5. Sorting

Simple Sorting, Quicksort, Mergesort

Problem

Input: An array $A = (A[1], ..., A[n])$ with length n.

Output: a permutation A' of A, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

Selection Sort

5 6 2 8 4 1 $(i = 1)$

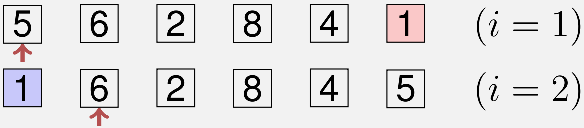
- Iterative procedure as for Bubblesort.

Selection Sort

5 6 2 8 4 1 $(i = 1)$

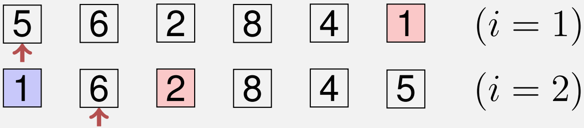
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



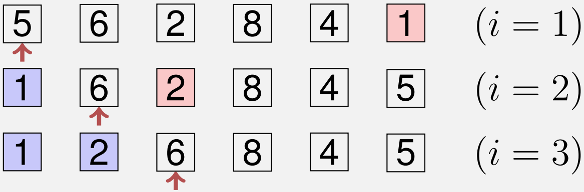
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



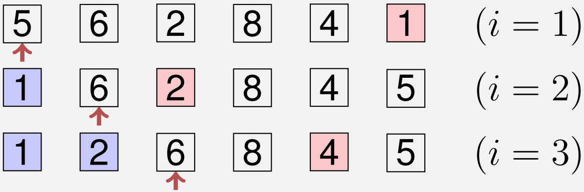
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



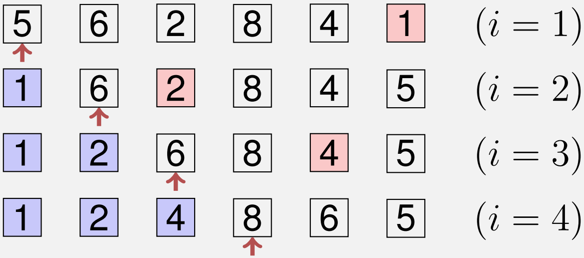
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



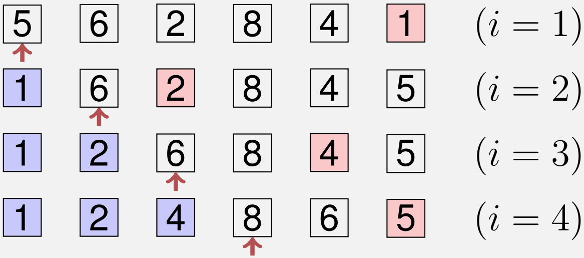
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



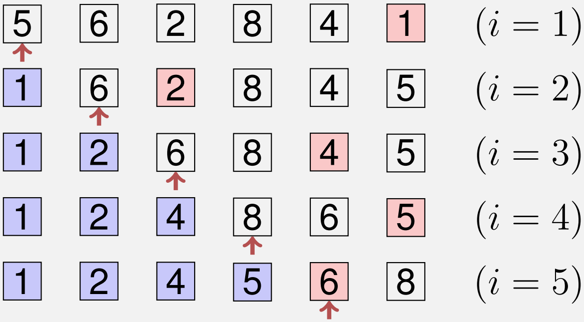
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$
1	2	4	5	6	8	$(i = 5)$

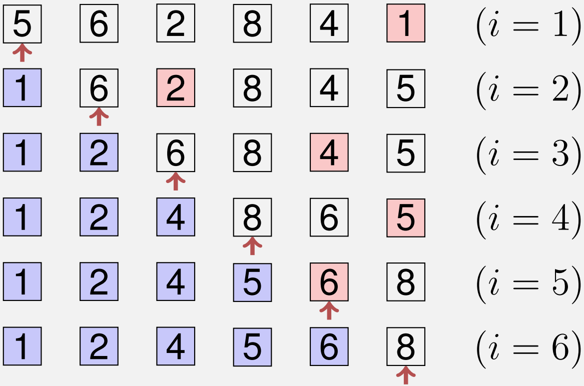
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



