Recursion

CS 1
Admin

- How's the project coming?
- After these slides, read chapter 13 in your book
  - Yes that is out of order, but we can read it stand alone
Objectives

• To understand the basic techniques for analyzing the efficiency of algorithms.
• To know what searching is and understand the algorithms for linear and binary search.
• To understand the basic principles of recursive definitions and functions and be able to write simple recursive functions.
  - Breaking problems into pieces
Objectives

- To understand sorting in depth and know the algorithms for selection sort and merge sort.
- To appreciate how the analysis of algorithms can demonstrate that some problems are intractable and others are unsolvable.
Solving problems

- **In Math**
  - many problems are solved using smaller versions of the same problem
  - Fibonacci numbers

- **In Philosophy**
  - inductive proofs
  - show base case is true
  - show that each later case follows from simple step and earlier case
Use the same technique in CS

- We can use the same inductive technique in our programming
  - recursion.
Searching

- Searching through data:
  - Looking for a particular value in a collections
- Search through a list of student records for one with your banner id so you can register
A simple Searching Problem

• Here is the specification of a simple searching function from your book:

```python
def search(x, nums):
    # nums is a list of numbers and x is a number
    # Returns the position in the list where x occurs
    # or -1 if x is not in the list.
```

• Here are some sample interactions:

```python
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

• It uses lists of integers which is overly simplistic but imagine Images or Strings
Python built in search

- To check to see if a value is in a list or not (what we were doing last time)
  - Python provides an easy way to do it.
  - If 'John' in names:
    - #Hooray I'm here
  - Use keyword in to check to see if some value is in a list.
  - Returns true if value is in list, returns false otherwise

- If we know it is in the list and want to know where
  - names.index('John')
  - Use index method on list object.
Searching for an object

- We have a list of students and a bannerId.
  - We want to find the student record in the list with just the bannerId.
  - The student record has the rest of the interesting stuff like GPA and student name and more.
  - So how do we search?
Searching for an object

- We have a list of students and a bannerId.
  - We want to find the student record in the list with just the bannerId.
  - The student record has the rest of the interesting stuff like GPA and student name and more.
  - So how do we search?
    - So you do it by flipping through and looking.
    - Computer can't do that.

- Without any assumptions
  - Computer must begin from beginning and look through data to find it.
  - (This is what in and index do)
Assumptions about data

- In the last slide I said without any assumptions about data
  - But what if we can make assumptions about the data?
  - How can we make it easier to search?
Assumptions about data

• In the last slide I said without any assumptions about data
  
  – But what if we can make assumptions about the data?
  – How can we make it easier to search?
  – If the data is ordered (sorted) then we don't have to look at every piece of data
    • We know by looking as one item, that lots of the data is either more than what we are looking for or less than what we are looking for
Introduce binary search

- Describe binary search
  - 'phone book' metaphor
- Let's implement it.
- Yup, this is one of those 'under the hood' things.
Comparing Algorithms

• Which search algorithm is better, linear or binary?
  - The linear search is easier to understand and implement
  - The binary search is more efficient since it doesn’t need to look at each element in the list

• Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?
Comparing Algorithms

- One way to conduct the test would be to code up the algorithms and try them on varying sized lists, noting the runtime.
  - Linear search is generally faster for lists of length 10 or less
  - There was little difference for lists of 10-1000
  - Binary search is best for 1000+ (for one million list elements, binary search averaged .0003 seconds while linear search averaged 2.5 second)
Comparing Algorithms

- While interesting, can we guarantee that these empirical results are not dependent on the type of computer they were conducted on, the amount of memory in the computer, the speed of the computer, etc.?
- We could abstractly reason about the algorithms to determine how efficient they are. We can assume that the algorithm with the fewest number of “steps” is more efficient.
Comparing Algorithms

• How do we count the number of “steps”?
• Computer scientists attack these problems by analyzing the number of steps that an algorithm will take relative to the size or difficulty of the specific problem instance being solved.
Comparing Algorithms

• For searching, the difficulty is determined by the size of the collection – it takes more steps to find a number in a collection of a million numbers than it does in a collection of 10 numbers.

• *How many steps are needed to find a value in a list of size* $n$?

• In particular, what happens as $n$ gets very large?
Comparing Algorithms

• Let’s consider linear search.
  - For a list of 10 items, the most work we might have to do is to look at each item in turn – looping at most 10 times.
  - For a list twice as large, we would loop at most 20 times.
  - For a list three times as large, we would loop at most 30 times!

• The amount of time required is linearly related to the size of the list, $n$. This is what computer scientists call a *linear time* algorithm.
Comparing Algorithms

