CodeClub JU : Session 3

Recursion

Siddhanth Gupta
Rohan Paul
Angshuman Ghosh

BCSE-3

siddhanthgupta@gmail.com

Jadavpur University - Dept. of CSE

2014-2015 Session
1st April 2015

Outline

Introduction

Things we hope to cover today

- Recursion
- that's it.

Introduction

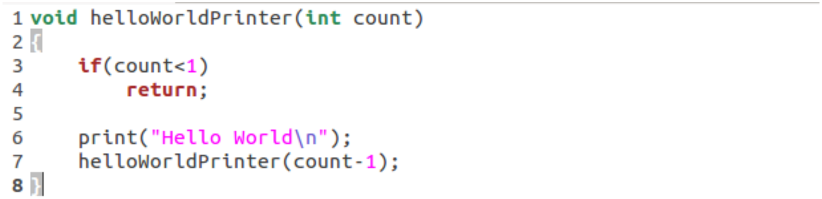
Recursion provides a simple, powerful way of approaching a variety of problems. It is often hard, however, to see how a problem can be approached recursively; it can be hard to "think" recursively.

It is also easy to write a recursive program that either takes too long to run or doesn't properly terminate at all.

In this session we'll go over the basics of recursion and hopefully help you develop, or refine, a very important programming skill.

Introduction

Basically, a function is said to be recursive if it calls itself.
Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.



```
void helloWorldPrinter(int count)
{
    if(count<1)
        return;

    print("Hello World\n");
    helloWorldPrinter(count-1);
}
```

Introduction

A recursive algorithm has 3 parts:

1. **Base Case** (i.e., when to stop) : The base case is the solution to the "simplest" possible problem
2. **Working towards the Base Case** : This is where we make the problem simpler
3. **Recursive Call**

Key Considerations

A recursive algorithm has 3 parts:

1. **It handles a simple "base case" without using recursion.** : The base case is the solution to the "simplest" possible problem
2. **It avoids cycles** : In many recursive programs, you can avoid cycles by having each function call be for a problem that is somehow smaller or simpler than the original problem. As the problem gets simpler and simpler (in this case, we'll consider it "simpler" to print something zero times rather than printing it 5 times) eventually it will arrive at the "base case" and stop recursing.

Key Considerations

A recursive algorithm has 3 parts:

1. **Each call of the function represents a complete handling of the given task.** Sometimes recursion can seem kind of magical in the way it breaks down big problems. But the recursive call must account for the complete solution of the problem.

Why use recursion

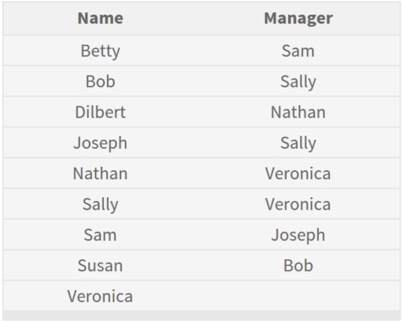
The problem we illustrated above is simple, and the solution we wrote works, but we probably would have been better off just using a loop instead of bothering with recursion.
Where recursion tends to shine is in situations where the problem is a little more complex. Recursion can be applied to pretty much any problem, but there are certain scenarios for which you'll find it's particularly helpful.

Hierarchies, Networks, or Graphs

In algorithm discussion, when we talk about a graphs, we're usually talking about a network of things, people, or concepts that are connected to each other in various ways.
For example, a road map could be thought of as a graph that shows cities and how they're connected by roads. Graphs can be large, complex, and awkward to deal with programatically. They're also very common in algorithm theory and algorithm competitions. Luckily, working with graphs can be made much simpler using recursion.

