Mergesort and Quicksort

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 2.2), Prof. Janicki's course slides, and https://www.cs.princeton.edu/~rs/AlgsDS07/04Sorting.pdf

Mergesort: Top Down Approach Basic idea

Divide and Conquer Algorithm.

Basic plan

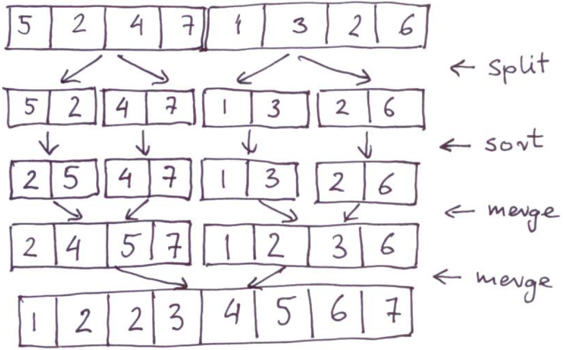
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Mergesort: Example (see demo for merge)

Trace of selection sort (array contents just after each exchange)



Mergesort: Example (see demo for merge)

ALGORITHM 2.4 Top-down mergesort

```
public class Merge
{
   private static Comparable[] aux;      // auxiliary array for merges

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];    // Allocate space just once.
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {  // Sort a[lo..hi].
      if (hi <= lo) return;
      int mid = lo + (hi - lo)/2;
      sort(a, lo, mid);         // Sort left half.
      sort(a, mid+1, hi);       // Sort right half.
      merge(a, lo, mid, hi);    // Merge results (code on page 271).
   }
}
```

- Note the use of the aux[] array. It eliminates the overhead of creating new arrays during merges.
- As the book points out, the proper way is to make aux[] local to Sort, and pass it as an argument to Merge().

Merging: Java implementation

Abstract in-place merge

```
public static void merge(Comparable[] a, int lo, int mid, int hi)
{  // Merge a[lo..mid] with a[mid+1..hi].
   int i = lo, j = mid+1;

   for (int k = lo; k <= hi; k++)  // Copy a[lo..hi] to aux[lo..hi].
      aux[k] = a[k];

   for (int k = lo; k <= hi; k++)  // Merge back to a[lo..hi].
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi )              a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
}
```

- Note that, this implementation is not exactly “in-place” and hence the name “abstract in-place” merge.
- An algorithm is in-place if it requires $N + O(\log N)$ memory.

Mergesort analysis: Memory

- How much memory does mergesort require?
  A. Too much!
  - Original input array = N.
  - Auxiliary array for merging = N.
  - Local variables: constant.
  - Function call stack: $\log_2 N$.
  - Total = 2N + O($\log$ N).
- How much memory do other sorting algorithms require?
- In-place merge is complicated - see [Kronrud, 1969]

Binary Trees: Tree source: https://www.interviewcake.com/concept/java/binary-tree

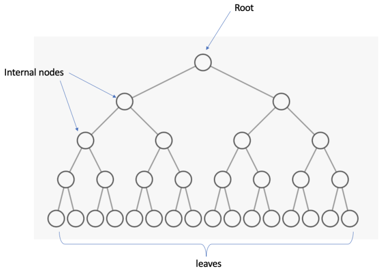
Binary Tree: A tree where each internal node has at most two children.

Full Binary Tree: A binary tree where each internal node has exactly two children.

Internal node: is node in a tree with atleast one child.

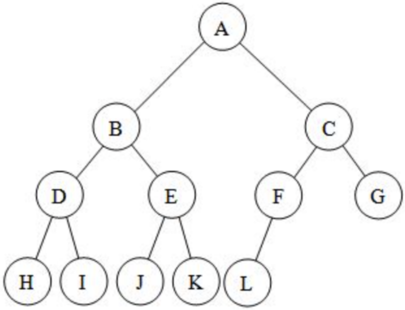
Leaf node: is node in a tree with no children.