• Now, let’s consider binary search.
  - Suppose the list has 16 items. Each time through the loop, half the items are removed. After one loop, 8 items remain.
  - After two loops, 4 items remain.
  - After three loops, 2 items remain
  - After four loops, 1 item remains.

• If a binary search loops $i$ times, it can find a single value in a list of size $2^i$. 
Comparing Algorithms

• To determine how many items are examined in a list of size \( n \), we need to solve \( n = 2^i \) for \( i \), or \( i = \log_2 n \).

• Binary search is an example of a log time algorithm – the amount of time it takes to solve one of these problems grows as the log of the problem size.
Comparing Algorithms

- This logarithmic property can be very powerful!
- Suppose you have the New York City phone book with 12 million names. You could walk up to a New Yorker and, assuming they are listed in the phone book, make them this proposition: “I’m going to try guessing your name. Each time I guess a name, you tell me if your name comes alphabetically before or after the name I guess.” How many guesses will you need?
Comparing Algorithms

- Our analysis shows us the answer to this question is \( \log_{10} 120000 \).
- We can guess the name of the New Yorker in 24 guesses! By comparison, using the linear search we would need to make, on average, 6,000,000 guesses!
Comparing Algorithms

- Earlier, we mentioned that Python uses linear search in its built-in searching methods. Why doesn’t it use binary search?
  - Binary search requires the data to be sorted
  - If the data is unsorted, it must be sorted first!
Let's implement linear search

- Use the ClassPrep.zip from the website.
Recursive Problem-Solving

• The basic idea between the binary search algorithm was to successfully divide the problem in half.
• This technique is known as a *divide and conquer* approach.
• Divide and conquer divides the original problem into subproblems that are smaller versions of the original problem.
Recursive Problem-Solving

- In the binary search, the initial range is the entire list. We look at the middle element... if it is the target, we’re done. Otherwise, we continue by performing a binary search on either the top half or bottom half of the list.
Recursive Problem-Solving

Algorithm: binarySearch - search for x in nums[low]...nums[high]

mid = (low + high)//2
if low > high
    x is not in nums
elsif x < nums[mid]
    perform binary search for x in nums[low]...nums[mid-1]
else
    perform binary search for x in nums[mid+1]...nums[high]

• The basic binary search algorithm
Recursive Definitions

- A description of something that refers to itself is called a recursive definition.
- In the last example, the binary search algorithm uses its own description – a “call” to binary search “recurs” inside of the definition – hence the label “recursive definition.”
Recursive Definitions

• Have you had a teacher tell you that you can’t use a word in its own definition? This is a *circular* definition.

• In mathematics, recursion is frequently used. The most common example is the factorial:

\[ n! = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases} \]

• For example, \( 5! = 5(4)(3)(2)(1) \), or \( 5! = 5(4!) \)

• Use board since it looks ugly cross platform on slide.
Recursive Definitions

- Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1. This is called a base case for the recursion.
- When the base case is encountered, we get a closed expression that can be directly computed.
Recursive Definitions

• All good recursive definitions have these two key characteristics:
  − There are one or more base cases for which no recursion is applied.
  − All chains of recursion eventually end up at one of the base cases.

• The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.
Recursive Functions

• We’ve seen previously that factorial can be calculated using a loop accumulator.
• If factorial is written as a separate function:

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```
Recursive Functions

- We’ve written a function that calls itself, a recursive function.
- The function first checks to see if we’re at the base case \((n==0)\). If so, return 1. Otherwise, return the result of multiplying \(n\) by the factorial of \(n-1\), \(\text{fact}(n-1)\).
Recursive Functions

>>> fact(4)
24
>>> fact(10)
3628800
>>> fact(100)
933262154439441526816992338856266700490715968264381621468592963895217
59999322991560894146397615618286253697920827223758251185210916864
0000000000000000000000000000L

- Remember that each call to a function starts that function anew, with its own copies of local variables and parameters.
Recursive Functions
Binary Search

● Now let's implement binary search
  - For the list of students
Admin

- Next project
- Reminder: Quiz
- Only one class next week.
Example: String Reversal

- Python lists have a built-in method that can be used to reverse the list. What if you wanted to reverse a string?
- If you wanted to program this yourself, one way to do it would be to convert the string into a list of characters, reverse the list, and then convert it back into a string.
Example: String Reversal

• Using recursion, we can calculate the reverse of a string without the intermediate list step.
  - That's a waste of space

• Think of a string as a recursive object:
  - Divide it up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end of it
Example: String Reversal

- `def reverse(s):
  return reverse(s[1:]) + s[0]

- The slice `s[1:]` returns all but the first character of the string.
- We reverse this slice and then concatenate the first character (`s[0]`) onto the end.
- Will this work?
Example: String Reversal

• >>> reverse("Hello")

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    reverse("Hello")
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
