

CS2010: Data Structures and Algorithms II

Merge sort and quick sort

Ivana.Dusparic@scss.tcd.ie

Divide and Conquer Sorting

- › Divide problem into smaller parts
- › Independently solve the parts
- › Combine these solutions to get overall solution
- › 2 common approaches:
 - Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
 - Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Merge vs quick

- › In Java, `Arrays.sort()` uses **QuickSort** for sorting primitives and **MergeSort** for sorting Arrays of Objects.
 - Why does it matter for Objects and not for primitive data types?

Mergesort

Merge sort

- › Top down merge sort
 - Recursive
 - Divide array in 2 halves, sort each array recursively, merge the arrays
- › Bottom up merge sort
 - Iterative
 - Iterate through array merging subarrays of size 1, size 2, 4, 8, etc

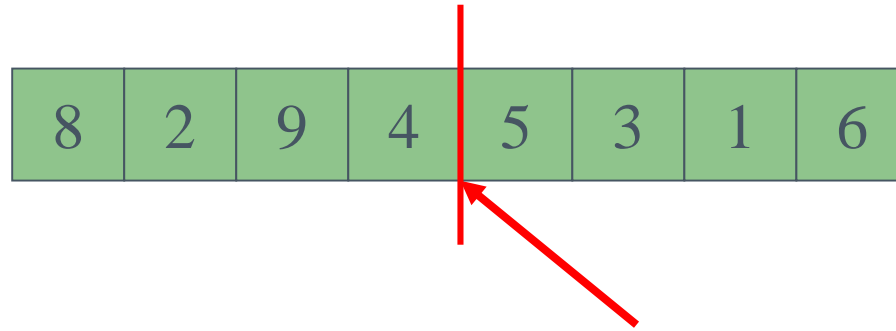
Top down merge sort

a[]															
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
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom up merge sort

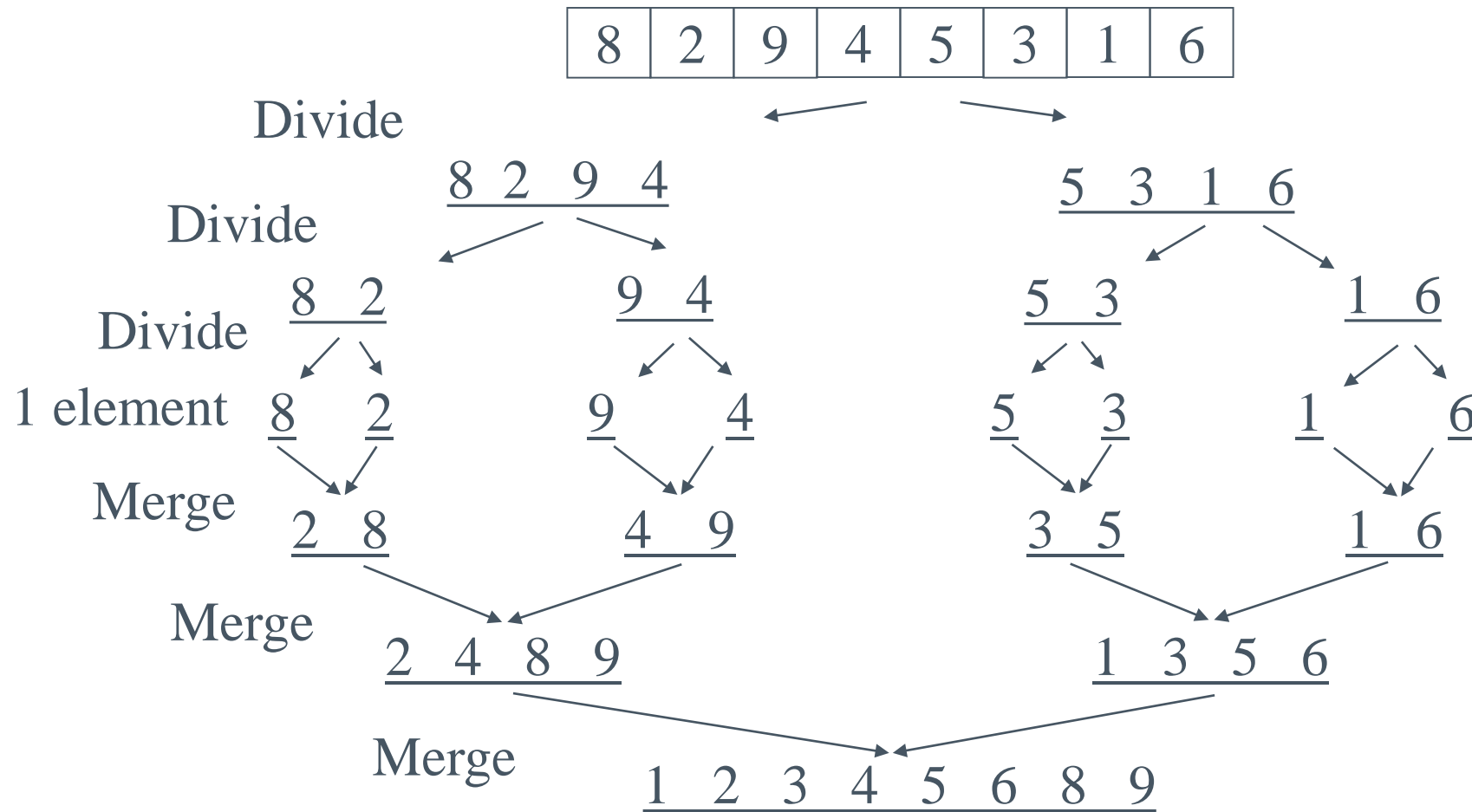
a[i]															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
E	M	G	R	E	S	O	R	E	T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

