Recursion in Python

Recursion

- A recursive function is one that calls itself.

```
def i_am_recursive(x) :
    maybe do some work
    if there is more work to do :
        i_am_recursive(next(x))
    return the desired result
```

- Infinite loop? Not necessarily, not if `next(x)` needs less work than x.

Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.

$$n! = n(n-1)(n-2)\ldots(1)$$

$$n! = n(n-1)!$$

base case

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

Recursive Definitions

- A recursive definitions should have 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.

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

Recursive Definitions

- Every recursive function definition includes two parts:
  - Base case(s) (non-recursive)
    One or more simple cases that can be done right away
  - Recursive case(s)
    One or more cases that require solving "simpler" version(s) of the original problem.
    - By "simpler", we mean "smaller" or "shorter" or "closer to the base case".

Example: Factorial

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

  $2! = \quad 2 \times 1$

  $3! = \quad 3 \times 2 \times 1$

  $4! = \quad 4 \times 3 \times 2 \times 1$

- alternatively:

  $0! = 1$ (Base case)

  $n! = n \times (n-1)!$ (Recursive case)

  So $4! = 4 \times 3!$

  And $3! = 3 \times 2!$, $2! = 2 \times 1!$, $1! = 1 \times 0!$

Recursion conceptually

$4! = 4(3!)$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = 24$

$\qquad 3! = 3(2!)$ $\qquad\qquad\qquad\qquad\qquad\qquad = 6$

$\qquad\qquad 2! = 2(1!)$ $\qquad\qquad\qquad\qquad = 2$

$\qquad\qquad\qquad 1! = 1\,(0!) = 1(1) = 1$

Compute the base case

make smaller instances of the same problem

build up the result

Recursive Factorial in Python

```
# 0! = 1 (Base case)
# n! = n × (n-1)! (Recursive case)
def factorial(n):
    if n == 0:      # base case
        return 1
    else:           # recursive case
        return n * factorial(n-1)
```

Inside Python Recursion

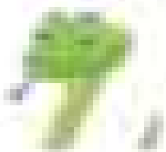
S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`

A n=2 `factorial(2)? = 2 * factorial(1)`

C n=1 `factorial(1)? = 1 * factorial(0)`

K n=0 `factorial(0) = 1`



Recursive vs. Iterative Solutions

- For every recursive function, there is an equivalent iterative solution.
- For every iterative function, there is an equivalent recursive solution.
- But some problems are easier to solve one way than the other way.
- And be aware that most recursive programs need space for the stack, behind the scenes

Factorial Function (Iterative)

```
def factorial(n):
    result = 1    # initialize accumulator var
    for i in range(1, n+1):
        result = result * i
    return result
```

Versus (Recursive):

```
def factorial(n):
    if n == 0:     # base case
        return 1
    else:           # recursive case
        return n * factorial(n-1)
```

Iteration to Recursion: exercise

- Mathematicians have proved
  $\pi^2/6 = 1 + 1/4 + 1/9 + 1/16 + \ldots$
- We can use this to approximate π
- Compute the sum, multiply by 6, take the square root

```
def pi_series_iter(n) :
    result = 0
    for i in range(1, n+1) :
        result = result + 1/(i**2)
    return result

def pi_approx_iter(n) :
    x = pi_series_iter(n)
    return (6*x)**(.5)
```

Let's convert this to a recursive function (see file pi_approx.py for a sample solution.)

Recursion on Lists

- First we need a way of getting a smaller input from a larger one:
  - Forming a sub-list of a list:

```
>>> a = [1, 11, 111, 1111, 11111, 111111]
>>> a[1:]
[11, 111, 1111, 11111, 111111]
>>> a[2:]
[111, 1111, 11111, 111111]
>>> a[3:]
[1111, 11111, 111111]
>>> a[3:5]
[1111, 11111]
>>>
```

the "tail" of list a

Recursive sum of a list

```
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

What if we already know the sum of the list's tail? We can just add the list's first element!

Tracing sumlist

```
def sumlist(items):
    if items== []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

```
>>> sumlist([2,5,7])
sumlist([2,5,7]) = 2 + sumlist([5,7])
                       5 + sumlist([7])
                           7 + sumlist([])
                                   0
```

After reaching the base case, the final result is built up by the computer by adding 0+7+5+2

Fibonacci Numbers

- A sequence of numbers:

0
1
1
2
3
5
8
13
...

