Merge Sort: an example of “Divide and Conquer”
Sorting

Problem statement: given a list $L$ of data items, sort them (from smallest to largest)

Data could be numbers, characters strings . . . any data type that supports “comparison”

We may sort a list of “records” based on different “fields” within each record

. . . and what we are sorting are just “pointers” to these records

The list could be implemented as an array or some type of linked list
A generic recursive sorting algorithm

Input: a list \( L \)
Output: a sorted list

if \(|L| \leq 1\) then  // \(|L|\) means length of \( L \)
    return \( L \)
else
    split \( L \) into two nonempty sublists \( L_1 \) and \( L_2 \)
    recursively sort \( L_1 \) and \( L_2 \)
    return \( \text{merge}(L_1, L_2) \)
A linear time merge algorithm

\[\text{merge}(L_1, L_2)\]
Input: sorted lists \(L_1\) and \(L_2\)
Output: a sorted list
Assume: singly linked list of nodes

if \(L_1 = \text{null}\) then
  return \(L_2\)
else if \(L_2 = \text{null}\) then
  return \(L_1\)
else  // both \(L_1\) and \(L_2\) are nonempty
  if \(L_1.\text{head} \leq L_2.\text{head}\) then
    \(L' \leftarrow \text{merge}(L_1.\text{tail}, L_2)\)
    return new Node\((L_1.\text{head}, L')\)
  else
    \(L' \leftarrow \text{merge}(L_1, L_2.\text{tail})\)
    return new Node\((L_2.\text{head}, L')\)

Analysis:
- total number of recursive calls: \(\leq |L_1| + |L_2|\)
- time per recursive call: \(O(1)\)
- total time: \(O(|L_1| + |L_2|)\)
An array implementation

Input: sorted arrays $A[0..m), B[0..n)$
Output: sorted array $C[0..m+n)$

$i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$
while $i < m$ and $j < n$ do
    if $A[i] \leq B[j]$ then
        $C[k++] \leftarrow A[i++]$
    else
        $C[k++] \leftarrow B[j++]$

while $i < m$ do
    $C[k++] \leftarrow A[i++]$

while $j < n$ do
    $C[k++] \leftarrow B[j++]$

Running time analysis:

each loop iteration moves an item to $C$

$\Rightarrow$ total number of loop iterations $m + n$
Back to our recursive sorting algorithm . . .

Let $n = |L|$

Total running time:

- the “local computation” time: $O(n)$, plus
- the time spent in the recursive calls

The total running time is determined by the strategy used to split $L$ into two sublists

**Unbalanced strategy:** always split $L$ into sublists of size $n - 1$ and 1

total time is $O(T)$, where

$$T = n + (n - 1) + (n - 2) + \cdots + 1$$

$\implies O(n^2)$ running time (essentially insertion sort)
Balanced strategy: the Merge Sort Algorithm

Always split $L$ into two sublists of (roughly) equal size

Running time analysis using a **recursion tree**:

- Every node in the tree corresponds to a single call
  - its children correspond to the recursive calls
- We associate with each node with the subproblem size and local cost of the corresponding call
- We add up all the local costs — usual level by level
Example: $n = 8$

More generally, assume $n = 2^k$

At level $j = 0, \ldots, k$:

- $2^j$ nodes, each with local cost $2^{k-j}$
- Cost per level: $2^k = n$
- Total cost: $n(k + 1) = O(n \log n)$
More observations

Good news:

- Merge Sort is a *stable* sort: data items of equal value retain their relative positions

Bad news:

- The array implementation of Merge Sort is not an *in place*: $O(n)$ auxiliary space is needed