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.

Naming Instructions Place your code in source files with the file names `TODO.`, respectively. Do some rudimentary input validation and exit the program with an error message on any erroneous input.

Programs

1. Pressure is a measure of force applied to the surface of an object per unit area. There are several units that can be used to measure pressure:

   - Pascal (Pa) which is one Newton per square meter
   - Pound-force Per Square Inch (psi)
   - Atmosphere (atm) or standard atmospheric pressure
   - The torr, an absolute scale for pressure

   To convert between these units, you can use the following formulas.

   - 1 psi is equal to 6,894.75729 Pascals, 1 psi is equal to 0.068045964 atmospheres
   - 1 atmosphere is equal to 101,325 Pascals
   - 1 torr is equal to $\frac{1}{760}$ atmosphere and $\frac{101{,}325}{760}$ Pascals

   You could write several functions to compute back and forth between these four scales. However, for this exercise, you will write a single function to convert a single value to all four at once. Specifically you will implement the following function:

   ```
   1 int convertPressure(double *pa,
   2                     double *psi,
   3                     double *atm,
   4                     double *torr,
   5                     Scale scale);
   ```

   Note:

   - You will need to define an enumerated type called `Scale` that has four values: `PASCAL, PSI, ATM, TORR`, the value passed to the function will serve as a flag to indicate which scale the conversion is *from*; the value stored in the corresponding variable should be used convert to the remaining three.

     For example, if `scale == PASCAL` the value stored in the variable `pa` should be used to convert to `psi`, `atm` and `torr`.

   - The return value will be a flag indicating an error-level: 0 for no error, otherwise a non-zero value for various types of errors (negative pressure values, null pointers, invalid scales, etc.). You are encouraged to define another `Error`

enumerated type for this purpose.

- Place your enum declaration and the prototype declaration in a file called `pressure.h`
- Place your actual function implementation in a file called `pressure.c`
- Create a main function in `pressureDemo.c` that prompts the user for one of the scales (an integer 1–4 in the order listed above) and a value to convert and outputs the result of all four scales (or indicates an appropriate error message).
- Hint: you may find it easier to convert to a universal scale first, then convert to the other scales.

2. **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.

   - Write a function that takes an integer array and returns the sum of its elements between two given indices, i, j (inclusive for both):
     `int subSum(const int *a, int size, int i, int j);`
   - Write a function that takes an integer array and returns the sums all of its elements:
     `int sum(const int *a, int size);`
   - 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);`
   - 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);`
   - 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, $[2, 3, 2]$ 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);
     ```

   - The k-th order statistic of an array is the k-th largest element. For our pur-

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

3. **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_utils.h` with their definitions in a source file named `matrix_utils.c`. You should test your functions by writing a driver program, but you need not hand it in.

   - Write a function that constructs a new $n \times n$ *identity matrix*. An identity matrix is a square matrix with ones on the diagonal and zeros everywhere else.
     `int **getIdentityMatrix(int n);`
   - 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);`
   - 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);`

- 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);`
- 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);`