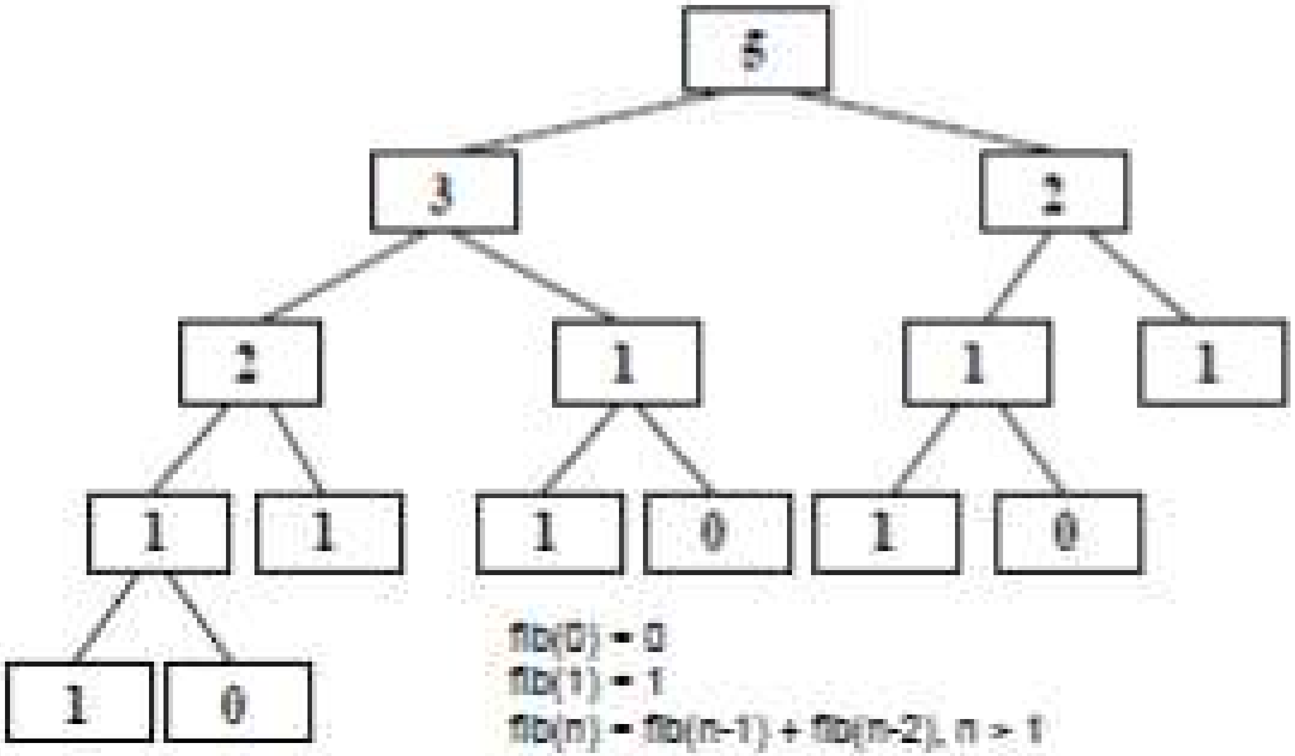
Recursive Definition

- Let fib(n) = the nth Fibonacci number, $n \geq 0$
  - fib(0) = 0 (base case)
  - fib(1) = 1 (base case)
  - fib(n) = fib(n-1) + fib(n-2), $n > 1$

Two recursive calls!

