CSc 120
Introduction to Computer Programming II

Adapted from slides by Dr. Saumya Debray

12: Recursion
Volunteers!

• Volunteers needed in front of the class for an activity:
  • Must be able to do simple addition
    (ex: 5 + 10 = 15)
  • Must be able to speak
How much money is in this cup?

If the cup is empty:

   Answer “zero” to the person on your right.
   (your_answer = 0)

else if the cup is not empty:

   Take out a coin. Pass the cup to the person on your left and ask them:

   “How much money is in this cup?”

   When they answer, tell the person on your right the sum of your coin and their answer

   (your_answer = your_coin + their_answer)
Challenge

Can we express that procedure in Python?

Idea:

```python
>>> cup = [5, 10, 1, 5]
>>> how_much_money(cup)
21
```

Write Python code that models the cup passing example.
function: how_much_money

def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])

Usage:
>>> how_much_money([5, 10, 1, 5])
21
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:]),

how_much_money([5,10,1,5])
 | how_much_money([10,1,5])
 |  | how_much_money([1,5])
 |  |  | how_much_money([5])
 |  |  |  | how_much_money([])
 |  |  |  | how_much_money returned 0
 |  |  | how_much_money returned 5
 |  | how_much_money returned 6
 | how_much_money returned 16
how_much_money returned 21
Manual expansion of calls

```python
>>> 5 + how_much_money([10, 1, 5])
21
```

```python
>>> 5 + (10 + how_much_money([1,5]))
21
```

```python
>>> 5 + (10 + (1 + how_much_money([5])))
21
```

```python
>>> 5 + (10 + (1 + (5 + how_much_money([]))))
21
```
Recursion

A function is *recursive* if it calls itself:

```python
def how_much_money( ... ):
    ...
    how_much_money( ... ) ← recursive call
    ...
```

The call to itself is a *recursive call*
Recursion

A solution to a problem is *recursive* when it is constructed from the solution to a simpler version of the same problem.
Recursion

A solution to a problem is *recursive* when it is constructed from the solution to a simpler version of the same problem.

```python
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])
```

*simpler version of the problem*
Recursion

• Recursive functions have two kinds of cases:
  – *base case(s)*:
    o *do some trivial computation and return the result*
  – *recursive case(s)*:
    o *the expression of the problem is a simpler case of the same problem*
    o *the input is reduced or the size of the problem is reduced*

• *Note*: the recursive call is given a smaller problem to work on
  – e.g., it makes progress towards the base case
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])

The convention is to handle the base case(s) first.
Problem 1

Write a recursive function to count the number of coins in a cup. *The len function is not allowed.*

Usage:

```python
>>> count_coins([10, 5, 1, 5])
4
```
def count_coins(cup):
    if cup == []:
        return 0
    else:
        return 1 + count_coins(cup[1:])
def count_coins(cup):
    if cup == []:
        return 0
    else:
        return 1 + count_coins(cup[1:])

base case: cup == []
recursive case: cup != []

recursive call is on a smaller problem
Problem 2

Write a recursive function to count the number of nickels in a cup.

Usage:

```python
>>> count_nickels([10, 5, 1, 5, 1])
2
```
def count_nickels(cup):
    if cup == []:
        return 0
    else:
        if cup[0] == 5:
            return 1 + count_nickels(cup[1:])
        else:
            return count_nickels(cup[1:])
Problem 3

Write a recursive function that returns the total length of all the elements of a list of lists (a 2-d list).

Usage:

```python
>>> total_length([[1,2], [8,2,3,4], [2,2,2]])
9
```
Solution

def total_length(alist):
    if alist == []:
        return 0
    else:
        return len(alist[0]) + total_length(alist[1:]),

base case:  
alist == []

recursive case:  
alist != []

recursive call is on a smaller problem
Problem 4

Recall that factorial is defined by the equation:

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1 \]

and

\[ 0! = 1 \]

Write a recursive function that computes the factorial of a number.