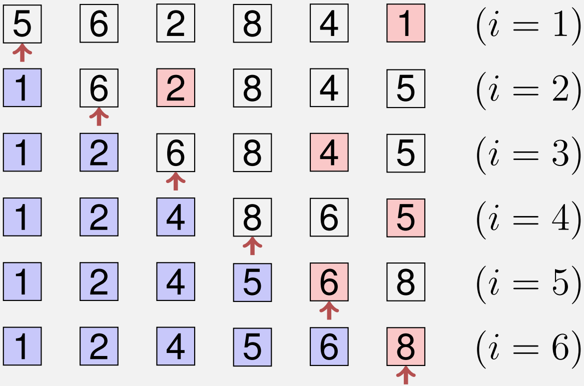
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



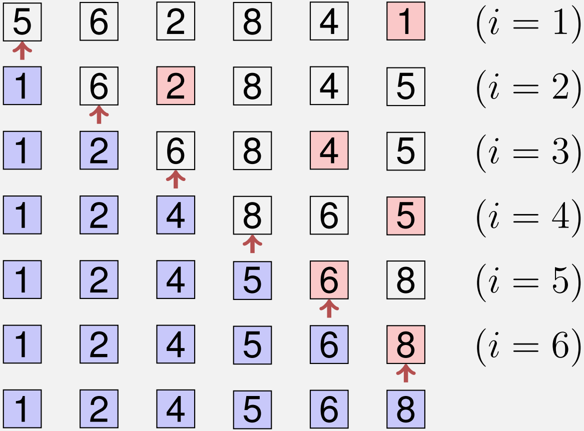
- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Selection Sort



- Iterative procedure as for Bubblesort.
- Selection of the smallest (or largest) element by immediate search.

Algorithm: Selection Sort

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sorted Array A

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

Analysis

Number comparisons in worst case:

Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case:

Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case: $n - 1 = \Theta(n)$

Best case number comparisons:

Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case: $n - 1 = \Theta(n)$

Best case number comparisons: $\Theta(n^2)$.

Insertion Sort

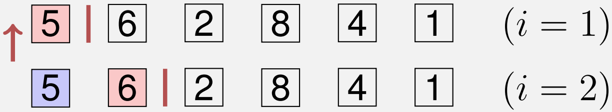
5 | 6 2 8 4 1 $(i = 1)$

Insertion Sort

↑ 5 | 6 2 8 4 1 $(i = 1)$

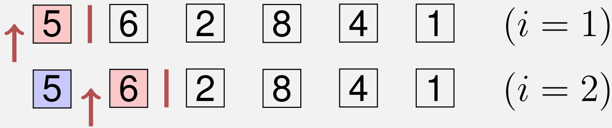
- Iterative procedure: $i = 1 \dots n$

Insertion Sort



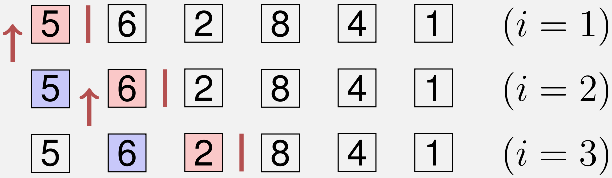
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.

Insertion Sort



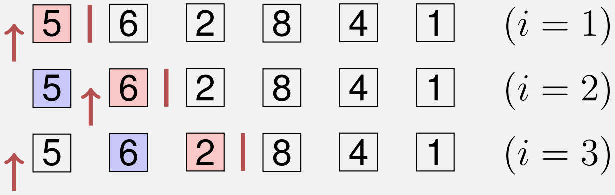
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



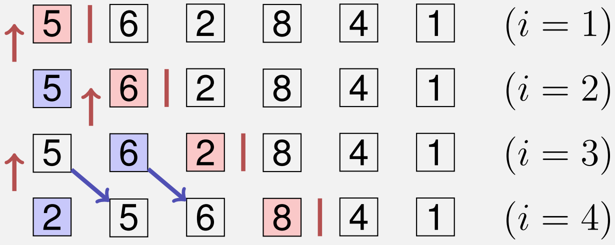
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



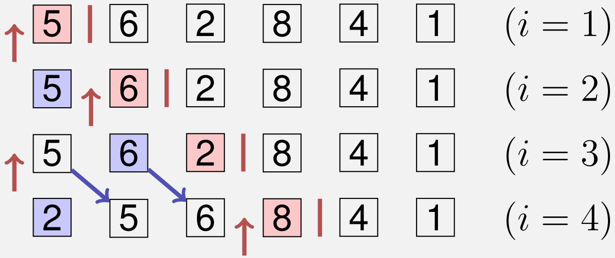
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



