# VE280 Lab2

**Out**: 00:00, May 26, 2020; **Due**: 23:59, June 02, 2020

## Ex1. Eratosthenes Sieve

**Problem**: Zed brags that he can immediately point out all prime numbers from a sequence of integers. You don't believe that so you are going to write a program to check. Given a sequence of integers within a certain range, you need to find out all the prime numbers without changing their order. Recall that a prime number is a natural number greater than 1 divisible only by itself and 1.

**Requirement**: To check if an integer is a prime, intuitively we can achieve that by dividing integers smaller than itself. However, there is a more efficient algorithm called `Eratosthenes Sieve`, and you are required to implement it for this exercise. The algorithm works as follows:

1. For all integers within given range, mark them as prime
2. For 0 and 1, mark them as not prime
3. Find the next smallest integer `i` that is marked as prime
4. For all integers that can be divided by `i`, mark them as not prime
5. Repeat Steps 3-4 until no more `i` can be found in Step 3

After running the above algorithm, you have generated a lookup table. To check if a given integer is a prime, simply check that lookup table and see if it is marked as prime or not.

**Input Format**: The first line: an integer n `(1 <= n <= 20)` that stands for the length of the sequence. The second line: $n$ integers within range `[0,100000]`.

```
# sample
5
3 4 5 6 7
```

**Output Format**: All the primes in original order.

```
# sample
3 5 7
```

**Takeaway**: In this exercise, we used more space to store the lookup table, but thanks to that we can solve the problem in less time. `Time-space Tradeoff` is very common when you try to solve problems and hope this exercise gives you a basic sense.

**Tags**: `#array` `#time-complexity`

## Ex2. Final Exam Grade Ranking

**Problem**: In a high school, all final exam grades have been released. The teachers want to rank the students according to their total score. If two students have the same total score, just keep the order the same as in the input. As a programming expert, can you help design a system that solves this problem elegantly?

**Requirement**: You are required to use a `struct` to store information for each student. Also, you are required to use `qsort()` in `<cstdlib>` to achieve sorting. Check http://www.cplusplus.com/reference/cstdlib/qsort/ on how to use that.

**Input Format**: The first line: an integer n `(1<= n <= 10)` that stands for the number of students. The following $n$ lines: Name(string) ChineseScore(int) MathScore(int) EnglishScore(int). You can assume that there is no space in name and the length is less than `16`.

```
# sample
3
sam 110 132 140
tom 105 140 140
david 90 150 120
```

**Output Format**: Students sorted in descending total score.

```
# sample
tom 105 140 140
sam 110 132 140
david 90 150 120
```

**Takeaway**: In this exercise, we sorted an array of structs. Whenever you want to `compact information`, struct could be a nice choice. Also, we learned how to write a `compare function` for `qsort()`. Referring to `documentation` is the most efficient way to learn how to use a new function.

**Tags**: `#function` `#struct` `#qsort` `#pointer`

## Ex3. Which Building Will Be Destroyed?

**Problem**: Scientists have predicted an upcoming earthquake. You live in an square area full of buildings. The scientists say that how the earthquake destroys buildings will follow the following algorithm:

1. Equally divide the square into 4 smaller squares, and the left-upper part will all be destroyed.
2. For each of the remaining 3 smaller squares, also divide them into 4 smaller squares, and the left-upper part will be destroyed
3. Repeat until the square can no longer be divided

To save your life, you need to write a program to calculate which buildings will be destroyed.

**Requirement**: You will use simple recursion to solve this exercise. Recursion is to call the function itself inside the function body, as if the function has already been implemented. See `Takeaway` for some hints.

**Input Format**: An integer $n$ `(1 <= n <= 10)` that stands for a `2^n by 2^n` square.

```
# sample
3
```

**Output Format**: A `2^n by 2^n` square , where 0 indicates destroyed, 1 indicates safe.

```
# sample
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 1
0 0 0 0 1 1 1 1
0 0 0 1 0 0 0 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
```

**Takeaway**: We used simple recursion in this exercise. Many people think recursion is the first difficult concept when learning programming. However if you are familiar with the concept of procedural abstraction and specification comment, you can understand recursion easily by imagining the function as a black box with properly defined behavior, and calling the function inside itself is like using that black box in advance.

**Tags**: `#abstraction` `#specification` `#recursion`

## Files

1. *lab2.html*: description in html, better readability
2. *lab2.pdf*: description in pdf
3. *lab2.zip*: starter files

## Submission

Compress each **.cpp** file into a **.tar** file and submit to JOJ.
Do not change the name of the **.cpp** file.

---

Created by: Yiqing Fan
Last update: May 25, 2020
@ UM-SJTU Joint Institute