TUTORIAL 5

BME 121 2016

Jeff Luo

Topics

- Recursion
- Software Testing
- Practice Problems for Midterm

Recursion

- In math, we can sometimes have functions defined this way:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

- This is the Fibonacci sequence defined as a recurrence relation
  - Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

- In computer code, we can also write methods using the principle of recursion
- **Recursion**: process of repeating in a self-similar way.
- **Recursive Methods**: a kind of method where it calls itself to solve a sub-problem.

```
int Fib(int n)
{
    if(n == 0 || n == 1) // base case
    {
        return n;
    }
    else   // recursive case
    {
        return Fib(n – 1) + Fib(n – 2);
    }
}
```

Recursive Method Calls

- Note: we tend to write the code that will stop repeated calls (base cases) first, then the code that will cause repeated calls (recursive cases)

Practice 1 – Recursion

- We have a type of triangle made of blocks, as follows:
  - The topmost row has 1 block.
  - The next row (row 2) has 2 blocks.
  - Row 3 has 3 blocks.
  - …
- Write method `int Triangle(int r)` which recursively calculates the total number of blocks for a triangle that has `r` rows (assume `r` >= 1)

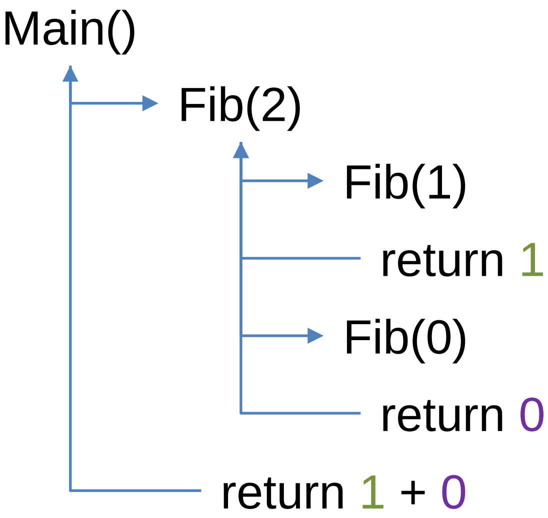
- Hint:
  - When solving a problem using recursion, think about the base case(s) and the recurrence relationship.

- Some problems are easier to solve by thinking of them as recursion problems, while others aren't so.

Recursion – How the computer executes the methods

- Remember: Only 1 method is active at a time in a program.
- With Recursion, only 1 copy of a recursive method is active at a time. Each copy gets it's own independent memory space for local variables.

```
int Fib(int n)
{
  if(n == 0 || n == 1) {
    return n;
  } else {
    return Fib(n – 1) + Fib(n – 2);
  }
}
```



Recursion – How the computer executes the methods

- Fib(4)
  - Fib(3)
    - Fib(2)
      - Fib(1)
        - return 1
      - Fib(0)
        - return 0
      - return 1 + 0
    - Fib(1)
      - return 1
    - return 1 + 1
  - Fib(2)
    - Fib(1)
      - return 1
    - Fib(0)
      - return 0
    - return 1 + 0
  - return 2 + 1

- Recursive Fib is easy to code, but actually inefficient…
  - Fib(2) is calculated twice here
- For Fib(n), generally half the calculations are duplicates
- How can we make it more efficient?

Recursion and Loops

Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

- All recursion methods can be rewritten as loops, and vice versa

```
int Fib(int n)
{
   if(n == 0 || n == 1) {
      return n;
   }  else {
      return Fib(n – 1) + Fib(n – 2);
   }
}
```

- The loop version is always more efficient, but not always straightforward to code

```
int Fib(int n)
{
   if(n == 0 || n == 1) {
      return n;
   } else {
      // store the last 2 values
      int nMinus1 = 1, nMinus2 = 0, result = 0;
      while(n >= 2) {
         // calculate the next value
         result = nMinus1 + nMinus2;
         // update the last 2 values
         nMinus2 = nMinus1;
         nMinus1 = result;
         n--;
      }
      return result;
   }
}
```

Recursion and Loops

- All recursion methods can be rewritten as loops

```
int Factorial(int n)
{
   // Base Case
   if(n == 0)
      return 1;
   // Recursive Case
   else
      return n * Factorial(n - 1);
}
```

- `Factorial(0) = 1`
- `Factorial(1) = 1 * 1`
- `Factorial(2) = 2 * 1 * 1`
- `Factorial(3) = 3 * 2 * 1 * 1`
- …

```
int Factorial(int n)
{
   int result = 1;
   // multiply every number from 1 to n
   for(int i = 1; i <= n; i++)
   {
      result *= i;
   }
   return result;
}
```

Practice 2 – Recursion

- Here's the mathematical definition of a function that calculates the greatest common divisor (gcd) of two integers m and n, as a recurrence relation:

$$Precondition: m \geq n > 0$$

$$\gcd(m, n) = \begin{cases} n & if\ n\ divides\ m\ with\ no\ remainder \\ \gcd(n, remainder\ of\ m \div n) & otherwise \end{cases}$$

- Write the method GCD: `int GCD(int m, int n)`
- Some expected answers:
  - GCD(5, 3) == 1
  - GCD(6, 4) == GCD(4, 6) == 2

Recursive Max for a 1D Array

- Previously, we wrote Max() by comparing every number in the array sequentially, updating a reference point whenever we find a bigger value, then return the final value of the reference point.