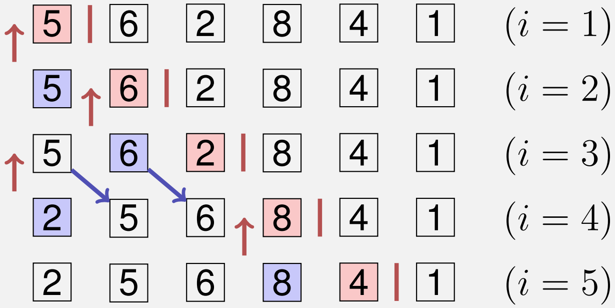
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



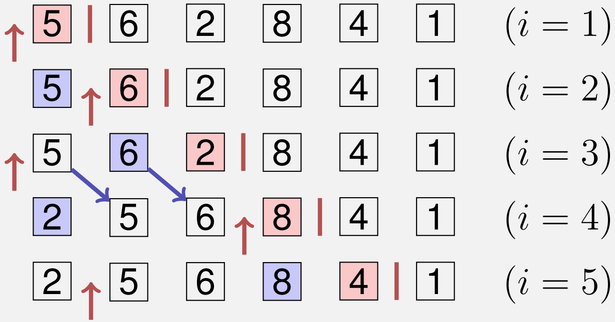
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



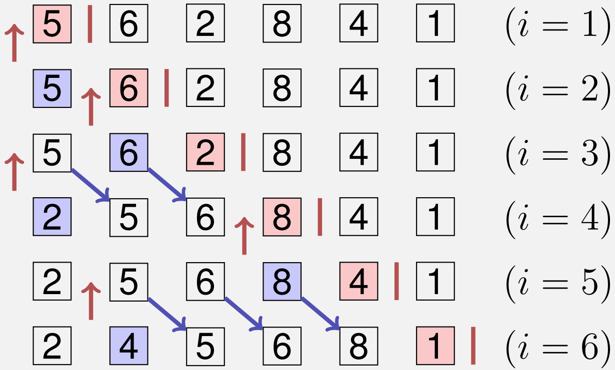
- Iterative procedure: $i = 1 ... n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



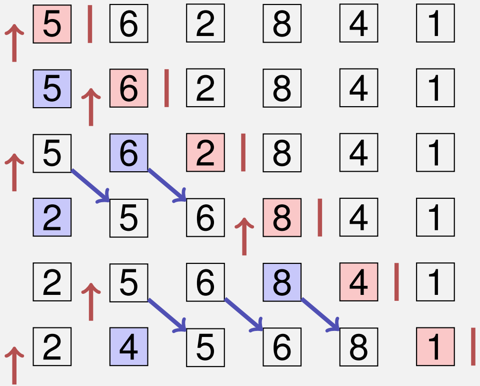
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

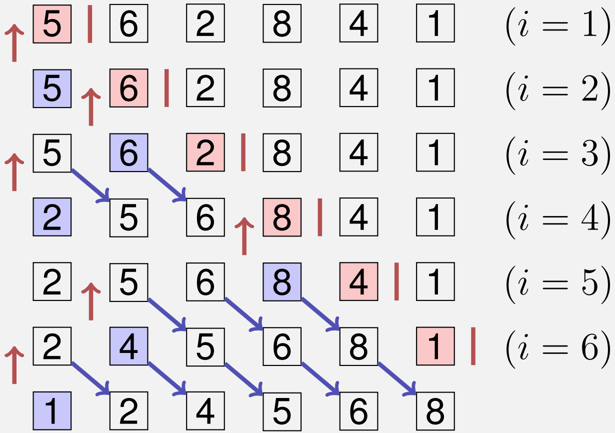
Insertion Sort



$(i = 1)$
$(i = 2)$
$(i = 3)$
$(i = 4)$
$(i = 5)$
$(i = 6)$

- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

! Many element movements in the worst case.

? What is the advantage of this algorithm compared to selection sort?

Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

! Many element movements in the worst case.

? What is the advantage of this algorithm compared to selection sort?

