CEN360
ALGORITHMS DESIGN AND ANALYSIS
Fall – 2018
Lab 6: Divide and Conquer Algorithms

1. Fake Coin Problem: [50pts]

- *Decrease-by-a-constant-factor strategy*
  - *Divide n coins into two piles of n/2 coins each*
  - *Leave one extra coin aside if n is odd*
  - *Put the two piles on the scale*
  - *If the piles weigh the same, the coin put aside must be fake*
  - *Otherwise, proceed in the same manner with the lighter pile*

To implement fake coin algorithm, you can use the sum function for weight. Create a 2D array of the size [M] [N], N has four different values; 1.000, 10.000, 100.000 and 1.000.000 (should be defined at the top using #define keyword) and M is always 1000, it should be defined as a constant at the top of your code. M stands for the number of tests for a given N value.

Fill arr[M][N] array with ones. Randomly select an index for each of N coins and set the value of this index as 2. You should do this for each 1…M indexes for once. Those will be your fake counterfeit coins for each of N coins, while the ones are representing genuine ones. Keep the track of fake coins for each test, and store the number of steps to find the fake coin. Fill in the blanks in the following table for each N value.

	N=1.000	N=10.000	N=100.000	N=1.000.000
Min steps found				
Max steps found				
Number of Average Steps in 1000 tests				
Total Program Execution Time in sec.				

2. The Sum of Two Integers Problem using Binary Search Algorithm: [50pts]

Problem: There is a sequence of **n** positive integers in strictly increasing order in an array at the cells numbered from **0** up to **n-1**. You read a positive integer value **v** from the user.

Goal: Determine whether if there exist two integers **x** and **y** (not necessarily distinct) in the sorted sequence such that **x + y = v**.

Example:

A "yes"-input with n = 12 and v=30

0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	11	13	17	19	23	29	31	37

A "no"-input with n = 12 and v=29

0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	11	13	17	19	23	29	31	37

A first Attempt:

A naïve algorithm (brute force algorithm) to solve this problem is to enumerate all possible pairs in the sorted sequence and check if they sum to **v**. Given a sequence of length **n**, there are:

$$1 + 2 + \cdots + n = \frac{n * (n + 1)}{2}$$

such pairs. This is on the order of $\mathbf{n^2}$, but can we do better than this?

Hint: *We can take advantage of the fact that the given sequence is sorted!*

Binary Search Solution:

If we rearranged the equation and put it in the form **y = v − x**, we can rephrase the problem in terms of whether if such a **y** exists in the sequence for every **x** in the sequence. The idea is then to let **x** run over the sequence, compute **y** as **v − x** and use binary search to see if **y** exists in the sequence.

The repeated binary search algorithm:

```
create random array arr[] with the size of n
array values should be in the range of 1 to n
sort this array in increasing order
read v from the user
i = 0
while i < n
      x = arr[i]
      y ← v − x
      if BinarySearch(y) = “yes”
            print values of x and y and their indexes
            return "yes"
      i++
return "no"
```

You can use the following quick sort algorithm:

```
// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
```

```
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

You can use the following binary search algorithm:

```
// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r-l)/2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```