CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

13: Recursion
Volunteers!

• Volunteers needed in front of the class for an activity:
  • Must be able to do simple addition
    (ex: $6 + 10 = 16$)
  • Must be able to speak
How much money is in this cup?

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)

Else: # the cup is empty:

Answer “zero” to the person on your right.

(your_answer = 0)
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:

```python
>>> how_much_money([5, 10, 1, 5])  
21
```
Calls and returns

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

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

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

>>> 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 value
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 to print the numbers from 1 through n, one per line.

Usage:

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

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 value
Problem 5

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:

```python
>>> fact(4)
24
```
Solution

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

base case:
- n == 0

recursive case:
- n != 0

recursive call is on a smaller value
Write a recursive function that implements join.

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

Usage:

```python
>>> join(['aa', 'bb', 'cc'], '--')
'aa-bb-cc'
```
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.

∴ 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

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

```python
>>> 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
```
How recursion works

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

>>> fact(4)
24
```

**runtime stack**

- fact(4)
  - fact(3)
    - fact(2)
      - Base case: return 1
  - Value of n and return value
  - Value from recursive call

**stack**: 2

**stack**: 3

**stack**: 4

**stack**: top
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 of `n`
- Value from recursive call
- Return value

Runtime stack:

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

Stack top

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

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

>>> fact(4)
24
```

![Stack frame diagram](image)

- **Value of n**: The value of `n` passed to the recursive call.
- **Value from recursive call**: The value returned from the recursive call.
- **Return value**: The final value returned from the recursive function.

![Runtime stack](image)

- Stack frame for `fact()`
- Stack top
- `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
```

Stack frame for `fact()`

- Value of `n`
- Value from recursive call
- Return value

Runtime stack

- `fact(4)`
- `fact(3)`
- `fact(2)`
- `fact(1)`
- `fact(0)`

Stack top
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()`

<table>
<thead>
<tr>
<th>fact(0)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fact(2)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>fact(3)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>fact(4)</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Runtime stack

Value of `n`, value from recursive call, return value
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()**: The stack frame contains the values of `n`, `fact(n-1)` and the return value.

- **Runtime stack**: The values are computed as follows:
  - `fact(0)`: Base case, returns 1.
  - `fact(1)`
  - `fact(2)`
  - `fact(3)`
  - `fact(4)`

- **Value computation**: Each recursive call computes the factorial of the next lower value until the base case is reached.

- **Execution order**: The computation proceeds from `fact(4)` to `fact(0)`.

- **Return value**: The final value is computed as `4 * 6 = 24`.

- **Stack top**: The stack is cleared after each recursive call, ensuring the correct order of execution.
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++, ...)
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 the list `L` is empty, return 0.
- **Recursive case**: return the first element of the list `L[0]` plus the result of the recursive call on the rest of the list `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).
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:])
```
Versions of sumlist

Version 1

```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)

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

- \[ n = \text{len}(L) \]
  - if \( n == 0 \):
    - return 0
  - else:
    - return \( \text{sumlist}(L[:n-1]) + L[n-1] \)

**Def sumlist(L):**

- \[ \text{if } \text{len}(L) = 0: \]
  - return 0
- \[ \text{elif } \text{len}(L) == 1: \]
  - return \( L[0] \)
- \[ \text{else:} \]
  - return \( \text{sumlist}(L[:\text{len}(L)//2]) + \text{sumlist}(L[\text{len}(L)//2:]] \)

---

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

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

\[
\text{sumlist}([1]) \rightarrow 1 \rightarrow \text{sumlist}([1,3]) \\
\text{sumlist}([3]) \rightarrow 3 \rightarrow \text{sumlist}([4,6,8])
\]
sumlist: example

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

1
1
4
4
3
3

sumlist: example

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

sumlist([3])
  3

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

1 3 4
sumlist: example

sumlist([1,3])
  sumlist([1])  sumlist([3])
  1             3
  ↓             ↓
  4             4

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

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

1

57
sumlist: example

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

sumlist([1,3])

sumlist([1])

1

sumlist([3])

3

sumlist([4])

4

sumlist([4,6,8])

sumlist([6,8])

sumlist([6])

sumlist([8])
sumlist: example