Root: Is a node that is either a parent or an ancestor of every other node.



Binary Trees contd..

Complete Binary Tree: A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Tree source: `https://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html`

Mergesort: Time Complexity

Proposition: Mergesort uses $\Theta(N \log N)$ compares to sort an array of length N.

Proof Sketch:

Proof. Sketch.

The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N}_{\text{merge}} \quad \text{for } N > 1, \ \text{with} \, C(1) = 0,$$

where $\lceil x \rceil$ is the smallest integer $\geq x$, i.e. $\lceil 1.5 \rceil = 2$, $\lceil 3.1 \rceil = 4$, and $\lfloor x \rfloor$ is the biggest integer $\leq x$, i.e. $\lfloor 1.5 \rfloor = 1$, $\lfloor 3.1 \rfloor = 3$.
We solve the recurrence when N is a power of 2:

$$D(N) = 2D(N/2) + N, \ \text{for } N > 1, \ \text{with} \, D(1) = 0.$$

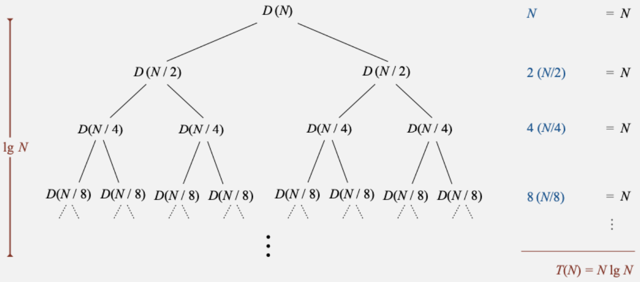
The result holds for all N, but general proof is a little bit messy. □

Mergesort: Time complexity and Recursion Tree

A **recursion tree** is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Mergesort: Bottom-up

Basic plan:

- Pass through file, merging to double size of sorted subarrays. Do so for subarray sizes 1, 2, 4, 8, . . . , N/2, N.
- In particular, merge sub arrays of size one in the first pass to form sorted sub arrays of size two.
- In the second pass merge the sub arrays of size two to form sorted sub arrays of size four.
- Keep performing the passes and merging subarrays, until you do a merge that encompasses the whole array.