! The search domain (insertion interval) is already sorted. Consequently: binary search possible.

Algorithm: Insertion Sort

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sorted Array A

for $i \leftarrow 2$ **to** n **do**
    $x \leftarrow A[i]$
    $p \leftarrow$ BinarySearch($A[1...i-1], x$); // Smallest $p \in [1, i]$ with $A[p] \geq x$
    for $j \leftarrow i-1$ **downto** p **do**
        $A[j+1] \leftarrow A[j]$
    $A[p] \leftarrow x$

Analysis

Number comparisons in the worst case:

[3]With slight modification of the function BinarySearch for the minimum / maximum: $\Theta(n)$

Analysis

Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.
Number comparisons in the best case

[3]With slight modification of the function BinarySearch for the minimum / maximum: $\Theta(n)$

Analysis

Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

Number comparisons in the best case $\Theta(n \log n)$.[3]

Number swaps in the worst case

[3]With slight modification of the function BinarySearch for the minimum / maximum: $\Theta(n)$

Analysis

Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

Number comparisons in the best case $\Theta(n \log n)$.[3]

Number swaps in the worst case $\sum_{k=2}^{n} (k-1) \in \Theta(n^2)$

[3] With slight modification of the function BinarySearch for the minimum / maximum: $\Theta(n)$

5.1 Quicksort

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

Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1

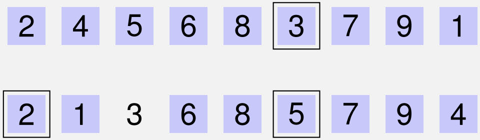
Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1

Quicksort (arbitrary pivot)

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

Quicksort (arbitrary pivot)



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

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

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

Quicksort (arbitrary pivot)

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

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

Algorithm Quicksort($A[l, \dots, r]$)

Input : Array A with length n. $1 \leq l \leq r \leq n$.
Output : Array A, sorted between l and r.
if $l < r$ **then**
    Choose pivot $p \in A[l, \dots, r]$
    $k \leftarrow$ Partition($A[l, \dots, r], p$)
    Quicksort($A[l, \dots, k-1]$)
    Quicksort($A[k+1, \dots, r]$)

Analysis: number comparisons

Best case.

Analysis: number comparisons

Best case. Pivot = median; number comparisons:

$$T(n) = 2T(n/2) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \mathcal{O}(n \log n)$$

Worst case.

Analysis: number comparisons

Best case. Pivot = median; number comparisons:

$$T(n) = 2T(n/2) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \mathcal{O}(n \log n)$$

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

$$T(n) = T(n-1) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analysis (randomized quicksort)

Theorem

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

Practical considerations

Worst case recursion depth $n - 1$[4]. 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.

[4] stack overflow possible!

Practical considerations.

Practically the pivot is often the median of three elements. For example: Median3($A[l], A[r], A[\lfloor l + r/2 \rfloor]$).

5.2 Mergesort

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

Mergesort

Divide and Conquer!

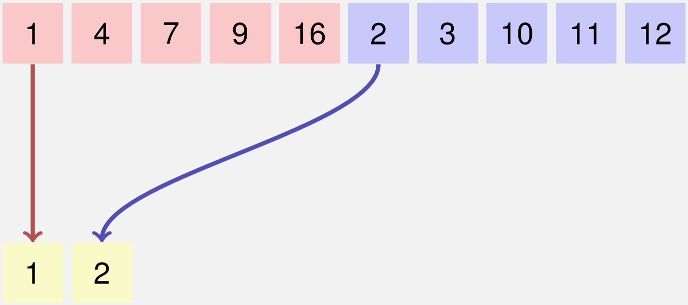
- Assumption: two halves of the array A are already sorted.
- Minimum of A can be evaluated with two comparisons.
- Iteratively: sort the pre-sorted array A in $\mathcal{O}(n)$.

