **Arrays** In this exercise you will write several functions that operate on arrays. You will place all the function prototypes in a header file named `array_utils.h` with their definitions in a source file named `array_utils.c`. You should test your functions by writing a driver program, but you need not hand it in.
   (a) Write a function that takes an integer array and returns the sum of its elements between two given indices, i, j:
   ```
   int subSum(const int *a, int size, int i, int j);
   ```
   (b) Write a function that takes an integer array and returns the sums all of its elements:
   ```
   int sum(const int *a, int size);
   ```
   (c) In an array, a contiguous subarray consists of all the elements between two indices i, j; the sum of the subarray is the sum of all the elements between i, j. The *maximum* contiguous subarray problem is the problem of finding the i, j whose subarray sum is maximal (it may not be unique). Write a function that takes an integer array and returns the sum of the maximum contiguous subarray.
   ```
   int maxSubArraySum(const int *a, int size);
   ```
   (d) Write a function that, given an integer array and an integer x determines if the array contains x within two provided indices, i, j
   ```
   int containsWithin(const int *a, int size, int x, int i, int j);
   ```
   (e) Write a function that, given an integer array and an integer x determines if the array contains x anywhere within the array.
   ```
   int contains(const int *a, int size, int x);
   ```
   (f) Write a function that takes *two* integer arrays and determines if they are *equal*: that is, each index contains the same element. If they are equal, then you should return true (non-zero), if not return false (zero).
   ```
   int isEqual(const int *a, const int *b, int size);
   ```
   (g) Write a function that takes *two* integer arrays and determines if they both contain the same elements (though are not necessarily equal) regardless of their multiplicity. That is, the function should return true even if the arrays' elements appear a different number of times (for example, $2, 2, 3$ would be equal to an array containing $3, 2, 3, 3, 3, 2, 2, 2$).
   ```
   int containsSameElements(const int *a, int sizeOfA, const int *b, int sizeOfB);
   ```

(h) An array of size n represents a *permutation* if it contains all integers $0, 1, 2, \ldots, (n-1)$ exactly once. Write a function to determine if an array is a permutation or not.
`int isPermutation(const int *a, int size);`

(i) The k-th order statistic of an array is the k-th largest element. For our purposes, k starts at 0, thus the minimum element is the 0-th order statistic and the largest element is the $n-1$-th order statistic. Another way to view it is: suppose we were to *sort* the array, then the k-th order statistic would be the element at index k in the sorted array. Write a function to find the k-th order statistic:
`int orderStatistic(const int *a, int size, int k);`
Note: you may use the selection sort algorithm in the code below to help you. However, since the array `a` is labeled `const` in the prototype above, you cannot sort it directly. Think about implementing a deep-copy function for this purpose.

```
1  void selectionSort(int *a, int size) {
2    int i, j, min_index;
3    for(i=0; i<size-1; i++) {
4      min_index = i;
5      for(j=i+1; j<size; j++) {
6        if(a[min_index] > a[j]) {
7          min_index = j;
8        }
9      }
10     //swap
11     int t = a[i];
12     a[i] = a[min_index];
13     a[min_index] = t;
14   }
15 }
```

2. **Matrices** In this exercise you will write several functions that operate on *matrices* or two-dimensional arrays. For this exercise, we will restrict our attention to *square* matrices–collections of numbers with the same number of columns and rows. You will place all the function prototypes in a header file named `matrix.h` with their definitions in a source file named `matrix.c`. You should test your functions by writing a driver program, but you need not hand it in.

(a) Write a function that takes two matrices and determines if they are equal (all of their elements are the same).
`int isEqual(int **A, int **B, int n);`

(b) Write a function that takes a matrix and an index i and returns a new *array* that is equal to the i-th row of the matrix.
`int *getRow(int **A, int n, int i);`

(c) Write a function that takes a matrix and an index j and returns a new *array* that is equal to the j-th column of the matrix.
`int *getCol(int **A, int n, int j);`

(d) The product of two square $n \times n$ matrices:

$$C = A \times B$$

is defined as follows. The entries for C are:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

where $1 \leq i, j \leq n$ and c_{ij} is the (i, j)-th entry of the matrix C. Write the following function to compute the product of two $n \times n$ square matrices:

```
int ** product(int **A, int **B, int n);
```

(e) Iterated Matrix Multiplication is where you take a square matrix, A and multiply it by itself n times:

$$A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$$

Write a function to compute the n-th power of a matrix A. You should use the matrix multiplication algorithm above, but do so such that you have *no memory leaks* (points will be deducted for incorrect implementations.

```
int **matrixPower(int **A, int size, int n);
```

3. Reimplement the array methods in Java. A Java source file, `ArrayUtils.java` has been provided with the necessary method signatures.

4. Reimplement the matrix methods in Java. A Java source file, `MatrixUtils.java` has been provided with the necessary method signatures.