Sorting Algorithms (ascending order).

Algorithm - set of steps that solve a problem.

Insertion sort - most intuitive, simplest, how most of us would sort deck of cards.

Insertion-Sort (LEO...n-I) // Sorts list L

For i = 1...n-1

move L[i] left until (in correct sorted position)
L[i] is before all elements that are > L[i]

ex: 26 5 37 1 61 11 59

5 26 37 1 61 11 59

5 26 37 1 61 11 59

1 5 26 37 61 11 59

1 5 11 26 37 61 59

1 5 11 26 37 59 61
Run Times

worst-case? For each element, must compare with all previous elements \(\Rightarrow O(n^2)\).

when? reverse sorted list

average-case: Also \(O(n^2)\). Roughly \(n\) comparisons for each element

best-case? already sorted list: \(O(n)\)

for each element, first comparison always fails

If nearly sorted, very fast

Also useful for small lists.

<SKIP>

in-place sorting - uses constant amount of extra space

stable sort - preserves input order of equal elements

Merge Sort: 
(1) Split lists into small sublists
(2) Merge small sublists to create larger (sorted) sublists
(3) Repeat (2) until entire list is sorted
\[ L = 26 \ 5 \ 37 \ 1 \ 61 \ 11 \ 59 \ 15 \]

\[
\begin{bmatrix}
26 & 5 & 37 & 1 \\
61 & 11 & 59 & 15
\end{bmatrix}
\]

\[
\begin{bmatrix}
26 & 5 \\
37 & 1 \\
61 & 11 \\
59 & 15
\end{bmatrix}
\]

26 \ 5 \ 37 \ 1 \ 61 \ 11 \ 59 \ 15
Bulk of algorithm is merging.

How to merge 2 sorted lists?

- Sublists of size 1:
  
  \([26] \ [5]\) - Just compare the 2 elements \(\Rightarrow [5] \ [26]\)

- Sublists of size 2?

  \(A: [5 \ 26], \ B: [1 \ 37]\)

  \(A^\uparrow P^\uparrow B^\uparrow P^\uparrow\)

(Compare \(A\) to \(B\), place smaller in new list \(C\), increment corresponding ptr)

\(C: [1 \ 5 \ 26 \ 37]\)

(Another sublist \(D:\)

\(D^\uparrow P^\uparrow\)

\(E: [1 \ 5 \ 11 \ 15 \ 26 \ 37 \ 59 \ 61 \ 85 \ 100]\)

What if one sublist larger than the other?

Just copy remaining items of larger sublist.
Merge$(A, B)$

- $Aptr = ptr$ to list $A$, $Bptr = ptr$ to list $B$
- while $(Aptr < A.length$ and $Bptr < B.length)$
  - Compare $A$ element to $B$ element
  - Place smaller element in $C$, increment corresponding $ptr$
  - if $Aptr < A.length$ copy $A$ elements into $C$
  - else if $Bptr < B.length$ $B$
  - return $C$

$\text{Runtime} = O(n)$

// Sorts list $L$ $\text{(start...end)}$
$\text{MergeSort} \ (L, \text{start...end})$ these will change for each recursive call

Notice, base case, if $\text{start} = \text{end}$ (list of size $1$, do nothing)
if ($\text{start} < \text{end}$) // Similar to Binary Search

- $\text{mid} = (\text{start} + \text{end})/2$

$\text{mergeSort} \ (L, \text{start...mid})$
$\text{mergeSort} \ (L, \text{mid+1...end})$

$L = \text{merge} \ (L, \text{start...mid}, L, \text{mid+1...end})$

$\text{Runtime}^*$

Each merge takes at most $O(n)$
How many merges? in terms of $n$, $\log n$

Total: $O(n \log n)$
At each level, total merge time is $O(n)$.

# levels? $O(\log n) = \#$ times we can split list of size $n$ in $\frac{1}{2}$ until we get lists of size 1.

Total: $O(n \log n)$. 
Our implementation (of Merge) not in-place.

Stable? Yes, ordering among equal elements is preserved.

Example of Divide-and-Conquer (and combine) algorithm.

Divide: problem into smaller sub-problems  (Split list in 1/2)
Conquer: (solve) sub-problems         (Merge)
Combine: solutions                    (Join left, right)