Divide-and-Conquer via Recursion
The Divide-and-Conquer Paradigm

- This is an important technique to efficiently solving computational problems.

- It is commonly used for
  - Efficient sorting
  - Multiplying large numbers
  - Multiplying matrices
  - Finding a closest pair of points in Euclidean space

- It is usually implemented using recursion.
Binary Search
An example of Divide-and-Conquer

- In `binarySearch(L, k)`, we make one comparison: `k` compared to `L[mid]`.
- Based on the outcome of this comparison, we either stop, search the left half, or search the right half.
- Thus the problem of searching for `k` in `L` is reduced to search for `k` in `L[:mid]` or `L[mid+1:]`.
- Earlier we implemented binary search using iteration (i.e., a `while`-loop).
- We can easily implement it using recursion.
Binary Search: Recursive Implementation

- We will define a function: 
  ```python
def recursiveBinarySearch(L, k, left, right)
``` 

- This function looks for \( k \) in \( L[left:right+1] \). Returns \textbf{True} if found, \textbf{False} otherwise.

- To look for \( k \) in the entire list, we can call the function as: 
  ```python
print recursiveBinarySearch(L, k, 0, len(L)-1)
```
def recursiveBinarySearch(L, k, left, right):

    # Base case
    if left > right:
        return False

    # Recursive cases
    mid = (left + right)/2
    if k == L[mid]:
        return True
    if k < L[mid]:
        return recursiveBinarySearch(L, k, left, mid-1)
    if k > L[mid]:
        return recursiveBinarySearch(L, k, mid+1, right)
Wrapper Functions

- Sometimes for recursive functions we have to create extra parameters that control the recursion.
- However, the calling code should not be expected to deal with these extra parameters.
- So such recursive functions usually come with “wrapper functions” that hide the extra parameters.

```python
# Wrapper function for binary search
def binarySearch(L, k):
    recursiveBinarySearch(L, k, 0, len(L)-1)
```
Divide-and-Conquer

- **Divide step:** Partition the problem into sub-problems.

- **Conquer step:** Solve each sub-problem separately.

- **Combine step:** Combine the solutions of the sub-problems into a solution of the original problem.
Algorithms such as selection sort, insertion sort, bubble sort, etc. are all extremely slow for large lists.

This is because they take about $N^2$ time on a list of size $N$.

Algorithms that are based on “divide-and-conquer” such as merge sort or quick sort are much faster.

These algorithms run in about $N \log N$ time on a list of size $N$. 
Merge Sort: Main Idea

- **Divide Step:** Partition the list into two halves.

- **Conquer Step:** Sort each of the halves separately.

- **Combine Step:** “Merge” the two sorted halves into a sorted whole.
# The merge sort function; sorts the sublist \( L[first:last+1] \)
def generalMergeSort(L, first, last):
    # Base case: if first == last then it is already sorted

    # Recursive case: \( L[first:last+1] \) has size 2 or more
    if first < last:
        # divide step
        mid = (first + last)/2

        # conquer step
        generalMergeSort(L, first, mid)
        generalMergeSort(L, mid+1, last)

        # combine step
        merge(L, first, mid, last)