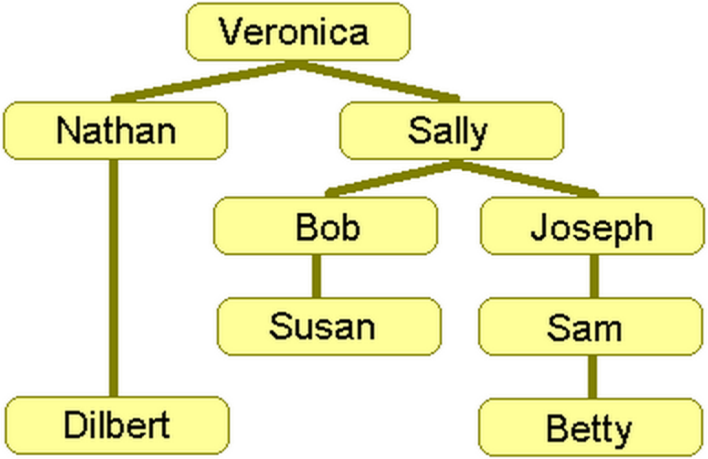
Hierarchies, Networks, or Graphs

Given the following relationship between employees and their respective managers, and given an manager's name, find the number of employees which report to the manager, **either DIRECTLY, or INDIRECTLY**

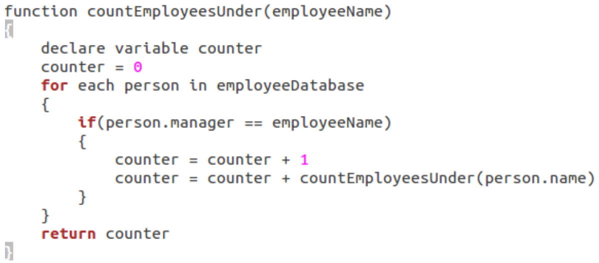


Name	Manager
Betty	Sam
Bob	Sally
Dilbert	Nathan
Joseph	Sally
Nathan	Veronica
Sally	Veronica
Sam	Joseph
Susan	Bob
Veronica	

Hierarchies, Networks, or Graphs



Hierarchies, Networks, or Graphs



```
function countEmployeesUnder(employeeName)
{
    declare variable counter
    counter = 0
    for each person in employeeDatabase
    {
        if(person.manager == employeeName)
        {
            counter = counter + 1
            counter = counter + countEmployeesUnder(person.name)
        }
    }
    return counter
}
```

Takeaway points before starting actual problems

Remember that each time you make a recursive call, you get a new copy of all your local variables. This means that there will be a separate copy of counter for each call. If that wasn't the case, we'd really mess things up when we set counter to zero at the beginning of the function.

Takeaway points before starting actual problems

A very important thing to consider when writing a recursive algorithm is to have a clear idea of our function's "mission statement."

For example, in this case I've assumed that a person shouldn't be counted as reporting to him or herself. This means "countEmployeesUnder('Betty')" will return zero. Our function's mission statment might thus be "Return the count of people who report, directly or indirectly, to the person named in employeeName - not including the person named employeeName."

Takeaway points before starting actual problems

what would have to change in order to make it so a person did count as reporting to him or herself.??

we just change the line "counter = 0" to "counter = 1" at the beginning of the function. This makes sense, as our function has to return a value 1 higher than it did before. A call to "countEmployeesUnder('Betty')" will now return 1.

BUT IS THIS CORRECT??

Takeaway points before starting actual problems

NO!!

For example, "countEmployeesUnder('Sam')" would now give an incorrect answer of 3. To see why, follow through the code: First, we'll count Sam as 1 by setting counter to 1. Then when we encounter Betty we'll count her as 1. Then we'll count the employees who report to Betty – and that will return 1 now as well.

We need to get rid of the line "counter = counter + 1", recognizing that the recursive call will now count Betty as "someone who reports to Betty"

Takeaway points before starting actual problems

We must have a very clear specification of what each function call is doing or else we can end up with some very difficult to debug errors.

Even if time is tight it's often worth starting out by writing a comment detailing exactly what the function is supposed to do. Having a clear "mission statement" means that we can be confident our recursive calls will behave as we expect and the whole picture will come together correctly.

Takeaway points before starting actual problems

We must have a very clear specification of what each function call is doing or else we can end up with some very difficult to debug errors.

Even if time is tight it's often worth starting out by writing a comment detailing exactly what the function is supposed to do. Having a clear "mission statement" means that we can be confident our recursive calls will behave as we expect and the whole picture will come together correctly.