MERGESORT

Sorting Recursively

Do you remember any iterative sorting methods? How fast are they?

Can we produce a good sorting method by thinking recursively?

We try to divide and conquer: break into subproblems and put their solutions together.

Divide and Conquer

Let's try it on the following list of letters:

```
P E N G U I N S
```

Break it into two lists:

```
P E N G
U I N S
```

Sort them:

```
E G N P
I N S U
```

Is putting these two lists together easier than sorting the whole list?

Merging

Combining two sorted lists into one sorted list is called **merging**.

```
 E  G  N  P
 I  N  S  U
```

Starting at the front of both lists, we keep picking the smallest letter:

```
*E* G  N  P
 I  N  S  U

    G  N  P
*I* N  S  U

   *G* N  P
    N  S  U
```

etc.

Code for Mergesort

We'll implement it for arrays. We only care about the order of elements so we assume only that the elements are `Comparable`.

Also, we'll try sorting the array 'in place', rather than creating a new array to contain the result.

```
/** Sort list[s..e] in non-decreasing order. */
public static void mergeSort(Comparable[] list,
                             int s,
                             int e) {

    if (s < e) { // More than one element

        int mid = (s + e) / 2;
        mergeSort(list, s, mid);
        mergeSort(list, mid + 1, e);

        ...merge list[s..m] and list[m+1..e]
            into list[s..e]...

    } // else just 1 element, so already sorted.

}
```


Except possibly for the merge portion, this was very simple to write. What can we do to understand how it works?

- if the recursive calls work, the method will clearly work
- trace it

Exercise: Write a requires clause for the method.

Exercise: Write a wrapper method for the common case of mergesorting an entire array.

Exercise: Would `mergeSort` still work if the two recursive calls were:

```
mergeSort(list, s, mid-1);
mergeSort(list, mid, e);
```


Header for Merge

Let's make it a helper method.

What header should it have? Imagine calling it from mergeSort:

```
merge(list, s, mid, e);
```

This gets us started with the header.

```
/** Merge list[s..m] with list[m+1..e].
  * Requires: the two sublists are sorted
  * in non-decreasing order.
  * Ensures: list[s..e] is sorted. */
private static void merge(Comparable[] list,
                          int s,
                          int m,
                          int e)
```


Body of Merge

```
int p1 = s; // index of current candidate in list[s..m]
int p2 = m + 1; // index of current candidate
                // in list[m+1..e]

// Temporary storage to accumulate the merged result.
Comparable[] merged = new Comparable[e - s + 1];
int p = 0; // index to put next element into merged

// merged[0..p] contains list[s..p1-1] merged
// with list[m+1..p2-1]
while (p1 != m + 1 && p2 != e + 1) {
    if (list[p1].compareTo(list[p2]) < 0) {
        merged[p] = list[p1];
        ++p1;
    } else {
        merged[p] = list[p2];
        ++p2;
    }
    ++p;
}

... to be continued ...
```

What's true at the end of the `while` loop?

What's left to do?

```
/*

                                                              */
if (p1 != m + 1) {
    System.arraycopy(list, p1,
                     list, s + p,
                     m + 1 - p1);
}
System.arraycopy(merged, 0, list, s, p);
```

Exercise: Finish the requires clause for the method.

Exercise: If we had copied the remainder of the unfinished sublist to `merged` and then copied all of `merged` back to `list`, would that have changed the big-O time complexity?

Exercise: Rewrite `mergeSort` and `merge` to follow the Java convention for ranges: passing `s` and `e` as the start and end of a sublist of `list` means `list[s..e)` (i.e. `list[s..e-1]`).

Exercise: Write a recursive merge method.

Exercise: Write a mergesort for linked lists.

Exercise: Does the following approach to an in-place merge work: If the next element is in the first list, then leave it there and move forward in the first list, otherwise swap it with the next element in the second list and move forward in both lists.

Efficiency of Sorting Methods

What sorting methods other than mergesort do you know? How efficient are they?

Let's see how merge sort compares.

With loops, we know how to analyze efficiency. What to do with recursion?

Strategy: Determine (1) the total number of calls to `mergeSort` and (2) the time per call. Multiply them together.

(1) Number of Calls to `mergeSort`

Suppose we call `mergeSort` with a list of 8 elements. What happens?

Number of calls in total:

Other sizes of list

Let $C(n)$ be the total number of calls to `mergeSort` if it is called initially with a list of n elements.