Merge

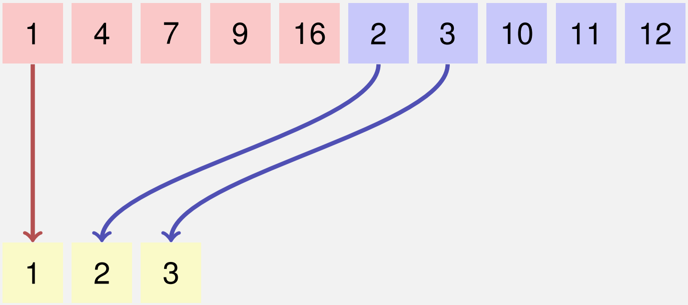
1	4	7	9	16	2	3	10	11	12

Merge

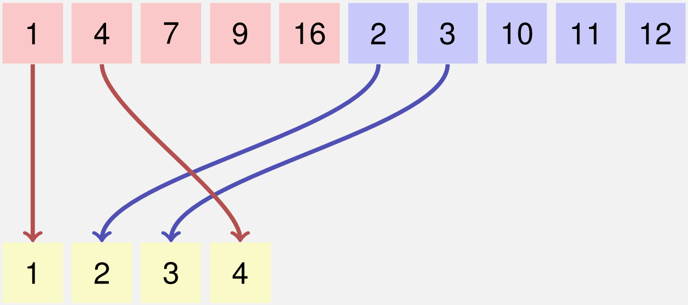
Merge



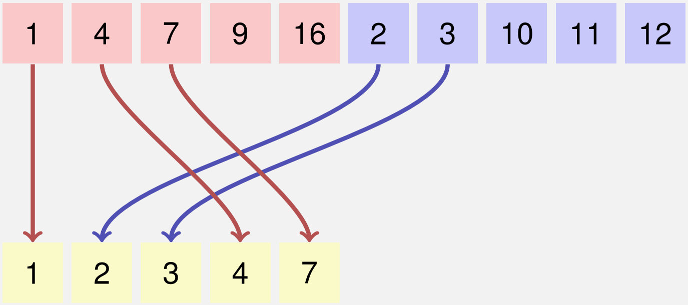
Merge



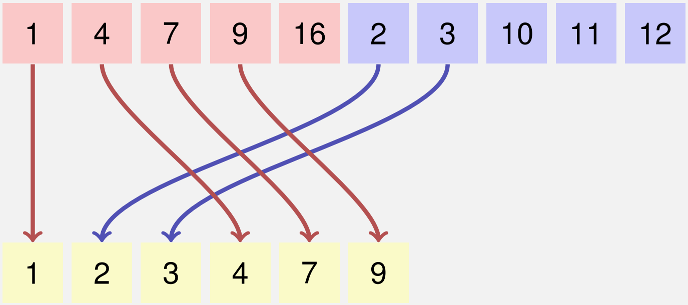
Merge



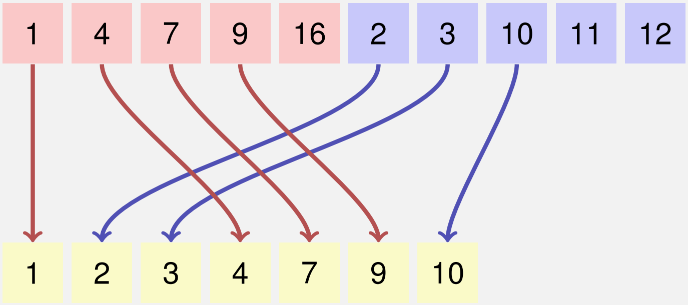
Merge



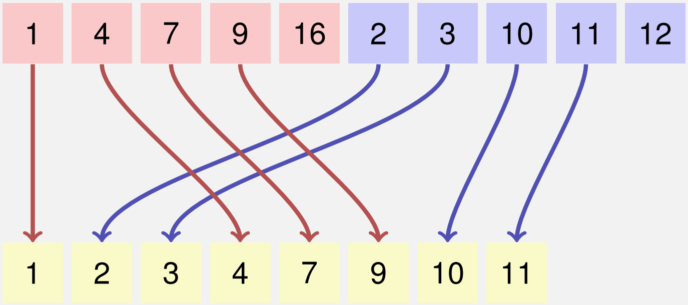
Merge



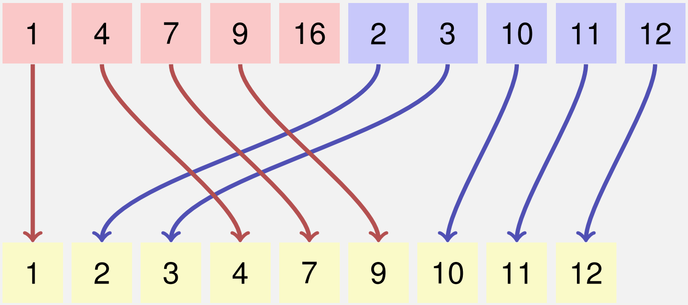
Merge



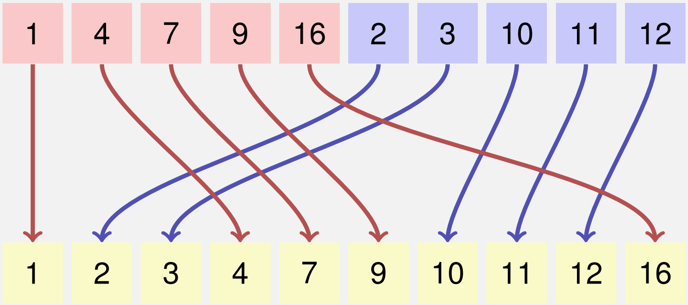
Merge



Merge



Merge



Algorithm Merge(A, l, m, r)

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

Output : $A[l, \dots, r]$ 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 $B[1,\ldots,k]$ is sorted and $B[k] \leq A[i]$, if $i \leq m$ and $B[k] \leq A[j]$ if $j \leq r$.

Proof by induction:
Base case: the empty array $B[1,\ldots,0]$ is trivially sorted.
Induction step ($k \to k+1$):

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

Analysis (Merge)

Lemma

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

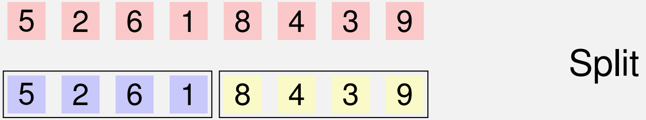
5	2	6	1	8	4	3	9

Mergesort

5	2	6	1	8	4	3	9

Split

Mergesort



Mergesort

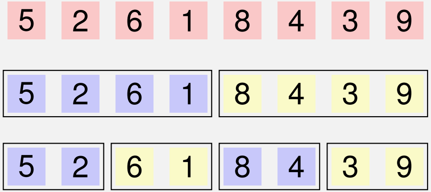
5 2 6 1 8 4 3 9

Split

[5 2 6 1] [8 4 3 9]