Top down merge sort



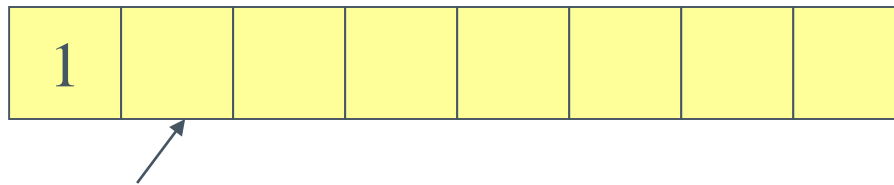
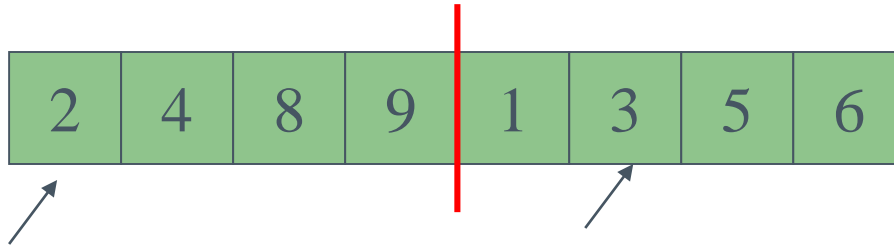
- › Divide it in two at the midpoint
- › Conquer each side in turn (by recursively sorting)
- › Merge two halves together

Top down merge sort



Top down merge sort

- › The merging requires an auxiliary array
 - Requires extra space



Auxiliary array

Top down merge sort Java implementation

- › What methods do we need?

- › public method that passes in array to be sorted

```
public static void sort (Comparable [] a)
```

- › Recursive method with original and auxiliary arrays, and indices of the subarray to be sorted

```
private static void sort (Comparable [] a, Comparable [] aux, int lo,  
int hi)
```

- › Merge method, to merge sorted subarrays, with the 2 arrays to be merged, lowest, highest and midpoint indices

```
private static void merge (Comparable [] a, Comparable [] aux, int lo,  
int mid, int hi)
```

Top down merge sort

- › Create an auxiliary array of the same size as the original one
- › Kick off recursion by passing in 0 and array length-1 as indices (ie the full original array)

```
public static void sort(Comparable[] a)
{
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length - 1);
}
```

Top down merge sort

› Recursive method

- Repeat until lo and hi are equal, ie get to array of length 1
- Note: $\text{mid} = \text{lo} + (\text{hi} - \text{lo}) / 2$ to avoid integer overflow

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Top down merge sort

- › Merge method
- › Copy the original array into auxiliary one, and then merge elements back into the original one in sorted order

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
```

```
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];
```

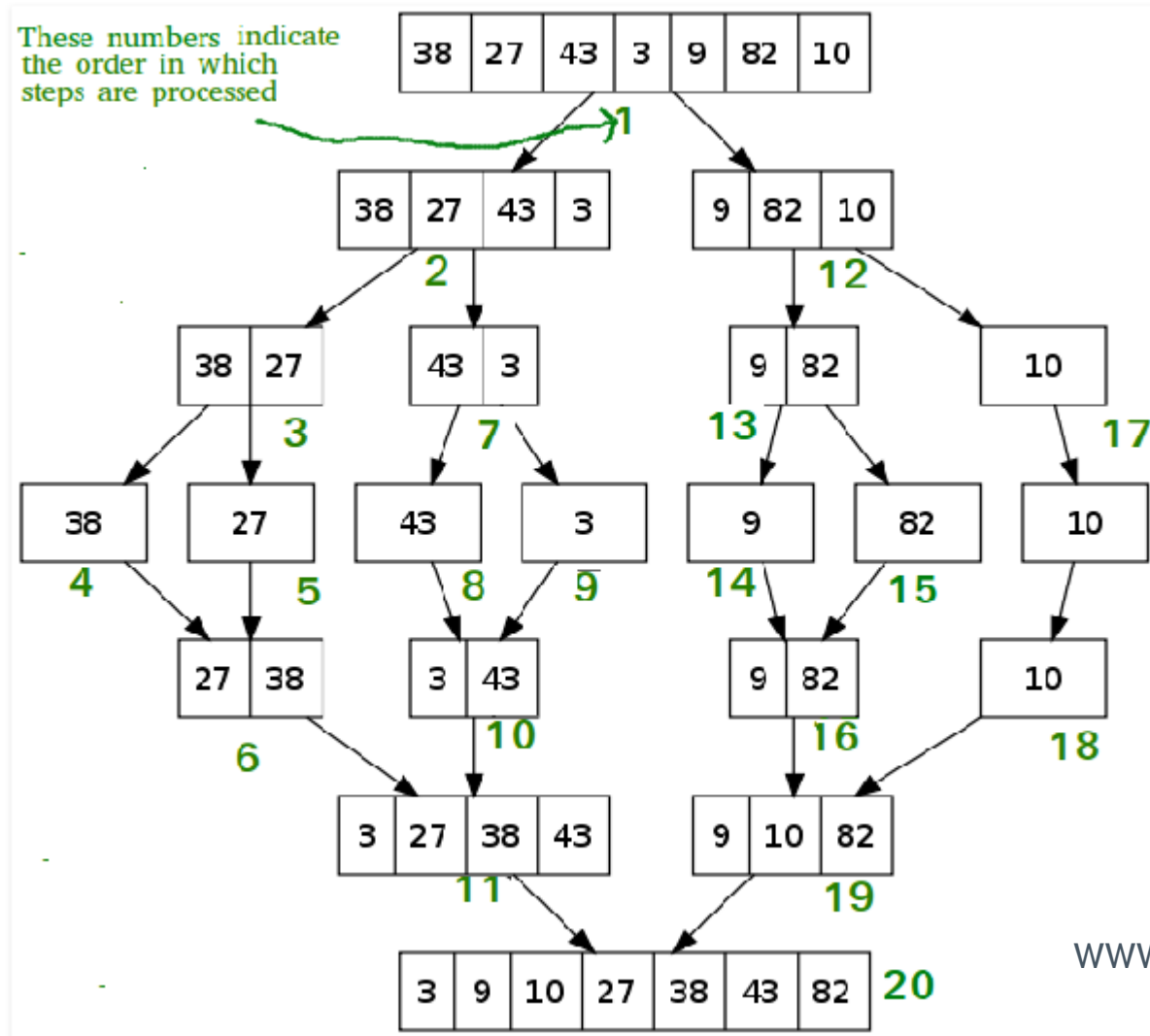
copy

