CSC148H Week 4

January 26, 2015

Announcements

- Assignment 1 - due next week, February 4 at 22:00
- Quiz 1 - this Friday during lecture! Please bring a pen.
- Office hours held in DH3097B

Recursion

- Provides an elegant and powerful alternative for performing repetitive tasks.
- Occurs when a function makes one or more calls to itself during execution.

Factorial Example

- Try solving 5! without recursion, this looks like: 5! = 5 x 4 x 3 x 2 x 1 = 120
- Let's try that again with recursion, this looks like: 5! = 5 x 4!
  - Notice how we use the factorial, !, again on a smaller case with 4!
  - 4! = 4 x 3!
  - 3! = 3 x 2!
  - 2! = 2 x 1!
  - 1! = 1 x 0!

Factorial Example

- When the problem gets smaller, we need to be able to solve it without recursion
- $5! = 5 \times 4!$
- $4! = 4 \times 3!$
- $3! = 3 \times 2!$
- $2! = 2 \times 1!$
- $1! = 1 \times 0!$
- $0! = 1$

Factorial Example

- Going to our original problem, we can now solve 5! using recursion
  - $\mathbf{0}! = 1$
  - $1! = 1 \times \mathbf{0}! = 1$
  - $2! = 2 \times \mathbf{1}! = 2$
  - $3! = 3 \times \mathbf{2}! = 6$
  - $4! = 4 \times \mathbf{3}! = 24$
  - $5! = 5 \times \mathbf{4}! = \mathbf{120}$

What is Recursion?

- Recursion: solving a problem by reducing it to subproblems, then combining the subproblem solutions to solve the original problem
- Subproblems must have the same structure as the original problem and be easier to solve
- Some subproblems are so simple that they can be solved directly (without reducing them further)
- Recursive functions: functions that call themselves as helper functions

Base Case

- The base case is the simplest case of a problem
- We can solve it directly, without subdividing further
- In our example, the base case is asking when n is 0

Factorial Example

- We can solve our factorial recursion problem in Python.
- What is our base case?
- What is the recursive structure?

```python
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

Tracing Our Factorial Example

```
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

Evaluate when n = 5
factorial(5)

5

```
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

5   4 = n - 1

Tracing Our Factorial Example

4

```
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

4   3 = n - 1

3

```
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

3   2 = n - 1

Tracing Our Factorial Example

2

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

2

1 = n - 1

1

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

1

0 = n - 1

Tracing Our Factorial Example

factorial(1) = 1 * factorial(0)
= 1 * 1
= 1

1

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

1

0 = n - 1
factorial(0) = 1

0

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

factorial(0) = 1

Tracing Our Factorial Example

factorial(2) = 2 * factorial(1)
= 2 * 1
= 2

2

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

2

1 = n - 1
factorial(1) = 1

factorial(3) = 3 * factorial(2)
= 3 * 2
= 6

3

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

3

2 = n - 1
factorial(2) = 2

Tracing Our Factorial Example

factorial(4) = 4 * factorial(3)
= 4 * 6
= 24

4

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

4

3 = n - 1
factorial(3) = 6

factorial(5) = 5 * factorial(4)
= 5 * 24
= 120

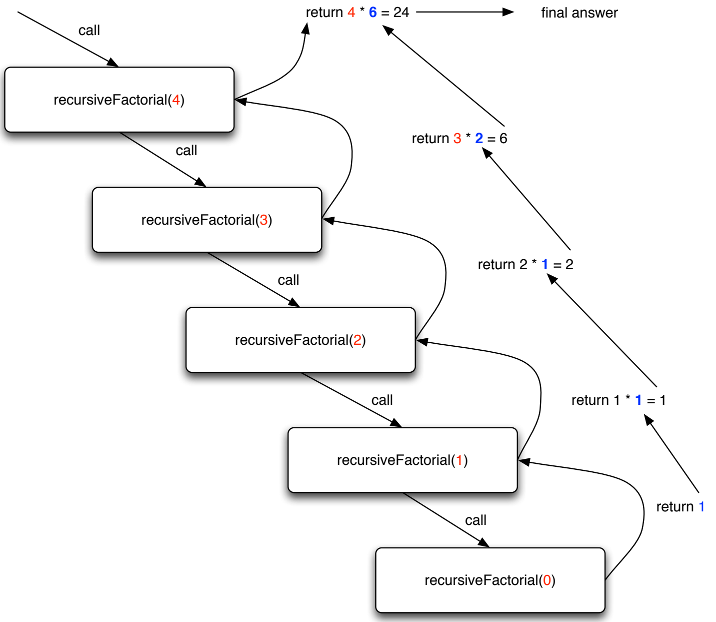
5

