Bunnies

Recursion

Objectives

- Identify recursive algorithms
- Write simple recursive algorithms
- Understand recursive function calling
  - With reference to the call stack
- Understand recursive linear and binary search

Rabbits

A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?

Liber Abaci, Leonardo Pisano Bigollo (aka Fibonacci), 1202

Bunnies

- What happens if you put a pair of rabbits in a field?
  - More rabbits!
- Assumptions
  - Rabbits take one month to reach maturity and
  - Each pair of rabbits produces another pair of rabbits one month after mating
  - Rabbits never die
  - Bunny heaven!

… and more Bunnies

- How many pairs of rabbits are there after 5 months?
  - Month 1: start – **1 pair**
  - Month 2: the rabbits are now mature and mate – **1 pair**
  - Month 3: – the first pair give birth to two babies – **2 pairs**
  - Month 4: the first pair give birth to 2 babies, the pair born in month 3 are now mature – **3**
  - Month 5: the 3 pairs from month 4, and 2 new pairs – **5**

… and even more Bunnies

- After 5 months there are 5 pairs of rabbits
  - i.e. the number of pairs at 4 months (3) plus the number of pairs at 3 months (2)
  - Why?
- While there are 3 pairs of bunnies in month 4 only 2 of them are able to mate
  - the ones alive in the previous month
- This series of numbers is called the Fibonacci series

month:	1	2	3	4	5	6
pairs:	1	1	2	3	5	8

Fibonacci Series

- The n^{th} number in the Fibonacci series, *fib*(n), is:
  - 0 if $n = 0$, and 1 if $n = 1$
  - *fib*($n - 1$) + *fib*($n - 2$) for any $n > 1$
- e.g. what is *fib*(23)
  - Easy if we only knew *fib*(22) and *fib*(21)
    - Then the answer is *fib*(22) + *fib*(21)
  - What happens if we actually write a function to calculate Fibonacci numbers like this?

Recursive Functions

Calculating the Fibonacci Series

- Let's write a function just like the formula
  - $fib(n) = 0$ if $n = 0$, 1 if $n = 1$,
  - otherwise $fib(n) = fib(n - 1) + fib(n - 2)$

