MergeSort.

Divide and Conquer Pattern Applied to Sorting Equals the MergeSort Algorithm.

Records.


CSI 310: Lecture 16
First, note \( n \) is a local variable.

\[
\begin{align*}
\text{int} & \text{ fact}(\text{int} \ n) \\
& \{ \text{ if}(n==0) \text{ return } 1; \text{ else return fact}(n-1); \} \\
& \text{int} \text{ fact}(\text{int} \ n)
\end{align*}
\]
VARIAABLES

Different ACTIVATIONS—Different AUTOMATIC

The storage used for AUTOMATIC VARIABLES is in the activation record.

The EVENT of "calling one function once" is an ACTIVATION.

Each time a function is called, an ACTIVATION RECORD is created.
4. Returned to.

now, automatic variables are in the activation record of the activation

RETURNED TO.

3. The ACTIVATION whose CALL originally created this activation is

"goes away"

2. This ACTIVATION's ACTIVATION RECORD

returns value (if any) is saved for use by the caller.

1. When a function ACTIVATION executes the return statement: . . .
Only true "logically"...this data is actually stored in the called activation.

ACTIVATION is destroyed, and its record gets recycled.

Really: When an ACTIVATION executes the return, that
spot within the function's body is this activation CALLED a function.

Different one (i.e. automatic variables, (2) A pointer to the activation (A)
activation's local (i.e. automatic) variables, (2) A pointer to the activation (A)
An definition of Activation Record: The data structure that holds (1) An
will control what THAT activation does.

Really: A new FUNCTION Activation is created, and the function's body
Wrong: Control "jumps" or "goes to" the function's body.

What HAPPENS when the computer executes a FUNCTION CALL?
{ return Solution;
{
Solution = MERGE (Ansl, Anst, Anst2);
ListNode * Anst = MERGE sort (Subproblem2);
ListNode * Ansl = MERGE sort (Subproblem1);
}
//RECURSE one or more times:
Subproblem2 = Instance; //SPLIT removed half the original list;
Subproblem1 = SPLIT (Instance);
ListNode * Subproblem1, Subproblem2;
}
else
{
Solution = Instance; //TRIVIAL SORT of a 1-element list;
}
if (Instance (Instance) == 1)
{ ListNode * Solution;
ListNode * MERGE sort (Instance * Instance);
}
particular activation of the (recursive) function \( \text{mergeSort} \) is doing.

**VERY IMPORTANT:** AUTOMATIC VARIABLES are used to keep track of what a
(2) The return value is the addr. of the first node.

The nodes removed.

\[
\begin{align*}
\text{post:} & \quad \text{Instance}\rightarrow\text{orignal value, but with approx. half} \\
\text{pre:} & \quad \text{Instance}=\text{the addr. of the first node of a} \\
\text{ListNode * Split (ListNode * Instance)} & \quad \text{with strings} \\
\end{align*}
\]

\[
\begin{align*}
\text{compared using} & \quad \text{ListNode is a linked-list node type; each node's data a string. Strings will be} \\
\text{Merge sort explained using Pre/Postconditions:} & \\
\end{align*}
\]
We will then illustrate the (recursive) 
MERGE_sort hal.
Sample list of items to sort:

- Cat
- Aardvark
- Bat
- Ape
- Caterpillar
- Dog
- Ant
- Zebra
SO, NO COPYING of strings is ever done!

2. The fastest way to change the positions is to move ListNodes from one spot to another, possibly in a different linked list.

1. Implement the ADT list (element collection where ORDER MATTERS and multiple occurrences are OK) with linked lists.

Main data structure ideas for sorting strings using linked lists:
is called "Merging"

3. Combine the two sorted groups into one large sorted list. This combining

INDEPENDENTLY!

2. Sort each of these smaller groups (by recursive calls). That means

size.

I. Divide the elements to be sorted into two groups of equal (or almost equal)

paradigm or pattern is applied to the problem of sorting a sequence:

MergeSort is the algorithm that is invented when the divide-and-conquer
<table>
<thead>
<tr>
<th>Input</th>
<th>Problem</th>
<th>ANSWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 31 25</td>
<td>5 2 6</td>
<td>20 8</td>
</tr>
<tr>
<td>3 6</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

University at Albany Computer Science Dept.
First Recursive Activation...

<table>
<thead>
<tr>
<th>Problem</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 12 31 25 5</td>
<td>? ? ? ?</td>
</tr>
<tr>
<td>8 20 2 3 6</td>
<td>? ? ? ?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Recursive Solution is:

Problem
Input

Problem
ANSWER

Problem

5 9 12 25 31

9 12 25 31 5
See the Returned Answer!
Again on 2nd half.

Next, Recurse.
Recursive Solution is:

\[
\begin{array}{c|c|c|c}
\text{Problem} & \text{Input} & \text{ANSWER} \\
2 & 3 & 6 \\
8 & 20 & 2 \\
\end{array}
\]
By 2nd Recursion

Answer Ret.

0 8 6 20 2
8 20 3 6

9 12 25 31 5
5 9 12 31 9
**No More Recursion!**

Get the answer by Merging:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>20</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

**Problem Input**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>20</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>12</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>12</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

**Problem Answer**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>20</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>20</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>12</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>12</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

**University at Albany Computer Science Dept.**
Only true "logically"...this data is actually stored in the called activation.

ACTIVATION is destroyed, and its Record gets recycled.

Really: When an ACTIVATION executes the return operation, that spot within the function's body that activation CALLED a function different one (whose CALL operation created this one. (3) The return

An Activation Record: The data structure that holds (1) An

Definition of Activation Record: The data structure that holds what THAT activation does.

Really: A new FUNCTION Activation is created, and the function's body

Wrong: Control "jumps" or "goes to" the function's body.

What HAPPENS when the computer executes a FUNCTION CALL?