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Recursive Call Tree



fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

Iterative Fibonacci

```
def fib(n):
    x = 0
    next_x = 1
    for i in range(1,n+1):
        x, next_x = next_x, x + next_x
    return x
```

SIMULTANEOUS ASSIGNMENT

Faster than the recursive version. Why?

String Reversal

- Write a function to reverse a given string
  - Divide it up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end

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

>>> 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
```

- What happened? There were 1000 lines of errors!

String Reversal

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

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

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

Fast Exponentiation

- One way to compute a^n

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

  - multiply a by itself n times.

Fast Exponentiation

- Another way to compute a^n
  - divide and conquer.
- $a^n = a^{n//2}(a^{n//2})$?

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

  - $2^8 = 2^4(2^4)$
  - $2^9 = 2^4(2^4)2$

Fast Exponentiation

```
def recPower(a, n) :
    # raises a to the int power n
    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
```

- temporary variable *factor* is used so that we don't need to calculate $a^{n//2}$ more than once

Sorting Algorithms

- The sorting problem
  - take a list of n elements
  - and rearrange it so that the values are in increasing (or decreasing) order.
- *Selection sort*
  - For n elements, we find the smallest value and put it in the 0^{th} position.
  - Then we find the smallest remaining value from position 1 to (n−1) and put it into position 1.
  - The smallest value from position 2 to (n−1) goes in position 2.
  - …

Naive Sorting: Selection Sort

```
def selSort(nums):
    # sort nums into ascending order

    n = len(nums)

    # For each position in the list (except the very last)

    for bottom in range(n-1):
        # find the smallest item in nums[bottom]...nums[n-1]

        mp = bottom                    # bottom is smallest initially
        for i in range(bottom+1, n):    # look at each position
            if nums[i] < nums[mp]:      # this one is smaller
                mp = i                  # remember its index

        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Divide and Conquer

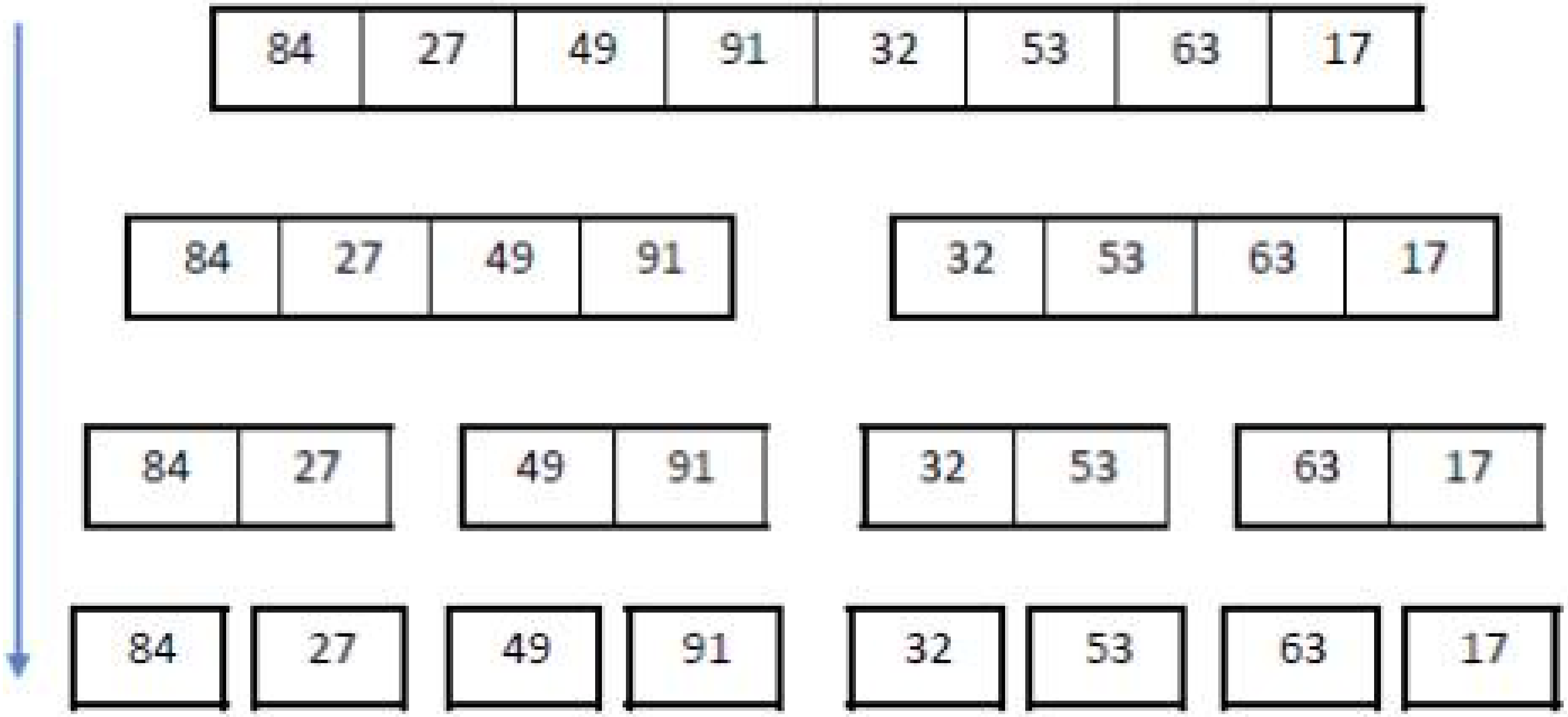
- In computation:
  - Divide the problem into "simpler" versions of itself.
  - Conquer each problem using the same process (usually recursively).
  - Combine the results of the "simpler" versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort, Quicksort, and many, many more