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

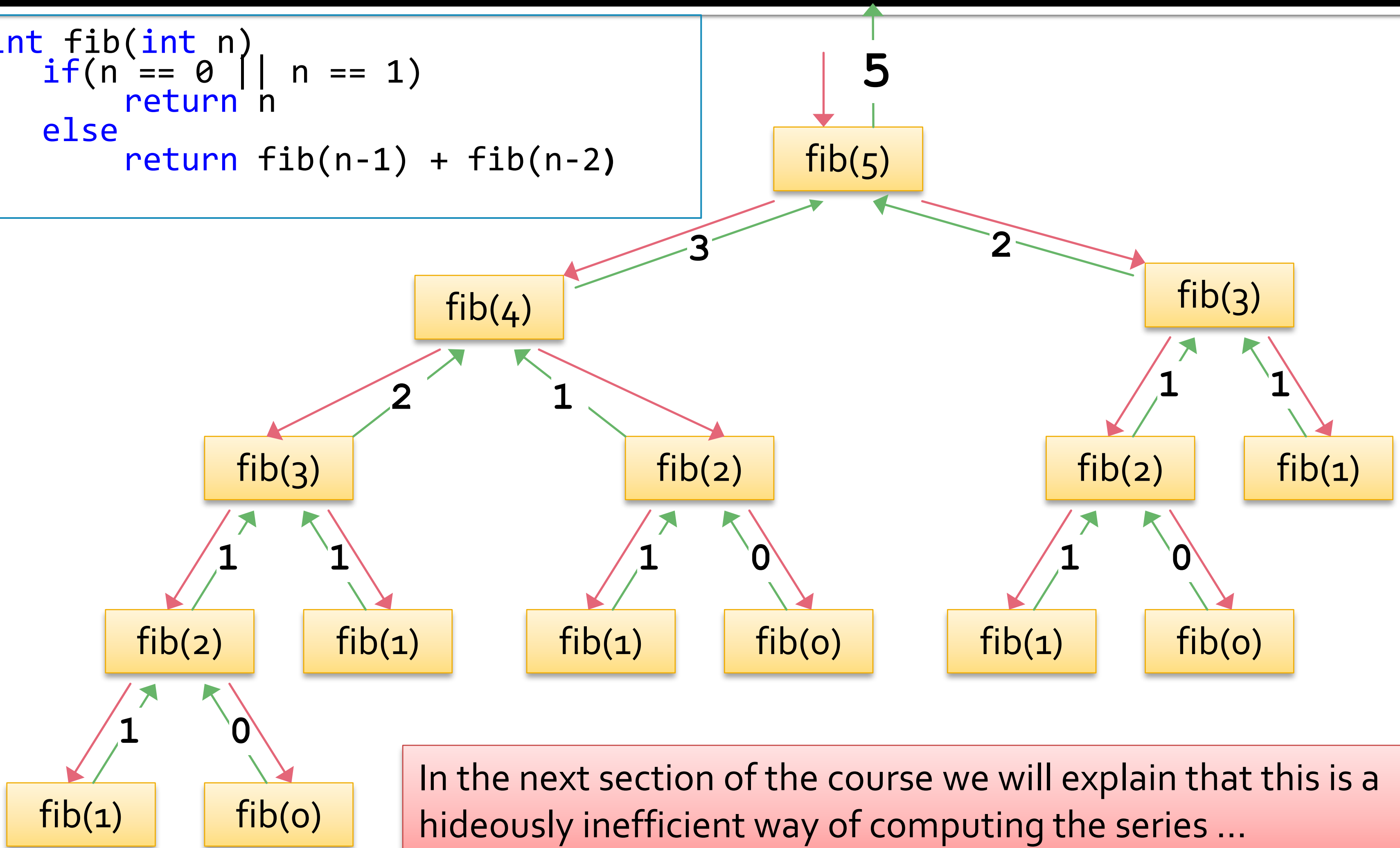
The function calls itself

Recursive Functions

- The Fibonacci function is *recursive*
  - A recursive function calls itself
  - Each call to a recursive method results in a *separate* call to the method, with its own input
- Recursive functions are just like other functions
  - The invocation is pushed onto the call stack
  - And removed from the call stack when the end of a method or a return statement is reached
  - Execution returns to the previous method call

Analysis of fib(5)

```
int fib(int n)
    if(n == 0 || n == 1)
        return n
    else
        return fib(n-1) + fib(n-2)
```



In the next section of the course we will explain that this is a hideously inefficient way of computing the series ...

Recursion and the Call Stack

- When a function is called its data is pushed onto the call stack, and execution switches to the function
- When a recursive function call is made, execution switches to the current invocation of that function
  - The call stack records the line number of the previous function where the call was made from
    - Which will be the previous invocation of the same function
  - Once the function call ends it also returns to the previous invocation

Recursive Function Anatomy

- Recursive functions do not use loops to repeat instructions
  - But use recursive calls, in if statements
- Recursive functions consist of two or more cases, there must be at least one
  - Base case, and one
  - Recursive case

Base Case

- The base case is a smaller problem with a simpler solution
  - This problem's solution must *not* be recursive
    - Otherwise the function may never terminate
- There can be more than one base case
  - And base cases may be implicit

Recursive Case

- The recursive case is the same problem with smaller input
  - The recursive case must include a recursive function call
  - There can be more than one recursive case

Finding Recursive Solutions

- Define the problem in terms of a smaller problem of the same type
  - The recursive part
  - e.g. `return fib(n-1) + fib(n-2);`
- And the base case where the solution can be easily calculated
  - This solution should not be recursive
  - e.g. `if (n == 0 || n == 1) return n;`

Steps Leading to Recursive Solutions

- How can the problem be defined in terms of smaller problems of the same type?
  - By how much does each recursive call reduce the problem size?
  - By 1, by half, ...?
- What is the base case that can be solved without recursion?
  - Will the base case be reached as the problem size is reduced?

Recursive Searching

Linear Search
Binary Search

Linear Search Algorithm

```
int linSearch(int arr[], int n, int x){
    for (int i=0; i < n; i++){
        if(x == arr[i]){
            return i;
        }
    } //for
    return -1; //target not found
}
```

The algorithm searches an array one element at a time using a for loop

Recursive Linear Search

- Base cases
  - Target is found, or the end of the array is reached
- Recursive case
  - Target not found

```
int recLinearSearch(int arr[], int next, int n, int x){
    if (next >= n){
        return -1;
    } else if (x == arr[next]){
        return next;
    } else {
        return recLinearSearch(arr, next+1, n, x);
    }
}
```

Binary Search

- Requires that the array is sorted
  - In either ascending or descending order
  - Make sure you know which!
- A *divide and conquer* algorithm
  - Each iteration divides the problem space in half
  - Ends when the target is found or the problem space consists of one element

Binary Search Sketch

Search for 32

Guess that the target item is in the middle, that is index = 15 / 2 = 7

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The array is sorted, and contains 16 items indexed from 0 to 15

Binary Search Sketch

Search for 32

45 is greater than 32 so the target must be in the lower half of the array

Repeat the search, guessing the mid point of the lower sub-array (6 / 2 = 3)

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Everything in the upper half of the array can be ignored, halving the search space

Binary Search Sketch

Search for 32

21 is less than 32 so the target must be in the upper half of the subarray

Repeat the search, guessing the mid point of the new search space, 5

The target is found so the search can terminate

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The mid point = (lower subarray index + upper index) / 2

Thinking About Binary Search

- Each sub-problem searches a sub-array
  - Differs only in the upper and lower array indices that define the sub-array
  - Each sub-problem is smaller than the last one
  - In the case of binary search, half the size
- There are two base cases
  - When the target item is found and
  - When the problem space consists of one item
    - Make sure that this last item is checked

Recursive Binary Search

```
int binarySearch(
    int arr[], int start, int end, int x){

    int mid = (start + end) / 2;
    if (start > end){
        return - 1; //base case
    } else if(arr[mid] == x){
        return mid; //second base case
    } else if(arr[mid] < x){
        return binarySearch(arr, mid + 1, end, x);
    } else { //arr[mid] > target
        return binarySearch(arr, start, mid - 1, x);
    }
}
```