```
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        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++];
    }
```

merge

```
}
```

Top down merge sort



Top down merge sort

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

Merge sort exercise

Illustrate merge sort by showing table partitioning and numbering the order in which steps happen (like in the lecture notes example)

7	8	2	4	5	4	3	1
---	---	---	---	---	---	---	---

Merge sort running time

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

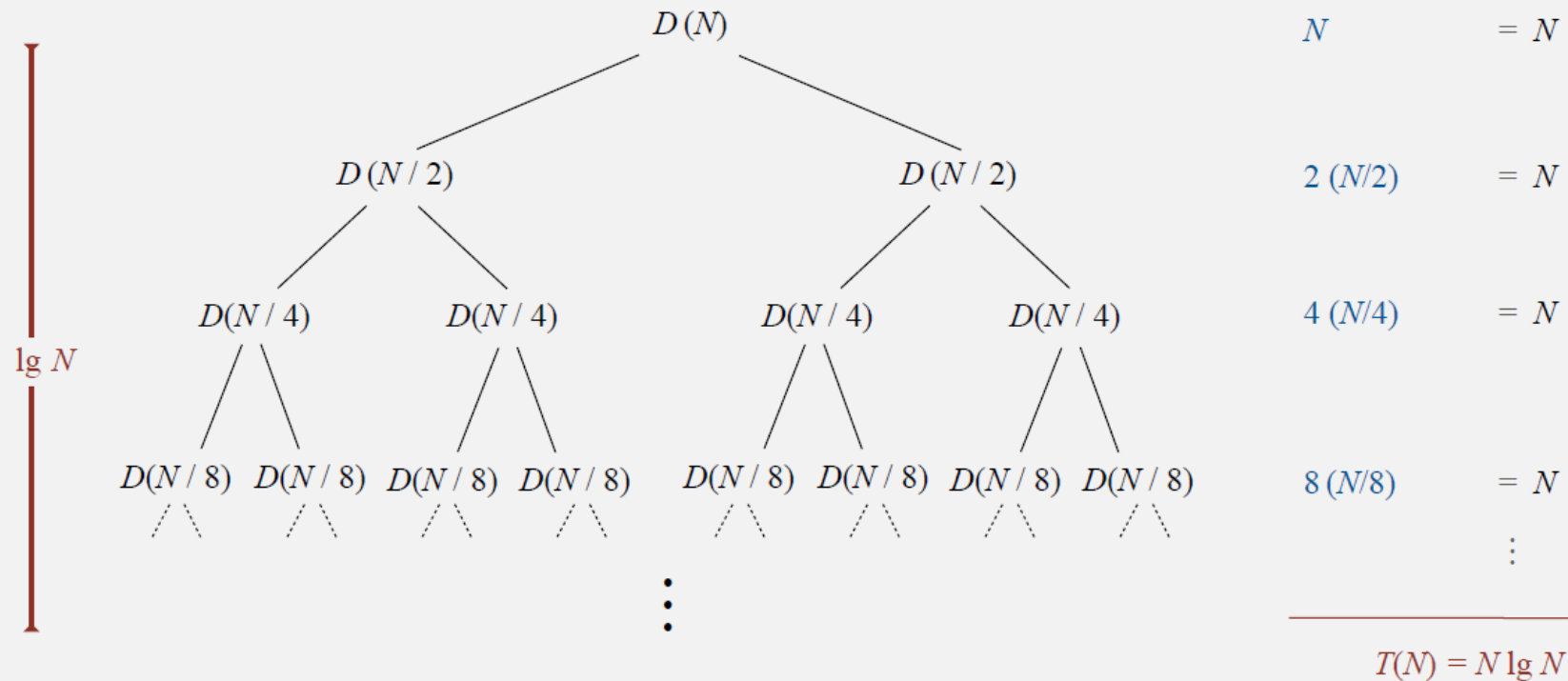
Merge sort

- › Number of compares $< N \lg N$
 - Linearithmic
 - Both average and worst
 - Stable – use “less than” - favours left hand value to right hand one even when they’re equal
- › Number of array accesses $< 6 N \lg N$
- › Memory use – auxiliary array of size N
- › Proofs in Sedgewick

Merge sort

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]



Merge sort

Key point. Any algorithm with the following structure takes $N \log N$ time:

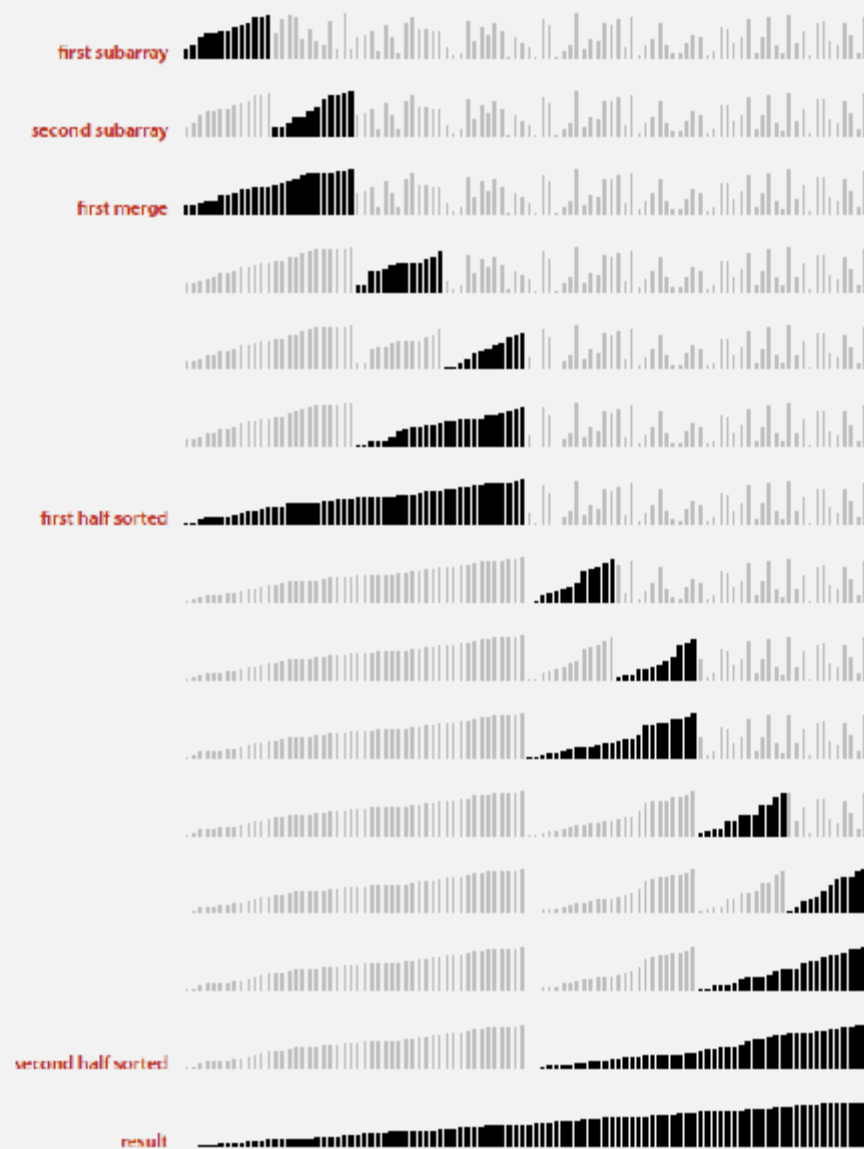
```
public static void linearithmic(int N)
{
    if (N == 0) return;
    linearithmic(N/2); ← solve two problems
    linearithmic(N/2); ← of half the size
    linear(N); ← do a linear amount of work
}
```

Merge sort improvement

- › Too much overhead for small subarrays
- › Cut off to insertion sort for ~10 items

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Mergesort with cutoff to insertion sort: visualization



Merge sort further improvements

- › Stop if already sorted
 - Is largest item in first half smaller than smallest in second half

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (a, aux, lo, mid);
    sort (a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```


Merge sort further improvements

- › Eliminate the time (but not the space) taken to copy to the auxiliary array used for merging
- › Use two invocations of the sort method
 - one that takes its input from the given array and puts the sorted output in the auxiliary array
 - the other takes its input from the auxiliary array and puts the sorted output in the given array.

Merge sort further improvements

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)      aux[k] = a[j++];
        else if (j > hi)  aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else              aux[k] = a[i++];
    }
}
```