Divide and Conquer: Merge Sort

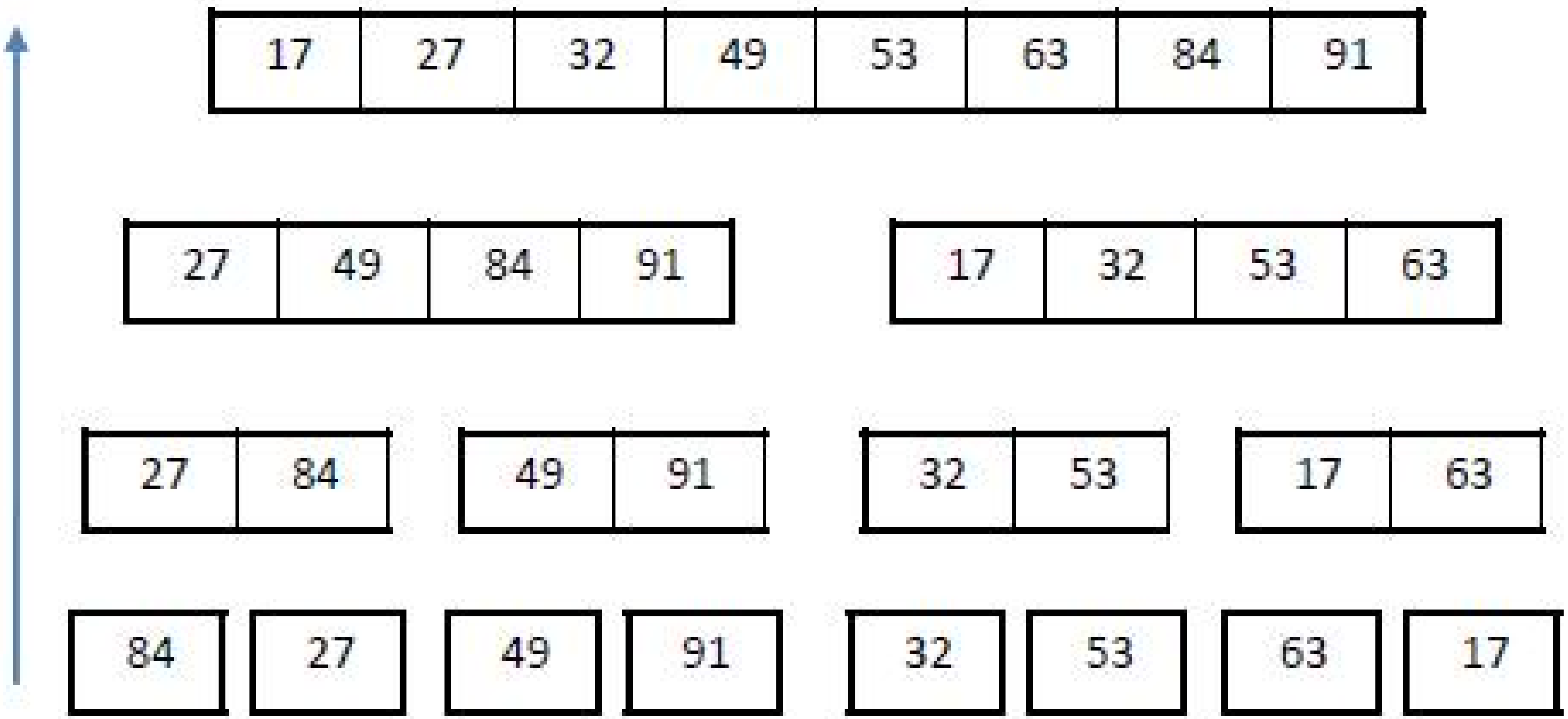
- *merge sort*
  - Merging: combining two sorted lists into a single sorted list

```
split nums into two halves
sort the first half
sort the second half
merge the two sorted halves back into nums
```