We know $C(8) = 15$.

Determine $C(n)$ for some other values of n.
To make it easier, consider only powers of 2.

Results:

n	1	2	4	8	16	...
$C(n)$						

General rule: $\boxed{}$

Proving the rule

We only derived the rule for powers of two, so the theorem should only make its claim for powers of 2:

Theorem: for $n \geq 1$ any power of 2,
$C(n) = 2n - 1$.

An equivalent theorem:

Theorem: for all integers $p \geq 0$,
$C(2^p) = 2 \times 2^p - 1$.

Let's prove it, by induction on p.

Prove: For all integers $p \geq 0$, $C(2^p) = 2 \times 2^p - 1$.

Base Case: Prove $C(2^0) = 2 \times 2^0 - 1$.
Consider a call to `mergeSort` with a list of length $2^0 = 1$.
The if condition fails and we immediately return.
The total number of calls is thus just 1.
Therefore $C(2^0) = 1$ which is the same as $2 \times 2^0 - 1$.

Let k be an arbitrary integer ≥ 0.
Induction Hypothesis: Assume $C(2^k) = 2 \times 2^k - 1$.

Induction Step: Prove $C(2^{k+1}) = 2 \times 2^{k+1} - 1$.
Consider a call to `mergeSort` with a list of length 2^{k+1}.
Since $k \geq 0$, $2^{k+1} \geq 2$.
So the if-condition succeeds, and we make two recursive calls.
Each call involves $2^{k+1}/2 = 2^k$ elements.
By the induction hypothesis, $C(2^k) = 2 \times 2^k - 1$.
So each recursive call causes a total of $2 \times 2^k - 1$ calls.
The total number of calls, for an initial list of length 2^{k+1}

$$\begin{aligned} &= 1 + 2(2 \times 2^k - 1) \\ &= 1 + 2(2^{k+1} - 1) \\ &= 1 + 2 \times 2^{k+1} - 2 \\ &= 2 \times 2^{k+1} - 1 \end{aligned}$$

Thus $C(2^{k+1}) = 2 \times 2^{k+1} - 1$.

Conclusion: For all integers $p \geq 0$, $C(2^p) = 2 \times 2^p - 1$.

(2) Time per call

We now know the total number of calls made to `mergeSort`.

Let's figure out the time required per call so that we can put it all together.

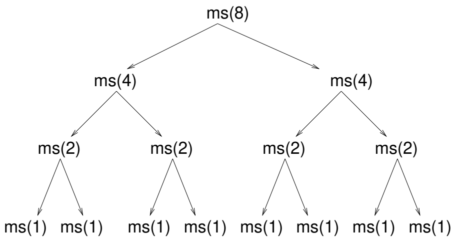
In big-O terms, how much time is required to execute a call to `mergeSort` on a list of n items?

Immediate problem: The size of the list is different every time we recurse.

Hm.

Another way to analyze `mergeSort`

Do you notice anything about the rows in the tree of calls?



Analysis of `mergeSort`

How much time is required to merge two sorted lists into one list with total size s?

So how much time in total is required for all the merging in any single row? How many rows are there?

So what is the total amount of time required?

Exercise: What effect would it have on the time required for mergesort if we split the list into 3 pieces instead of 2?

Exercise: How much time would a (good version of) mergesort take for linked lists?

Comparing with other sorts

Many familiar sorts are $O(n^2)$ in the worst case, including:
bubble sort, selection sort, insertion sort

Some other sorts are $O(n \log n)$ in the worst case (like mergesort), for example heap sort.

But that's only part of the story. We also care about

- the *average* case.
  This is much harder to determine, because it requires arguing about the probability of seeing various cases.
- special cases, such as a list that is only slightly out of sorted order, is already sorted, or is sorted in the reverse order.

Analysis of Recursive Code

Our analysis of `mergeSort` required us to unwind the recursion. If you truly understand recursive code, you don't have to unwind it in order to understand or analyze it (but it's ok to unwind it).

Recurrence relations are suited to analyzing efficiency of recursive code:

REC RELN		CODE
initial condition	$\leftrightarrow$	degenerate case
general rule	$\leftrightarrow$	recursive case

Induction is suited to proving correctness of recursive code:

PROOF		CODE
base case	$\leftrightarrow$	degenerate case
induction step	$\leftrightarrow$	recursive case