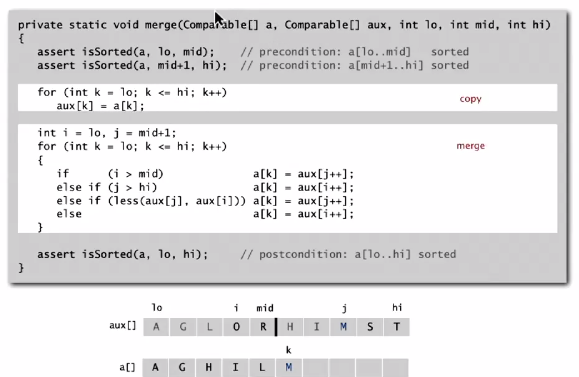
Mergesort

Divide array into two halves, recursively sort each half and merge each half.

Merging = compare the values in each sorted half of a copy of an array (left to right each side) and insert the smaller into the array, then increment on the side that had a value added



**Assertions** are statements to test assumptions about your program:

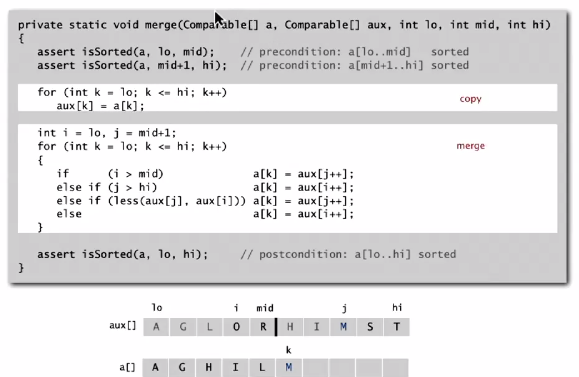
* Help detect logic bugs
* Document code

(Basically what you expect, while testing that these conditions hold)

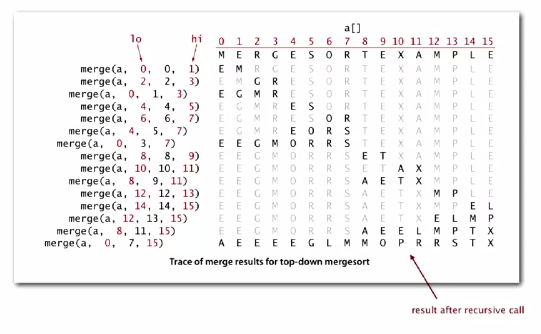
Can enable or disable assertions at runtime (java –ea MyProgram == enable, java –da MyProgram == disable)

Best practice is to use assertions to test internal invariants and to disable them in production code.

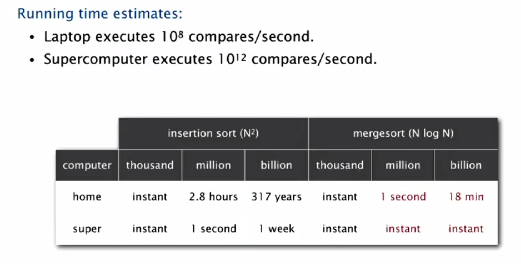
Mergesort implementation



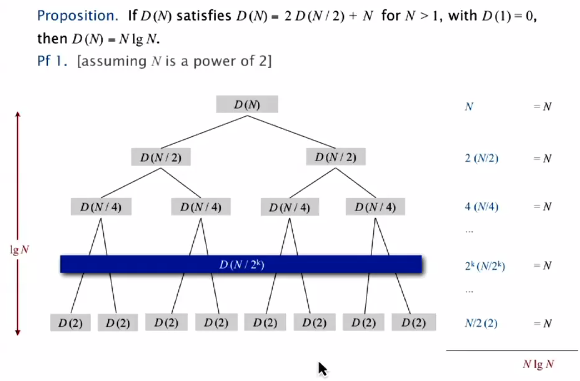
Mergesort trace



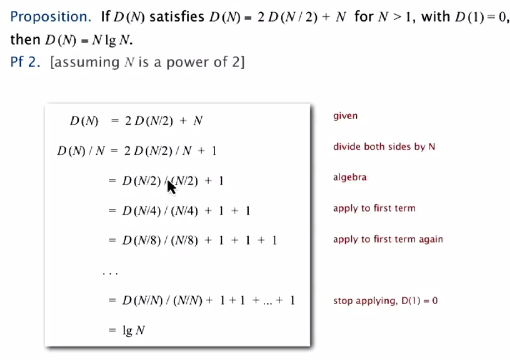
Very fast runtime (N log N):



Divide-and-conquer recurrence mathematical proof visualization

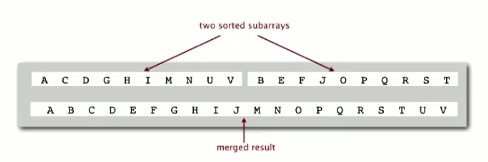


Further mathematical proof



Memory analysis:

Memory required is N (linear) due to the array for the final merge. You can therefore only sort something half the capacity of memory due to the extra array that will be size N.