...  File "C:/Program Files/Python 2.3.3/z.py", line 8, in reverse
    return reverse(s[1:]) + s[0]
RuntimeError: maximum recursion depth exceeded

• Noooooooooooooooooo!!!!
Example: String Reversal

- Remember: To build a correct recursive function, we need a base case that doesn’t use recursion.
- We forgot to include a base case, so our program is an infinite recursion. Each call to reverse contains another call to reverse, so none of them return.
Example: String Reversal

• Each time a function is called it takes some memory. Python stops it at 1000 calls, the default “maximum recursion depth.”

• What should we use for our base case?
Example: String Reversal

• Each time a function is called it takes some memory. Python stops it at 1000 calls, the default “maximum recursion depth.”
• What should we use for our base case?
• Following our algorithm, we know we will eventually try to reverse the empty string. Since the empty string is its own reverse, we can use it as the base case.
Example: String Reversal

- `def reverse(s):
  if s == "":
    return s
  else:
    return reverse(s[1:]) + s[0]
- `>>> reverse("Hello")`
  'olleH'
a nice problem for recursion.

- **Problem**: test whether a sentence is a palindrome
  - **Palindrome**: a string that is equal to itself when you reverse all characters
    - A man, a plan, a canal–Panama!
    - Go hang a salami, I'm a lasagna hog
    - Madam, I'm Adam
  - how would you design a recursive solution to this problem?
Sample code minus solution

IsPalindrome(str):

...
Thinking Recursively

- We need
  - a base case
  - a simpler version of the problem.
- Consider various ways to simplify inputs
  - Here are several possibilities:
    - Remove the first character
    - Remove the last character
    - Remove both the first and last characters
    - Remove a character from the middle
    - Cut the string into two halves
- do any sound good?
Recursive solutions: Simplification

• Combine solutions with simpler inputs into a solution of the original problem
  • Most promising simplification: remove first and last characters
    - "adam, I'm Ada", is a palindrome too!
  • Thus, a word is a palindrome if
    • The first and last letters match, and
    • Word obtained by removing the first and last letters is a palindrome
Simplification scenarios

- What if first or last character is not a letter?
  - Ignore it
- If the first and last characters are letters, check whether they match;
  - if so, remove both and test shorter string
  - If last character isn't a letter, remove it and test shorter string
  - If first character isn't a letter, remove it and test shorter string
Base Cases

• Find solutions to the simplest inputs
  • Strings with two characters
    – No special case required; step two still applies
  • Strings with a single character
    – They are palindromes
  • The empty string
    – It is a palindrome
So let's write it.

- In PyCharm
Recursion vs. Iteration

- There are similarities between iteration (looping) and recursion.
- In fact, anything that can be done with a loop can be done with a simple recursive function! Some programming languages use recursion exclusively.
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.
Recursion vs. Iteration

• In the factorial and binary search problems, the looping and recursive solutions use roughly the same algorithms, and their efficiency is nearly the same.
  - Though factorial is uglier in looping
Recursion vs. Iteration

• So… will recursive solutions always be as efficient or more efficient than their iterative counterpart?

• The Fibonacci sequence is the sequence of numbers 1,1,2,3,5,8,…
  – The sequence starts with two 1’s
  – Successive numbers are calculated by finding the sum of the previous two
Recursion vs. Iteration

• Loop version:
  - Let’s use two variables, $\text{curr}$ and $\text{prev}$, to calculate the next number in the sequence.
  - Once this is done, we set $\text{prev}$ equal to $\text{curr}$, and set $\text{curr}$ equal to the just-calculated number.
  - All we need to do is to put this into a loop to execute the right number of times!
Recursion vs. Iteration

- `def loopfib(n):
  # returns the nth Fibonacci number

  curr = 1
  prev = 1
  for i in range(n-2):
    curr, prev = curr+prev, curr
  return curr

- Note the use of simultaneous assignment to calculate the new values of `curr` and `prev`.
- The loop executes only \( n-2 \) times since the first two values have already been “determined”.
Recursion vs. Iteration

• The Fibonacci sequence also has a recursive definition:

\[
\begin{align*}
1 & \text{ if } n < 3 \\
& \text{ otherwise }
\end{align*}
\]

• This recursive definition can be directly turned into a recursive function!

• def fib(n):
  if n < 3:
    return 1
  else:
    return fib(n-1)+fib(n-2)
Recursion vs. Iteration

- This function obeys the rules that we’ve set out.
  - The recursion is always based on smaller values.
  - There is a non-recursive base case.
- So, this function will work great, won’t it? – Sort of…
Recursion vs. Iteration

- The recursive solution is extremely inefficient, since it performs many duplicate calculations!
Recursion vs. Iteration

- To calculate $\text{fib}(6)$, $\text{fib}(4)$ is calculated twice, $\text{fib}(3)$ is calculated three times, $\text{fib}(2)$ is calculated four times… For large numbers, this adds up!
Recursion vs. Iteration

- Recursion is another tool in your problem-solving toolbox.
- Sometimes recursion provides a good solution because it is more elegant or efficient than a looping version.
- At other times, when both algorithms are quite similar, the edge goes to the looping solution on the basis of speed.
- Avoid the recursive solution if it is terribly inefficient, unless you can’t come up with an iterative solution (which sometimes happens!)