Divide (Split)



Conquer (Merge)



Merge Outline

- Input: Two lists *a* and *b*, already sorted
- Output: A new list containing the elements of *a* and *b* merged together in sorted order.
- Algorithm:

1. Create an empty list *c*, set *index_a* and *index_b* to 0
2. While *index_a* < length of *a* and *index_b* < length of b
   a. Add the smaller of *a*[*index_a*] and *b*[*index_b*] to the end of *c*, and increment the index of the list with the smaller element
3. If any elements are left over in *a* or *b*, add them to the end of *c*, in order
4. Return *c*

Divide and Conquer: Merge Sort

```
def merge(lst1, lst2, lst3):
    # merge sorted lists lst1 and lst2 into lst3
    # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0  # all start at the front
    n1, n2 = len(lst1), len(lst2)

    # Loop while both lst1 and lst2 have more items
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]: # top of lst1 is smaller
            lst3[i3] = lst1[i1] #  copy it into current spot in lst3
            i1 = i1 + 1
        else:                   # top of lst2 is smaller
            lst3[i3] = lst2[i2] #  copy itinto current spot in lst3
            i2 = i2 + 1
        i3 = i3 + 1             # item added to lst3, update position
```

Divide and Conquer: Merge Sort

```
# Here either lst1 or lst2 is done. One of the following loops
# will execute to finish up the merge.

    # Copy remaining items (if any) from lst1
    while i1 < n1:
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1

    # Copy remaining items (if any) from lst2
    while i2 < n2:
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
```

Divide and Conquer: Merge Sort

```
def mergeSort(nums):
    # Put items of nums into ascending order
    n = len(nums)

    if n > 1:     # Do nothing if nums contains 0 or 1 items

        m = n/2  # split the two sublists
        nums1, nums2 = nums[:m], nums[m:]
                # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
                # merge the sorted pieces back
        merge(nums1, nums2, nums)
```

Comparing Sorts

- Selection Sort
- For a list of size n.
  - To find the smallest element, the algorithm inspects all n items.
  - The next time through the loop, it inspects the remaining $n-1$ items.
- The total number of iterations is:

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{n(n+1)}{2}$$

- contains an n^2 term: the number of steps in the algorithm is proportional to the square of the size of the list: *quadratic* or n^2 algorithm.