← merge from a[] to aux[]

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(aux, a, lo, mid);
    sort(aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

↑
assumes aux[] is initialize to a[] once,
before recursive calls

↑
switch roles of aux[] and a[]

Bottom up mergesort

Merge sort bottom up

- › Pass through array merging subarrays of size 1, 2, 4, etc

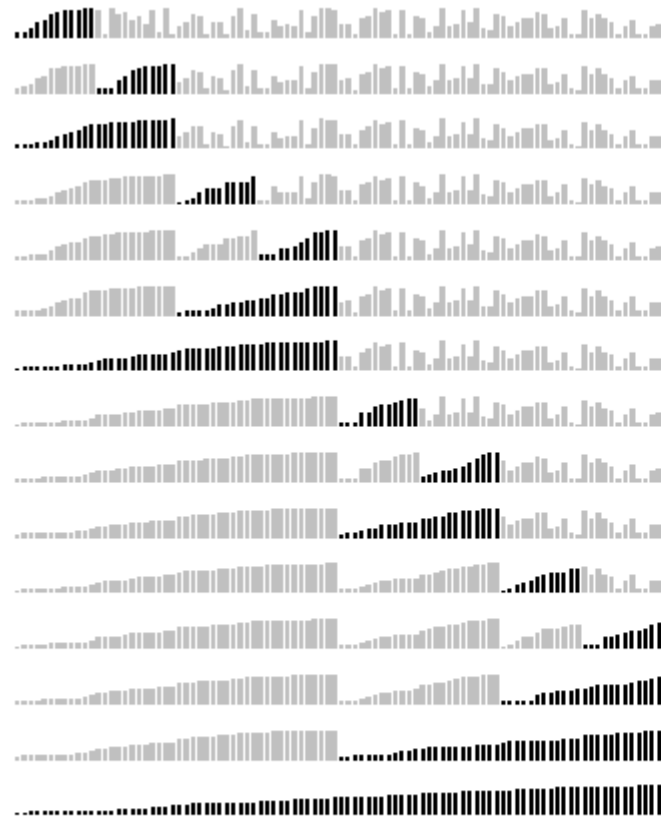
```
public class MergeBU
{
    private static void merge(...)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int N = a.length;
        Comparable[] aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

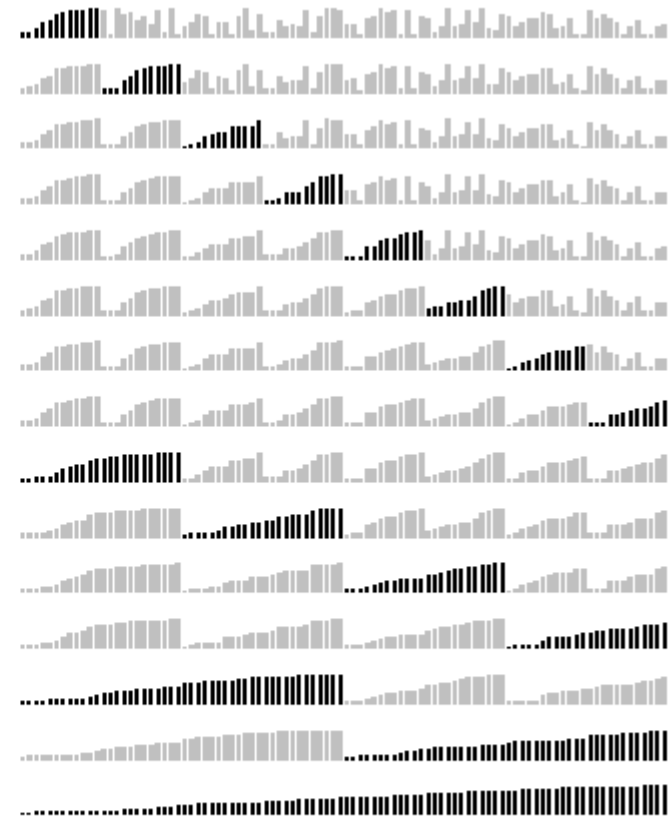
Merge sort bottom up

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

Merge sort top down vs bottom up



top-down mergesort (cutoff = 12)



bottom-up mergesort (cutoff = 12)

Timsort

- › adaptive sort, combination of
 - natural merge sort – exploit pre-existing order by identifying naturally occurring non-descending sequences (so ascending or equal) - Look for at least 2 elements
 - Use insertion sort to make initial runs
- › Java 7 (for non primitive data types), Python, Android

Quicksort

Quicksort

- › One of top 10 algorithms of 20th century in science and engineering
 - “the greatest influence on the development and practice of science and engineering in the 20th century”
 - <https://www.computer.org/csdl/mags/cs/2000/01/c1022.html>
 - “one of the best practical sorting algorithm for general inputs”
 - inspiration for developing general algorithm techniques for various applications

Quicksort

- › Invented by Tony Hoare in 1959
 - Visiting student in Russia, needed to sort the words before looking them up in dictionary
 - Insert sort was too slow so he developed quicksort, but couldn't implement it until learnt ALGOL and its ability to do recursion
- › Further improvements
 - Sedgwick, Bentley, Yaroslavskiy
 - Dual-pivot implementation in 2009, now implemented in Java 7 onwards

Quicksort

1. Shuffle the array $a[]$ (we'll talk later why)
2. Partition the array so that, for some j
 - $a[j]$ is in place (called pivot)
 - There is nothing larger than $a[j]$ to the left of it
 - There is nothing smaller to the right of it
