Topics for Today

- Recursive functions
  - examples and different approaches
- Percolation Project
  - use vertical percolation function from lab to start experimental part before wave function is completed
  - use data files from lab to test your wave percolation function

Reading: Zelle, Chapter 13.2

Clicker Question

Fibonacci number are defined as

\[ F(0) = 0, \ F(1) = 1 \]
\[ F(n) = F(n-1) + F(n-2) \]

Assume you need to compute \( F(1000) \). You would do this by write a program using

A. an array of size 1000 and fill the array using a for-loop
B. using a for-loop, but no array or list structure
C. write a recursive function having two base cases
D. compute \( F(1000) \) using a closed form solution
Clicker Question (part. only)

What is the approximate solution to the following recursive definition?

\[ F(n) = F\left(\lfloor n/2 \rfloor \right) + 1 \]
\[ F(1) = 0 \]

A. It is approximately n/2
B. It is approx. \log n (base 2)
C. None of the listed options
D. It is approximately \sqrt{n}/2

Recursive functions

- Euclid’s algorithm
- String reversal
- Anagrams
- Searching in a sorted list/array
- Fast exponentiation
- Paths searching in a grid (Project 2, Part 2)
- SUDOKU (Lab)
Example: Euclid’s GCD Algorithm

```python
def nonrec_gcd1(m, n):
    # assume m<=n
    while m != 0:
        m, n = n%m, m
    return n

def nonrec_gcd2(m, n):
    while m != 0:
        temp = m
        m = n%m
        n = temp
    return n
```

Euclid's algorithm determines the greatest common divisor (GCD). It dates back to the ancient Greeks (first recorded 300BC).

Recursive gcd(m,n)

```python
def gcd(m, n):
    if m == 0:
        return n
    else:
        return gcd(n%m, m)
```

What pairs of numbers require the most iterations? Two successive Fibonacci numbers.

euclid.py

Monday, March 2, 2009
Example: String Reversal

- For “recursion” generate “noisrucer”
- Python lists have a built-in method that can be used to reverse a list: L.reverse()
- How to reverse a string?

**Solution 1**: use the list operation

- convert the string into a list of characters
- reverse the resulting list (using L.reverse())
- convert the new list back into a string

Solution 2: Recursive Reversal

- Goal is to use recursion to reverse a string without the intermediate list step:
  - Divide the string up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end of it
- Recursion needs to terminate
  - choose empty string as the termination condition
Idea of recursive reversal

```python
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.

Recursive String Reversal - complete

```python
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]

>>> reverse("Hello")
'olleH'
```

reverse.py
Clicker Question

```python
>>> myL = "rotavator"
>>> print reverse(reverse(myL)[2: ])
```

A. tavator  
B. rotavat  
C. rot  
D. tor

---

Example: Anagrams

- An *anagram* is formed by rearranging the letters of a word.
- ```python
>>> anagrams("abc")
['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```
- The number of anagrams of a word is the factorial of the length of the word.
- Write a recursive function generating all anagrams of a given word.
Recursive Idea

- Slice the first character off the given word
- Make a recursive call that generates a list containing all anagrams formed by the remaining characters of the word
- Consider every word w in this returned list:
  - Place the first character in all possible locations within word w

Example

- st = “abc”. Stripping off the “a” leaves “bc”.
- Generating all anagrams of “bc” gives the list [“bc”, “cb”].
- To form the anagram of the original string, place “a” in all possible locations within these two smaller anagrams:
  [“abc”, “bac”, “bca”, “acb”, “cab”, “cba”]
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        list_without_first = anagrams(s[1:]),

        for w in list_without_first:
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])

        return ans

Comments of function anagrams

- The outer for-loop iterates through each anagram in the list returned from the recursive call.
- The inner loop goes through each position in an anagram w and creates range(len(w)+1) new strings
  - the original first character inserted into all possible positions.
- Insertion is done by the operation w[:pos]+s[0]+w[pos:]
  - w[:pos] gives the part of w up to, but not including, pos.
  - w[pos:] gives everything from pos to the end.
  - Inserting s[0] between them effectively inserts it into w at pos.
\[ F(n) = F(n-1) + F(n-2); \ F(0)=0, \ F(1)=1 \]

# iterative function computing the n-th Fibonacci number

```python
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

```python
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!