Usage:

```
>>> fact(4)
24
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
Write a recursive function `sumlist(L)` that returns the sum of the elements in L.

Usage:

```python
>>> sumlist([2,4,6,10])
22
```
Write a recursive function `string_len(s)` that returns the length of string `s`.

Usage:

```python
>>> >>> string_len("I wandered lonely as a cloud")
28
>>> >>>
```
Write a recursive function `joint_all()` that takes a list `alist` and returns a string consisting of every element of `alist` concatenated together.

Usage:

```python
>>> joint_all([1,2,3,4,5])
'12345'

>>> joint_all(['aa','bb'])
'aabb'
```
Write a recursive function that implements join.

That is, write a function \texttt{join(alist, sep)} that takes a list \texttt{alist} and creates a string consisting of every element of \texttt{alist} separated by the string \texttt{sep}.

Usage:

\[
>>> \texttt{join(['aa', 'bb', 'cc'], '---')}
\]

'a-a-b-b-c-c'
the runtime stack
How recursion works

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

>>> fact(4)
24
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n+1)

>>> fact(4)
24
```

We need the value of \( n \) both before and after the recursive call.

\[ \therefore \text{its value has to be saved somewhere} \]

“somewhere” ≡ “stack frame”
How recursion works

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

>>> fact(4)
24
```

Python's runtime system* maintains a stack:

- push a "frame" when a function is called
- pop the frame when the function returns

* "runtime system" = the code that Python executes to make everything work at runtime

"frame" or "stack frame": a data structure that keeps track of variables in the function body, and their values, between the call to the function and its return
How recursion works

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

>>> fact(4)
24
```

Python's runtime system* maintains a stack:

- push a "frame" when a function is called
- pop the frame when the function returns

* "runtime system" = the code that Python executes to make everything work at runtime
How recursion works

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

>>> fact(4)
24
```
How recursion works

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

>>> fact(4)
24
How recursion works

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

>>> fact(4)
24
```
How recursion works

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

>>> fact(4)
24
```

value of n  value from recursive call  return value

stack frame for fact()
How recursion works

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

>>> fact(4)
24
```

```
stack	frame	for	fact()

value	to
recursive	call

return
tvalue

stack
top

value	of	

stack

runtime
stack

fact(4)
4

fact(3)
3

fact(2)
2

fact(1)
1
```
How recursion works

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

>>> fact(4)
24
```

value of n  value from recursive call  return value

stack frame for fact()  

stack top  

fact(0)  0

fact(1)  1

fact(2)  2

fact(3)  3

Fact(4)  4

runtime stack
How recursion works

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

>>> fact(4)
24
```

<table>
<thead>
<tr>
<th></th>
<th>fact(0)</th>
<th>fact(1)</th>
<th>fact(2)</th>
<th>fact(3)</th>
<th>fact(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stack frame for fact()  
value of n  
value from recursive call  
return value  

stack top
How recursion works

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

>>> fact(4)
24
```

The diagram illustrates the stack frame for `fact()` and shows the recursive calls and return values.
How recursion works

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

>>> fact(4)
24
```

[Diagram showing stack frames and recursive calls]
How recursion works

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

>>> fact(4)
24
```

The stack frame for `fact()`

<table>
<thead>
<tr>
<th>stack frame for fact()</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of n</td>
</tr>
<tr>
<td>value from recursive call</td>
</tr>
<tr>
<td>return value</td>
</tr>
</tbody>
</table>

Runtime stack:

- `fact(0)`: 0 1
- `fact(1)`: 1 1 1
- `fact(2)`: 2 1 2
- `fact(3)`: 3 2 6
- `fact(4)`: 4
How recursion works

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

```python
>>> fact(4)
24
```
How recursion works

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

>>> fact(4)
24
```

Value of n, value from recursive call, return value, runtime stack, stack frame for `fact()`.
The runtime stack

• The use of a runtime stack containing stack frames is not specific to recursion
  – all function and method invocations use this mechanism
  – not just in Python, but other languages as well (Java, C, C++, ...)

43
Problem 5

Write a recursive function to print the numbers from 1 through n, one per line.

Usage:

```python
>>> print_n(6)
1
1
2
2
3
3
4
4
5
5
6
```
def print_n(n):
    if n == 0:
        return
    else:
        print_n(n-1)
        print(n)
Recursion How to

To write a recursive function, figure out:

*What values are involved in the computation?*
  – these will be the arguments to the recursive function

*Base case(s)*
  – when does the recursion stop?
  – what is the simple value or data that can be computed and returned?

*Recursive case(s)*
  – what is the "smaller problem" to pass to the recursive call?
  – what does a single round of computation involve?
Recursion how to: sumlist

```python
def sumlist(L):
    if len(L) = 0:
        return 0
    else:
        return L[0] + sumlist(L[1:])
```

- **Base case**: `if len(L) = 0:
  return 0`
- **Recursive case**: `else:
  return L[0] + sumlist(L[1:]);`

One round of computation adds L[0] and the result of the recursive call.

Argument to recursive call is "rest of the list" after L[0] (recurses on a smaller problem).
Write a function `even_positions(s)` that returns a string consisting of all the even-numbered positions of the Python string `s`.

Usage:

```python
>>> even_positions("abc")
'ac'

>>> even_positions("abcdef")
'ace'
```
Recursion

• The recursive case can be written many ways
• Consider summing the elements in a list

```python
def sumlist(L):
    if len(L) = 0:
        return 0
    else:
        return L[0] + sumlist(L[1:])
```

• Options:
  o Recurse on L[:-1] and then add in L[-1]
  o Recurse on each half and add the results
Write sumlist() in the two ways described in the exercise.
Versions of sumlist

<table>
<thead>
<tr>
<th>Version 1</th>
</tr>
</thead>
</table>

```python
def sumlist(L):
    if len(L) == 0:
        return 0
    else:
        return L[0] + sumlist(L[1:])
```

- **Base case**: `if len(L) = 0:
  return 0`

- **Recursive case**: `return L[0] + sumlist(L[1:])`

One round of computation adds `L[0]` and the result of the recursive call.

Argument to recursive call is "rest of the list" after `L[0]`
(recurses on a smaller problem)
Versions of sumlist

Version 2
(variation on version 1)

def sumlist(L):
    n = len(L)
    if n == 0:
        return 0
    else:
        return sumlist(L[:-1]) + L[-1]

argument to recursive call is "rest of the list" up to the last element
(recurses on a smaller problem)

add the last element of L
## Versions of sumlist

<table>
<thead>
<tr>
<th>Version 2</th>
<th>Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(variation on version 1)</td>
<td>(&quot;smaller&quot; need not be by just 1)</td>
</tr>
</tbody>
</table>

**def** sumlist(L):

```python
    n = len(L)
    if n == 0:
        return 0
    else:
        return sumlist(L[:-1]) + L[-1]
```

**def** sumlist(L):

```python
    if len(L) == 0:
        return 0
    elif len(L) == 1:
        return L[0]
    else:
        return sumlist(L[:len(L)//2]) + sumlist(L[len(L)//2:])
```

- **better for parallel execution**
- Argument to each recursive call is half of the current list (recurses on a smaller problem)
def sumlist(L):
    if len(L) == 0:
        return 0
    elif len(L) == 1:
        return L[0]
    else:
        return sumlist(L[:len(L)//2]) + sumlist(L[len(L)//2:])
recursive sumlist

input list

split into two halves

add the halves (recursively)

return the sum of the sums
sumlist: example

sumlist([1,3,4,6,8])
sumlist: example

sumlist([1,3])

sumlist([1,3,4,6,8])

sumlist([4,6,8])
sumlist: example

sumlist([1,3,4,6,8])

sumlist([1,3])

sumlist([1])  sumlist([3])

sumlist([4,6,8])
sumlist: example

sumlist([1,3,4,6,8])

- sumlist([1,3])
- sumlist([1]) \rightarrow 1
- sumlist([3]) \rightarrow 3

sumlist([4,6,8])

- sumlist([4])
- sumlist([6])
- sumlist([8])
sumlist: example

sumlist([1,3,4,6,8])

sumlist([1,3])

sumlist([1])
1

sumlist([3])
3

sumlist([4,6,8])

4
sumlist: example

sumlist([1,3,4,6,8])

sumlist([1,3])

sumlist([1])  sumlist([3])
1  3

sumlist([4,6,8])

sumlist([4])

sumlist([6,8])

sumlist([1,3,4,6,8])

sumlist([4,6,8])
sumlist: example

sumlist([1,3,4,6,8])

- sumlist([1,3,4,6,8])
- sumlist([4,6,8])
- sumlist([6,8])

sumlist([1,3])

- sumlist([1,3])
- sumlist([1])
- sumlist([3])

sumlist([4])

- sumlist([4])
- sumlist([4,6,8])

1 3 4 4 62
sumlist: example

\[
\begin{align*}
\text{sumlist}([1]) & \rightarrow 1 \\
\text{sumlist}([3]) & \rightarrow 3 \\
\text{sumlist}([4]) & \rightarrow 4 \\
\text{sumlist}([6]) & \rightarrow 4 \\
\text{sumlist}([8]) & \rightarrow 4
\end{align*}
\]

\[
\begin{align*}
\text{sumlist}([1,3,4,6,8]) & \rightarrow \text{sumlist}([1,3]) \\
\text{sumlist}([4,6,8]) & \rightarrow \text{sumlist}([4,6]) \\
\text{sumlist}([6,8]) & \rightarrow \text{sumlist}([6])
\end{align*}
\]
sumlist: example

sumlist([1,3,4,6,8])

sumlist([1,3])

sumlist([1])  sumlist([3])

1  3

4

sumlist([4,6,8])

sumlist([4])

sumlist([4])

4

sumlist([6,8])

sumlist([6])  sumlist([8])

6  8

4

64
sumlist: example

\[
\text{sumlist([1,3,4,6,8])} \\
\text{sumlist([1,3])} \quad \text{sumlist([4,6,8])} \\
\text{sumlist([1])} \quad \text{sumlist([3])} \quad \text{sumlist([4])} \quad \text{sumlist([6,8])} \\
\text{sumlist([1])} \quad \text{sumlist([3])} \quad \text{sumlist([4])} \quad \text{sumlist([6,8])} \\
1 \quad 3 \quad 4 \quad 6 \\
4 \quad 3 \quad 4 \quad 14 \\
6 \quad 6 \quad 8 \\
\]

1 + 3 + 4 + 6 + 8 = 14
sumlist: example

sumlist([1,3,4,6,8])
  sumlist([1,3])
    sumlist([1])  sumlist([3])
     1            3
  sumlist([4,6,8])
    sumlist([4])
     4
    sumlist([6])
     6
  sumlist([6,8])
    sumlist([6])
     6
    sumlist([8])
     8

14
18
sumlist: example

sumlist([1,3,4,6,8])

sumlist([1,3])
sumlist([1])
sumlist([3])

sumlist([4,6,8])
sumlist([4])
sumlist([6])
sumlist([8])

1 3 4
4

22
Recursion: flow of values

**Version 1**

```python
def sumlist1(L):
    if len(L) = 0:
        return 0
    else:
        return L[0] + sumlist1(L[1:])
```

The process of summing the list `[11, 22, 33]` using the recursive function `sumlist1`: 66 + 55 + 33 + 0 = 154.
recursion: example
binary search
Searching an unsorted list

- Problem: Given an unsorted list $L$ and a value $a$, determine whether or not $a$ is in $L$. 

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
3 & 81 & 2 & 14 & 64 & 12 & 8 & 29 & 14 & 5 \\
\end{array}
\]
Searching an unsorted list

- Problem: Given an unsorted list \( L \) and a value \( a \), determine whether or not \( a \) is in \( L \).

- Linear search: sequentially look at (possibly) all values in the list.
Searching a sorted list

• Problem: Given a sorted list $L$ and a value $a$, determine whether or not $a$ is in $L$. 

$$
\begin{array}{c}
L \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
$$

$$
\begin{array}{c}
a \quad 29 \\
\end{array}
$$
Searching a sorted list

• Problem: Given a **sorted** list $L$ and a value $a$, determine whether or not $a$ is in $L$.

$$L = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Pick a value $i$ in range(len($L$))

Say, $i = 4$

$a \geq L[4]$?

Q: Can $L[3]$ be $a$?
Searching a sorted list

- Problem: Given a **sorted** list $L$ and a value $a$, determine whether or not $a$ is in $L$.

L sorted and $a > L[4]$ means $a$ cannot be any of these elements.

pick a value $i$ in range(len($L$))
say, $i = 4$

$a > L[4]$
Binary search: recursive solution

binary search - find an item in a sorted list
  if the list is empty
    the item is not found (return False)
  look at the middle of the list
  if we found the item
    then done (return True)
  else
    if the item is less than the middle
      search in the lower half of the list
    else
      search in the upper half of the list
EXERCISE

Write a recursive function `bin_search(alist, item)` that searches for `item` in `alist` and returns `True` if found and `False` otherwise.