(where does equal go?)
3. Sort each subarray recursively

Quicksort

› To sort an array S

1. If the number of elements in S is 0 or 1, then return. The array is sorted.
2. Pick an element v in S . This is the *pivot* value.
3. Partition $S - \{v\}$ into two disjoint subsets, $S_1 = \{\text{all values } x \leq v\}$, and $S_2 = \{\text{all values } x \geq v\}$.
4. Return $\text{QuickSort}(S_1), v, \text{QuickSort}(S_2)$

Quicksort example

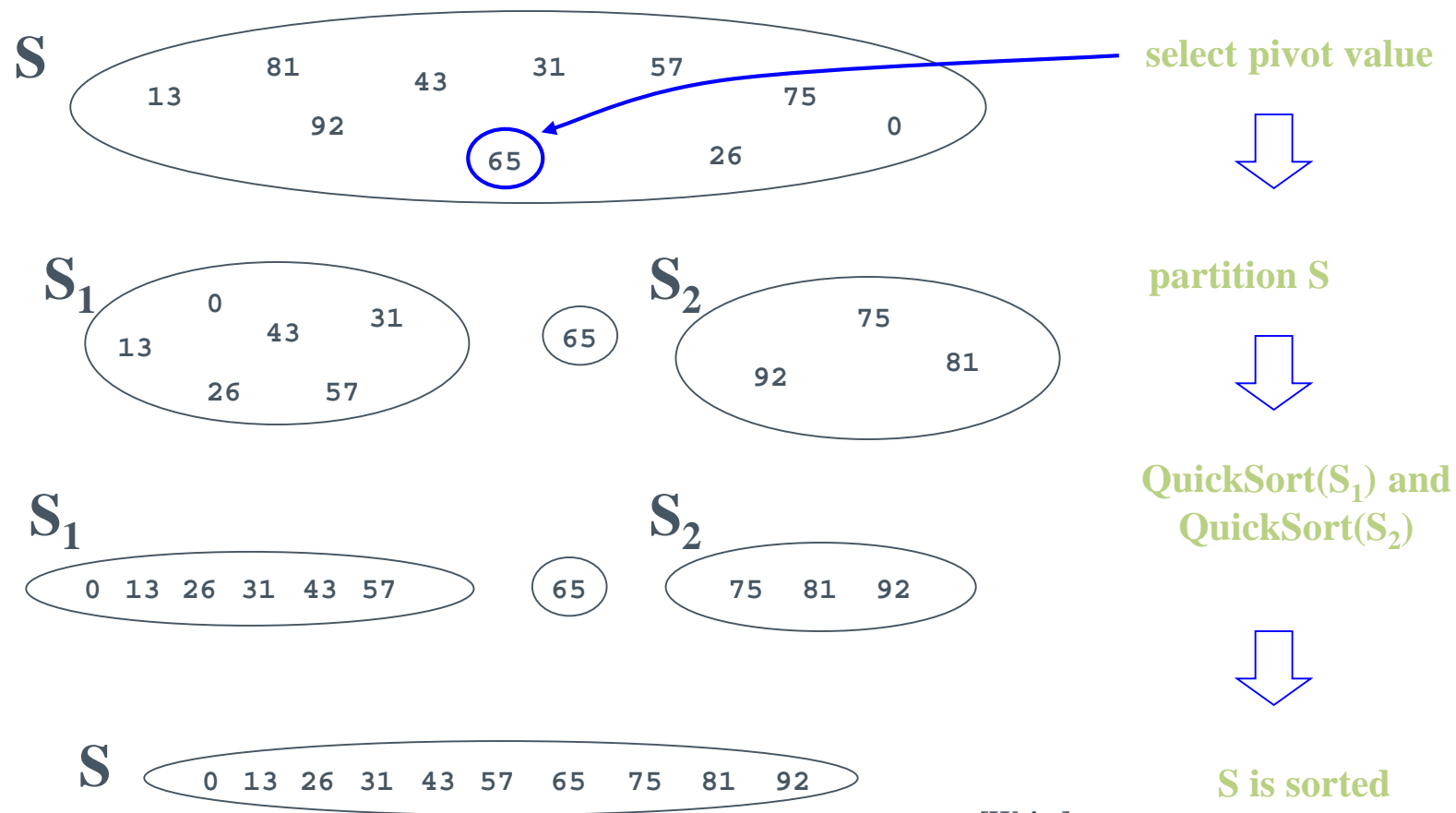
input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	← R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

partitioning item

not greater

not less

Quicksort example



[Weiss]

Quicksort - details

- › Implement partitioning
 - › recursive
- › Pick a pivot
 - › want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

Quicksort – partitioning

- › Need to partition the array into left and right sub-arrays
 - the elements in left sub-array are \leq pivot
 - elements in right sub-array are \geq pivot
- › How do the elements get to the correct partition?
 - Choose an element from the array as the pivot
 - Make one pass through the rest of the array and swap as needed to put elements in partitions

Quicksort – picking a pivot

- Ideally median value
 - › Expensive, calculating median
 - › Approximate: choose a median of first, middle and last values
- Choose pivot randomly
 - › Need a random number generator
- Choose the first element
 - › Ok if array shuffled, bad if array sorted – worst case for quicksort

Quicksort – in-place partitioning

- › If we use an extra array, partitioning is easy to implement, but not so much easier that it is worth the extra cost of copying the partitioned version back into the original.
- › Partition in-place

Quicksort – in-place partitioning example

Repeat until i and j pointers cross.

- Scan i from left to right so long as $(a[i] < a[l_o])$.
- Scan j from right to left so long as $(a[j] > a[l_o])$.
- Exchange $a[i]$ with $a[j]$.



Quicksort – in-place partitioning example

	i	j	a[]															
			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

Partitioning trace (array contents before and after each exchange)

Quicksort – partition code

