National University of Singapore
School of Continuing and Lifelong Education
TIC1002: Introduction to Computing and Programming II
Semester II, 2019/2020

Release date: 30 Jan 2020
Due: 16th Feb 2020, 23:59

Problem Set 2
Recursion and Sorting

Task 1: Memoisation…. Did you just misspelled memorization? [10 marks]

The **`Fibonacci()`** example from lecture illustrated that a) Recursion is indeed simpler / elegant to code, but also b) Precious time may be wasted due to the recalculation of known results. Let us learn a powerful technique to get the best of both worlds in this task.

Memoization (not a typo!) is a problem solving technique where **we keep previously calculated results** for future use. Combining memoization with recursion give you a very powerful problem solving tool!

Let us use **`factorial()`** to illustrate the technique:

```
int factorial_m( int N, int known_result[])
{
    if (known_result[N] == 0) {
         //Cannot rely on known calculation, have to work
         if (N == 0){
             //Fill in the result for future use
             known_result[0] = 1;
         } else {
             //Same, record result for future use
             known_result[N] = N * factorial_m( N-1, known_result);
         }
    }

    //At this point, known_result[N] should have N!
    // Either it is previously known (i.e. failed the if condition) OR
    // It has just been calculated in the recursive portion above.
    return known_result[N];
}
```

The factorial function now takes in an array **`known_result[]`** which keeps track of previously calculated result, **`known_result[K]`** stores the answer for **`K!`**.

At the beginning, the **`known_result[]`** should be initialized to all zeroes to indicate no known answers (zero is chosen because **factorial(N)** cannot be zero). You can see that as answers are calculated, the respective elements in the **`known_result`** array will be filled. If the same **`known_result[]`** array is used again, we can cut down some of the unnecessary calculations.

For example:

```
int known[20] = {0};   //support up to factorial(19).

factorial_m( 3, known );   // Call ONE
factorial_m( 3, known );   // Call TWO
factorial_m( 5, known );   // Call Three
```

For **Call ONE**, each of the recursive calls (**`f(3)`** ➔ **`f(2)`** ➔ **`f(1)`** ➔ **`f(0)`**) will find that the respective element in **`known[3]`**, **`known[2]`**, **`known[1]`** and **`known[0]`** are all "**`0`**", i.e. no known result. The normal recursion will kick in and fill up the answers in **`known[0], known[1], known[2]`** and **`known[3]`**. In this example, there is no saving at all.

However, at **Call TWO**, **`fac(3)`** will find that **`known[3]`** already has an answer and proceed to return it **without any recursive call!**

For **Call THREE**, **`fac(5)`** will only calculates two values at **`known[5]`** and **`known[4]`**. As soon as the recursion hits **`fac(3)`**, the known result will be used instead of recursion.

Observations:

1. **Memoisation** is not only applicable to recursive functions. You can use it for non-recursive processing too, especially if the calculation takes a long time.
2. **`Factorial()`** can only benefit from memorization if we call factorial() multiple times for different inputs. i.e. if we stop the example after call one, there is no benefit gained as **`fac(3)`**, **`fac(2)`**, **`fac(1)`** and **`fac(0)`** are calculated and used once only.

Point (2) above should give you a hint that memorization will help functions like **`Fibonacci()`** greatly as multiple redundant calculations are performed even in the same Fibonacci call.

Your task is to utilize memorization technique for the recursive **`Fibonacci()`** function. Note that in the template code, we gave the original Fibonacci function with a small addition. There is a global variable "**`call_count`**" which will be incremented every time the Fibonacci function is called. This is just a simple way for us to check whether your memoization is implemented properly.

For actual usage outside of this task, you can just remove this global variable safely. In the original recursive `Fibonacci()` function:

Fibonacci	20	30	40
Result	6765	832040	102334155
Call Count	**13529**	**1664079**	**204668309**

With memoization, the call count reduces drastically:

Fibonacci	20	30	40
Result	6765	832040	102334155
Call Count	**20**	**30**	**40**

Note that the results reported are from independent single `Fibonacci()` function call, not consecutive ones. Once the memorization is properly implemented, you should find that the recursive function's execution speed is comparable to the iterative version!

Task 2. No dinner for Polly? [10 marks]

Polly's teacher asked her to calculate the final value of S-polynomials (Simplified polynomial). Can you help her?

A S-polynomial takes the form of:

$$C_1 * X^1 + C_2 * X^2 + \cdots + C_{n-1} * X^{n-1} + C_n * X^n$$

where C_i are **integer coefficients**. Given a set of coefficients $\mathbf{C_i}$ and the value of **X**, we can calculate the final value (i.e. total) easily. For example, the S-polynomial below:

$3X + 2X^2 + 1X^3$ can be evaluated to **22** if X is 2. [$22 = 3*2 + 2*2^2 + 1*2^3$].

$3X + 2X^2 + 1X^3$ can be evaluated to **54** if X is 3. [$54 = 3*3 + 2*3^2 + 1*3^3$].

Implement a function where

`int polynomial(int xValue, int termArray[], int nTerm)`
`xValue` is the value of the X variable in the S-polynomial. `termArray[]` is an array with `nTerm` number of integer coefficients. Note that termArray[0] is C_1, termArray[1] is C_2 ..... termArray[n-1] is C_n.
This function returns the **value of the S-polynomial**.

Additional Requirements

Restrictions (must be respected, otherwise 0 mark)
No predefined math library functions allowed. Your code must make all calculation directly in the function, i.e. no helper function allowed (or needed).

Implementation	Maximum Mark
Iterative (Nested loop of any kind)	**5 marks**
Iterative (Only a single loop)	**7 marks**
Recursive: Without ANY form of loop	**10 marks**

Sample results

`xValue`	`termArray[]`	`nTerm`	Meaning	Return result
2	{3, 2, 1}	3	$3X + 2X^2 + 1X^3$	22
3	{3, 2, 1}	3	$3X + 2X^2 + 1X^3$	54
11	{7, 5, 3, 2}	4	$7X + 5X^2 + 3X^3 + 2X^4$	33957
13	{7, 5, 3, 2}	4	$7X + 5X^2 + 3X^3 + 2X^4$	64649

Notes

The given `printPolynomial()` function in the template print out the S-polynomial in a more human friendly format on screen.

<u>Task 3: There is no point in sorting! [10 marks]</u>

Sorting algorithm can be easily expanded to any other **comparable** items. As long as there is a way to say "item A is **before** item B", then a collection of such items can be sorted.

In this task, we will try to sort an array of **2D points**, i.e. reusing the **Point** structure covered in lecture 2:

```
struct Point {
    int X, Y;
};
```

The ordering we want to achieve is "***sort by X- coordinate, tie breaks by Y-coordinate***". Below is an example of **sorted** array of points:

Point Index	**0**	**1**	**2**	**3**	**4**
X	5	11	11	11	13
Y	73	19	34	68	5

Note the interesting example of Point 1, 2 and 3: Since they have the same X- coordinates, these points are ordered by the Y coordinate ("Tie breaks by Y-coordinate").

You can implement any of the 3 sorting algorithms taught in this course: Insertion, Selection or Bubble Sort. However, **the restriction is that you can only call sort() ONCE** to achieve the final result.

Note: The auto-grader can only confirm the sorting order (i.e. is it sorted properly), but not the further requirements (sorting algorithm, cannot use sort more than once etc). Hence, the further requirements will be manually verified after submission deadline.