- Intuition: computer always compares 2 numbers at a time, so why not…
  - Split the whole array into 2 halves
    - Split each sub-array into halves, forming 4 partitions
      - Split those 4 partitions into halves, forming 8…
      - Until there's only 1 number in each partition
        - Then
      - Compare 2 (single number) partitions to determine the max of those 2
    - Compare and determine the max of 2 sub-maxes…
  - Until there's no more partitions to compare

Recursive Max for a 1D Array

```
int MyMax(int[] values)
{
    if(values.Length == 0) {
        return 0;
    }
    return MaxRecursive(values, 0, values.Length - 1);
}
int MaxRecursive(int[] values, int start, int end)
{
    if(start == end) {
        return values[start];
    }
    else {
        int middle = (end - start) / 2 + start;      // compute middle index
        return Math.Max(MaxRecursive(values, start, middle), MaxRecursive(values, middle + 1, end));
    }
}
```

Main() calls MyMax()

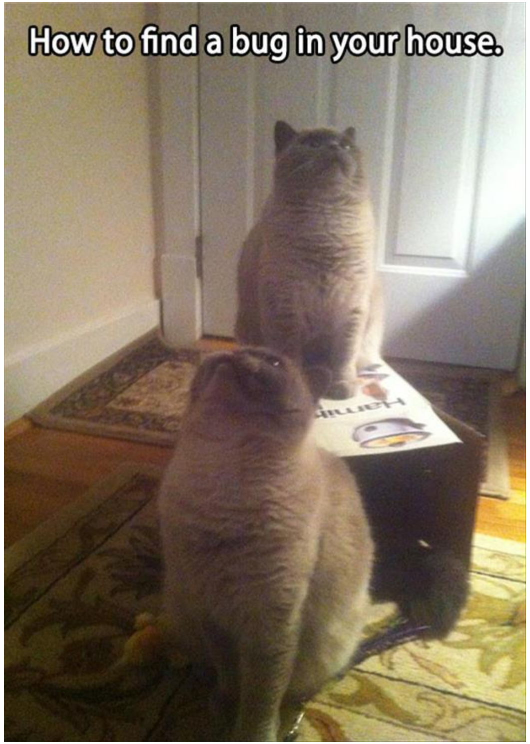
MyMax() starts the recursion with the indices set to 0 and Length – 1

If the indices match, then we're at the smallest partition, so just return the number.

Return the max of the left half and right half

Software Testing

- How do we know if our program is working as expected?
  - Compile it
  - Run it
  - **Test it**
- Software Testing: process of executing a program or application with the intent of finding the software bugs.
- We do this by designing tests for every method of our program.
- We test the program one method at a time.



Terminology

- Software **Fault**: a (unexecuted) defect in the software
- Software **Error**: incorrect internal software state (snapshot of the values in all variables) that is the manifestation of some fault (an executed fault)
- Software **Failure**: external, observable incorrect behavior (an error that's visible by user).
- **Testing**: evaluating software by observing its execution.
- **Debugging**: finding (and fixing) a fault, given a failure.
- **Validation**: evaluating software prior to release to ensure compliance with intended usage. (Are we building the right system?)
- **Verification**: determining whether products of a given phase of the development process fulfill requirements established in a previous phase. (Are we building the system right?)

Software Testing

- **Test Case**: a chosen pair of input and expected output used to examine program behavior; a single test.
  - Eg: Method: `int Fib(int n)`
    Input: 5 Expected Output: 5
  - If we run the method with the input and we don't get the expected output, then there must be some bug in the code
- **Test Set**: a set of test cases which together examines and ensures that a method is correct

Test Set for `int Fib(int n)`

Test Case	Input	Expected Output
1	0	0
2	1	1
3	2	1
4	5	5
5	8	21
6	12	144
7	20	6765
8	31	1346269

Exhaustive Testing

- Can we test every possible input value?
  - Technically yes
- How long would it take to test this method?

```
int M(int x) {
    return x;
}
```

- Assuming 3GHz CPU, performing 3 billion tests per second:
- **int**: 4,294,967,295 possible input values (32-bit int)
  - 1.43 seconds
- **long**: 18,446,744,073,709,551,615 possible input values (64-bit int).
  - 6148914691.24 seconds
  - ~= 71168 days
  - ~= 195 years

Exhaustive Testing

- How long would it take to test this method?

```
int N(int x, int y) {
    return x + y;
}
```

- Assuming 3GHz CPU, performing 3 billion tests per second:
- **int**: $4{,}294{,}967{,}295^2$ possible input values (32-bit int)
  - 6148914688.37 seconds
  - ~= 195 years
- **long**: $18{,}446{,}744{,}073{,}709{,}551{,}615^2$ possible input values (64-bit int).
  - ~= 1.13×10^{29} seconds

Age of the universe: 13.82 billion years (ESA Planck project), or about 4.36117×10^{17} seconds.

Exhaustive Testing

- It's not possible to exhaustively test every method, considering combinatorics effects…

- Pick good representative test inputs:
  - According to what the method should do
  - Common expected inputs from users, including spelling mistakes
  - Boundary values: size 0 array, empty string "", Int32.MaxValue, Int32.MinValue, …

Practice 3 – Software Testing

- Design a test set for each of these methods:
    - int Max(int[] values)
    - string UniqueLetters(string input)
        - Returns the unique letters in the input string in the order of their appearance from left to right

Software Engineering Life Cycle

- Waterfall Model/Process:

Requirements → Spec → Design → Code → Test/Verification → Ship

- Test Driven Development:

Requirements → Spec → Design → Write Test Sets → Code → Run Tests → Ship

Practice 4 – Test Driven Development

- Here's a test set for `bool Magic(int x)`
- A) What's does the method do?
- B) Implement it (on your own time), and rename the method appropriately

Test Case	Input	Expected Output
1	0	false
2	1	false
3	2	true
4	3	false
5	4	false
6	5	true
7	6	false
8	7	true
9	8	false
10	13	true
11	17	true

Practice 5 – Test Driven Development

- Here's a test set for `int[] Magic(int x)`

- A) What's does the method do?

- B) Implement it (on your own time), and rename the method appropriately

Test Case	Input	Expected Output
1	1	[0]
2	3	[0,3,6]
3	2	[0,2]
4	4	[0,4,8,12]

Software Testing

- How many different solutions can solve this test set?

- Is this test set specific enough?

Test Case	Input	Expected Output
1	[0]	0
2	[1]	1
3	[0, 0]	0

Test Driven Development

- Often times, in addition to a set of test cases, a description of the expected behavior of a method is given for TDD.

- Eg: `int[] Sort(int[] values)` should return a copy of the input array sorted from smallest to largest

Test Case	Input	Expected Output
1	[0]	[0]
2	[2,1]	[1,2]
3	[3,1,2]	[1,2,3]

Future Testing Topics

- Designing Tests for Classes and Objects
  - (so far the ones shown are for methods that definitely return something)
- Designing Tests for methods that use Random
- Designing Tests for multi-component software systems
- Writing Test Code
  - (almost the same as any code we've written)
- Running Test Code
  - (different dotnet command)

Review Problems for Midterm From 2015 Textbook

- Solve at least one problem from each chapter. The book arranges problems from easiest to hardest. Coloured problems are highly recommended.
  - Those with the 2012 edition, please partner up!

- Chapter 2: int, double, bool, string, arithmetic (+ - * / %) and comparison (== != >= <= > <) operators, type.Parse(), Write/WriteLine, ReadLine, placeholders & formatting
  - Page 96: #1, 2, 3, 5, 6, 9, 11, and 17
- Chapter 4: boolean (&& || ! ^) operators, if else, switch case, inline if statement (textbook calls it conditional operator), Random
  - Page 185: #1, 3, 6, 7, and 8 using switch case
- Chapter 5: for and while loops, nested loops
  - Page 224: #1, 2, 3, 7, 8, 9, 10

Review Problems for Midterm From 2015 Textbook

- Chapter 6: 1D and 2D Array
  - 1D array practices: Page 265: #2, 3, 6
    - Note: Two parallel arrays are 2 separate 1D arrays that have the same length, and where the pairs of values (one from each array) at the same index together store some interesting information, eg string[] name, int[] age where name[0] and age[0] stores the name and age for one specific person. Arrays (1D or manyD) each store only 1 type of information, so if we need to store different types, we use parallel arrays.
  - 2D array practices:
    - Hard code a 2D array of integers using the initializer syntax, then write the following methods:
    - int Max(int[,] values) which returns the max value in the 2D array
    - double Average(int[,] values) which calculates and returns the average of all the values in the 2D array

- Chapter 7: Designing and Writing Methods
  - Page 307: #3, 4, 5, 6, 9, 10

Review Problems for Midterm From 2015 Textbook

- Chapter 9: Defining Classes and Using Objects
  - Page 417: #1
  - Also, separately, create a class called Pizza with the following:
    - Fields:
      - Size, which stores the letter S, M, or L
      - Kind, which stores the string Vegetarian, Mediterranean, Canadian, or Supreme
    - Methods:
      - Constructor, which sets the size and kind of the pizza
      - GetPrice, which calculates and returns the price of the pizza according to the following:
        - S $10, M $15, L $20, with additional cost of Vegetarian $1, Mediterranean $2, Canadian $0, or Supreme $3
      - ToString, which returns a string that represents the pizza, showing its size, kind, and price:
        - “Small Canadian $10”
  - In Main, create 3 Pizza objects and fill in the values with inputs from the user, then display each pizza by calling ToString and also calculate and display the total order price with 13% tax added.