CSC148H Lecture 3

Dan Zingaro
OISE/UT

September 22, 2008

Cannibals and Missionaries

The puzzle: Three missionaries and three cannibals are on the left side of a river. The goal is to get them all to the right side using a boat, but obeying these constraints:

- The boat can only travel with one or two people at a time (i.e. no empty crossings)
- Cannibals can never outnumber missionaries on a river bank (or cannibals do their thing ...)
- Note that this condition is not violated when the number of missionaries on a bank is zero, no matter how many cannibals are on that bank
- We must solve the puzzle with as few moves as possible

Representing the State Space

We can represent the current state of our solution as a triple (c, m, b), where:

- c represents the number of cannibals on the left side of the river
- m represents the number of missionaries on the left side of the river
- b is true exactly when the boat is on the left side of the river

TPS: Using this representation, how do we represent the start state? Final (winning) state? How about the state where exactly two cannibals and two missionaries are on the right side of the river?

Solution Plan

- We will write a procedure, **solve**, to solve the puzzle
- It will take four parameters: three representing the (c, m, b) triple, and a list of moves tracing our progression from the start state to the current triple

```
def solve (c, m, b, moves):
  ...

solve (3, 3, True, [(3, 3, True)])
```

Base Case

- So, given our state (c, m, b), we're apparently supposed to make a move that brings us closer to the solution
- But, there are so many possibilities. How do we know which to try?
- Easier question: what if we ask **solve** to solve the puzzle if we already start from the winning state?

```
solve (0, 0, False, [(0, 0, False)])
```

Base Case...

When we find a solution, we'll print it. The procedure below only works when we start at the winning state, so far:

```
def solve (c, m, b, moves):
  if c == 0 and m == 0 and b == False:
    print moves
    return

solve (0, 0, False, [(0, 0, False)])
```

One Step from the End

- Imagine we're at state $(1, 1, \text{true})$
- What move can we make to finish the puzzle?
- Imagine we're at state $(2, 0, \text{true})$
- What move can we make to finish the puzzle now?
- **solve** does not know how to deal with this situation yet
- But, if **solve** tries all possible moves from here, it will eventually hit the winning state
- . . . and **solve** knows what to do from there from before!

One Step from the End...

```
def solve (c, m, b, moves):
  if c == 0 and m == 0 and b == False:
    print moves
    return
  for all possible moves:
    let newC, newM, not b be the move
    if moves.count((newC, newM, not b)) == 0:
      newMoves = moves + [((newC, newM, not b))]
      solve (newC, newM, not b, newMoves)

solve (1, 1, True, [(1, 1, True)])
```

N Moves From the End

- At the top of **solve**, we also have to add checks to ensure we don't violate the constraints of the puzzle
- For example, if there's at least one missionary on a bank and more cannibals than missionaries there, we should immediately **return** (why?)
- Once we do that, **solve** works if we start one step from winning, since it tries all available moves
- More importantly, it works no matter how far from the end we start (i.e. we can call it with $(3, 3, \text{True})$)
- Why is this the case?

Recursion

- Our solution to the cannibals and missionaries puzzle is a realization of a more general technique called recursion
- We have recursion when a function calls itself
- Two important aspects:
  - The base case terminates the recursion
  - The recursive step gets us closer to the base case

Base Case

- The base case is the simplest case of a problem
- We can solve it directly, without recursing further
- In our puzzle, the base case is when we already have three cannibals, three missionaries and the boat on the right side of the river

Recursive Case

- When the problem is too tough to solve directly, we use recursion
- In our puzzle, this was the case whenever we hadn't reached the winning condition
- It's critical that recursion brings us closer to the base case, or we might recurse indefinitely
- Recall that we checked if a move already existed in **moves** before adding it to **moves** and recursing
- If we didn't do this, we could flip-flop between two moves over and over, never reach the base case, and never stop recursing

