9. Sorting II

Mergesort, Quicksort

9.1 Mergesort

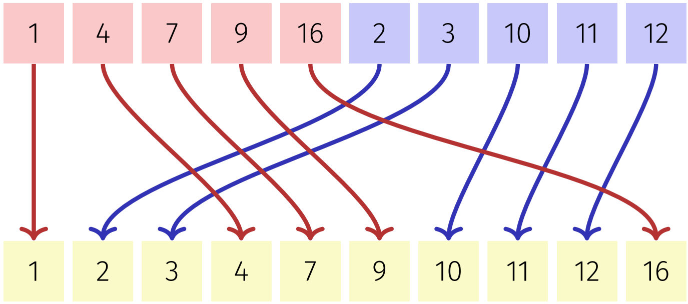
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort

Divide and Conquer!

- Assumption: two halves of the array A are already sorted.
- Minimum of A can be evaluated with a single element comparison.
- Iteratively: merge the two presorted halves of A in $\mathcal{O}(n)$.

Merge



Algorithm Merge(A, l, m, r)

Input: Array A with length n, indexes $1 \le l \le m \le r \le n$.
$A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sorted

Output: $A[l, \ldots, r]$ sorted

$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$
while $i \le m$ and $j \le r$ **do**
    if $A[i] \le A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$
    else $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$;
while $i \le m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \le r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

Correctness

Hypothesis: after k iterations of the loop in line 3 $B[1, \dots, k]$ is sorted and $B[k] \le A[i]$, if $i \le m$ and $B[k] \le A[j]$ if $j \le r$.

Proof by induction:

Base case: the empty array $B[1, \dots, 0]$ is trivially sorted.

Induction step $(k \to k+1)$:

- wlog $A[i] \le A[j]$, $i \le m, j \le r$.
- $B[1, \dots, k]$ is sorted by hypothesis and $B[k] \le A[i]$.
- After $B[k+1] \leftarrow A[i]$ $B[1, \dots, k+1]$ is sorted.
- $B[k+1] = A[i] \le A[i+1]$ (if $i+1 \le m$) and $B[k+1] \le A[j]$ if $j \le r$.
- $k \leftarrow k+1, i \leftarrow i+1$: Statement holds again.

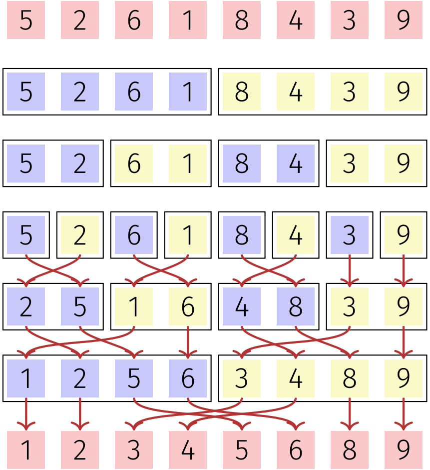
Analysis (Merge)

Lemma 12

If: array A with length n, indexes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ and $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted.
Then: in the call of Merge(A, l, m, r) a number of $\Theta(r - l)$ key movements and comparisons are executed.

Proof: straightforward(Inspect the algorithm and count the operations.)

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithm (recursive 2-way) `Mergesort`(A, l, r)

Input: Array A with length n. $1 \leq l \leq r \leq n$
Output: $A[l, \ldots, r]$ sorted.

```
if l < r then
    m ← ⌊(l + r)/2⌋          // middle position
    Mergesort(A, l, m)       // sort lower half
    Mergesort(A, m + 1, r)   // sort higher half
    Merge(A, l, m, r)        // Merge subsequences
```

Analysis

Recursion equation for the number of comparisons and key movements:

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + \Theta(n) \in \Theta(n \log n)$$

Algorithm `StraightMergesort`(A)

Avoid recursion: merge sequences of length $1, 2, 4, \ldots$ directly

Input: Array A with length n
Output: Array A sorted

$length \leftarrow 1$
while $length < n$ **do** // Iterate over lengths n
    $r \leftarrow 0$
    while $r + length < n$ **do** // Iterate over subsequences
        $l \leftarrow r + 1$
        $m \leftarrow l + length - 1$
        $r \leftarrow \min(m + length, n)$
        Merge(A, l, m, r)
    $length \leftarrow length \cdot 2$

Analysis

Like the recursive variant, the straight 2-way mergesort always executes a number of $\Theta(n \log n)$ key comparisons and key movements.