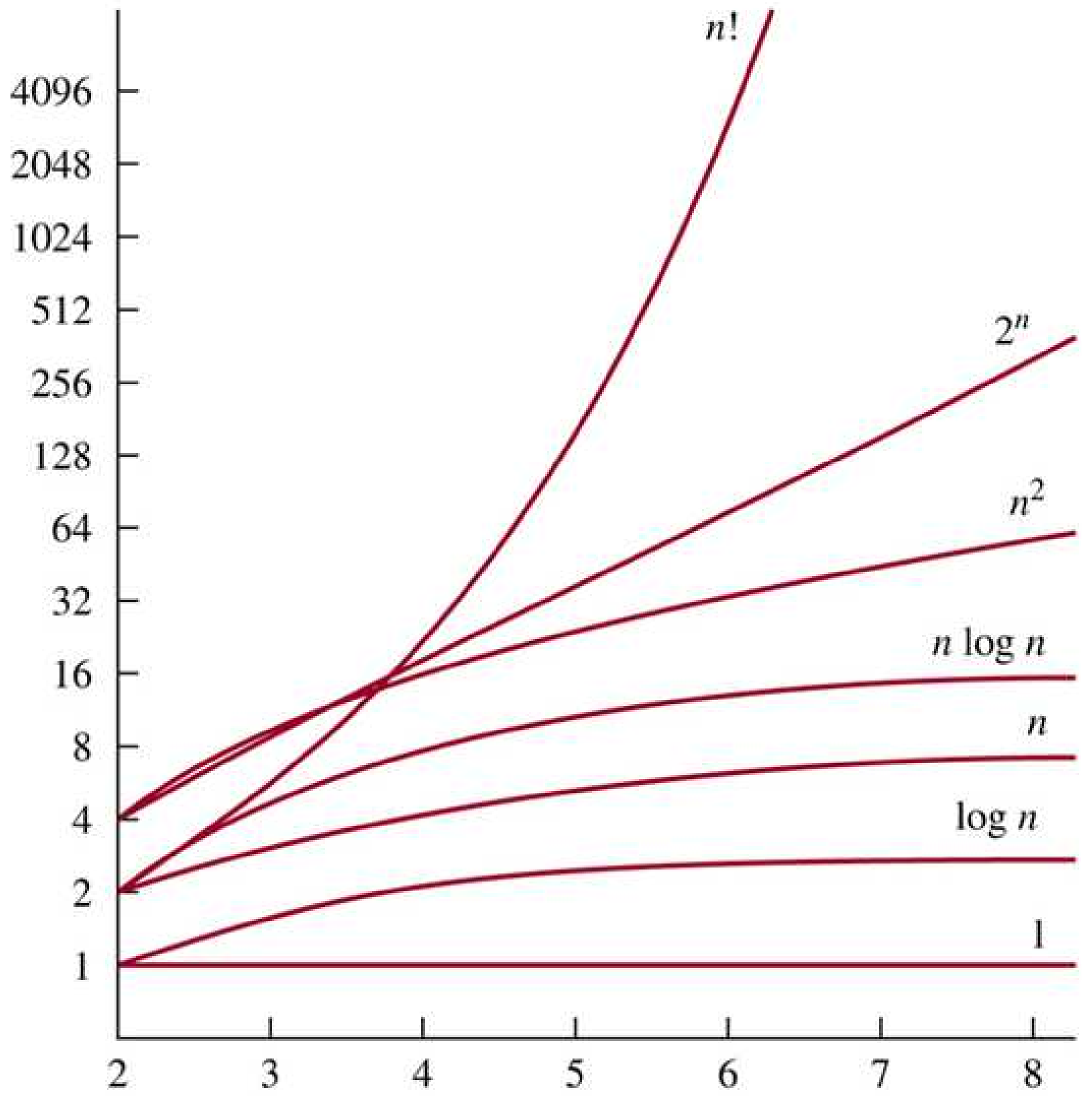
- *Merge Sort*
- n, $n/2$, $n/4$, $\cdots$, 1

  => $\log_2 n$ *levels*

  => total work required to sort n items: $n\log_2 n$.

  ($n\log n$ algorithm)

Comparing Algorithms



Some more recursive functions

1. Write a recursive function count_matches(some_list, value) that takes a list and a value and counts the number of elements in the list that are equal to the value. Save your function definition in a file called count_matches.py

Example:

```
>>> count_matches([0, 1, 0, 4, 2, 0], 0)
3
>>> count_matches(["a", "b", "c"], 1)
0
>>> count_matches([], "a")
0
```

Some hints: Be sure to review how a recursive function should break a list apart into the first element of the list and the rest of the list.

```python
def count_matches(some_list,value):
    # the following is the base case #
    if len(some_list) < 2:
        if len(some_list) == 0:
            return 0
        elif some_list[0] == value:
            return 1
        else:
            return 0
    return count_matches([some_list[0]],value) + count_matches(some_list[1:],value)
```

- 2. Write a recursive function double_each(some_list) that takes a list and returns a new list that has each element in the input list repeated twice.

The original input list must remain unchanged. For example, you may not assign new values to the original list, such some_list[i] = x.

Save your function definition in a file called double_each.py

- Example:

```
>>> nums = [1, 2, 3]
>>> double_each(nums)
[1, 1, 2, 2, 3, 3]
>>> nums
[1, 2, 3]
>>> double_each([])
[]
```

```
def double_each(some_list):
    if len(some_list) == 0:
        return []
    elif len(some_list) == 1:
        return [some_list[0]] * 2
    return double_each([some_list[0]]) + double_each(some_list[1:])
```

- 3. Write a recursive function sums_to(nums, k) that takes a list of integers and returns True if the sum of all the elements in the list is equal to k and returns False otherwise. Save your function definition in a file called sums_to.py
- Example:
- `>>> nums = [1, 2, 3]`
- `>>> sums_to(nums, 6)`
- `True`
- `>>> sums_to(nums, 5)`
- `False`
- `>>> sums_to([], 1)`
- `False`

- Note: You are *not* allowed to use any python sum function in any form, nor sum the list and then at the end check whether it equals k. In addition, you must write sums_to as a single recursive function: That is, you may not use a main function and a recursive helper function, (Obeying this restriction gives you the opportunity to write a very simple function definition.)

Think: What is the base case and for what value of k would sums_to return True? In the recursive call, what input to sums_to would lead towards the base case?

```
def sums_to(nums,k):
    if len(nums) == 0:
        if k == 0:
            return True
        else:
            return False
    return sums_to(nums[1:],k-nums[0])
```

4. Write a recursive function is_reverse(string1, string2) that takes two strings and returns True if string1 is the same string as string2 except in the reverse order. It returns False otherwise. Save your function definition in a file called is_reverse.py. To obtain each character in a string, you can use indexing just as you would with a list. You can also get substrings using slicing.

For example,

```
>>> s = "tacos"          >>> s[1:]        >>> s[-1]
>>> s[0]                  'acos'           'taco'
't'
>>> s[-1]                >>> s[0:-1]
s'                        'taco'
```

Note: You are *not* allowed to compare the lengths of the two input strings with each other. You may only check for empty strings. Again, this restriction leads to a simple function definition. (Remember that you have the Boolean operators and and or available. *Do not use the reverse function in any form. *

Example:

```
>>>
is_reverse("abc","cba")      >>>is_reverse("abc","abc" )     >>>is_reverse("abc","dcba")
True                          False                           False
>>> is_reverse("abc","cb")   >>> is_reverse("","")
False                         True
```

```
def is_reverse(string1,string2):
    if len(string1) != len(string2):
        return False
    if len(string1) == 0 and len(string2) == 0:
        return True
    if string1[0] == string2[-1]:
        return is_reverse(string1[1:], string2[:len(string2)-1])
    else:
        return False
```