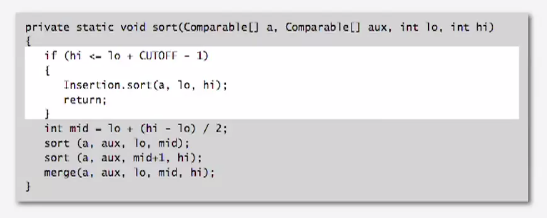


Other algorithms (like selection, insertion and shellsort) use <= c log N extra memory and are ‘in-place’

IMPROVEMENTS

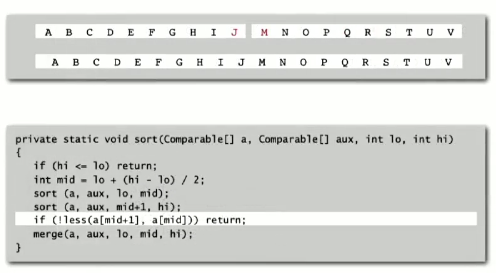
Too much overhead due to recursive calls- use insertion sort on small subarrays (20% speed increase).

Implementation:

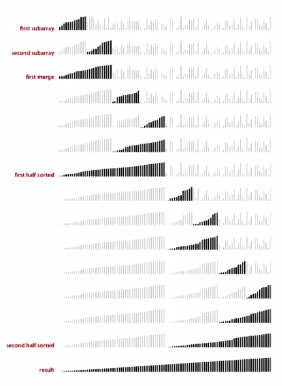


Also, stop sorting if the two halves have already been sorted

Implementation:

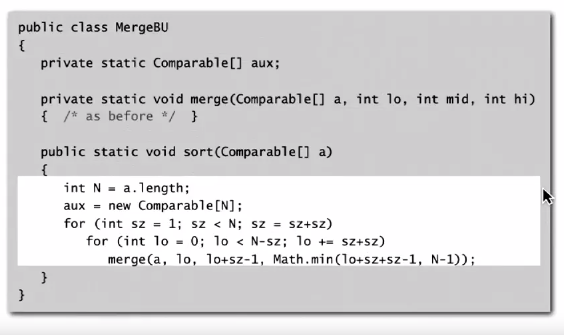


Mergesort visualization

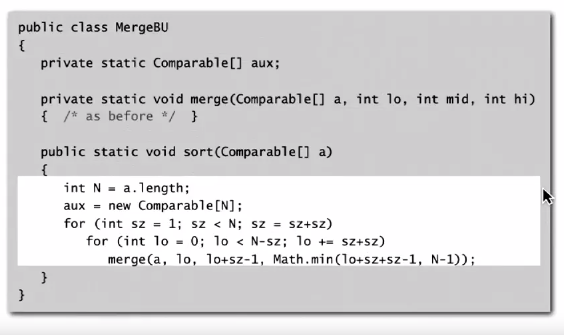


Bottom up Mergesort

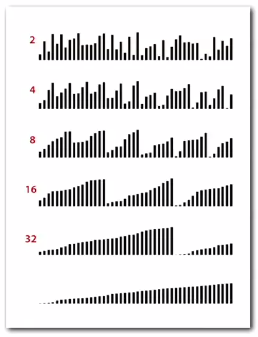
Basically, merge subarrays of size 2 and up until you merge the entirety of the array.



Implementation



Bottom up mergesort visualized



Mergesort doesn’t necessarily divide perfectly in half, so final subarrays may be smaller. However, merge operation doesn’t care about this and merges at the same speed with no issue.

It uses the same amount of memory as the recursive version of mergesort.

**Complexity of sorting**

**Computational complexity:** framework to study efficiency of algorithms for solving particular problem x

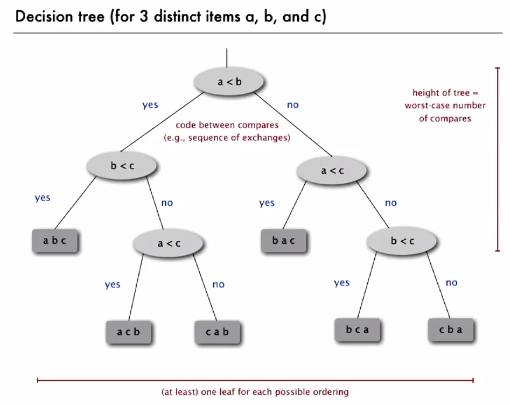
* **Model of computation:** operations that the algorithms are allowed to perform
* **Cost model:** count different operations (e.g. comparisons in sorts)
* **Upper bound**: cost guarantee provided by some algorithm for solving X (can be no greater)
* **Lower bound:** proven limit on cost guarantee on all algorithms for X (can be no lower)
* **Optimal algorithm:** the lower bound == the upper bound, so the best possible cost guarantee for X