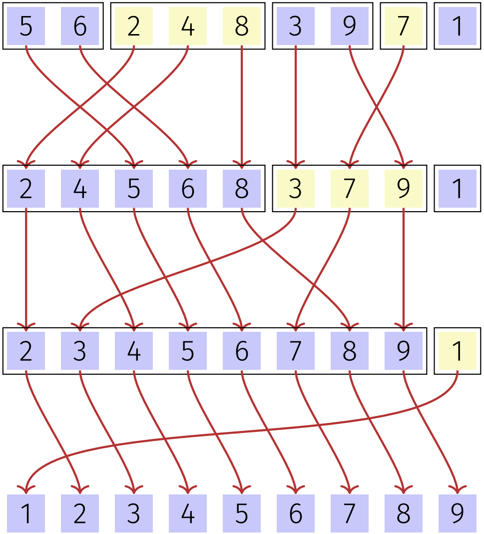
Natural 2-way mergesort

Observation: the variants above do not make use of any presorting and always execute $\Theta(n \log n)$ memory movements.

How can partially presorted arrays be sorted better?

❗ Recursive merging of previously sorted parts (*runs*) of A.

Natural 2-way mergesort



Algorithm `NaturalMergesort`(A)

Input: Array A with length $n > 0$
Output: Array A sorted

repeat
    $r \leftarrow 0$
    while $r < n$ **do**
        $l \leftarrow r + 1$
        $m \leftarrow l$; **while** $m < n$ **and** $A[m+1] \geq A[m]$ **do** $m \leftarrow m + 1$
        if $m < n$ **then**
            $r \leftarrow m + 1$; **while** $r < n$ **and** $A[r+1] \geq A[r]$ **do** $r \leftarrow r + 1$
            Merge(A, l, m, r);
        else
            $r \leftarrow n$
until $l = 1$

Analysis

Is it also asymptotically better than StraightMergesort on average?

❗No. Given the assumption of pairwise distinct keys, on average there are $n/2$ positions i with $k_i > k_{i+1}$, i.e. $n/2$ runs. Only one iteration is saved on average.

Natural mergesort executes in the worst case and on average a number of $\Theta(n \log n)$ comparisons and memory movements.

9.2 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Quicksort

What is the disadvantage of Mergesort?

Requires additional $\Theta(n)$ storage for merging.

How could we reduce the merge costs?

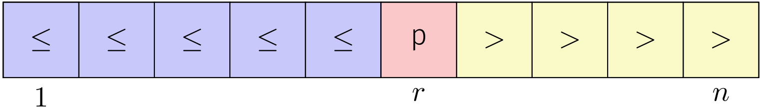
Make sure that the left part contains only smaller elements than the right part.

How?

Pivot and Partition!

Use a pivot

1. Choose a (an arbitrary) **pivot** p
2. Partition A in two parts, one part L with the elements with $A[i] \leq p$ and another part R with $A[i] > p$
3. Quicksort: Recursion on parts L and R



Algorithm `Partition`(A, l, r, p)

Input: Array A, that contains the pivot p in $A[l, \ldots, r]$ at least once.
Output: Array A partitioned in $[l, \ldots, r]$ around p. Returns position of p.

while $l \leq r$ **do**
- **while** $A[l] < p$ **do**
  - $l \leftarrow l + 1$
- **while** $A[r] > p$ **do**
  - $r \leftarrow r - 1$
- swap($A[l]$, $A[r]$)
- **if** $A[l] = A[r]$ **then**
  - $l \leftarrow l + 1$

return l-1

Algorithm `Quicksort`(A, l, r)

Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted in $A[l, \ldots, r]$.

if $l < r$ **then**
    Choose pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ `Partition`(A, l, r, p)
    `Quicksort`$(A, l, k - 1)$
    `Quicksort`$(A, k + 1, r)$

Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1
2	1	3	6	8	5	7	9	4
1	2	3	4	5	8	7	9	6
1	2	3	4	5	6	7	9	8
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Analysis: number comparisons

Worst case. Pivot = min or max; number comparisons:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = d \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analysis: number swaps

Result of a call to partition (pivot 3):

2	1	3	6	8	5	7	9	4

? How many swaps have taken place?

! 2. The maximum number of swaps is given by the number of keys in the smaller part.

Analysis: number swaps

Thought experiment

- Each key from the smaller part pays a coin when it is being swapped.
- After a key has paid a coin the domain containing the key decreases to half its previous size.
- Every key needs to pay at most $\log n$ coins. But there are only n keys.

Consequence: there are $\mathcal{O}(n \log n)$ key swaps in the worst case.

Randomized Quicksort

Despite the worst case running time of $\Theta(n^2)$, quicksort is used practically very often.

Reason: quadratic running time unlikely provided that the choice of the pivot and the pre-sorting are not very disadvantageous.

Avoidance: randomly choose pivot. Draw uniformly from $[l, r]$.

Analysis (randomized quicksort)

Expected number of compared keys with input length n:

$$T(n) = (n-1) + \frac{1}{n}\sum_{k=1}^{n}(T(k-1) + T(n-k)),\ T(0) = T(1) = 0$$

Claim $T(n) \leq 4n\log n$.
Proof by induction:
Base case straightforward for $n = 0$ (with $0\log 0 := 0$) and for $n = 1$.
Hypothesis: $T(n) \leq 4n\log n$ for some n.
Induction step: $(n - 1 \to n)$

Analysis (randomized quicksort)

$$
\begin{aligned}
T(n) &= n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \overset{\text{H}}{\leq} n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} 4k\log k \\
&= n - 1 + \sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
&\leq n - 1 + \frac{8}{n}\left((\log n - 1)\sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k \right) \\
&= n - 1 + \frac{8}{n}\left((\log n) \cdot \frac{n(n-1)}{2} - \frac{n}{4}\left(\frac{n}{2} + 1\right)\right) \\
&= 4n\log n - 4\log n - 3 \leq 4n\log n
\end{aligned}
$$

Analysis (randomized quicksort)

Theorem 13

On average randomized quicksort requires $\mathcal{O}(n \log n)$ comparisons.

Practical Considerations

Worst case recursion depth $n - 1$[9]. Then also a memory consumption of $\mathcal{O}(n)$.

Can be avoided: recursion only on the smaller part. Then guaranteed $\mathcal{O}(\log n)$ worst case recursion depth and memory consumption.

[9] stack overflow possible!

Quicksort with logarithmic memory consumption

Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted between l and r.

while $l < r$ **do**
    Choose pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition(A, l, r, p)
    if $k - l < r - k$ **then**
        Quicksort($A[l, \ldots, k-1]$)
        $l \leftarrow k + 1$
    else
        Quicksort($A[k+1, \ldots, r]$)
        $r \leftarrow k - 1$

The call of Quicksort($A[l, \ldots, r]$) in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

Practical Considerations.

- Practically the pivot is often the median of three elements. For example: Median3$(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.
- There is a variant of quicksort that requires only constant storage. Idea: store the old pivot at the position of the new pivot.
- Complex divide-and-conquer algorithms often use a trivial $(\Theta(n^2))$ algorithm as base case to deal with small problem sizes.

