CS 31 Worksheet 3

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.
Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

Concepts

Arrays (1D, 2D), Pass by reference/value, Arrays as parameters in functions

1) Write a function that takes in an array of integers foo, and a number n and return true if they are "equal".

```
int foo [5] = {7, 8, 2, 1, 6};
int n = 78216;
equality(foo, n) → returns True

int bar [3] = {3, 2, 1};
int n = 312;
equality(bar, n) → returns False
```

Function header: `bool equality(int foo[], int n);`

```
bool equality(int foo[], int size, int n) {
     for (int i = size-1; i >= 0; i--) {
          if (foo[i] != (n%10) || (n <=0 )) {
               return false;
          }
          n /= 10;
     }
     return true;
}
```

2) Create a function that accepts three parameters: (1) an integer array, (2) the size of the array, and (3) a target value, *k*. Then return true or false depending on whether there is a pair of numbers within the array whose sum is *k*.

Sample: `[1, 5, 6, 20, 10], size=5, target = 25`

Output: `True. Because 20 + 5 = 25`

Sample: `[1, 23, 3], size = 3, target = 22`
Output: `False. There are no pairs whose sum is 22.`

Function header: `bool isPair(int foo[], int size, int k);`

```
bool isPair(int foo[], int size, int k) {
      for (int i = 0; i < size; i++) {
            for (int j = i + 1; j < size; j++) {
                  if (foo[i] + foo[j] == k)
                        return true;
            }
      }
      return false;
}
```

3) After Halloween Frank has *N* candies, but he must share *N/2* with his sister, Mandy. He wants to maximize the number of unique candies, all designated by a different integer, he can have after dividing his stash in half. Implement the following function to find the maximum number of unique candies.

Sample: `[10, 10, 10, 10, 2, 5]`
Output: `3. Frank can keep [2, 5, 10] for himself and share [10, 10, 10] with his sister.`

Sample: `[2, 2, 10, 10, 10, 2]`
Output: `2. Frank can keep [2, 10, 10] for himself, a maximum of 2 unique candies, and share [2, 2, 10] with his sister.`

Function header: `int maxUnique(int candies[], int k)`
`// k is the size of the array candies[]`

```
int maxUnique(int candies[], int k) {
      int temp[k];
      int counter = 0;
      for (int i = 0; i < k; i++) {
            bool isNewCandy = true;
            for (int j = 0; j < counter; j++) {
                  if (candies[i] == temp[j]) {
                        isNewCandy = false;
                        break;
                  }
            }
            if (isNewCandy) {
                  temp[counter] = candies[i];
                  counter++;
            }
      }
      return counter;
}
```

4) Write the function zeroLeft which takes in an array of 1's and 0's and modifies the array so all of the 0's are at the left and all of the 1's are at the right.

```
int foo [7] = {1, 1, 0, 0, 0, 1, 0}
zeroLeft(foo, 7);
// foo = {0, 0, 0, 0, 1, 1, 1}
```

Function header: `void zeroLeft(int array[], int array_len);`

```
void zeroLeft(int array[], int array_len) {
      int zeroCount = 0;
      for (int i = 0; i < array_size; i++) {
             if (array[i] == 0) {
                  zeroCount++;
            }
      }
      for (int j = 0; j < array_size; j++) {
            if (zeroCount <= 0) {
                  array[j] = 1;
            } else {
                  array[j] = 0;
                  zeroCount--;
            }
      }
}
```

5) Write a function with the following header:

`void exclusiveProduct(int nums[], int len)`
where *nums* is an array of positive numbers and *len* is the length of *nums*.

This function should alter *nums* such that *nums*[i] is the product of all *nums*[j] where j != i (in which *nums*[j] is the value before changes).