```
private int partition(Comparable[] numbers, int lo, int hi) {  
    int i = lo;  
    int j = hi+1;  
    Comparable pivot = numbers[lo];  
    while(true) {  
        while((numbers[++i].compareTo(pivot) < 0)) {  
            if(i == hi) break;  
        }  
        while((pivot.compareTo(numbers[--j]) < 0)) {  
            if(j == lo) break;  
        }  
        if(i >= j) break;  
        Comparable temp = numbers[i];  
        numbers[i] = numbers[j];  
        numbers[j] = temp;  
    }  
    numbers[lo] = numbers[j];  
    numbers[j] = pivot;  
    return j;  
}
```

Quicksort – example

- › Partitioning one array – need to do this recursively on the array left of j and right of j

	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 – recursive code

```
public void sort(Comparable[] numbers) {  
    recursiveQuick(numbers, 0, numbers.length-1);  
}  
  
public void recursiveQuick(Comparable[] numbers, int lo, int hi) {  
    if(hi <= lo) {  
        return;  
    }  
    int pivotPos = partition(numbers, lo, hi);  
    recursiveQuick(numbers, lo, pivotPos-1);  
    recursiveQuick(numbers, pivotPos+1, hi);  
}
```


Quicksort – iterative version?

- › With the help of auxiliary stack

Quicksort – performance

- › How many compares to partition the array of length N ?
 - › How many recursive calls? – depth of recursion
 - › Best case analysis – for shuffled elements?
 - › Worst case analysis – for sorted elements?
-
- › TurningPoint polls

Quicksort – best case analysis

What is the number of compares?

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort – worst case analysis

What is the number of compares?

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort

- › Make sure to always avoid worst case performance by shuffling the array at the start!
- › Alternatively – pick a random pivot in each subarray
- › Quicksort is therefore a randomized algorithm
 - Uses random numbers to decide what to do next somewhere in its logic

Quicksort – performance

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (n^2)			mergesort ($n \log n$)			quicksort ($n \log n$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Quicksort - performance

Average case. Expected number of compares is $\sim 1.39 n \lg n$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Maths in Sedgwick

Quicksort – properties summary

- › Not stable because of long distance swapping.
- › No iterative version (without using a stack).
- › Pure quicksort not good for small arrays.
- › “In-place”, but uses auxiliary storage because of recursive call ($O(\log n)$ space).
- › $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

Quicksort improvements

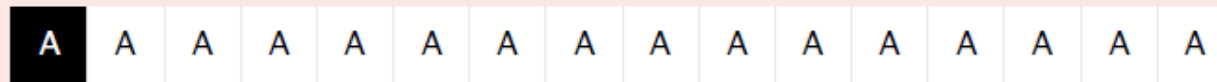
- › Use insertion sort for small arrays
 - Cut off to insertion sort at subarray size ~ 10
- › Use median for pivot value (median of 3 random items, ie first, last, middle)
- › 3-way quicksort, dual pivot, 3-pivot

Quicksort – all items the same

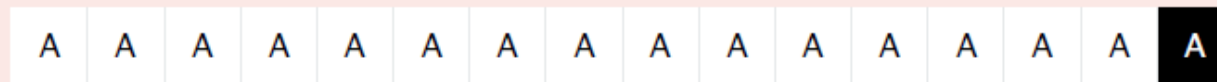
