Recursion

CMPT 145

Objectives

After this topic, students are expected to

1. recognize recursive definitions and recursive algorithms
2. describe the general form of a recursive definition, as well as recursive algorithm.
3. identify the base case and general case for recursive algorithms
4. design recursive algorithms use the template for recursive algorithms
5. define the terms of activation records and system stack.
6. understand how recursion works in terms of activation records and the system stack.
7. analyze the time complexity of simple recursive algorithms.

Dispelling concerns about recursion

- Recursion is a form of repetition based on function calls, instead of loops.
- You may like loops better because you practice those more.
- Recursion is easier than loops. Proof:
  - Recursion expresses a meaningful relationship.
  - Understanding relationships is what humans do best.
  - Therefore, recursion is what humans do best.

  Q.E.D.

Recursion

- Definitions that refer to themselves are said to be "recursive".
- Example:

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \textit{(base case)} \\ n \cdot (n-1)! & \text{if } n > 0 \quad \textit{(inductive step)} \end{cases}$$

- Suppose we re-formulate the ! operation as a function:

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & \text{if } n > 0 \end{cases}$$

- It's easy to re-formulate this as a recursive function.

Recursive Algorithms

- Recursive functions are those that call themselves.
- Recall that each time a recursive call is made, that call gets its own copy of local variables.

Recursive Factorial Algorithm

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Thinking recursively: The delegation metaphor

- A function call is like delegation. You "hire" a "delegate" to do some work for you.
- You give the delegate 3 things:
  1. Some data.
  2. Some instructions.
  3. A place to work.
- You wait until your delegate returns to you with the answer you asked for.
- If you gave your delegate a copy of the same instructions you are using, it's a recursive function.

Note:

It does not matter at all that your delegate is using the same instructions you are using.

Designing Recursive Algorithms

Recursive Structure

Every recursive function is essentially a conditional.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Designing Recursive Algorithms

Base case(s)

At least one branch of the conditional has a very simple return.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Designing Recursive Algorithms

Recursion Property

The base case expresses a solution to the smallest problem that can be solved by the function.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Designing Recursive Algorithms

General (aka Recursive or Inductive) Case(s)

At least one branch of the conditional has a call to the function itself.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Designing Recursive Algorithms

Recursion Property

The recursive case expresses a true relationship between the problem you are given, and the solution to a smaller, related problem.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

A template for recursive algorithms

```
def <function name>(<data>)
   if <base case test>:
      <base case task>
   else:
     Calculate <new params> for smaller, related problem
     <subSolution> = Call <name> recursively on <new params>
     <solution> = Combine <subSolution> with <data>
     return <solution>
```

Sum of Squares

Recursive case

Calculate the sum of squares: $1^2 + 2^2 + 3^2 + \cdots + n^2$, for $n > 0$

- Let

$$s(n) = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

- Rewrite using $n - 1$:

$$s(n-1) = 1^2 + 2^2 + 3^2 + \cdots + (n-1)^2$$

  ...but only if $n > 1$ of course. What happens if $n = 1$?

- Subtract:

$$s(n) - s(n-1) = n^2$$

  because they have the same terms except the last one

- Rearrange:

$$s(n) = n^2 + s(n-1)$$

Sum of Squares

Base case

Calculate the sum of squares: $1^2 + 2^2 + 3^2 + \cdots + n^2$, for $n > 0$

- Let
$$s(n) = 1^2 + 2^2 + 3^2 + \cdots + n^2$$
- Then the smallest n we can consider is $n = 1$
- So:
$$s(1) = 1^2 = 1$$

Sum of Squares

Use the template to build a function for Sum of Squares

Thinking recursively: Rules of thumb

Numerical Recursion:

- Parameters often include at least one integer.
- Base cases of $n = 0$ or $n = 1$ is very common.
- Smaller subproblem is often $n - 1$; sometimes you can use $n/2$.
- Building up a solution always uses *associativity*, e.g., $1 + 2 + 3 = (1 + 2) + 3$ or $1 \times 2 \times 3 = (1 \times 2) \times 3$

Note:

The right choice is right because it fits your problem.

Example: Binary search

```python
def bin_search(C, target, start, end):
    """
    Determine if target occurs in collection C, between
    indices start and end
Pre-conditions:
    C: a collection of data items in increasing order
    target: the target key
    start: first offset of C to be searched
    end: last offset of C to be searched
Return: True if C contains the target
    """
    if end < start:
        return False

    mid = (start + end) // 2

    if C[mid] == target:
        return True
    elif C[mid] < target:
        return bin_search(C, target, mid+1, end)
    else:
        return bin_search(C, target, start, mid-1)
```

Thinking recursively: Understanding the problem

- Recursion always makes sense as a relationship.
- Relationships come from understanding the problem.
  - Draw diagrams.
  - Do short mathematical derivations.
  - Write down the solution to small examples.
  - Try to find relationships between problems of different sizes.
  - Staring at Python code is not helpful.

