Being ignorant is not so much a shame, as being unwilling to learn.
~ *Benjamin Franklin*

CS1010 Programming Methodology
Week 12: Recursion

To students:

- Recursion is an important topic which will be used very often in the next module CS2040. Hence it is very important that you learn the basics of recursion well.
- Please also be reminded that PE2 is on **4 Nov (Sat)**. Please check out the CS1010 website PE page for more information.

1. **Tracing recursive codes**

(a) [AY2010/2011 Semester 1 Exam, Q1.2]
Given the following function, what does **f(5)** compute?

```
// Precond: n >= 0
int f(int n) {
    if (n == 0)
        return 0;
    else
        return (2 * n + f(n-1));
}
```

(b) Trace the function below manually, and write out the return value of **q(12)**.

```
// Precond: n >= 0
int q(int n) {
   if (n < 3)
      return n+1;
   else
      return q(n-3) + q(n-1);
}
```

Exploration: Would you be able to write an iterative version? Run both versions on large input, such as 50. What do you observe?

(c) [AY2011/2012 Semester 1 Exam, Q1.5]
What does following function compute?

```
int mystery(int x, int y) {
   if (x == 0)
      return y;
   else if (x < 0)
      return mystery(++x, --y);
   else
      return mystery(--x, ++y);
}
```

A. It returns the value of y.
B. It returns the value of x – y.
C. It returns the value of x + y.
D. It returns the value of x * y.
E. It will give compile-time error.

2. **Summing digits in an integer.**
   Summing digits in a non-negative integer *n* can be easily written using a loop. Is writing a recursive code for it just as easy? Write a recursive function **int sum_digits(int n)** to sum up the digits in *n*. (This question is discussed in lecture so this is some kind of revision.)

   A sample run is shown below:

```
Enter a non-negative integer: 970517
Sum of its digits = 29
```

3. **Recursion on array**
   Study the program **q3.c** below and trace the recursive function **mystery(int [] , int)**.

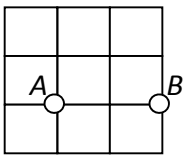
   What is the smaller version of the task on which the recursive call works? How does the original problem relate to this smaller problem? What does the function compute?

```
... // Omitted for brevity

int main(void) {
   int list[SIZE];
   scan_array(list, SIZE);
   printf("Answer = %d\n", mystery(list, SIZE));
   return 0;
}

// Read in values for array arr
void scan_array(int arr[], int size) {
   int i;

   printf("Enter %d values: ", size);
   for (i=0; i<size; i++)
       scanf("%d", &arr[i]);
}

// Precond: n > 0
int mystery(int arr[], int n) {
   int m;

   if (n == 1)
       return arr[0];
   else {
       m = mystery(arr, n-1);
       return (arr[n-1] > m) ? arr[n-1] : m;
   }
}
```

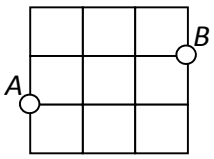
4. [AY2010/2011 Semester 1 Exam, Q4]
   Write a recursive function **int largest_digit_pair(int n)** to determine the largest pair of digits of a positive integer **n** starting from the right to the left.

   For example, if **n** is 5064321, then the pairs are 21, 43, 6 and 5, and hence the answer is 43.

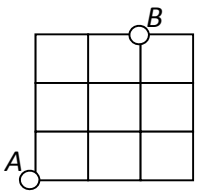
5. **North-East Paths**
   In a special town where pedestrians are only allowed to move northwards or eastwards, each of the following examples shows the total number of unique NE-paths, **ne(x, y)**, to get from point *A* to point *B*, where *B* is x rows north and y columns east of *A*. Assume that x and y are non-negative integers. By convention, ne(0, 0) = 1.



ne(0, 2) = 1



ne(1, 3) = 4



ne(3, 2) = 10

   Write a recursive function **int ne(int, int)** to compute the number of NE-paths.

   The following are some sample runs.

```
Enter rows and columns apart: 0 2
Number of NE-paths = 1
```

```
Enter rows and columns apart: 1 3
Number of NE-paths = 4
```

```
Enter rows and columns apart: 3 2
Number of NE-paths = 10
```

6. **Reversing an Array**
   Write a program **reverse_array.c** to randomly assign values into an integer array, and then reverse the array using recursion.

