Outline

Recursive Definitions

A Function Can Call Itself

- A recursive definition of a function is one which makes a function call to the function being defined.
- The function call is then a recursive function call.
- A definition is circular if it leads to an infinite sequence of function calls.
- To prevent this, *the function must call itself with a parameter smaller than the one it is using*.
  The function must test for when the parameter has reached the minimum size (the base case): this must be handled without a recursive call.

Recursive Definitions

The Call Stack

The function call stack can handle recursive functions easily. There is no reason why a function can't push an activation record onto the call stack with variables for the current function while calling that same function. The earlier version of that function will resume when the recursive call is completed. When the base case is finally met, there will be no further recursive calls, and no further activation records will be pushed onto the stack.

Without a base case, the stack would overflow, producing a run-time error.

Recursive Definitions

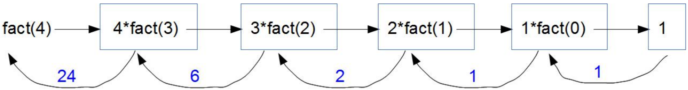
The Factorial Function

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

The Factorial Function - using Python

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

Recursive Definitions



The Factorial Function - using Python

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

See fact.py

String Reversal

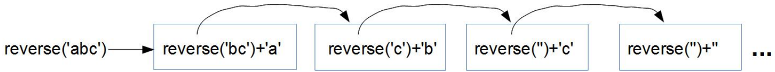
Circular Definition

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

String Reversal

Circular Definition

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



String Reversal

Definition with Base Case

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

See reverse.py

Anagrams

Anagrams

An anagram of a word is another word spelled using the same letters but rearranged. Rearrangements are also called permutations. For example: TORSO is an anagram for ROOST. A recursive strategy to produce all anagrams of a given word is:

- remove the first letter from the word.
- for all anagrams of the smaller word, insert it in all possible positions.

How many permutations of the letters in TORSO is there?

Anagrams

Anagrams

An anagram of a word is another word spelled using the same letters but rearranged. Rearrangements are also called permutations. For example: TORSO is an anagram for ROOST. A recursive strategy to produce all anagrams of a given word is:

- remove the first letter from the word.
- for all anagrams of the smaller word, insert it in all possible positions.

  How many permutations of the letters in TORSO is there? $5! = 120$

Anagrams

Anagrams Using Recursion

```
def anagrams(s):
   if s == "":
      return [s]
   else:
      ans = []
      for w in anagrams(s[1:]):
         for pos in range(len(w)+1):
            ans.append(w[:pos]+s[0]+w[pos:1])
      return ans
```

See anagrams.py

Fast Exponentiation

Naive Iteration is $\Theta(n)$

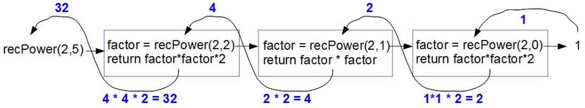
```
# power.py
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

Fast Exponentiation

Divide and Conquer Recursion is $\Theta(log\ n)$

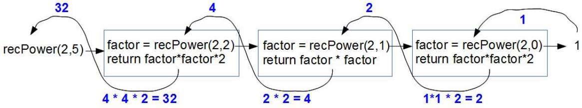
```
# power.py
def recPower(a, n):
    if n == 0:
        return 1
    else:
        factor = recPower(a, n // 2)
        if n % 2 == 0:
            return factor * factor
        else:
            return factor * factor * a
```

Fast Exponentiation




```
# power.py
def recPower(a, n):
    if n == 0:
        return 1
    else:
        factor = recPower(a, n // 2)
        if n % 2 == 0:
            return factor * factor
        else:
            return factor * factor * a
```

Fast Exponentiation



Let's compare the number of multiplications performed while using recursive definition of power function and naive definition of power function:

naive (running time $\Theta(n)$): 4 multiplications
recursive (running time $\Theta(\log n)$): 5 multiplications

If we try 2^{10}, then
naive: 9 multiplications
recursive: 6 multiplications

Binary Search

Iteration

```python
def search(items, target):
   low = 0
   high = len(items) - 1
   while low <= high:
      mid = (low + high) // 2
      item = nums[mid]
      if target == item:
         return mid
      elif target < item:
         high = mid - 1
      else:
         low = mid + 1
   return -1
```

Binary Search

Pseudocode Using Recursion

```
Algorithm:  binary search
-- search for x in nums[low]...nums[high]
  if low > high
    x is not in nums
  mid = (low + high) // 2
  if x== nums[mid]:
    x is at mid position
  elif x < nums[mid]
    binary search for x in nums[low]...nums[mid-1]
  else
    binary search for x in nums[mid+1]...nums[high]
```

Binary Search

Python Code Using Recursion

```
def search(items, target):
   return recBinSearch(target, items, 0, len(items)-1)
def recBinSearch(x, nums, low, high):
   if low > high:
      return -1
   mid = (low + high) // 2
   item = nums[mid]
   if x == item:
      return mid
   elsif x < item:
      return recBinSearch(x, nums, low, mid-1)
   else:
      return recBinSearch(x, nums, mid+1, high)
```

In-Class Work

1. Show pictorial representation of the call `recPower(3,7)`.
2. Figure out exactly how many multiplications does `recPower(3,7)` do.
3. Write and test a recursive function `Maximum` to find the largest number in a list.
   Hint: the maximum is the larger of the first item and maximum of all the other items.