# Mergesort and Quicksort

Reference: Chapters 7-8, Algorithms in Java, 3rd Edition, Robert Sedgewick.



## Mergesort and Quicksort

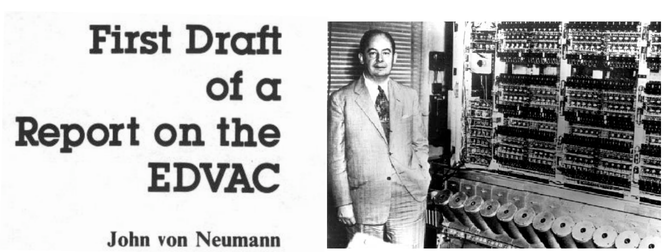
Two great sorting algorithms.
- Full scientific understanding of their properties has enabled us to hammer them into practical system sorts.
- Occupies a prominent place in world's computational infrastructure.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.
- Java sort for objects.
- Perl stable, Python stable.

Quicksort.
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Perl, Python.



# Mergesort






## Mergesort

Mergesort.
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

*input*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |

*sort left*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |

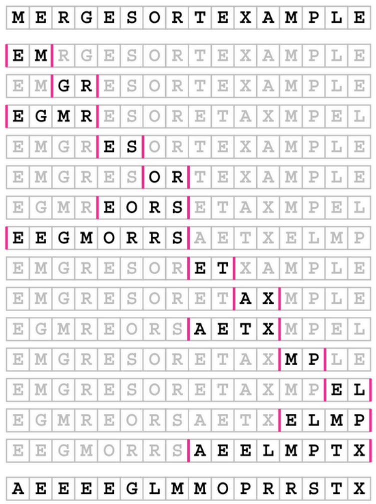
*sort right*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |

*merge*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

## Mergesort: Example

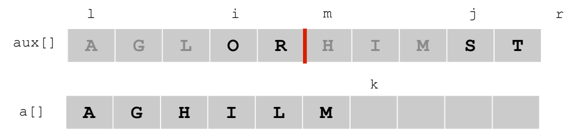




## Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently? Use an auxiliary array.



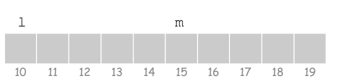

```
for (int k = l; k < r; k++)  aux[k] = a[k];
int i = l, j = m;
for (int k = l; k < r; k++) {
   if      (i >= m)                  a[k] = aux[j++];
   else if (j >= r)                  a[k] = aux[i++];
   else if (less(aux[j], aux[i]))    a[k] = aux[j++];
   else                              a[k] = aux[i++];
}
```



## Mergesort: Java Implementation

```
public class Merge {
   private static void sort(Comparable[] a,
                            Comparable[] aux, int l, int r) {
      if (r <= l + 1) return;
      int m = l + (r - l) / 2;
      sort(a, aux, l, m);
      sort(a, aux, m, r);
      merge(a, aux, l, m, r);
   }

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






## Mergesort Analysis: Memory

Q. How much memory does mergesort require?

- Original input array = N.
- Auxiliary array for merging = N.
- Local variables: constant.
- Function call stack: $\log_2 N$.
- Total = $2N + O(\log N)$.

Q. How much memory do other sorting algorithms require?

- $N + O(1)$ for insertion sort and selection sort.
- In-place = $N + O(\log N)$.

Challenge for the bored. In-place merge. [Kronrud, 1969]

## Mergesort Analysis: Running Time

Def. T(N) = number of comparisons to mergesort an input of size N.

Mergesort recurrence.

$$T(N) \le \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{T(\lceil N/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor N/2 \rfloor)}_{\text{solve right half}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$

including already sorted

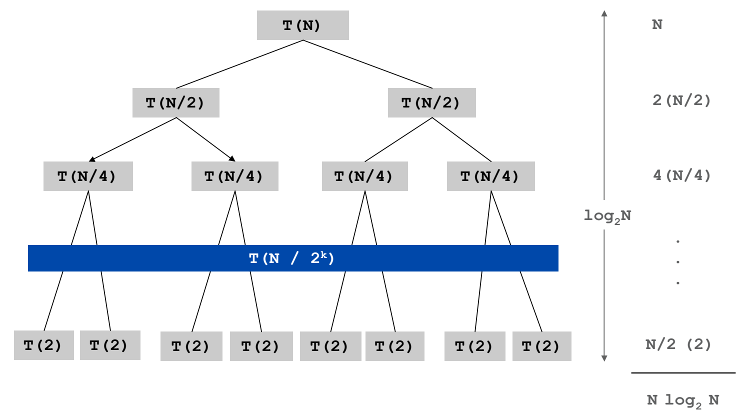
Solution. $T(N) = O(N \log_2 N)$.

- Note: same number of comparisons for any input of size N.
- We prove $T(N) = N \log_2 N$ when N is a power of 2, and = instead of ≤.



## Proof by Recursion Tree

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$






## Proof by Induction

Claim. If T(N) satisfies this recurrence, then $T(N) = N \log_2 N$.

assumes N is a power of 2

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. [by induction on n]

- Base case: n = 1.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$



## Mergesort: Practical Improvements

Use sentinel. Two statements in inner loop are array-bounds checking.

Use insertion sort on small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 elements.

Stop if already sorted.

- Is biggest element in first half ≤ smallest element in second half?
- Helps for nearly ordered lists.

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

## Sorting Analysis Summary

Running time estimates:

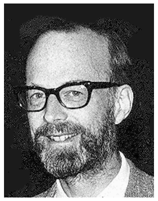
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

| | Insertion Sort ($N^2$) | | | Mergesort (N log N) | | |
|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 sec | 18 min |
| super | instant | 1 second | 1.6 weeks | instant | instant | instant |

Lesson 1. Good algorithms are better than supercomputers.



# Quicksort



Sir Charles Antony Richard Hoare
1980 Turing Award

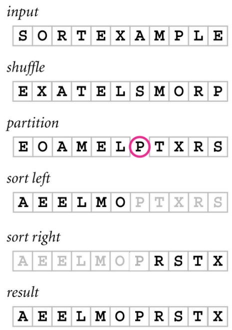


## Quicksort

Quicksort.

- Shuffle the array.
- Partition array so that:
  - element `a[i]` is in its final place for some `i`
  - no larger element to the left of `i`
  - no smaller element to the right of `i`
- Sort each piece recursively.

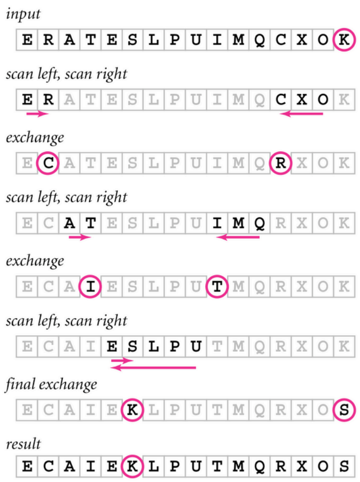
Q. How do we partition in-place efficiently?

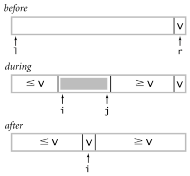



*Quicksort*



## Quicksort Partitioning




*Partitioning for quicksort*

## Quicksort Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | U | I | C | K | S | O | R | T | E | X | A | M | P | L | E |
| E | R | A | T | E | S | L | P | U | I | M | Q | C | X | O | K |
| E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| A | C | E | E | I | K | L | P | O | R | M | Q | S | X | U | T |
| A | C | E | E | I | K | L | P | O | M | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | X | U | T |
| A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |



## Quicksort: Java Implementation

```java
public class Quick {

   public static void sort(Comparable[] a) {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int l, int r) {
      if (r <= l) return;
      int m = partition(a, l, r);
      sort(a, l, m-1);
      sort(a, m+1, r);
   }
```



## Quicksort: Java Implementation

```java
private static int partition(Comparable[] a, int l, int r) {
   int i = l - 1;
   int j = r;

   while(true) {

      while (less(a[++i], a[r]))      find item on left to swap
         if (i == r) break;

      while (less(a[r], a[--j]))      find item on right to swap
         if (j == l) break;

      if (i >= j) break;              check if pointers cross
      exch(a, i, j);                  swap
   }

   exch(a, i, r);                     swap with partitioning element
   return i;                          return index where crossing occurs
}
```



## Quicksort Implementation Details

Partitioning in-place. Using a spare array makes partitioning easier, but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The `(i == r)` test is redundant, but the `(j == l)` test is not.

Preserving randomness. Shuffling is key for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to partitioning element.

## Quicksort: Performance Characteristics

Worst case. Number of comparisons is quadratic.

- N + (N-1) + (N-2) + … + 1 ≈ $N^2$ / 2.
- More likely that your computer is struck by lightning.

Caveat. Many textbook implementations go quadratic if input:

- Is sorted.
- Is reverse sorted.
- Has many duplicates.



## Quicksort: Average Case

Average case running time.

- Roughly 2 N ln N comparisons. ← see next two slides
- Assumption: file is randomly shuffled.

Remarks.

- 39% more comparisons than mergesort.
- Faster than mergesort in practice because of lower cost of other high-frequency instructions.
- Caveat: many textbook implementations have best case $N^2$ if duplicates, even if randomized!



## Quicksort: Average Case

Theorem. The average number of comparisons $C_N$ to quicksort a random file of N elements is about 2N ln N.

- The precise recurrence satisfies $C_0 = C_1 = 0$ and for N ≥ 2:

$$\begin{aligned} C_N &= N+1+\tfrac{1}{N}\sum_{k=1}^{N}\left(C_k + C_{N-k}\right) \\ &= N+1+\tfrac{2}{N}\sum_{k=1}^{N} C_{k-1} \end{aligned}$$

- Multiply both sides by N and subtract the same formula for N-1:

$$N C_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

- Simplify to:

$$NC_N = (N+1)\,C_{N-1} + 2N$$



## Quicksort: Average Case

- Divide both sides by N(N+1) to get a telescoping sum:

$$\begin{aligned} \frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \vdots \\ &= \frac{C_2}{3} + \sum_{k=3}^{N} \tfrac{2}{k+1} \end{aligned}$$

- Approximate the exact answer by an integral:

$$\frac{C_N}{N+1} \approx \sum_{k=1}^{N} \tfrac{2}{k} \approx \int_{k=1}^{N} \tfrac{2}{k} = 2\ln N$$

- Finally, the desired result:

$$C_N \approx 2(N+1)\ln N \approx 1.39 N \log_2 N. \quad \blacksquare$$

## Sorting Analysis 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 |

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.



## Quicksort: Practical Improvements

Median of sample.

- Best choice of pivot element = median.
- But how would you compute the median?
- Estimate true median by taking median of sample.

Insertion sort small files.

- Even quicksort has too much overhead for tiny files.
- Can delay insertion sort until end.

Optimize parameters. $\approx$ 12/7 N log N comparisons

- Median-of-3 random elements.
- Cutoff to insertion sort for $\approx$ 10 elements.

Non-recursive version.

- Use explicit stack.
- Always sort smaller half first. guarantees O(log N) stack size



# 3-Way Quicksort



## Duplicate Keys

Equal keys. Omnipresent in applications when purpose of sort is to bring records with equal keys together.

- Sort population by age.
- Finding collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

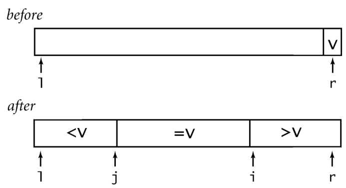
Typical application.

- Huge file.
- Small number of key values.

## 3-Way Partitioning

3-way partitioning. Partition elements into 3 parts:

- Elements between `i` and `j` equal to partition element `v`.
- No larger elements to left of `i`.
- No smaller elements to right of `j`.




Dutch national flag problem.

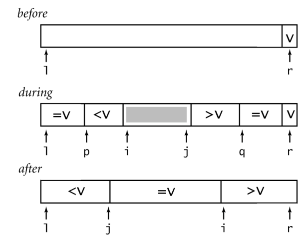
- Not done in practical sorts before mid-1990s.
- Incorporated into Java system sort, C qsort.



## Dutch National Flag Problem: Solution

Solution to Dutch national flag problem.

- Partition elements into 4 parts:
  - no larger elements to left of `i`
  - no smaller elements to right of `j`
  - equal elements to left of `p`
  - equal elements to right of `q`
- Afterwards, swap equal keys into center.




All the right properties.

- In-place.
- Not much code.
- Linear if keys are all equal.
- Small overhead if no equal keys.



## 3-way Quicksort: Java Implementation