```
def factorial(n):

    # base case
    if n == 0:
        return 1

    # recursive case
    else:
        return n * factorial(n-1)
```

5

4 = n - 1
factorial(4) = 24

Visualizing Our Trace



Binary Codes

- A binary code of length r is a string of r bits (0 or 1)
- There are 2 binary codes of length 1, 4 binary codes of length 2, 8 binary codes of length 3 ...
  - Length 1: 0, 1
  - Length 2: 00, 11, 10, 01
  - Length 3: 000, 111, 001, 010, 100, 011, 110, 101
  - Function: 2**n
- Given integer r, our task is to generate a list of all binary codes of length r

Binary Codes, Base Case

- First, what if `r` were 0?
- Can we write a function that generates a list of all 0-length binary codes?
- The proper return value is `['']`, because the only binary code of length 0 is the empty string
- Recall: 2**n, 2**0 $= 0$

Binary Codes, Base Case

```
def rec(i):
  if i == 0:
    return ['']
  else:
    return
```

```
>>> rec(0)
''
>>> rec(1)
>>>
```

Binary Codes, Recursive Structure

- Given a list of all length $r - 1$ binary codes, how can you construct a list of all length r binary codes?
- Remember that when the length of the desired binary codes increases by 1, the number of binary codes doubles (function: 2**n)
- So each binary code of length $r - 1$ will yield **two** binary codes of length r
- Recall: 2**n, 2**1 $= 2$

Recursive Case

- When the problem is too tough to solve directly, we use recursion
- In our example, the recursive case was asking for all binary codes of length greater than 0
- It's critical that recursion brings us closer to the base case, or we might recurse indefinitely
- In our example, we recurse on problems whose length is decreased by 1

Binary Codes, Recursive Structure

- Strategy
  - Take each binary code of length $r - 1$ and append a 0 to it
  - Take each binary code of length $r - 1$ and append a 1 to it
  - Combine all of these into a new list and return it

Binary Codes, Recursive Structure

```
def codes(r):
  '''(int) -> list of str
  Return all binary codes of length r.
  '''

  if r == 0:     # base case
    return ['']

  small = codes(r-1)

  lst = []

  for item in small:
    lst.append(item + '0')
    lst.append(item + '1')
  return lst
```

Tracing the Binary Codes Function

- We already know what the function does with argument 0
- When tracing with argument 1, substitute `['']` when a call with argument 0 is made
- Then you know what the function does with argument 1, so you can trace it for argument 2 using a similar process

Tracing the Binary Codes Function

```
def codes(r):
    '''(int) -> list of str
    Return all binary codes of length r. '''

    if r == 0: # base case
        return ['']

    small = codes(r-1)
    lst = []

    for item in small:

        lst.append(item + '0')
        lst.append(item + '1')

    return lst
```

Tracing the Binary Codes Function

Evaluate when r = 0

codes(0)

0

```
def codes(r):
    '''(int) -> list of str
    Return all binary codes of length r. '''

    if r == 0: # base case
        return ['']            codes(0) = ['']

    small = codes(r-1)
    lst = []

    for item in small:

        lst.append(item + '0')
        lst.append(item + '1')

    return lst
```

Tracing the Binary Codes Function

Evaluate when r = 1

codes(1)

1

```
def codes(r):
    '''(int) -> list of str
    Return all binary codes of length r. '''

    if r == 0: # base case
        return ['']

    small = codes(r-1)
    lst = []
    for item in small:

        lst.append(item + '0')
        lst.append(item + '1')

    return lst
```

codes(r-1)

= codes(0)

= ['']

'' ['']

['0']

['0', '1']

['0', '1']

Tracing the Binary Codes Function

Evaluate when r = 2

codes(2)

2

```
def codes(r):
    '''(int) -> list of str
    Return all binary codes of length r. '''

    if r == 0: # base case
        return ['']

    small = codes(r-1)
    lst = []

    for item in small:

        lst.append(item + '0')
        lst.append(item + '1')

    return lst
```

codes(r-1)

= codes(1)

= ['0', '1']

'0'

['0', '1']

['00']

['00', '01']

Tracing the Binary Codes Function

Evaluate when r = 2

codes(2)

2

```
def codes(r):
    '''(int) -> list of str
    Return all binary codes of length r. '''

    if r == 0: # base case
        return ['']

    small = codes(r-1)
    lst = []
    for item in small:

        lst.append(item + '0')
        lst.append(item + '1')

    return lst
```

codes(r-1)

= codes(1)

= ['0', '1']

'1' ['0', '1']

['00', '01', '10']

['00', '01', '10', '11']

['00', '01', '10', '11']

Announcements

- Assignment 1 - due next week, February 4 at 22:00
- Quiz 1 - this Friday during lecture (15 minutes)! Please bring a pen.
- Office hours held in DH3097B

Permutations

- Based on a factorial function, $n!$
- For a string of n characters, there are $n!$ permutations
- A permutation is an ordering of the elements
- e.g. the permutations of abc are abc, acb, bac, cab, bca, cba

Permutations...

Let's write a recursive function to generate all permutations of a string.

What is the base case?

- Empty string

What is the recursive structure of permutations?

- How many permutations of a one-character string exist?
- How many permutations of a two-character string exist?

Permutations, Base Case

```
def permutations(s):
    '''(str) -> list of str
    Return all permutations of s.
    '''

    if s == '':
        return ['']
```

Permutations...

```
def permutations(s):
  '''(str) -> list of str
  Return all permutations of s.
  '''

  if s == '':
    return ['']

  # generate all smaller permutations
  smaller = permutations(s[1:])
  bigger = []

  # Now, for each of the smaller permutations,
  # put s[0] in every possible position
  for p in smaller:
    for i in range(len(p) + 1):
      new_perm = p[:i] + s[0] + p[i:]
      bigger.append(new_perm)
  return bigger
```

Tracing Our Permutations Example

```python
def permutations(s):
    '''(str) -> list of str
    Return all permutations of s. '''
    if s == '':
        return ['']
    # generate all smaller permutations
    smaller = permutations(s[1:])
    bigger = []

    # Now, for each of the smaller permutations,
    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]

            bigger.append(new_perm)
    return bigger
```

Evaluate when s = 'abc'
permutations('abc')

'abc'

```python
def permutations(s):
    '''(str) -> list of str
    Return all permutations of s. '''
    if s == '':
        return ['']
    # generate all smaller permutations
    smaller = permutations(s[1:])
    bigger = []

    # Now, for each of the smaller permutations,
    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]

            bigger.append(new_perm)
    return bigger
```

permutations('bc')

Tracing Our Permutations Example

'bc'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])

    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]

            bigger.append(new_perm)

    return bigger
```

permutations('c')

'c'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])

    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]

            bigger.append(new_perm)

    return bigger
```

permutations('')

Tracing Our Permutations Example

''

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']            permutations('') = ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])

    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position


    for p in smaller:


        for i in range(len(p) + 1):


            new_perm = p[:i] + s[0] + p[i:]


            bigger.append(new_perm)

    return bigger
```

'c'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])      smaller = permutations('')
                                                 = ['']
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
p = ''                     smaller = ['']

    for p in smaller:

    i = 0                        range(len(p) + 1) = 1

        for i in range(len(p) + 1):