Mergesort: Bottom-up Java

Bottom-up mergesort

```
public class MergeBU
{
   private static Comparable[] aux;      // auxiliary array for merges

   // See page 271 for merge() code.

   public static void sort(Comparable[] a)
   {  // Do lg N passes of pairwise merges.
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)        // sz: subarray size
         for (int lo = 0; lo < N-sz; lo += sz+sz) // lo: subarray index
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Mergesort: Bottom-up Java

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
merge(a, 0, 0, 1)	**E**	**M**	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	**G**	**R**	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	**E**	**S**	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	**O**	**R**	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	**E**	**T**	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	**A**	**X**	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	**M**	**P**	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	**E**	**L**
sz = 2																
merge(a, 0, 1, 3)	**E**	**G**	**M**	**R**	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	**E**	**O**	**R**	**S**	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	**A**	**E**	**T**	**X**	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	**E**	**L**	**M**	**P**
sz = 4																
merge(a, 0, 3, 7)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	**A**	**E**	**E**	**L**	**M**	**P**	**T**	**X**
sz = 8																
merge(a, 0, 7, 15)	**A**	**E**	**E**	**E**	**E**	**G**	**L**	**M**	**M**	**O**	**P**	**R**	**R**	**S**	**T**	**X**

Trace of merge results for bottom-up mergesort

Mergesort: Analysis

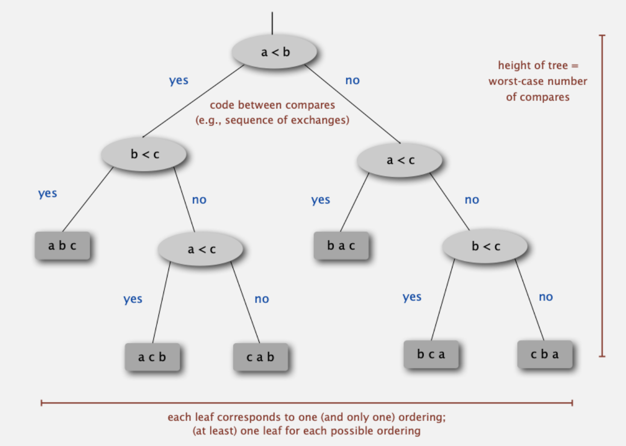
- Top-down mergesort uses between $1/2N \log_2 N$ and $N \log N$ compares to sort any array of length N.
- Top-down mergesort uses at most $6N \log_2 N$ array accesses to sort an array of length N.
- Bottom-up mergesort also uses between $1/2N \log_2 N$ and $N \log_2 N$ compares and at most $6N \log_2 N$ array accesses to sort an array of length N.

Mergesort: Practical improvements

- Mergesort has too much overhead for tiny subarrays. Therefore, use insertion sort for < 10 items.
- Stop if already sorted; that is, do not call merge if the sub arrays are sorted. This can be done by a simple check $a[mid] \leq a[mid+1]$
- Eliminate the copy to the auxiliary array.

Lower Bound for Sorting: Decision Tree

For each particular sequence any sorting can be represented as a decision tree. (below for keys a, b, c, note that $3! = 6$).



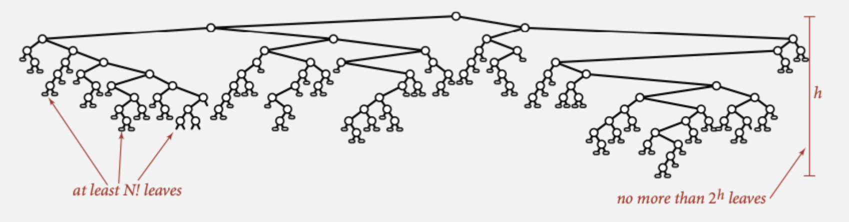
Lower Bound for Sorting

Proposition: Any compare-based sorting algorithm must use at least $N \log_2 N$ compares in the worst-case; that is, $T(N) = \Omega(N \log_2 N)$.

Proof Sketch:

- Assume array consists of N distinct values a_1 through a_N.
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.

Lower Bound for Sorting



- $2^h \geq N! \Rightarrow h \geq \log_2(N!)$, by Sterling's approximation we have $\log_2(N!) \approx N \log_2 N$.
- Hence $h \geq N \log_2 N$. Therefore, any compare-based sorting algorithm must use at least $N \log_2 N$ compares in the worst-case.

Quicksort - Idea I (C. A. R. Hoare 1962)

Quicksort honoured as one of top 10 algorithms of 20th century in science and engineering!

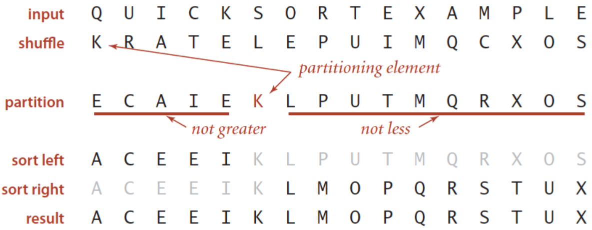
```
Quicksort(S):
If |S| ≤ 3 then
   Sort S
   Output the sorted list
Else
   Choose a splitter a_i ∈ S uniformly at random
   For each element a_j of S