Implementation of Recursion

- When you make a recursive call, information about the previous function invocation is stored on an activation stack, and control is transferred to the new function
- Further recursion causes more copies of the function to be stored on the stack
- When a function invocation terminates, the stack is popped, returning control to the previously running function at the point just after the recursive call
- e.g. if we have two possible moves at a point in our puzzle, all paths including the first move are tried prior to those involving the second move

Mystery Code: What is Printed?

```
def sums (i):
  if i == 0:
    return 0
  else:
    return i + sums (i - 1)

print sums(3)
```

- Calling the function with 3, the return value is 3 plus whatever the function returns with parameter $3 - 1 = 2$
- Calling the function with 2, the return value is 2 plus whatever the function returns with parameter $2 - 1 = 1$
- TPS: finish this line of reasoning, please

Another Example: Binary Strings

- There are 2^r binary strings of length r
- For example, for $r = 3$, we have: 000, 001, 010, 011, 100, 101, 110, 111
- We can iteratively generate these with three nested loops

```
for i in range (2):
  for j in range (2):
    for k in range (2):
      print str(i) + str(j) + str(k)
```

Binary Strings with Recursion

- But what if we want to generate all binary strings for a user-specified length r?
- We can write a recursive procedure
- The parameters will be
  - The length of codes to generate
  - The length of the code we're currently constructing
  - The current code itself

Binary Strings: Base Case

- The base case occurs when we have already generated a string of the desired length
- In this case, we just print it, though you can imagine storing it in an array or processing it

```
def codes (binlen, depth, cur):
  if depth == binlen:
    print cur
    return
```

Recursive Structure of Binary Strings

- Binary strings of length 2: 00, 01, 10, 11
- Binary strings of length 3: 000, 001, 010, 011, 100, 101, 110, 111
- How can we use binary strings of length 2 to build the binary strings of length 3?
- Each string b of length 2 yields two strings of length 3: $b0$ and $b1$

Binary Strings: Recursive Case

- If our current string **cur** is not yet at length **binlen**, what do we do?
- 1. Add a 0, then append all binary strings of length **cur** $-$ 1, and
- 2. Add a 1, then append all binary strings of length **cur** $-$ 1
- These two steps use smaller instances of the problem (i.e. are closer to the base case), so we can invoke our recursive procedure to solve them

Generating Binary Strings

```
def codes (binlen, depth, cur):
  if depth == binlen:
    print cur
    return
  codes (binlen, depth + 1, cur + "0")
  codes (binlen, depth + 1, cur + "1")
```

Alternative Implementation: Generating Binary Strings

```
def codes (binlen, depth):
  if depth == binlen:
    return [""]
  l = codes (binlen, depth + 1)
  m = []
  for elt in l:
    m.append (elt + "0")
    m.append (elt + "1")
  return m
```

Types of Recursion

- Linear recursion: a function that calls itself once (c.f. mystery code)
- Binary recursion: a function that calls itself twice (c.f. generating binary codes)
- N-ary recursion: a function that calls itself arbitrarily often (c.f. cannibals and missionaries)
- Mutual recursion: function A is not directly recursive, but A starts a chain of function calls that results in A being called recursively

What do These Functions Do?

```
def f1 (i):
  if i == 1:
    return 1
  else:
    return i * f1 (i - 1)

def f2 (i, acc):
  if i == 1:
    return acc
  else:
    return f2 (i - 1, acc * i)
```

Calling the functions as `f1(1000)` and `f2(1000, 1)`, what happens?

Tail Recursion

- If the only recursion in a function occurs as the last thing the function does, it is tail recursive
- Question: Which of the two functions on the previous slide is tail recursive?
- Many languages (especially functional ones) implement tail recursion elimination
- With tail recursion, keeping a stack of activation records is unnecessary, since we never have to unwind the stack and go back to finish a previous function invocation
- To eliminate tail recursion: replace the tail call with assignment statements that set the variables for the next call of the function

Tail Recursion Elimination

```
def f2Iter (i, acc):
  while True:
    if i == 1:
      return acc
    else:
      i, acc = i - 1, acc * i
```