What is the result of partitioning the following array (stop on equal keys)?



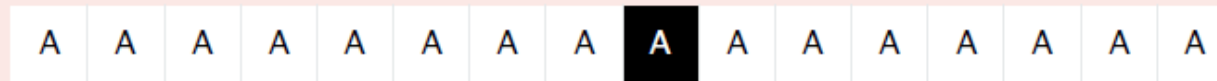
A.



B.



C.

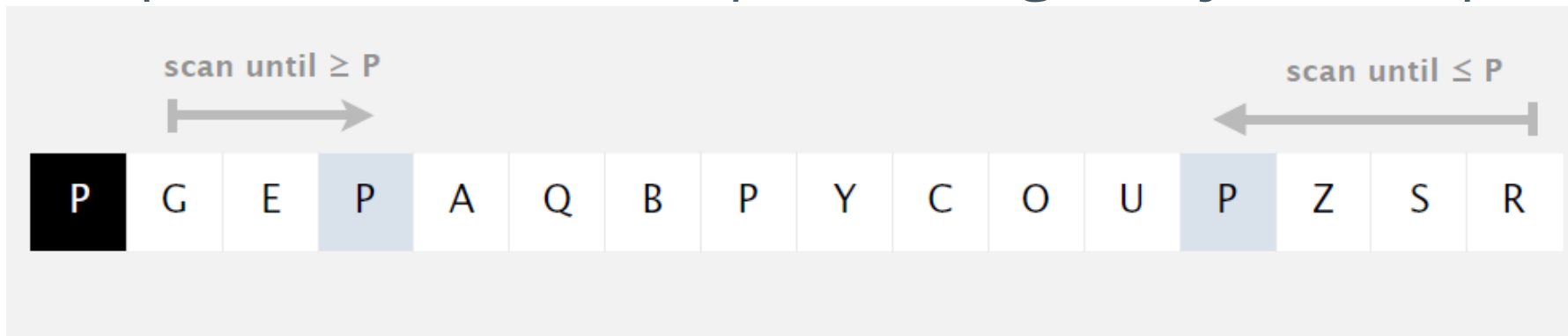


D.

I don't know.

Quicksort – stop at equal keys

- › `qsort()` in C bug reported in 1991 – “unbearably slow” for organ-pipe inputs (eg “01233210”)
 - In implementations and textbooks until then
- › N^2 time to sort organ-pipe inputs, and random arrays of 0s and 1s
- › Improvement now: stop scanning if keys are equal



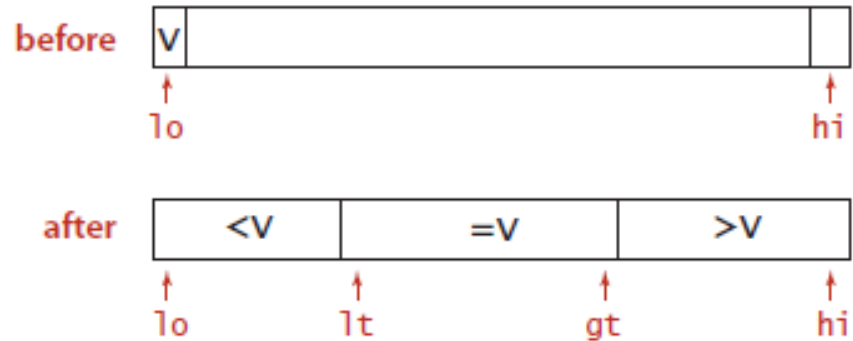
Quicksort – stop at equal keys

- › Problem – if all items equal to pivot are moved to one side of it
 - Consequence $\sim 1/2 n^2$ compares when all keys are equal
- › Stop when keys are equal
 - If all keys are equal, divides the array exactly
 - Why not put all items that are the same as partition item in place? 3-way partitioning

3-way partitioning

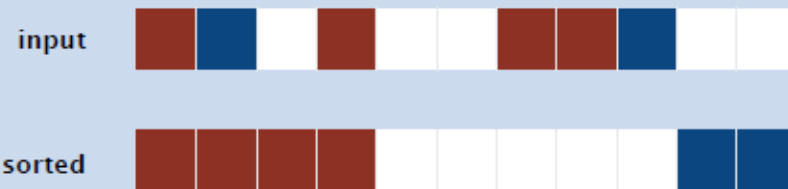
Goal. Partition array into **three** parts so that:

- Entries between **lt** and **gt** equal to the partition item.
- No larger entries to left of **lt**.
- No smaller entries to right of **gt**.



Dutch national flag problem

Problem. [Edsger Dijkstra] Given an array of n buckets, each containing a red, white, or blue pebble, sort them by color.



Operations allowed.

- $swap(i, j)$: swap the pebble in bucket i with the pebble in bucket j .
- $color(i)$: color of pebble in bucket i .

Requirements.

- Exactly n calls to $color()$.
- At most n calls to $swap()$.
- Constant extra space.

3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - ($a[i] < v$): exchange $a[lt]$ with $a[i]$; increment both lt and i
 - ($a[i] > v$): exchange $a[gt]$ with $a[i]$; decrement gt
 - ($a[i] == v$): increment i



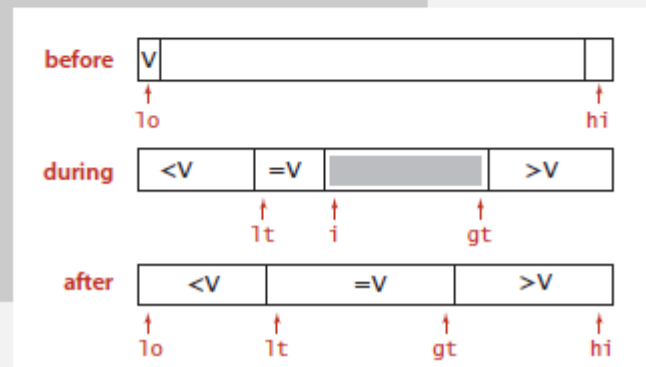
Demo <https://algs4.cs.princeton.edu/lectures/23DemoPartitioning.pdf>

3-way partitioning

- › Improves quick sort when there are duplicate keys

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }

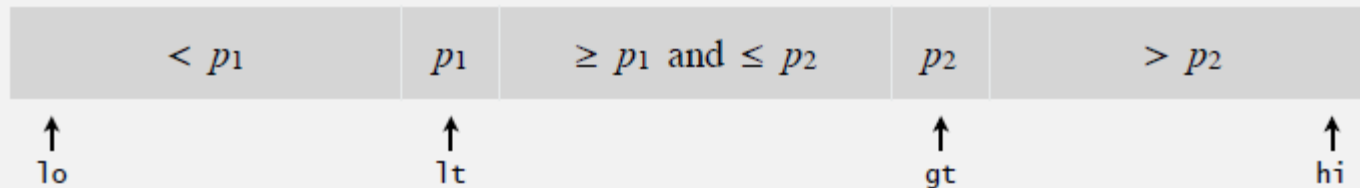
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```



2-pivot quick sort

Use **two** partitioning items p_1 and p_2 and partition into three subarrays:

- Keys less than p_1 .
- Keys between p_1 and p_2 .
- Keys greater than p_2 .



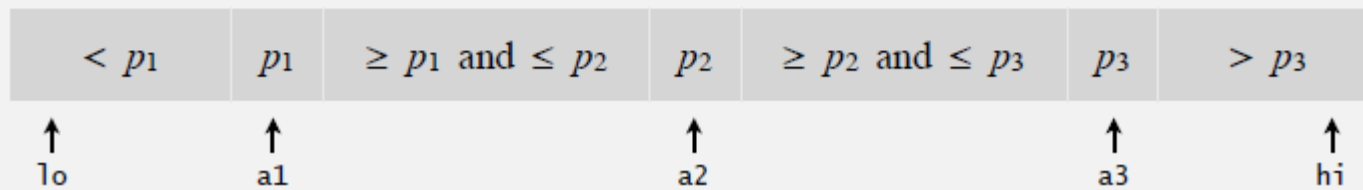
Recursively sort three subarrays.

3-pivot quick sort

Three-pivot quicksort

Use **three** partitioning items p_1 , p_2 , and p_3 and partition into four subarrays:

- Keys less than p_1 .
- Keys between p_1 and p_2 .
- Keys between p_2 and p_3 .
- Keys greater than p_3 .



Demos

- › <https://algs4.cs.princeton.edu/lectures/23DemoPartitioning.pdf>
- › Quicksort
- › 3-way partitioning
- › Dual pivot partitioning

Quicksort – cache improvements

- › Principle of locality
 - the same values, or related storage locations, are frequently accessed
 - Temporal locality
 - › If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future
 - Spatial locality
 - › If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future -> pre-fetch arrays