9.3 Appendix

Derivation of some mathematical formulas

$\log n! \in \Theta(n \log n)$

$$\log n! = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

$$\begin{aligned}
\sum_{i=1}^{n} \log i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \log i + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log i \\
&\geq \sum_{i=2}^{\lfloor n/2 \rfloor} \log 2 + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log \frac{n}{2} \\
&= (\underbrace{\lfloor n/2 \rfloor}_{> n/2 - 1} - 2 + 1) + (\underbrace{n - \lfloor n/2 \rfloor}_{\geq n/2})(\log n - 1) \\
&> \frac{n}{2} \log n - 2.
\end{aligned}$$

$[n! \in o(n^n)\ ]$

$$
\begin{aligned}
n \log n &\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log 2i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} \log i \\
&= \sum_{i=1}^{n} \log i + \left\lfloor \frac{n}{2} \right\rfloor \log 2 \\
&> \sum_{i=1}^{n} \log i + n/2 - 1 = \log n! + n/2 - 1
\end{aligned}
$$

$$
n^n = 2^{n \log_2 n} \geq 2^{\log_2 n!} \cdot 2^{n/2} \cdot 2^{-1} = n! \cdot 2^{n/2-1}
$$

$$
\Rightarrow \frac{n!}{n^n} \leq 2^{-n/2+1} \stackrel{n \to \infty}{\longrightarrow} 0 \Rightarrow n! \in o(n^n) = \mathcal{O}(n^n) \setminus \Omega(n^n)
$$

[Even $n! \in o((n/c)^n)\,\forall\, 0 < c < e$]

Konvergenz oder Divergenz von $f_n = \frac{n!}{(n/c)^n}$.

Ratio Test

$$\frac{f_{n+1}}{f_n} = \frac{(n+1)!}{\left(\frac{n+1}{c}\right)^{n+1}} \cdot \frac{\left(\frac{n}{c}\right)^n}{n!} = c \cdot \left(\frac{n}{n+1}\right)^n \longrightarrow c \cdot \frac{1}{e} \lesseqgtr 1 \text{ if } c \lesseqgtr e$$

because $\left(1 + \frac{1}{n}\right)^n \to e$. Even the series $\sum_{i=1}^{n} f_n$ converges / diverges for $c \lesseqgtr e$.

f_n diverges for $c = e$, because (Stirling): $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

[Ratio Test]

Ratio test for a sequence $(f_n)_{n\in\mathbb{N}}$: If $\frac{f_{n+1}}{f_n} \xrightarrow[n\to\infty]{} \lambda$, then the sequence f_n and the series $\sum_{i=1}^{n} f_i$

- converge, if $\lambda < 1$ and
- diverge, if $\lambda > 1$.

[Ratio Test Derivation]

Ratio test is implied by Geometric Series

$$S_n(r) := \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}.$$

converges for $n \to \infty$ if and only if $-1 < r < 1$.
Let $0 \leq \lambda < 1$:

$$\forall \varepsilon > 0 \, \exists n_0 : f_{n+1}/f_n < \lambda + \varepsilon \, \forall n \geq n_0$$
$$\Rightarrow \exists \varepsilon > 0, \exists n_0 : f_{n+1}/f_n \leq \mu < 1 \, \forall n \geq n_0$$

Thus

$$\sum_{n=n_0}^{\infty} f_n \leq f_{n_0} \cdot \sum_{n=n_0}^{\infty} \cdot \mu^{n-n_0} \quad \text{konvergiert.}$$

(Analogously for divergence)

L'Hospital's rule

Theorem 14

Let $f, g : \mathbb{R}^+ \to \mathbb{R}$ *differentiable functions with* $g'(x) \neq 0 \; \forall x > 0$.
If

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0,$$

or

$$\lim_{x\to\infty} f(x) = \pm\infty \text{ and } \lim_{x\to\infty} g(x) = \pm\infty,$$

then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

if the limit of $f'(x)/g'(x)$ *exists*

L'Hospital's rule

Example

Es gilt $\log^k(n) \in o(n)$, because with $f(x) = \log^k(x)$, $g(n) = x$, we can apply L'Hospital's rule and get

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} k\frac{\log^{k-1}(x)}{x}$$

After k iterations we get

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} k!\frac{1}{x} = 0.$$