Usage:

```python
>>> bin_search([4, 25, 28, 33, 47, 54, 65, 83], 65)
True
```

```python
>>> 
```
def bin_search(L, item):
    if L == []:
        return False
    mid = len(L) // 2
    if L[mid] == item:
        return True
    if item < L[mid]:
        return bin_search(L[0:mid], item)
    else:
        return bin_search(L[mid+1:], item)
Binary search: complexity

- The size of the search area is halved at each round of repetition

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>Approx. number of items left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n/2</td>
</tr>
<tr>
<td>2</td>
<td>n/4</td>
</tr>
<tr>
<td>3</td>
<td>n/8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>n/2^i</td>
</tr>
</tbody>
</table>

- The number of comparisons until we are done is i, where n/2^i = 1
  solving for i gives i = log n
- total no. of rounds of recursion = log_2(n)
Binary search: complexity

• The size of the search area is halved at each round of repetition (recursion)
  – total no. of rounds of recursion = \( \log_2(n) \)
  or the number of comparisons is \( \log_2(n) \)

• However, on each round of repetition, the work done is *not* a fixed amount due to slicing
  – slicing is \( O(n) \)

• Fix that by computing the indices and passing them as parameters.
Binary search: no slicing

def bin_search(L, item, lo, hi):
    if lo > hi:
        return False
    if lo == hi:
        return L[lo] == item
    mid = (lo+hi)//2
    if item <= L[mid]:
        return bin_search(L, item, lo, mid)
    else:
        return bin_search(L, item, mid+1, hi)
Binary search: complexity

• The size of the search area is halved at each round of recursion
  – total no. of rounds of recursion = \( \log_2(n) \)

• On each recursive step, the work done is a fixed amount
  – \( O(1) \)

\[ \therefore \text{Overall complexity: } O(\log n) \]
recursion: example
Example: merging two sorted lists

**Problem:** Given two sorted lists $L_1$ and $L_2$, merge them into a single sorted list

**Example:** $L_1 = [11, 22, 33], L_2 = [5, 10, 15]$

- Output: $[5, 10, 11, 15, 22, 33]$
  - can't just concatenate the lists
  - can't alternate between the lists
Merging: values involved

**Problem**: Given two sorted lists L1 and L2, merge them into a single sorted list

1. Values involved in the recursion: ???
Merging: values involved

**Problem**: Given two sorted lists \( L_1 \) and \( L_2 \), merge them into a single sorted list

1. Values involved in the computation in each (recursive) call: \( L_1 \) and \( L_2 \)

   So the recursive function will look something like

   ```python
   def merge(L1, L2):  # may need another argument
       ...
   ```
Merging: repetition

**Problem:** Given two sorted lists \( L1 \) and \( L2 \), merge them into a single sorted list.

2. What does the computation involve in each call?
Problem: Given two sorted lists $L_1$ and $L_2$, merge them into a single sorted list

2. What does the computation involve in each call?

move the smaller value into the merged list
Merging: repetition

**Problem:** Given two sorted lists $L_1$ and $L_2$, merge them into a single sorted list

2. How does the problem (or data) get smaller?

move the smaller value into the merged list
repeat on the remaining list values
Merging: base case

**Problem:** Given two sorted lists $L_1$ and $L_2$, merge them into a single sorted list

3. When can’t we make the data smaller?
Merging: base case

**Problem:** Given two sorted lists \( L_1 \) and \( L_2 \), merge them into a single sorted list

3. When can’t we make the data smaller?
   – when either \( L_1 \) or \( L_2 \) is empty

In this case, concatenate the other list into the merged list
Merging: base case

The code looks something like:

```python
def merge(L1, L2, merged):
    if L1 == []:
        return merged + L2
    elif L2 == []:
        return merged + L1
    else:
        ....
```
The code looks something like:

```python
def merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        ....
```
Merging: recursive case

**Problem:** Given two sorted lists $L_1$ and $L_2$, merge them into a single sorted list

4. What is "the rest of the computation"?
   - "repeat on the remaining list values"
Merging: recursive case

if L1[0] < L2[0]:
    new_merged = merged + [ L1[0] ]
    new_L1 = L1[1: ]
    new_L2 = L2
else:
    new_merged = merged + [ L2[0] ]
    new_L1 = L1
    new_L2 = L2[1: ]
return merge(new_L1, new_L2, new_merged)
def merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [L1[0]]
            new_L1 = L1[1:]
            new_L2 = L2
        else:
            new_merged = merged + [L2[0]]
            new_L1 = L1
            new_L2 = L2[1:]
    return merge(new_L1, new_L2, new_merged)
```python
>>> def merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [L1[0]]
            new_L1 = L1[1:]
            new_L2 = L2
        else:
            new_merged = merged + [L2[0]]
            new_L1 = L1
            new_L2 = L2[1:]
        return merge(new_L1, new_L2, new_merged)

>>> merge([11,22,33],[5,10,15,20,25],[])
[5, 10, 11, 15, 20, 22, 25, 33]
```
recursion: flow of values
Recursion: flow of values

def merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [L1[0]]
            new_L1 = L1[1:]
            new_L2 = L2
        else:
            new_merged = merged + [L2[0]]
            new_L1 = L1
            new_L2 = L2[1:]
        return merge(new_L1, new_L2, new_merged)

values are computed and passed down as arguments into the recursive call
Recursion: flow of values

```python
def merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [L1[0]]
            new_L1 = L1[1:]
            new_L2 = L2
        else:
            new_merged = merged + [L2[0]]
            new_L1 = L1
            new_L2 = L2[1:]
        return merge(new_L1, new_L2, new_merged)
```

the computation of each round of repetition takes place as values are passed up as return values
recursion: application
merge sort
Sorting

• Problem: Given a list L, sort the elements of L into a list sortedL

• Important problem
  – arises in a wide variety of situations
  – many different algorithms, with different assumptions and characteristics
  – we will consider just one algorithm
Algorithm: mergesort

input list

split into two halves

sort the halves recursively

merge the sorted lists

Divide and conquer algorithm
Divide and Conquer

• An algorithm paradigm based on multi-branched recursion

  – Recursively break the problem down into two or more sub-problems (until they are trivial to solve)

  – Combine the solutions of the sub-problems to give the solution to the original problem
Mergesort

• Base case: \( \text{len}(L) \leq 1 \)
  – no further halving possible

• Recursive case:
  – setting up the next round of computation: splitting the list
  – smaller problem to recurse on: a list of half the size

• Each round of computation: merging the sorted lists
  – has to be done after the recursive call
def msort(L):
    if len(L) <= 1:
        return L
    else:
        split_pt = len(L)//2
        L1 = L[:split_pt]
        L2 = L[split_pt:]
        sortedL1 = msort(L1)
        sortedL2 = msort(L2)
        return merge(sortedL1, sortedL2, [])
Mergesort: example

msort([1, 3, 2, 5, 4])
Mergesort: example

\[
\text{msort([1, 3, 2, 5, 4])}
\]

\[
\text{msort([1, 3])}
\]

\[
\text{msort([2, 5, 4])}
\]
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])  msort([3])

msort([2, 5, 4])
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])

[1]

msort([3])

[3]

msort([2, 5, 4])

[1]

[3]
Mergesort: example

\[\text{msort}([1, 3, 2, 5, 4])\]

\[\text{msort}([1, 3, 2, 5, 4]) \rightarrow \text{msort}([1, 3]) \rightarrow \text{msort}([2, 5, 4]) \rightarrow \text{msort}([1]) \rightarrow \text{msort}([3]) \rightarrow \text{merge}([1], [3])\]
Mergesort: example

mergesort([1, 3, 2, 5, 4])

mergesort([1, 3])

mergesort([1])

mergesort([3])

mergesort([2, 5, 4])

merge([1], [3])

[1, 3]
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])

msort([3])

merge([1], [3])

[1, 3]

[1]

[3]
Mergesort: example

\[
\text{msort([1, 3, 2, 5, 4])}
\]

\[
\text{msort([1, 3])}
\]

\[
\text{msort([2, 5, 4])}
\]

\[
\text{msort([1])}
\]

\[
\text{msort([3])}
\]

\[
\text{msort([2])}
\]

\[
\text{msort([5, 4])}
\]

\[
\text{merge([1], [3])}
\]

\[
[1, 3]
\]
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])

msort([3])

msort([2])

msort([5, 4])

merge([1, 3])