 - Predictability of memory access
 - Implications for caching
 - › cache – stores data “nearer” to processor so that it can be accessed quicker in the future
- › 2-pivot and 3-pivot have smaller number of cache misses and smaller number of recursive calls to a subproblem larger than the size of a cache block
- › Multi-Pivot Quicksort: Theory and Experiments - by Kushagra, López-Ortiz, Munro, and Qiao
 - Original paper - <http://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.6>
 - Discussion: <https://cs.stanford.edu/~rishig/courses/ref/l11a.pdf>

Caching improvements

Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

A. ~~Fewer compares.~~

B. ~~Fewer exchanges.~~

C. Fewer cache misses.

entries scanned is a good proxy
for cache performance when
comparing quicksort variants

partitioning	compares	exchanges	entries scanned
1-pivot	$2 n \ln n$	$0.333 n \ln n$	$2 n \ln n$
median-of-3	$1.714 n \ln n$	$0.343 n \ln n$	$1.714 n \ln n$
2-pivot	$1.9 n \ln n$	$0.6 n \ln n$	$1.6 n \ln n$
3-pivot	$1.846 n \ln n$	$0.616 n \ln n$	$1.385 n \ln n$

Reference: Analysis of Pivot Sampling in Dual-Pivot Quicksort by Wild-Nebel-Martínez

Bottom line. Caching can have a significant impact on performance.

Merge vs quick

- › In Java, `Arrays.sort()` uses **QuickSort** for sorting primitives and **MergeSort** for sorting Arrays of Objects. This is because, merge sort is stable, so it won't reorder elements that are equal.
 - Why does it matter for Objects and not for primitive data types?
- › QuickSort in java
 - 2-pivot since 2009
- › MergeSort in java
 - Timsort

Sort algorithms summary

- › Use system sort - `Arrays.sort()`; - usually good enough
- › What to consider when picking an algorithm?

Compare performance to system sort in your assignment?

Comparator interface

- › Comparable interface
 - Uses natural order to compare things
 - Can override method `compareTo()` if want custom-defined criteria
- › But what if we have Objects we want to compare according to multiple custom-defined criteria?
- › Comparator interface
 - Can create multiple classes implementing Comparator and override compare method
 - Custom ordering
 - To use with system sort, pass as a second argument to `Array.sort(a, new MyCustomOrder());`

Sorting algorithms summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✓		$n \log_3 n$?	$c n^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✓	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✓		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✓		$3 n$	$2 n \lg n$	$2 n \lg n$	$n \log n$ guarantee; in-place
?	✓	✓	n	$n \lg n$	$n \lg n$	holy sorting grail

Quick algorithms exercise

› Which algorithm would work best to sort data as it arrives, one piece at a time, perhaps from a network?

1. Mergesort
2. Selection sort
3. Quicksort
4. Insertion sort

Another quick question

› Which algorithm would you use to sort 1 million of 32-bit integers?

1. Mergesort
2. Selection sort
3. Quicksort
4. Insertion sort
5. None of the above

https://www.youtube.com/watch?v=k4RRi_ntQc8