Topics for Today

- More recursive functions
  - Searching in a sorted sequence
  - (Exponentiation)
  - Recursive drawings
  - Searching in a grid

Reading: Zelle, Chapter 13.2

Searching

***Searching*:** look for a particular value in a list/array

- It is a basic operation you already used
- Python provides a number of built-in search-related methods
- A search should return
  - -1 if the element is not found
  - the position of the element if it is found

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
```

Searching in Python

```
if x in nums:
    # do something
```

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

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

index operation raises an exception if the target value does not appear

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?

Searching: assumptions

Entries are stored in a structure A so that

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

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

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

Have you ever played the number guessing game, where I pick a number between 1 and 100 and you try to guess it?

Binary Search - Idea

- 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:                    # No place left to look, return -1
     return  -1
  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)
```

bin_search_trace.py

Clicker Question 0 – Part. only

When do you plan on starting to study for exam 1?

A. I have been studying all along and I am almost ready
B. I will start on the weekend
C. I will start Tuesday after project 1 is submitted
D. The material is easy for me and I don't need to study

Clicker Question 1

On a list consisting of 500,000 elements, how many comparisons could a binary search make?

A. 50,000
B. 500,000
C. 23
D. 256

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
- Why are we counting comparisons?

Fast Exponentiation

- One way to compute a^n for an integer n is to multiply a by itself n times.
- This can be done with a simple accumulator loop:

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

Fast Exponentiation

- We can solve this problem using recursion.
- We know that $2^8 = 2^4(2^4)$.
  - If we know 2^4, we can calculate 2^8 using **one** multiplication.
  - How is 2^4 computed? Using 2^2 and **one** multiplication
  - How is 2^2 computed? Using 2 and **one** multiplication
  - We can calculate 2^8 using only three multiplications!

$$a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if n is even} \\ a^{n/2}(a^{n/2})(a) & \text{if n is odd} \end{cases}$$

Fast Exponentiation

```
def recPower(a, n):
   # raises a to the n-th power
   if n ==  0:
      return 1

   else:
       factor = recPower(a, n/2)

       if n%2 == 0:       # n is even
           return factor*factor
       else:              # n is odd
           return factor*factor*a
```

Use variable *factor* so that we don't calculate $a^{n/2}$ more than once

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

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.
- In the exponentiation problem, two different algorithms are used.
  - The looping version takes linear time to complete, while the recursive version executes in log time.

F(n) = F(n-1) + F(n-2); F(0)=0, F(1)=1

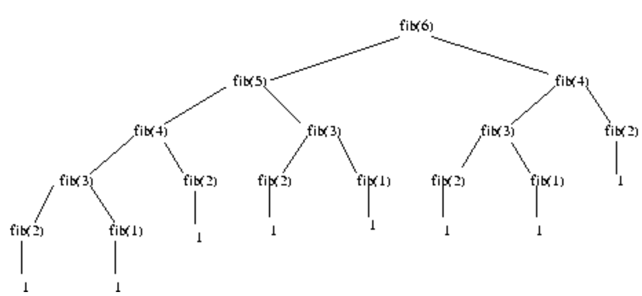
```
# iterative function computing the n-th Fibonacci number
def loopfib(n):
    curr = 1
    prev = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr

#recursive function computing the n-th Fibonacci number
def fib(n):
      if n < 3:
          return 1
      else:
          return fib(n-1)+fib(n-2)
```

fib_rec_trace.py

Recursive fib(n)

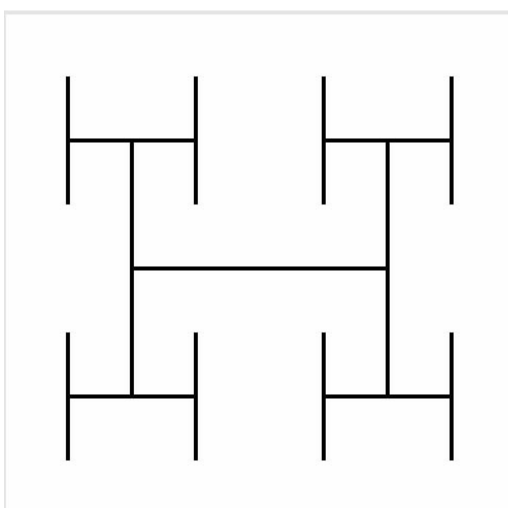
The recursive solution is extremely inefficient, since it performs many duplicate calculations!



Recursive drawings

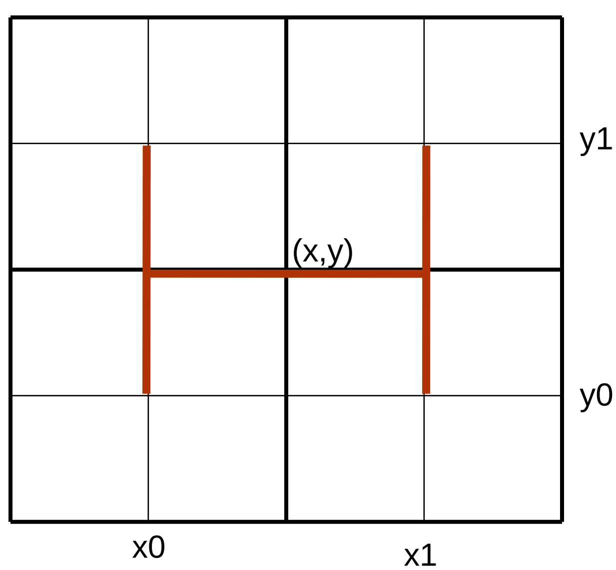
- Simple recursive drawings can lead to interesting pictures
- **H-tree drawings**
  - An H–tree of order 1 consists of drawing the letter H
  - An H-tree of order n is created by
    - drawing four H-trees of order n-1, each connected to the tip on an H
    - The side length of each H-tree of order n-1 has half of the side length of an order n H-tree

H-tree of order 2



Who needs to draw H-trees?

- Circuit design
  - used as a distribution network for routing time signals
- Multi-processor interconnection structures
  - Space–efficient embedding of a tree communication structure
- Running underground cables
  - forming a distribution center
- H-trees are an example of a fractal canopy (related to a Mandelbrot tree)



```
def draw_Htree(n, sz, x, y):
# n is order; (x,y) is center of drawing area of size sz
    if n > 0:
        x0 = x - sz/2
        x1 = x + sz/2
        y0 = y - sz/2
        y1 = y + sz/2

        curve(pos=[(x0,y),(x1,y)],    color=color.red)
        curve(pos=[(x0,y0),(x0,y1)],  color=color.red)
        curve(pos=[(x1,y0),(x1,y1)],  color=color.red)

        draw_Htree(n-1, sz/2, x0, y0)
        draw_Htree(n-1, sz/2, x0, y1)
        draw_Htree(n-1, sz/2, x1, y0)
        draw_Htree(n-1, sz/2, x1, y1)
```


4 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(2000)   # default is 1000
```

- Excessive recomputations
  - As done in Fibonacci code