                           new_perm = '' + 'c' + ''

            new_perm = p[:i] + s[0] + p[i:]


            bigger.append(new_perm)

    return bigger          bigger = ['c']
```

Tracing Our Permutations Example

'bc'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]


            bigger.append(new_perm)

    return bigger
```

smaller = permutations('c')
= ['c']

p = 'c'

smaller = ['c']

i = 0

range(len(p) + 1) = 2

new_perm = '' + 'b' + 'c'

'bc'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position

    for p in smaller:

        for i in range(len(p) + 1):

            new_perm = p[:i] + s[0] + p[i:]


            bigger.append(new_perm)

    return bigger
```

smaller = permutations('c')
= ['c']

p = 'c'

smaller = ['c']

i = 1

range(len(p) + 1) = 2

new_perm = 'c' + 'b' + ''

bigger = ['bc']

bigger = ['bc', 'cb']

Tracing Our Permutations Example

```
                 'abc'
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])        smaller = permutations('bc')
                                                 = ['bc', 'cb']
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
p = 'bc'                    smaller = ['bc', 'cb']
    for p in smaller:
  i = 0                             range(len(p) + 1) = 3
        for i in range(len(p) + 1):
                        new_perm = '' + 'a' + 'bc'
            new_perm = p[:i] + s[0] + p[:i]


            bigger.append(new_perm)

    return bigger
```

```
                 'abc'
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])        smaller = permutations('bc')
                                                 = ['bc', 'cb']
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
p = 'bc'                    smaller = ['bc', 'cb']
    for p in smaller:
  i = 1                             range(len(p) + 1) = 3
        for i in range(len(p) + 1):
                        new_perm = 'b' + 'a' + 'c'
            new_perm = p[:i] + s[0] + p[i:]

                       bigger = ['abc']
            bigger.append(new_perm)

    return bigger
```

Tracing Our Permutations Example

'abc'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])
```

smaller = permutations('bc')
= ['bc', 'cb']

```
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
```

p = 'bc'
smaller = ['bc', 'cb']

```
    for p in smaller:
```

i = 2
range(len(p) + 1) = 3

```
        for i in range(len(p) + 1):
```

new_perm = 'bc' + 'a' + ''

```
            new_perm = p[:i] + s[0] + p[i:]
```

bigger = ['abc', 'bac']

```
            bigger.append(new_perm)

    return bigger
```

'abc'

```
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])
```

smaller = permutations('bc')
= ['bc', 'cb']

```
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
```

p = 'cb'
smaller = ['bc', 'cb']

```
    for p in smaller:
```

i = 0
range(len(p) + 1) = 3

```
        for i in range(len(p) + 1):
```

new_perm = '' + 'a' + 'cb'

```
            new_perm = p[:i] + s[0] + p[i:]
```

bigger = ['abc', 'bac', 'bca']

```
            bigger.append(new_perm)

    return bigger
```

Tracing Our Permutations Example

```
              'abc'
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])    smaller = permutations('bc')
                                       = ['bc', 'cb']
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
p = 'cb'            smaller = ['bc', 'cb']
    for p in smaller:
i = 1                    range(len(p) + 1) = 3
        for i in range(len(p) + 1):
                    new_perm = 'c' + 'a' + 'b'
            new_perm = p[:i] + s[0] + p[i:]
                    bigger = ['abc', 'bac', 'bca', 'acb']
            bigger.append(new_perm)

    return bigger
```

```
              'abc'
def permutations(s):
    '''(str) -> list of str

    Return all permutations of s. '''

    if s == '':

        return ['']

    # generate all smaller permutations

    smaller = permutations(s[1:])    smaller = permutations('bc')
                                       = ['bc', 'cb']
    bigger = []


    # Now, for each of the smaller permutations,

    # put s[0] in every possible position
p = 'cb'            smaller = ['bc', 'cb']
    for p in smaller:
i = 2                    range(len(p) + 1) = 3
        for i in range(len(p) + 1):
                    new_perm = 'cb' + 'a' + ''
            new_perm = p[:i] + s[0] + p[i:]
                    bigger = ['abc', 'bac', 'bca', 'acb', 'cab']
            bigger.append(new_perm)

    return bigger    bigger = ['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```