Now You’re Thinking With Recursion!

CS 211

Adam Sweeney

December 5, 2017

Wichita State University, EECS

Introduction

- Even thought it's optional, being able to think recursively is still a very important skill

Agenda

- Chapter 14, Sections 14.3
- Requirements of a recursive algorithm
- Binary search

Requirements of a Recursive Algorithm

Requirements of a Recursive Algorithm

- There are three basic requirements
    - There is no infinite recursion
    - Each stopping case (base case) returns the correct value for that case
    - For recursive cases: If all recursive cases return the correct value, then the final value returned by the function is the correct value
- Each requirement will be discussed in more detail

There is No Infinite Recursion

- There are two aspects here that need to be considered
- The first is that we have a base case, or stopping case
    - This is a case with no recursion
    - In the example of printing numbers vertically, the base case was if our number was less than ten
    - In the example of the factorial, this was if our number less than or equal to one
- The second is that the recursive case must always reach a base case
    - Recursive calls may make more recursive calls and so on, but at some point we *must* reach a base case
    - In the example of printing numbers vertically, the recursive call always divided the number by ten
    - In the example of the factorial, the recursive call always subtracted one from the number

Each Stopping Case Returns the Correct Value for That Case

- If the recursive function is void, we can also say that it takes the correct action
- When it came to printing numbers vertically, the base case was if the number was less than ten
  - The action taken was to simply print the number
  - A single digit number is printed vertically simply by being printed, so this is correct
- When it came to the factorial, the base case was if the number was less than or equal to 1
  - The factorial of 1 and 0 is 1, so that is correct
  - We can't actually take the factorial of a negative number, so returning 1 as a default answer can also be considered to be correct

Recursive Cases Return the Correct Value

- The greater concept at play here is mathematical induction
- A very broad analogy follows:
  - If every piece of a puzzle is correctly placed, then the entire puzzle is solved
- If every recursive call returns the correct value (or performs the correct action), then the final result is correct
- This means that we have to validate our recursive call, but we don't have to validate every single recursive call
  - We would typically generalize the recursive call and show that to be correct
  - Thereby showing all recursive calls to be correct

Recursive Calls Do the Correct Action

- In the case of printing numbers vertically, the recursive case always made a recursive call using the number divided by ten
    - This eventually reduced the number to a single digit, which was the first digit of the number
    - It was also the first thing printed
    - When the recursive calls finished, the modulus 10 of the previous number was printed to the screen, which was always the last digit of that number
    - This resulted in the first, then second, then third, . . . , and so on being printed until the entire number was printed vertically

Recursive Calls Return the Correct Value

- In the case of the factorial, the recursive case was a return statement that was n * (n - 1)!
    - This is correct
    - For example, in the case of the number 5: $5 \cdot 4! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5!$
    - All recursive calls in after this manner are correct, by the nature of the factorial, so the final answer, the factorial of our original number, must also be correct

Binary Search

Binary Search

- Binary search is a fairly efficient method of searching through a **sorted** list
- The algorithm is quite simple

  Go to the middle element and compare against key (value being searched)

  If it matches, you're done

  If not, is key less than the value of the middle element?

  If yes, discard the right half of the list

  If not, discard the left half of the list

  Repeat until item is found or list is exhausted (not found)
- This much faster than an element-by-element check
- If you are checking elements individually, the average case is about half the size of your list
- But, for example, a binary search of a list with 100 elements can usually finish its search in 7 iterations

Implementations of Binary Search

- `https://repl.it/@sweenish/211-Ch-14-Thinking-Recursively`
- The main items to focus on are the functions `rbinarySearch()` and `ibinarySearch()`
    - These are the recursive and iterative versions, respectively
- What is interesting to note here is how similar the two versions look
- As well as the fact that the iterative version actually adheres to the principles that drive recursive algorithm design

The Moral of the Story

- Even though recursion is optional, being able to think recursively gives us another tool that we can use to solve problems
- Stocking our toolbox avoids the "When all you have is a hammer, everything looks like a nail" syndrome