*Example through sorting:*

* Model of computation: decision tree (can access info only through compares… Java Comparable framework)
* Cost model: the number of compares
* Upper bound: ~N log N from mergesort
* Lower bound: ~N log N
* Optimal algorithm: mergesort

Algorithms compute comparisons

The height of the decision tree Is the highest number of compares (worst case number)

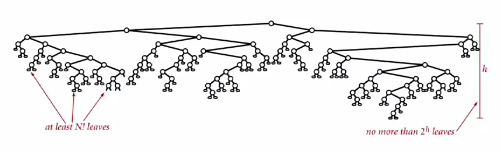


Lower bound uses similar decision tree and determines:

Any compare-based sorting algorithm must use at least log (N !) ~ N log N compares in the worst case.

Proof:

* Assumed: Array consists of N distinct values a1 through aN
* Worst case based on height h of decision tree.
* Binary tree of height h has at most 2h leaves
* N ! different orderings -> at least N ! leaves



Sterlings’s formula:

2^h >= # leaves >= N ! , therefore: h >= log ( N ! ) ~ N log N

Complexity in context:

* Mergesort **IS**  optimal in terms of compares, BUT
* Mergesort **IS NOT** optimal in respect to space usage.
* Lower bound may not hold if the algorithm has info about
  + The initial order of the input
  + The distribution of key values
  + The representation of the keys
* **Partially ordered arrays:** depending on the initial order of the input we may not need N log N compares (Insertion sort only requires N-1 compares if array is sorted)
* Depending on the input distribution of **duplicates**, we may not need N log N compares
* We can use digit/character compares instead of key compares for numbers and strings (digital properties of keys)

USE THEORY AS A GUIDE

E.g. Don’t try to design a sorting algorithm that guarantees ½ N log N compares (lower bound proves it’s not possible)

E.g. Consider designing an algorithm that is both time- and space- optimal.

Comparators

Similar to sorting a music library by artist, genre, etc.

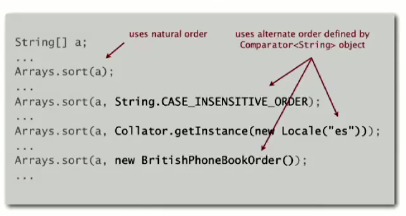


Comparator interface helps sorting using an alternate order- or multiple orders- on the same data.

Required property: must be a total order.

Java system sort:

* Create a Comparator object
* Pass as second argument to Arrays.sort()

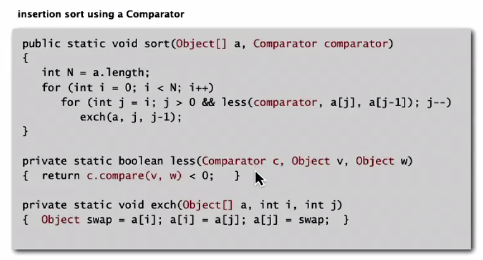


Decouple definition of the data type from the definition of what it means to compare two objects of that type. We can even make out own order for String comparisons, etc.

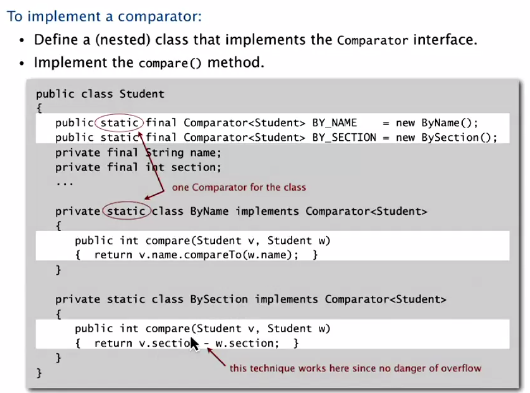
Comparator interface

To support comparators, we use Object instead of Comparable

Pass Comparator to sort() and less() (as a second argument) and use it in less()



To implement a comparator, you declare as use as below:



**Stability:** first sort by X, then sort by Y. It preserves the relative order of items with equal keys.

Stable sorts:

* Insertion sort : equal items are never moved past each other- only items less than the value move left
* Mergesort : so long as merge operation is stable , this is stable.
  + For example: in below code, if 2 keys are equal, it pulls from the left subarray first to merge. Therefore, with two sets of equal keys, the relative order is preserved.

NOT stable sorts: selection sort and shellsort: large moves that can move equal values past each other