Designing Recursive functions: Exercises

Write a recursive function for each of the following:

1. Calculate r^n for $n \geq 0$. Assume n is integer.
2. Calculates $1 - 2 + 3 - 4 + 5 \cdots n$
3. Determines if an integer n is even.
4. Determines if n is evenly divisible by m.

Our node ADT is recursive!

- This is not coincidence.
- Node sequences, like integers, are inherently recursive.
- A node sequence starting at `theNode` always contains a smaller sequence starting at `get_next(theNode)`!

Thinking recursively: Rules of thumb for Node chains

- Parameters include at least one node (not the List record)
- Base cases always include `theNode == None`
- Not uncommon to have multiple base cases, e.g. in search
- Smaller subproblem always follows the next field, e.g., `get_next(theNode)`. It's smaller because following the reference gets you closer to the end of the sequence.

Designing Recursive functions: Exercises

Write a recursive function for each of the following:

1. Displays the data values stored in a node-chain
2. Counts the number of nodes in a node-chain
3. Calculates the sum of the numbers stored in a node-chain

Assume that a node chain starts with the reference stored in the variable `chain`.

Thinking recursively

- Thinking recursively means understanding relationships in the problem.
- It would be a big mistake for anyone to try to understand recursion by simulating the computer.
- Still, we have to understand how recursion is implemented.
- Keep these two ideas separate. They are related, but not equivalent.
    1. Understanding recursive relationships.
    2. Understanding the implementation of recursion by a computer.

How does a function call work?

1. A function call creates a “frame”, and pushes the frame on a “system” or “call” stack.
2. Parameters and local variables are names in the frame.
3. Each name has a place to store a reference to a value
4. When the function returns, the return value is saved, but the frame is popped from the stack, and discarded.

How does Recursion Work?

1. A function call creates a “frame”, and pushes the frame on a “system” or “call” stack.
2. Parameters and local variables are names in the frame.
3. Each name has a place to store a reference to a value
4. When the function returns, the return value is saved, but the frame is popped from the stack, and discarded.

Recursion works because there is a call stack for functions.
There is nothing special about a recursive function.

The System Stack, a.k.a the Call Stack

- Frames (a.k.a. activation records), are stored on a stack provided to the app by the operating system.
  - This stack is called the system stack.
- Whenever a function is called, its activation record gets pushed onto the stack.
  - When a function returns, its activation record is popped.
- The activation record for the currently executing function is always on the top of the stack.

Depth of Recursion in Python

- Python's call stack is limited in size.
- If recursion gets too deep and uses up stack space, no more functions can be called.
- This is called a stack overflow, which results in a run-time error.
- Python's default is to limit recursion to about 1000 recursive calls.
  - This reflects a conservative attitude about recursion.
  - Very often, if recursion reaches the limit, it's because of an error.
  - This limit can be changed.

Depth of Recursion in Other Languages

- Every application's call stack is limited in size.
- If recursion gets too deep and uses up stack space, no more functions can be called.
- This is called a stack overflow, which results in a run-time error.
- Usually, stack overflow results in catastrophic failure of an app.
- Most languages do not limit recursion depth.
- It's your job to prevent stack overflow.

Python is not the whole story!

- Recursion in Python (and many languages) requires stack space.
- But the statement "Recursion always uses stack space" is false.
- Some languages can implement space efficient (i.e., $O(1)$) recursion. Python is not one of these.
- The technique is called Tail-call optimization
- It's good to be aware of the limitations of Python.
- It is more important not to over-generalize.

Time Complexity of Recursive Algorithms

- With loops:
  - Analyze the cost for body of the loop
  - Factor in the number of repetitions of the loop.
- With recursion:
  - Analyze the cost for body of the function, ignoring recursive calls
  - Factor in the number of times the function is called.

Example: Factorial

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

- Ignoring the function call, what's the cost of executing the function 1 time?
- How many times does the function get called?

Example: Sum of Squares

```
def sumsq(n):
    if n == 1:
        return 1
    else:
        return n*n + sumsq(n-1)
```

- Ignoring the function call, what's the cost of executing the function 1 time?
- How many times does the function get called?

Example: Exponentiation

```
def exp(r, n):
    if n == 0:
        return 1
    else:
        return r * exp(r, n-1)
```

- Ignoring the function call, what's the cost of executing the function 1 time?
- How many times does the function get called?

Example: Exponentiation

We can do better, if we've studied a little math.

$$r^{2n} = r^n \times r^n$$

- Rewrite the recursive function to use this property!
- Figure out the time complexity of the new version.

Example: Binary search

```
def bin_search(C, target, start, end):

    if end < start:
        return False

    mid = (start + end) // 2

    if C[mid] == target:
        return True
    elif C[mid] < target:
        return bin_search(C, target, mid+1, end)
    else:
        return bin_search(C, target, start, mid-1)
```

- Ignoring the function call, what's the cost of executing the function 1 time?
- How many times does the function get called?

Recursion

- It's fair to see recursion as just another way to do repetition
- We've seen no examples where recursion is the only way.
- But we will, right away!