      Put a_j in S⁻ if a_j < a_i
      Put a_j in S⁺ if a_j > a_i
   Endfor
   Recursively call Quicksort(S⁻) and Quicksort(S⁺)
   Output the sorted set S⁻, then a_i, then the sorted set S⁺
Endif
```

The splitter in this algorithm is generally referred to as the pivot.

Quicksort - Idea II

Basic Plan:

- Shuffle the array.
- Partition so that, for some i element a[i] (called the **pivot**) is in place; that is,
  - no larger element to the left of i, and
  - no smaller element to the right of i
- Sort each piece recursively.



Quicksort - Idea II Code

ALGORITHM 2.5 Quicksort

```
public class Quick
{
   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);          // Eliminate dependence on input.
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);  // Partition (see page 291).
      sort(a, lo, j-1);              // Sort left part a[lo .. j-1].
      sort(a, j+1, hi);              // Sort right part a[j+1 .. hi].
   }
}
```

When is it possible for $i = j$, and when is it possible for $j <= i$?

Quicksort - Idea II Code Example

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

no partition for subarrays of size 1

Quicksort - Idea II Partition Code

Partition returns the index of the pivot.

Quicksort partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{  // Partition into a[lo..i-1], a[i], a[i+1..hi].
   int i = lo, j = hi+1;            // left and right scan indices
   Comparable v = a[lo];            // partitioning item
   while (true)
   {  // Scan right, scan left, check for scan complete, and exchange.
      while (less(a[++i], v)) if (i == hi) break;
      while (less(v, a[--j])) if (j == lo) break;
      if (i >= j) break;
      exch(a, i, j);
   }
   exch(a, lo, j);         // Put v = a[j] into position
   return j;               // with a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```

Quicksort - Idea II Partition Example

			v							a[]								
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Quicksort - Idea II Analysis

- Partitioning in-place: Using a spare array makes partitioning easier, but is not worth the cost.
- Preserving randomness: Shuffling is key for performance guarantee. An alternate way to preserve randomness is to choose a random item for partitioning within partition().
- Handling items with keys equal to the partitioning item's key: When duplicates are present, it is (counter-intuitively) best to stop on elements equal to partitioning element. However, it is crucial to avoiding quadratic running time in certain typical applications.

Quicksort Performance Characteristics

Quick Sort is considered the fastest sorting when input is random and not small (> 15). WHY?

- Because it allows a very efficient implementation which is superfast for half of cases.
- Short inner loop: The inner loop of quicksort (in the partitioning method) increments an index and compares an array entry against a fixed value. This simplicity is one factor that makes quicksort quick – it is hard to have a shorter inner loop in a sorting algorithm.

  For example, mergesort and shellshort are typically slower than quicksort because they also do data movement within their inner loops.

Quicksort running time Analysis

- Best case: Number of compares is $\approx N \log N$; that is, $\Omega(N \log N)$
  - The best case is when partition creates equal size subarrays. In this case the recurrence relation is $T(n) = 2T(n/2) + cn$, and so $T(n) \in \Theta(N \log N)$.
  - Same bound of $\Omega(N \log N)$ reached when sub arrays are split in ratio $1 : 9$! In this case the recurrence relation is $T(n) = T(n/10) + T(9n/10) + cn$, and so $T(n) \in \Theta(N \log N)$.
- Worst case: Number of compares is $\approx 1/2N^2$; that is, $O(N^2)$
  - The worst case is when partition creates sub arrays of size zero all the time. In this case the recurrence relation is $T(n) = T(n-1) + T(0) + cn$, and so $T(n) \in O(N^2)$.

Quicksort running time Analysis

- Average case: Expected number of compares is $\approx 1.39N\log N$; that is, $\Theta(N\log N)$.
  - Although mergesort also has the same running time and quicksort does 39% more compares, quicksort is typically faster as it does much less data movement.

Quicksort: Practical improvements

All the below are suggestions are validated with refined math models and experiments.

- Best choice of pivot element $=$ median.
- Even quicksort has too much overhead for tiny subarrays. Therefore, use insertion sort for < 15 items.
- Non-recursive version using stack, and always sort smaller half first.

Sorting Summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort (N log N)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant