QuickXsort: Efficient Sorting with $n \log n - 1.399n + o(n)$ Comparisons on Average

Stefan Edelkamp[1] Armin Weiß[2]

[1]TZI, Universität Bremen, Germany

[2]FMI, Universität Stuttgart, Germany

Moscow, June 11, 2014

Sorting

Task: Sort a sequence of elements of some totally ordered universe using only pairwise comparisons.

$$\rightsquigarrow$$

QuickXsort

Hybrid algorithm: combination of Quicksort with some other algorithm X.

- QuickHeapsort (Cantone, Cincotti, CIAC 2000)
  (improvements: Diekert, W., CSR 2013)
- QuickWeakHeapsort (Edelkamp, Stiegeler, WAE 2000)
  (improvements: see proceedings)
- QuickMergesort (this talk)

Hybrid algorithm: combination of QuickSort with some other algorithm X.

- QuickHeapsort (Cantone, Cincotti, CIAC 2000) (improvements: Diekert, W., CSR 2013)
- QuickWeakHeapsort (Edelkamp, Stiegeler, WAE 2000) (improvements: see proceedings)
- QuickMergesort (this talk)

Objectives:

- Unified analysis of the average number of comparisons.

QuickXsort

Hybrid algorithm: combination of Quicksort with some other algorithm X.

- QuickHeapsort (Cantone, Cincotti, CIAC 2000)
  (improvements: Diekert, W., CSR 2013)
- QuickWeakHeapsort (Edelkamp, Stiegeler, WAE 2000)
  (improvements: see proceedings)
- QuickMergesort (this talk)

Objectives:

- Unified analysis of the average number of comparisons.
- Pushing the limits towards optimal average case in place sorting ($n \log n + \kappa n + o(n)$ comparisons with κ as small as possible).

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	≈ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	≈ -1.24
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
Insertionsort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,

* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	$\approx$ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	$\approx$ -1.24
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
InsertionSort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,

* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	≈ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	≈ -1.24
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
InsertionSort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,
* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	$\approx$ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	$\approx$ -1.24
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
InsertionSort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,

* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

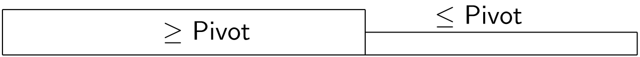
	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	$\approx$ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	$\approx$ -1.24
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
InsertionSort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,

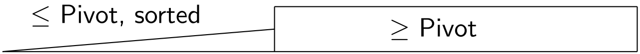
* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

QuickWeakHeapsort

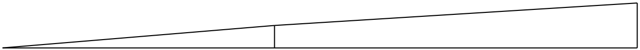
1. Partition



2. Sort one part with WeakHeapsort

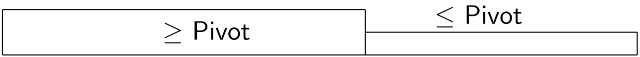


3. Sort the other part recursively with QuickWeakHeapsort

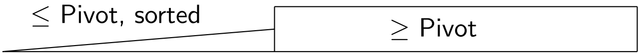


QuickWeakHeapsort

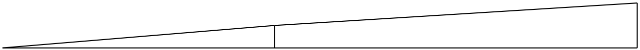
1. Partition



2. Sort one part with WeakHeapsort



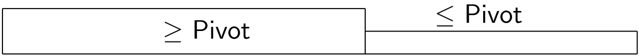
3. Sort the other part recursively with QuickWeakHeapsort



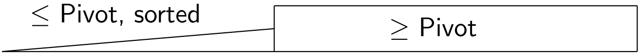
Advantage: WeakHeapsort can be implemented with less comparisons.

QuickWeakHeapsort

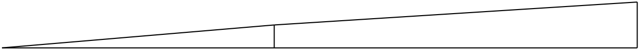
1. Partition



2. Sort one part with WeakHeapsort



3. Sort the other part recursively with QuickWeakHeapsort

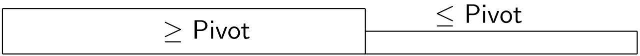


Advantage: WeakHeapsort can be implemented with less comparisons.

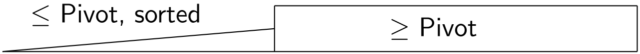
Other point of view: in-place implementation of ExternalWeakHeapsort

QuickHeapsort

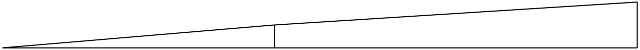
1. Partition



2. Sort one part with Heapsort

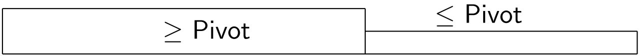


3. Sort the other part recursively with QuickHeapsort

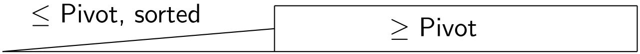


QuickHeapsort

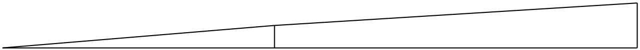
1. Partition



2. Sort one part with Heapsort



3. Sort the other part recursively with QuickHeapsort

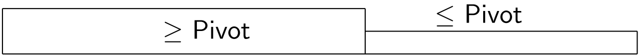


Advantage: during the sift down procedure only one comparison per level in the heap is needed.

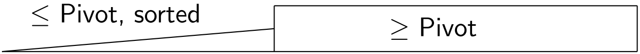
$\rightsquigarrow$ $n \log n + \mathcal{O}(n)$ comparisons on average

QuickHeapsort

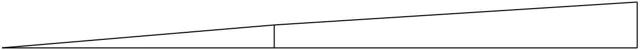
1. Partition



2. Sort one part with Heapsort



3. Sort the other part recursively with QuickHeapsort

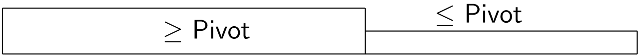


Advantage: during the sift down procedure only one comparison per level in the heap is needed.
$\rightsquigarrow$ $n \log n + \mathcal{O}(n)$ comparisons on average

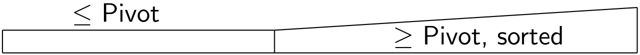
Other point of view: in-place implementation of ExternalHeapsort

QuickMergesort

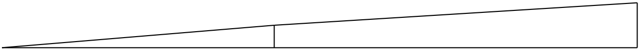
1. Partition



2. Sort one part with Mergesort

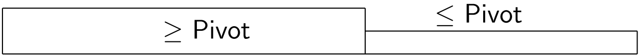


3. Sort the other part recursively with QuickMergesort

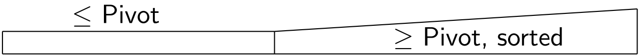


QuickMergesort

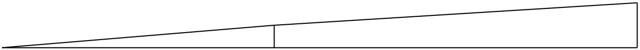
1. Partition



2. Sort one part with Mergesort



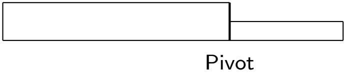
3. Sort the other part recursively with QuickMergesort



Advantage: Can be implemented in place.

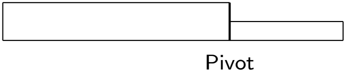
Step 2: Mergesort in place

After partitioning

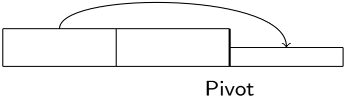


Step 2: Mergesort in place

After partitioning

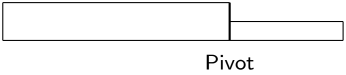


Apply MERGESORT recursively

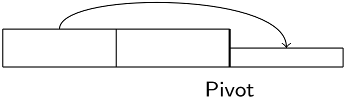


Step 2: Mergesort in place

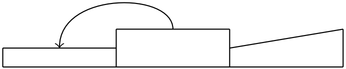
After partitioning

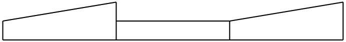


Apply MERGESORT recursively



Apply MERGESORT recursively





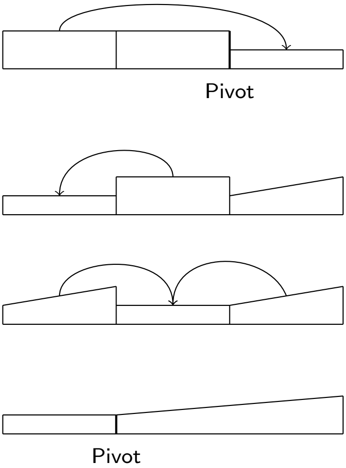
Step 2: Mergesort in place

After partitioning

Apply MERGESORT recursively

Apply MERGESORT recursively

Merge



QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

5	1	8	7	6	2	3	10	9	11	12	4

QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

5	1	8	7	6	2	3	10	9	11	12	4

QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

9	10	8	7	6	12	11	5	3	2	1	4

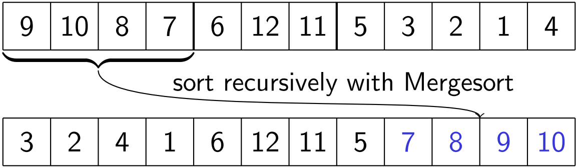
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

$$\underbrace{\boxed{9}\boxed{10}\boxed{8}\boxed{7}\boxed{6}\boxed{12}\boxed{11}}_{\text{sort with Mergesort}}\boxed{5}\boxed{3}\boxed{2}\boxed{1}\boxed{4}$$

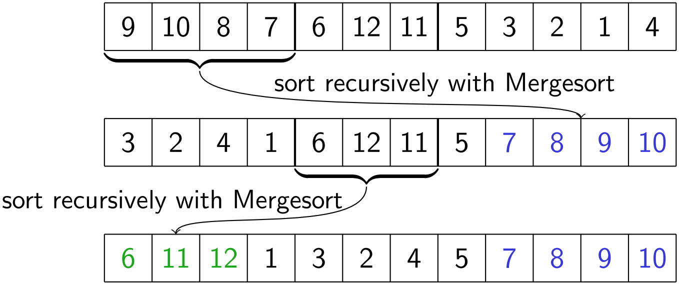
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



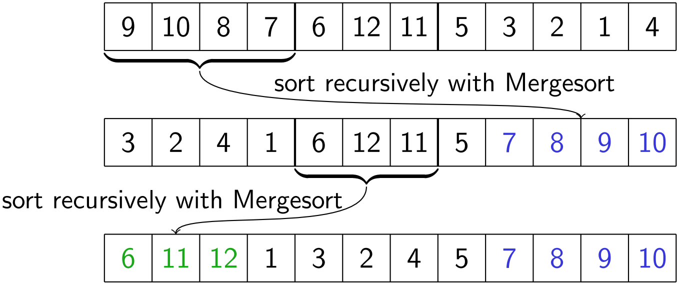
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



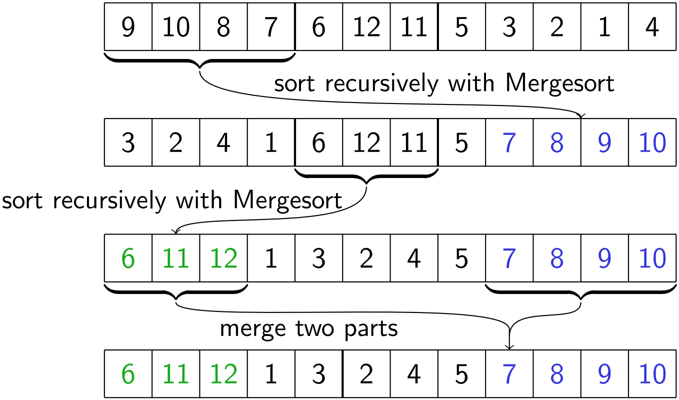
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



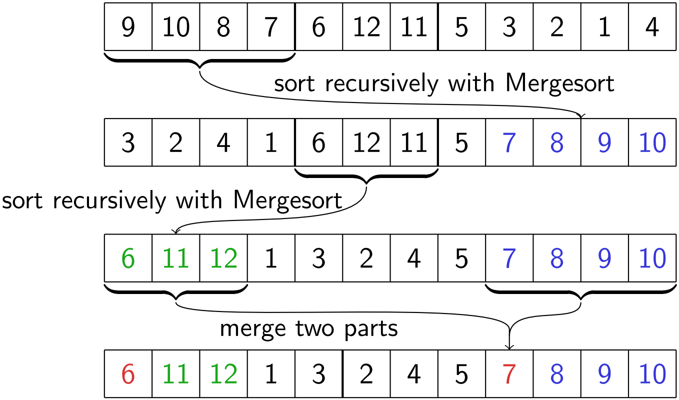
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



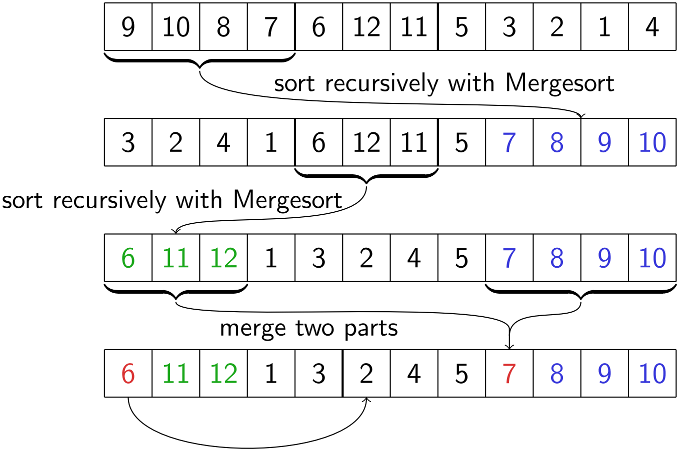
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



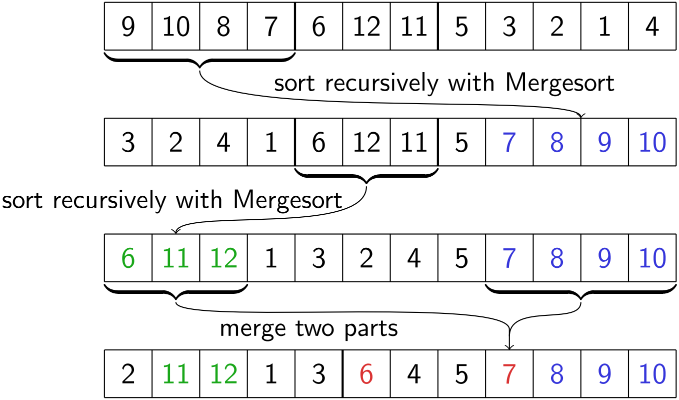
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



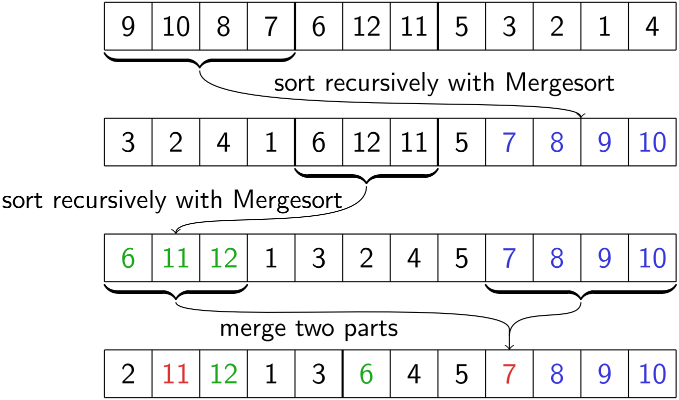
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



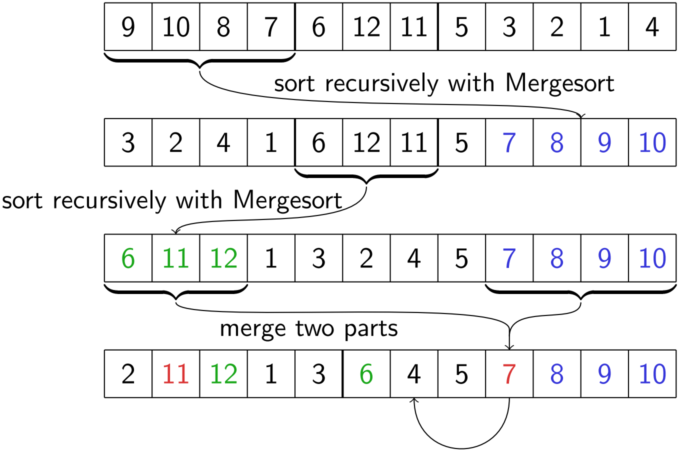
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



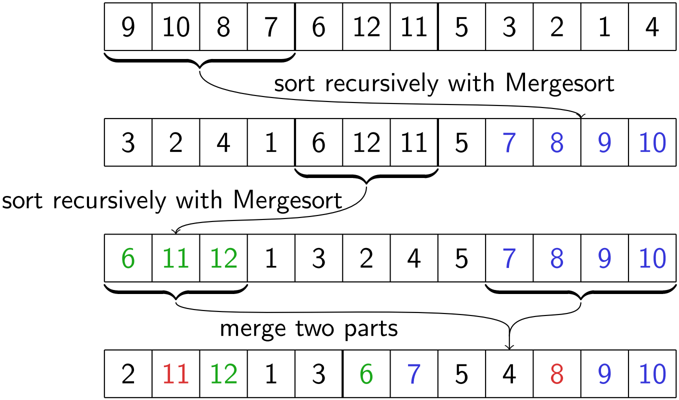
QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

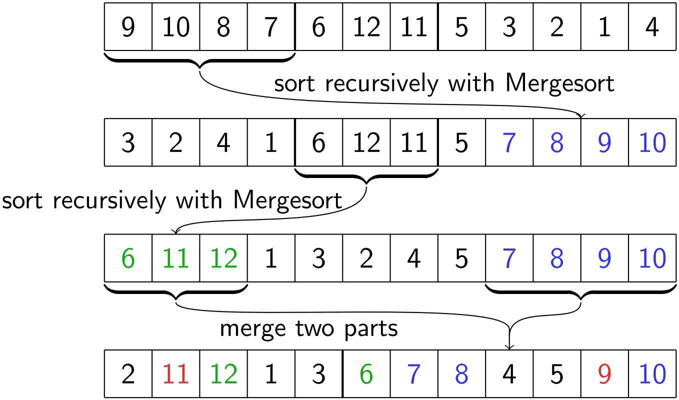


QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

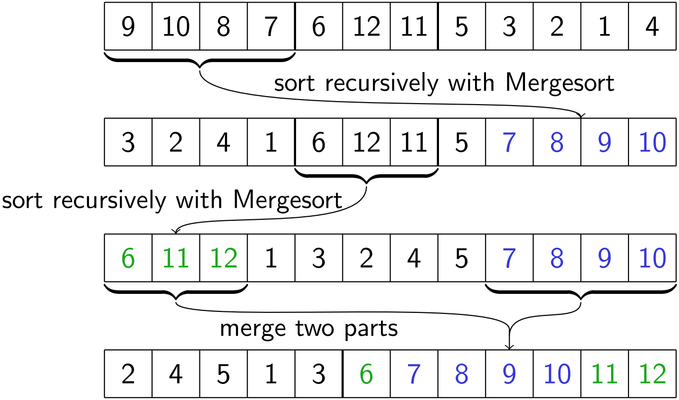


1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.



QuickMergesort

1. Partition the array according to some pivot element.
2. Sort one part with Mergesort.
3. Sort the remaining elements recursively with QuickMergesort.

9	10	8	7	6	12	11	5	3	2	1	4

sort recursively with Mergesort

3	2	4	1	6	12	11	5	7	8	9	10

sort recursively with Mergesort

6	11	12	1	3	2	4	5	7	8	9	10

merge two parts

2	4	5	1	3	6	7	8	9	10	11	12

sort recursively with QuickMergesort

Advantage:

- “External” algorithms can be implemented “internal”.

Advantage:

- "External" algorithms can be implemented "internal".

Assumptions:

- draw the pivot as median of $\sqrt{n}$ randomly selected elements
- uniform distribution of all input permutations

QuickXsort

Advantage:

- "External" algorithms can be implemented "internal".

Assumptions:

- draw the pivot as median of $\sqrt{n}$ randomly selected elements
- uniform distribution of all input permutations

Theorem (QuickXsort Average-Case)

If X is some sorting algorithm requiring at most $n \log n + cn + o(n)$ comparisons on average, then QuickXsort needs at most $n \log n + cn + o(n)$ comparisons on average.

Proof idea

General recurrence relation for the average number of comparisons: $(S(n) = n \log n + cn + o(n)$ = bound for the average number of comparisons of X)

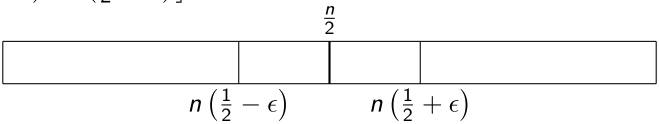
$$T(n) \le T_{\mathrm{pivot}}(n) + n + \sum_{k=1}^{n} \Big(\Pr[\text{pivot } = k] \cdot \max\{\, T(k-1) + S(n-k), T(n-k) + S(k-1) \,\} \Big)$$

Proof idea

General recurrence relation for the average number of comparisons: ($S(n) = n \log n + cn + o(n)$ = bound for the average number of comparisons of X)

$$T(n) \leq T_{\text{pivot}}(n) + n + \sum_{k=1}^{n} \Big(\Pr\left[\text{pivot } = k\right] \cdot \max\{\, T(k-1) + S(n-k), T(n-k) + S(k-1) \,\} \Big)$$

It is very unlikely that the pivot is chosen outside the interval $\left[n\left(\frac{1}{2} - \epsilon\right), n\left(\frac{1}{2} + \epsilon\right)\right]$.

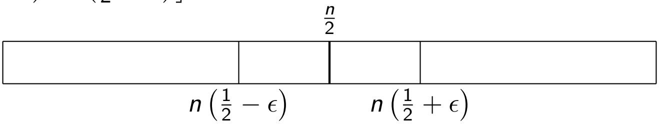


Proof idea

General recurrence relation for the average number of comparisons: ($S(n) = n \log n + cn + o(n)$ = bound for the average number of comparisons of X)

$$\begin{aligned}
T(n) &\leq T_{\text{pivot}}(n) + n + \sum_{k=1}^{n} \Big(\Pr[\text{pivot} \;= k] \\
&\qquad \cdot \max\{\, T(k-1) + S(n-k), T(n-k) + S(k-1) \,\} \Big) \\
&\approx T_{\text{pivot}}(n) + n + T(n/2) + S(n/2) \\
&= n + n\log(n/2) + cn + o(n).
\end{aligned}$$

It is very unlikely that the pivot is chosen outside the interval $\left[n\left(\frac{1}{2} - \epsilon\right), n\left(\frac{1}{2} + \epsilon\right) \right]$.



Worst case of QuickXsort

Worst case is $\Theta(n^{3/2})$.

Worst case of QuickXsort

Worst case is $\Theta(n^{3/2})$.

Trick to obtain a provable bound for the worst case (similar to Introsort (Musser, 1997)):

- Choose some slowly decreasing function $\delta(n) \in o(1) \cap \Omega(n^{-1/5})$, e. g., $\delta(n) = 1/\log n$.

Worst case of QuickXsort

Worst case is $\Theta(n^{3/2})$.

Trick to obtain a provable bound for the worst case (similar to Introsort (Musser, 1997)):

- Choose some slowly decreasing function $\delta(n) \in o(1) \cap \Omega(n^{-1/5})$, e. g., $\delta(n) = 1/\log n$.
- Whenever the pivot is more than $n \cdot \delta(n)$ off the median, choose the next pivot as median of the whole array using some linear time (worst case) algorithm.

Worst case of QuickXsort

Worst case is $\Theta(n^{3/2})$.

Trick to obtain a provable bound for the worst case (similar to Introsort (Musser, 1997)):

- Choose some slowly decreasing function $\delta(n) \in o(1) \cap \Omega(n^{-1/5})$, e. g., $\delta(n) = 1/\log n$.
- Whenever the pivot is more than $n \cdot \delta(n)$ off the median, choose the next pivot as median of the whole array using some linear time (worst case) algorithm.

Theorem (QuickXsort worst case)

Let X be a sorting algorithm with at most $n \log n + cn + o(n)$ comparisons on average and $n \log n + \mathcal{O}(n)$ comparisons in the worst case. Then QuickXsort *(with the above modification) performs at most $n \log n + cn + o(n)$ comparisons on average and $n \log n + \mathcal{O}(n)$ comparisons in the worst case.*

Corollary

QuickWeakHeapsort *performs at most* $n \log n - 0.91n + o(n)$ *comparisons on average.*

Corollary

QuickMergesort *is an internal sorting algorithm that performs at most* $n \log n - 1.26n + o(n)$ *comparisons on average.*

(See e.g. Knuth, *The Art of Computer Programming, Sorting and Searching* 5.2.4–13.)

QuickMergesort with base case

Further improvement for QuickMergesort:

- Sort small subarrays with some other algorithm Z.

QuickMergesort with base case

Further improvement for QuickMergesort:

- Sort small subarrays with some other algorithm Z.

Candidates for the base case algorithm Z:

- (Binary) InsertionSort
- MergeInsertion (Ford, Johnson, 1959)

QuickMergesort with base case

Further improvement for QuickMergesort:

- Sort small subarrays with some other algorithm Z.

Candidates for the base case algorithm Z:

- (Binary) InsertionSort
- MergeInsertion (Ford, Johnson, 1959)

Theorem

Let Z be some sorting algorithm with $n \log n + dn + o(n)$ *comparisons on average and at most* $\mathcal{O}(n^2)$ *other operations (e.g. moves). If base cases of size* $\Theta(\log n)$ *are sorted with Z,* QuickMergesort *needs at most* $n \log n + dn + o(n)$ *comparisons on average and* $\mathcal{O}(n \log n)$ *other instructions.*

QuickMergesort with base case InsertionSort

Insert the elements successively into the already sorted sequence:

- find the position of each element by binary search
- make place for the new element by moving all elements by one

QuickMergesort with base case InsertionSort

Insert the elements successively into the already sorted sequence:

- find the position of each element by binary search
- make place for the new element by moving all elements by one

$\rightsquigarrow$ quadratic number of moves.

QuickMergesort with base case InsertionSort

Insert the elements successively into the already sorted sequence:

- find the position of each element by binary search
- make place for the new element by moving all elements by one

$\rightsquigarrow$ quadratic number of moves.

Proposition (Average Case of InsertionSort)

InsertionSort *needs* $n \log n - (2 \ln 2 + c(n)) \cdot n + \mathcal{O}(\log n)$ *comparisons on average where* $c(n) \in [-0.005, 0.005]$.

QuickMergesort with base case InsertionSort

Insert the elements successively into the already sorted sequence:

- find the position of each element by binary search
- make place for the new element by moving all elements by one

$\rightsquigarrow$ quadratic number of moves.

Proposition (Average Case of InsertionSort)

InsertionSort *needs* $n \log n - (2 \ln 2 + c(n)) \cdot n + \mathcal{O}(\log n)$ *comparisons on average where* $c(n) \in [-0.005, 0.005]$.

Corollary

QuickMergesort *with base case* InsertionSort *uses at most* $n \log n - 1.38n + o(n)$ *comparisons on average.*

The algorithm:

1. Build pairs $a_i > b_i$.
2. Sort the values $a_1,\ldots,a_{\lfloor n/2 \rfloor}$ recursively.
3. Insert the elements $b_1 \ldots, b_{\lceil n/2 \rceil}$ into the linear chain by binary insertion following a special ordering.

Theorem (Hadian 1969, Knuth 1973)

MergeInsertion *needs at most* $n \log n - 1.329n + \mathcal{O}(\log n)$ *comparisons in the worst case.*

QuickMergesort with base case MergeInsertion

Theorem (Average Case of MergeInsertion)

A simplified version of MergeInsertion *needs at most* $n \log n - 1.3999 \cdot n + \mathcal{O}(\log n)$ *comparisons on average.*

QuickMergesort with base case MergeInsertion

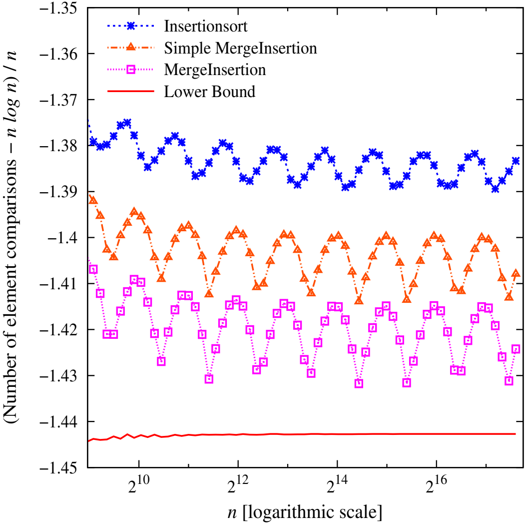
Theorem (Average Case of MergeInsertion)

A simplified version of MergeInsertion *needs at most* $n \log n - 1.3999 \cdot n + \mathcal{O}(\log n)$ *comparisons on average.*

Corollary

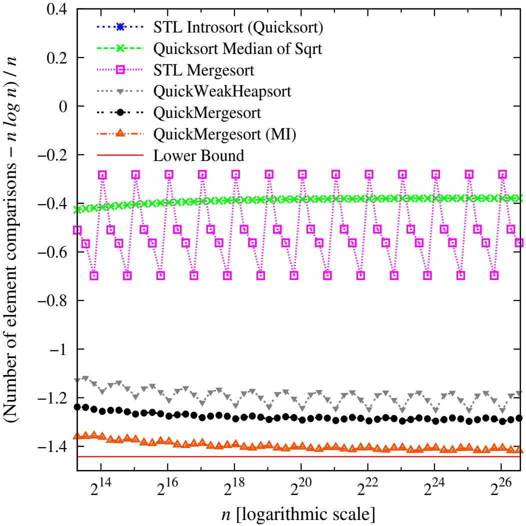
QuickMergesort *with* MergeInsertion *as base case needs at most* $n \log n - 1.3999n + o(n)$ *comparisons on average.*

Experiments on InsertionSort and MergeInsertion



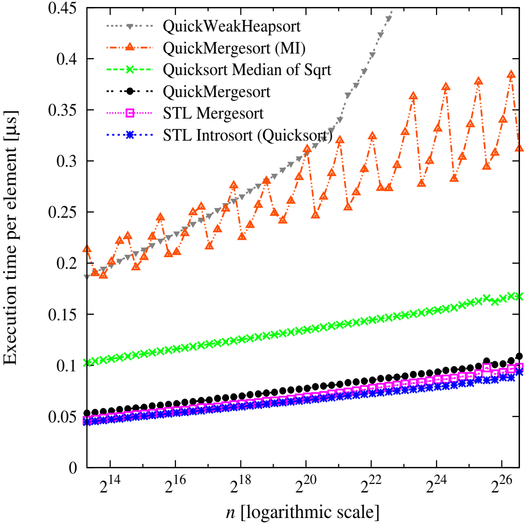
Sorting with $n \log n + \kappa n$ comparisons.

Experimental behavior of the linear term of QuickXsort



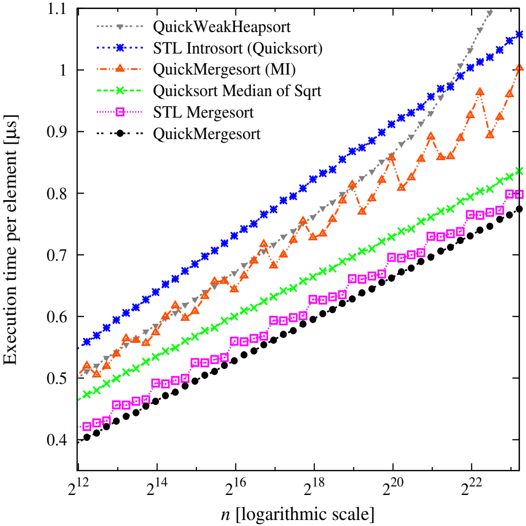
Sorting with $n \log n + \kappa n$ comparisons.

Running times of QuickXsort and other algorithms



Running time with integer data.

Running times of QuickXsort and other algorithms



Running time with more expensive comparisons simulated by calculating the logarithm of one operand in every comparison.

Constant-factor-optimal sorting with $n \log n + \kappa n + o(n)$ comparisons.

	Mem.	κ Worst	κ Avg.	κ Exper.
Information theo. lower bound		-1.44	-1.44	
BottomUpHeapsort	$\mathcal{O}(1)$	$\omega(1)$	–	[0.35,0.39]
QuickHeapsort	$\mathcal{O}(1)$	$\omega(1)$	-0.03	$\approx$ 0.20
	$\mathcal{O}(n)$ bits	$\omega(1)$	-0.99	$\approx$ -1.24
WeakHeapsort	$\mathcal{O}(n)$ bits	0.09	–	[-0.46,-0.42]
RelaxedWeakHeapsort	$\mathcal{O}(n)$	-0.91	-0.91	-0.91
ExternalWeakHeapsort #	$\mathcal{O}(n)$	-0.91	-1.26*	–
Mergesort	$\mathcal{O}(n)$	-0.91	-1.26	–
InPlaceMergesort	$\mathcal{O}(1)$	-1.25	–	–
InsertionSort	$\mathcal{O}(1)$	-0.91 †	-1.38 #	[-1.38,-1.39]
MergeInsertion	$\mathcal{O}(1)$	-1.32 †	-1.3999 #	[-1.43,-1.41]
QuickWeakHeapsort	$\mathcal{O}(n)$ bits	$\mathcal{O}(1)$ #	-0.91#	[-1.13,-1.25]
QuickMergesort #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.26	[-1.29,-1.27]
QuickMergesort (IS) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.38	–
QuickMergesort (MI) #	$\mathcal{O}(1)$	$\mathcal{O}(1)$	-1.3999	[-1.41,-1.40]

in this paper, – not analyzed,

* only for $n = 2^k$, † needs $\Theta(n^2)$ moves.

Conclusion and open problems

- Generic analysis of QuickXsort.
- QuickMergesort is highly competitive.
- Variant with base cases needs $\leq n \log n - 1.3999n + o(n)$ comparisons on average.

Conclusion and open problems

- Generic analysis of QuickXsort.
- QuickMergesort is highly competitive.
- Variant with base cases needs $\leq n \log n - 1.3999n + o(n)$ comparisons on average.

- Exact average case of MergeInsertion?
- How close can one get to the lower bound?

Conclusion and open problems

- Generic analysis of QuickXsort.
- QuickMergesort is highly competitive.
- Variant with base cases needs $\leq n \log n - 1.3999n + o(n)$ comparisons on average.

- Exact average case of MergeInsertion?
- How close can one get to the lower bound?

Thank you!