   For example, if the array contains { 6, 3, 0, 6, 8, 1, 5 }, then the reversed array is { 5, 1, 8, 6, 0, 3, 6 }. You should not use any additional array. You may assume that the array contains at most 15 elements.

7. [AY2012/2013 Semester 1 Exam, Q4]
   A positive integer can always be expressed as a product of prime numbers. For instance,

$$60 = 2 \times 2 \times 3 \times 5 = 2^2 \times 3^1 \times 5^1$$
$$78 = 2 \times 3 \times 13 = 2^1 \times 3^1 \times 13^1$$
$$13 = 13 = 13^1$$

   The following function **countPrimes()** takes a positive integer and counts the number of (possibly duplicate) prime numbers required to form the given integer.

```
countPrimes(60) returns 4 (two 2's, one 3 and one 5)
countPrimes(78) returns 3 (one 2, one 3 and one 13)
countPrimes(13) returns 1 (one 13)
```

```
int countPrimes(int number) {
   return countPrimesRec(number, 1, 0);
}
```

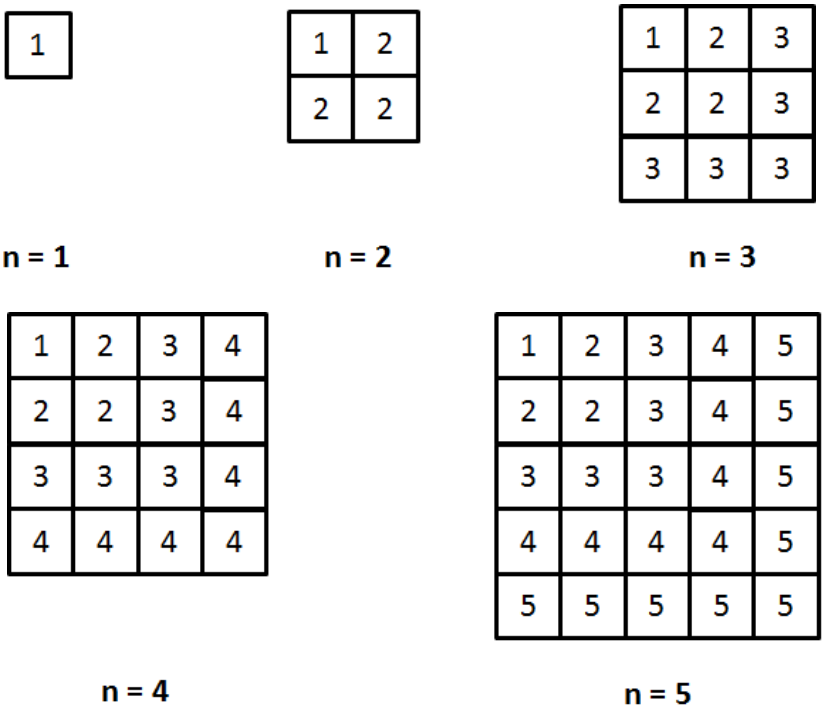
   This function in turn calls a recursive function **countPrimesRec**() that does the job of counting primes. The partial code for **countPrimesRec**() is given. You are to complete it by filling your code and the pre-conditions in the dashed boxes.

   Note that function **getPrime(*i*)** is considered given, which returns the *i*-th smallest prime number. For instance:

`getPrime(1)` returns 2; `getPrime(2)` returns 3; `getPrime(3)` returns 5

```
// Pre-conditions:
//
//
int countPrimesRec(int number, int index, int count) {
   int prime;

   if (number == 1)
      return count;
   prime = getPrime(index);
   if (number % prime) {
      
   }
   else {
      
   }
}
```

8. [AY2015/2016 Semester 1 Exam, Q13]
Observe the pattern in the following 2D arrays.



Write a recursive function **fill(int arr[][20], int n)** to fill a **n** x **n** 2D array **arr** with the pattern shown above.

You may assume that **n** is an integer in [1, 20].

9. Rewrite the binary search program using recursion. Use an auxiliary function as suggested in the lecture.