[1, 3]
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])

msort([3])

merge([1], [3])

[1]

[3]

[1, 3]

msort([2])

msort([5, 4])

msort([5])

msort([4])

[2]
Mergesort: example
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])  msort([2, 5, 4])

msort([1])  msort([3])  msort([5, 4])


merge([1], [3])  merge([5], [4])

[1, 3]  [5, 4]
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([1])

[1]

msort([3])

[3]

merge([1], [3])

[1, 3]

msort([2])

[2]

msort([5, 4])

msort([5])

[5]

msort([4])

[4]

merge([5], [4])

[4, 5]
Mergesort: example

msort([1, 3, 2, 5, 4])

dsplits into:

msort([1, 3])  msort([2, 5, 4])

and:

msort([1])  msort([3])

and:

msort([2])  msort([5, 4])

and:

merges:

merge([1], [3])

merge([5], [4])

merge([2], [4, 5])
Mergesort: example

```
msort([1, 3, 2, 5, 4])
```

```
msort([1, 3])
```  
```
msort([2, 5, 4])
```  
```
msort([1])
```  
```
msort([3])
```  
```
msort([2])
```  
```
msort([5, 4])
```  
```
merge([1], [3])
```  
```
merge([5], [4])
```  
```
merge([2], [4, 5])
```  
```
merge([2], [4, 5])
```
Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3])
msort([2, 5, 4])

msort([1])
msort([3])
msort([2])
msort([5, 4])

merge([2], [4, 5])

merge([1, 3], [2, 4, 5])
Mergesort: example

\[
\begin{align*}
\text{msort}([1, 3, 2, 5, 4]) & \rightarrow \text{msort}([2, 5, 4]) \\
\text{msort}([1, 3]) & \rightarrow \text{msort}([5, 4]) \\
\text{msort}([1]) & \rightarrow \text{msort}([4]) \\
\text{msort}([3]) & \rightarrow \text{msort}([2]) \\
\text{merge}([1], [3]) & \rightarrow [1, 3] \\
\text{merge}([5], [4]) & \rightarrow [4, 5] \\
\text{merge}([2], [4, 5]) & \rightarrow [2, 4, 5] \\
\text{merge}([1, 3], [2, 4, 5]) & \rightarrow [1, 2, 3, 4, 5]
\end{align*}
\]
Mergesort: complexity

Cost = \( \text{cost per round of repetition} \times \text{no. of rounds of repetition} \)

merging the sorted lists is \( O(n) \) *if slicing is removed from merge

*if slicing is removed from merge
Mergesort: complexity

\[
[a_0, a_1, \ldots, a_{n-1}]
\]

\[
[a_0, \ldots, a_{n/2}] \quad [a_{n/2+1}, \ldots, a_{n-1}]
\]

\[
\vdots
\]

\[
[a_0, a_1, a_2, a_3] \quad \ldots \quad [a_{n-4}, a_{n-3}, a_{n-2}, a_{n-1}]
\]

\[
[a_0] \quad [a_1] \quad [a_2] \quad [a_3] \quad \ldots \quad [a_{n-4}] \quad [a_{n-3}] \quad [a_{n-2}] \quad [a_{n-1}]
\]

\[2^k = n\]

Counts:
- 1 round: \(2^1 = 2\)
- 2 rounds: \(2^2 = 4\)
- \(k\) rounds: \(2^k = n\)
Mergesort: complexity

• No. of rounds of recursion:
  – if we start with a list of size n and have k rounds of recursion, then $2^k = n$
    $\therefore \log_2(2^k) = \log_2(n)$
    $\therefore k = \log_2(n)$

• Complexity of each round of recursion: $O(n)$

$\Rightarrow$ Worst-case complexity of mergesort: $O(n \log n)$*

*if slicing is removed from msort
recursion: summary
Recursion: summary

• Recursion offers a way to express repetitive computations cleanly and succinctly

• How to:
  – what are the values used in the recursive call?
  – base case: when does the recursion stop?
  – recursive case:
    o what does a single round of computation involve?
    o what is the “smaller problem” to recurse on?

• Recursion is an essential component of every good computer scientist’s toolkit