Example:

```
int foo[3] = {3, 1, 2}
exclusiveProduct(foo, 3)
//foo now equals {2, 6, 3}. At index 1, the new value is 2,
because it is the product of 1 and 2, all the integers in the
array **not** at index 1.
```

```
// case of length 1 is unspecified, it can be handled in any
way
// in this implementation, the sole element is set to 1

void exclusiveProduct(int nums[], int len) {
  if (len < 0)
    return; // returning breaks you out of the function

  int total = 1;
  for (int x = 0; x < len; x++) {
    total *= nums[x];
  }

  for (int x = 0; x < len; x++) {
    nums[x] = total / nums[x];
  }
}
```

6) Write a function with the following header:

```
bool rangeSearch(const int sorted_nums[], int len, int target,
                  int& start, int& end)
```

sorted_nums is a sorted array of positive numbers
len is the length of *sorted_nums*
target is a number to search for within the array

This function should return true if *target* is contained within the array and
false
otherwise. If the function returns true, *start* should be set to the first index where *target* appears and *end* should be set to the last index where *target* appears. If the function returns false, *start* and *end* should not be altered.

```
bool rangeSearch(const int sorted_nums[], int len, int target,
int& start, int& end) {
  bool startFound = false; // start of the target found
  bool endFound = false;   // end of the target found
  for (int x = 0; x < len; x++) {
    if (!startFound && sorted_nums[x] == target) {
      start = x;
      startFound = true;
    }
    else if (startFound && sorted_nums[x] != target) {
      end = x - 1;
      endFound = true;
      break;
    }
  }
  if (startFound && !endFound)
    end = len - 1;

  return startFound;
}
```

7) Create a function that accepts three parameters: (1) a SORTED integer array, (2) the size of the array, and (3) a target value, *k*. Then return true or false depending on whether there is a triplet of numbers within the array whose sum is *k*. A triplet is any group of three numbers that come from the array.

Sample: `[1, 5, 6, 20, 40], size=5, target = 27`
Output: `True. Because 1 + 6 + 20 = 27`

Sample: `[1, 23, 3], size = 3, target = 22`
Output: `False. There are no triplets whose sum is 22.`

```
bool hasTriplet(int arr[], int size, int k) {
  for (int i = 0; i < size - 2; i++) {
    if (i == 0 || (i > 0 && arr[i] != arr[i-1])) {
      int lo = i+1, hi = size - 1, sum = k - arr[i];
      while (lo < hi) {
        if (arr[lo] + arr[hi] == sum) {
          return true;
        } else if (arr[lo] + arr[hi] < sum) lo++;
        else hi--;
      }
    }
  }
  return false;
}
```

8) Write a function that takes two parameters: (1) a 2-dimensional integer array of size n x n, and (2) n as an integer. This function should print out the integers on the diagonal of the array.

```
Sample: {{1, 2, 3},   n: 3   Output: 159
         {4, 5, 6},
         {7, 8, 9}}
```

```
// The second [] in a 2D array passed as a parameter requires a number for size, which restricts the possible matrix sizes. The following solution is for 2D matrices of size 3x3.

void printDiagonals(int matrix[][3], int n) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			if (i == j) {
				cout << matrix[i][j];
			}
		}
	}
	cout << endl;
}
```

9) You are given an array of integers A, which has size n and whose elements have values ranging from 1 to n-1. Assume n < 500. Write a function *findDuplicate* that takes in the array A, and returns any element of A that is not unique (if multiple elements satisfy that property, return any one of them). The function prototype is given to you below:

```
int findDuplicate(const int A[], int n);

int A[7] = {1, 3, 5, 3, 2, 6, 1}
findDuplicate(A, 7) → 3 or 1
```

```
int findDuplicate(const int A[], int n) {
  bool isFound[500];

  for (int i = 1; i < 500; i++) {
    isFound[i] = false;
  }

  for (int i = 0; i < n; i++) {
    if (isFound[A[i]]) {
      return A[i];
    }

    isFound[A[i]] = true;
  }
  return -1; // this should never be called
}
```

10) Write a function that takes two parameters: (1) a 2-dimensional integer array of size n x n, and (2) n as an integer. This function should invert the array, that is, the rows of the original array should become the columns of the array and vice versa.

```
Sample: {{1, 2, 3},  n: 3  returns: {{1, 4, 7},
         {4, 5, 6},                  {2, 5, 8},
         {7, 8, 9}}                  {3, 6, 9}}
```

```
// The second [] in a 2D array passed as a parameter requires a
number for size, which restricts the possible matrix sizes. The
following solution is for 2D matrices of size 3x3.
void invert(int matrix[][3], int n) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        int temp = matrix[i][j];
                        matrix[i][j] = matrix[j][i];
                        matrix[j][i] = temp;
                }
        }
}
```