```java
private static void sort(Comparable[] a, int l, int r) {
   if (r <= l) return;
   int i = l-1, j = r;
   int p = l-1, q = r;

   while(true) {                                       4-way partitioning
      while (less(a[++i], a[r])) ;
      while (less(a[r], a[--j])) if (j == l) break;
      if (i >= j) break;
      exch(a, i, j);
      if (eq(a[i], a[r])) exch(a, ++p, i);
      if (eq(a[j], a[r])) exch(a, --q, j);
   }                                    swap equal keys to left or right
   exch(a, i, r);

   j = i - 1;                           swap equal keys back to middle
   i = i + 1;
   for (int k = l  ; k <= p; k++) exch(a, k, j--);
   for (int k = r-1; k >= q; k--) exch(a, k, i++);

   sort(a, l, j);        recursively sort left and right piece
   sort(a, i, r);
}
```



## Duplicate Keys

Theorem. [Sedgewick-Bentley] Quicksort with 3-way partitioning is optimal for random keys with duplicates.
Pf. Ties cost to entropy. Beyond scope of 226.

Practice. Randomized 3-way quicksort is linear time when many duplicates. (Try it!)

# Selection



## Selection

Selection. Find the $k^{th}$ largest element.

- Min: k = 1.
- Max: k = N.
- Median: k = N/2.

Application. Order statistics.

Easy. Min or max with O(N) comparisons; median with O(N log N).
Challenge. O(N) comparisons for any k.



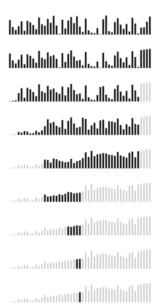
## Selection

Quick select.

- Partition array so that:
  - element `a[i]` is in its final place for some `i`
  - no larger element to the left of `i`
  - no smaller element to the right of `i`
- Repeat in one subarray, depending on `i`.

```
public static void select(Comparable[] a, int k) {
   StdRandom.shuffle(a);
   int l = 0;
   int r = a.length - 1;
   while (r > l) {
      int i = partition(a, l, r);
      if      (i > k) r = i - 1;
      else if (i < k) l = i + 1;
      else return;
   }
}
```

upon termination, `a[k]` contains k+1st smallest element





## Quick-Select Analysis

Property C. Quick-select takes linear time on average.

- Intuitively, each partitioning step roughly splits array in half.
- N + N/2 + N/4 + … < 2N comparisons.
- Formal analysis similar to quicksort analysis proves the average number of comparisons is

$$2N + k \ln\left(\frac{N}{k}\right) + (N-k)\ln\left(\frac{N}{N-k}\right)$$

Ex: (2 + 2 ln 2) N comparisons to find the median

Worst-case. The worst-case is $\Omega(N^2)$ comparisons, but as with quicksort, the random shuffle makes this case extremely unlikely.