Searching (and some follow ups)

Summary: Recursion vs. Iteration

- Some problems that are simple to solve with recursion are quite difficult to solve with loops.
- Every recursive program has an equivalent non-recursive program (it can be generated by program).
- A simple non-recursive version is generally more efficient than a recursive one
- Example when recursion is a poor choice: computing Fibonacci numbers

Common mistakes when using recursion

- Missing base case for terminating the recursion
  - Needs to exist in code and be executed
- No convergence
  - Make sure the problem size decreases
- Excessive memory requirements
  - May need to be increased for a correct program

```
from sys import setrecursionlimit
setrecursionlimit(5000)   # default is 1000
```

- Excessive re-computations
  - As done in recursive Fibonacci code

Clicker Question

```
def what(L):
    if L == []:
        return 0
    else:
        return what(L[1:]) + 1
```

What does function what do when given a list as an argument?

A. Generates an error
B. Returns the length of the list
C. Returns the length of the list + 1
D. Returns the number of characters in the list

Comments on percolation project

- Decide if you want to
  - detect percolation and stop as soon as bottom row is reached, or
  - explore the entire grid; in this case, detect percolation outside after the wave exploration is done
- A grid size of 75 by 75 may be large if your code is inefficient (omit this size if it takes too long)
- For testing:
  - Use small grid sizes (like 10 by 10), one grid per execution
  - Turn visualization on

Searching

***Searching*:** look for a particular value in a data collection (lists, arrays).

- It is a basic operation
- Python provides a number of built-in search-related methods
- A search returns
  - -1 if the element is not found
  - the position of the element in the collection if it is found

Searching in Python

```
>>> numL = [3, 1, 4, 2, 5]
>>> numL.index(4)
2
>>> numL.index(6)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    numL.index(6)
ValueError: list.index(x): x not in list

if x in numL:           if x not in numL:
    # do something            # do something
```

7

Searching in a list (what we want to do)

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

8 Wednesday, March 4, 2009

http://www.thescripts.com/forum/thread32188.html

Hi,

I've got a list with more than 500,000 ints. Before inserting new ints, I have to check that it doesn't exist already in the list.

Currently, I am doing the standard:

```
if new_int not in long_list:
    long_list.append(new_int)
```

but it is extremely slow… is there a faster way of doing this in python?

9

Searching: assumptions

Entries are stored in a structure A so that

- you can access an arbitrary element as A[i]
- you can scan/itertate over the structure from beginning to end

10

Example: Linear Search

```
def search(A, x):
    for i in range(len(A)):
        if A[i] == x:
            return i
    return -1
```

A linear search has to look at every element if x is not in A

If the elements stored in A are in arbitrary order, one needs to look at every one (until found or end is reached)

Would it help if the elements were in **sorted order**?

Binary Search in a sorted list

- Use two variables to keep track of the endpoints of the range in the sorted list/array where x could be.
- initially *low* is set to the first and *high* is set to the last location in A.
- Compare the middle element to x:
  - x is smaller than the middle element, then
    binary search for x in right half
  - x is larger than the middle element, then
    binary search for x in left half

Recursive Binary Search

```
def recBinSearch(A, x, low, high):

   if low > high:                    # base case if x not yet found
         return  -1                          # No place left to look,

   mid = (low + high) / 2
   item = A[mid]
   if item == x:                     # Found it! Return index
         return mid
   elif x < item:                     # Look in lower half
         return recBinSearch(A, x, low, mid-1)
   else:                              # Look in upper half
         return recBinSearch(A, x, mid+1, high)

def Search(A, x):
   return recBinSearch(A, x, 0, len(A)-1)
```

13 Wednesday, March 4, 2009

bin_search_trace.py

How good is binary search?

- It is the best way to search in a sorted structure
  - Need to be able to index any element
- It makes up to log n comparisons (log is base 2, ignore floor and ceiling)
  - Searching a list with 500,000 records takes at most 23 comparisons

14 Wednesday, March 4, 2009