```
sumlist([1,3,4,6,8])
  └── sumlist([4,6,8])
      └── sumlist([6,8])
          └── sumlist([8])
  ┌── sumlist([1,3,4,6,8])
      └── sumlist([1,3,4,6,8])
          └── sumlist([1,3])
              └── sumlist([1])
                  └── 1
              └── sumlist([3])
                  └── 3
          ┌── sumlist([4,6,8])
          │    └── sumlist([4])
          │        └── 4
          │    ┌── sumlist([6,8])
          │    │    └── sumlist([6])
          │    │          └── 6
          │    ┌── sumlist([8])
          │    │    └── sumlist([8])
          │    │          └── 8
          │    └── sumlist([6])
          │        └── 4
          └── sumlist([3])
              └── 3
          ┌── sumlist([1,3,4,6,8])
          │    └── sumlist([1])
          │        └── 1
          │    ┌── sumlist([3])
          │    │    └── sumlist([3])
          │    │          └── 3
          │    └── sumlist([4])
          │        └── 4
          └── sumlist([6,8])
              └── 4
```

1 3 4
sumlist: example

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

- sumlist([1,3,4,6,8]) → 14
sumlist: example

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

sumlist([1,3])

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

1  3

4

sumlist([4,6,8])

sumlist([4])

4

sumlist([6,8])

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

6  8

14

18

61
sumlist: example

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

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

1 3 4 4 6 8 14 18 22 62
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 \( L \).
Searching an unsorted list

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

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

• 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 $L$. 

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
L
\]

\[
\begin{array}{cccccccccc}
& 29 \\
\end{array}
\]
Searching a sorted list

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

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

$L = \begin{array}{ccccccccccc}
& & & & & & & & & & \\
\big| & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}$

a \hspace{1cm} 29

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

$\uparrow$

a > 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 \( L \).

Consider an example list:

\[
L = [2, 5, 7, 8, 14, \ldots, 9]
\]

- Pick a value \( i \) in range(len(\( L \)))
- Say, \( i = 4 \)
- \( a > L[4] \)

This means \( a \) cannot be any of these elements.

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
2 & 5 & 7 & 8 & 14 & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]
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 the code
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 repetition (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 repetition = \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

```python
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 repetition (recursion)
  – total no. of rounds of repetition = $\log_2(n)$

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

∴ Overall complexity: $O(\log n)$
recursion: example
Example: merging two sorted lists

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

Example: \( L1 = [11, 22, 33], \) \( L2 = [5, 10, 15] \)

• 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 repetition: ???
Merging: values involved

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

1. Values involved in the computation in each (recursive) call: L1 and L2

   So the recursive function will look something like

   ```python
   def merge(L1, L2):
     ...
   ```
Merging: repetition

**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?
Merging: repetition

**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:
        ....
```
Merging: base case

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

**Version 1**

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

```
sumlist1([11, 22, 33])
  ^
  |  66
  |   
  |  sumlist1([22, 33])
  |     ^
  |     |  55
  |     |   
  |     |  sumlist1([33])
  |     |     ^
  |     |     |  33
  |     |     |   
  |     |     |  sumlist1([])
  |     |     |     ^
  |     |     |     |  0
```
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)
```

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

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

>>> merge([11, 22, 33], [5, 10, 15, 20, 25])
[5, 10, 11, 15, 20, 22, 25, 33]
>>> 
```
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
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

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

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

msort([1, 3])

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

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

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

\[
\text{msort}([1]) \quad \text{msort}([3])
\]
Mergesort: example

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

msort([1, 3]) →-msort([1]) ← [1]

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

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

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

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

\[
[1] \\
[3]
\]

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

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

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

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

\[
[1] \quad [3]
\]

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

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

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

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

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

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

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

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

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

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

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

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

[1]

msort([3])

[3]

merge([1], [3])

[1, 3]

msort([2, 5, 4])

msort([2])

[2]

msort([5, 4])

msort([5])

msort([4])
Mergesort: example

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

msort([1, 3])

msort([1])

msort([3])

msort([2])

msort([5, 4])

msort([2, 5, 4])

msort([5])

msort([4])

merge([1], [3])

[1, 3]
Mergesort: example

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

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

msort([1, 3])

msort([1])

[1]

merge([1], [3])

[1, 3]

msort([3])

[3]

msort([2])

[2]

msort([2, 5, 4])

msort([5, 4])

msort([5])

[5]

merge([5], [4])

[4, 5]

msort([4])

[4]

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

[2, 4, 5]
Mergesort: example

```
Mergesort: example

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

114
Mergesort: example

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

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

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

[1, 2, 3, 4, 5]
Mergesort: complexity

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

merging the sorted lists is $O(n)^*$

*if slicing is removed from merge
Mergesort: complexity

\[
\begin{align*}
[a_0, a_1, \ldots, a_{n-1}] \\
[a_0, \ldots, a_{n/2}] & \quad [a_{n/2+1}, \ldots, a_{n-1}] \\
\vdots
\end{align*}
\]

\[2^k = n\] rounds
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: \( O(n) \)

\[ \Rightarrow \text{Worst-case complexity of mergesort: } O(n \log n) \]
recursion: summary
Recursion: summary

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

• How to:
  – what are the values used in 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