Split

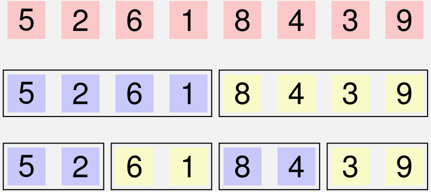
Mergesort



Split

Split

Mergesort

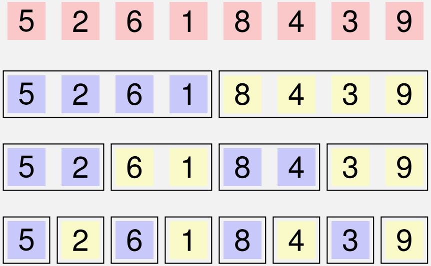


Split

Split

Split

Mergesort

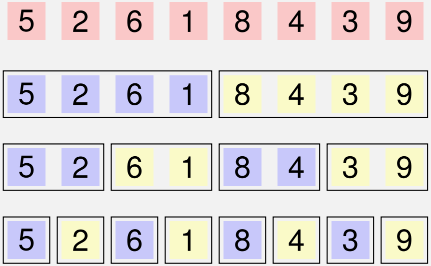


Split

Split

Split

Mergesort



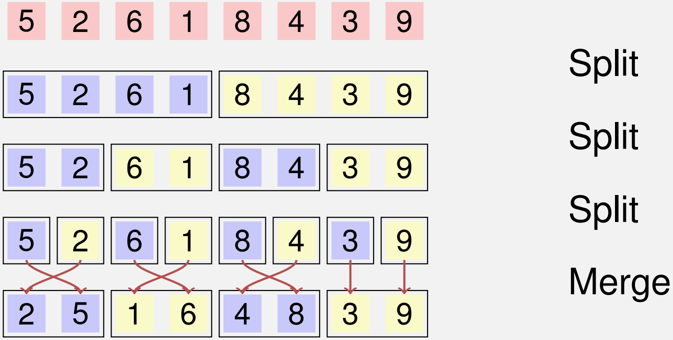
Split

Split

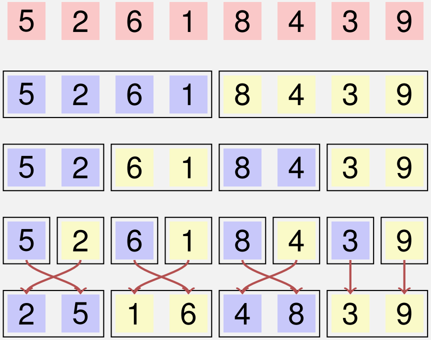
Split

Merge

Mergesort


5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
Split
Split
Split
Merge

Mergesort



Split

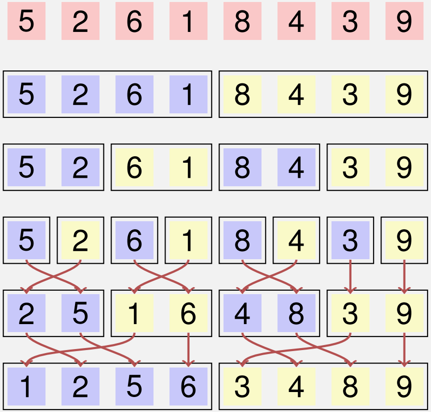
Split

Split

Merge

Merge

Mergesort



Split

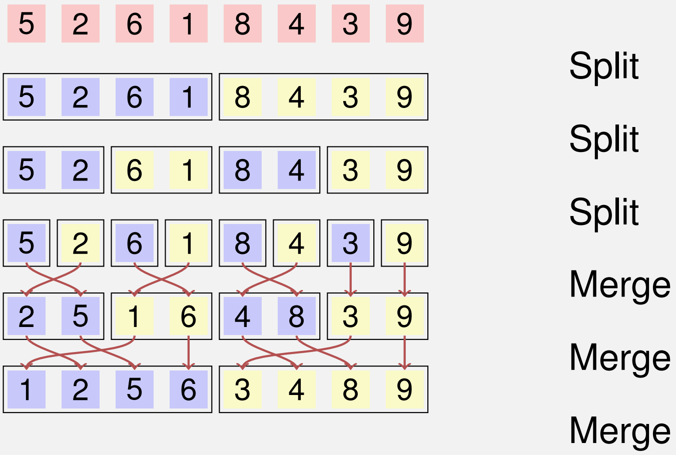
Split

Split

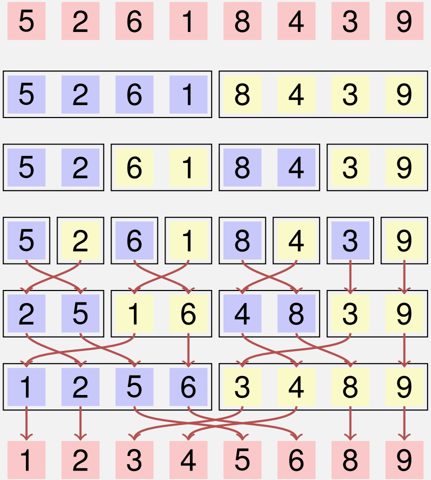
Merge

Merge

Mergesort


5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
1 2 5 6 3 4 8 9
Split
Split
Split
Merge
Merge
Merge

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 : Array $A[l, \dots, r]$ sorted.

if $l < r$ **then**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$ // 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:

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

Analysis

Recursion equation for the number of comparisons and key movements:

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

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?

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.

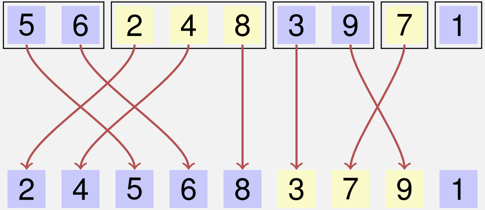
Natural 2-way mergesort

5	6	2	4	8	3	9	7	1

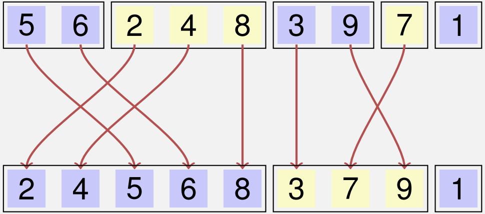
Natural 2-way mergesort

5	6	2	4	8	3	9	7	1

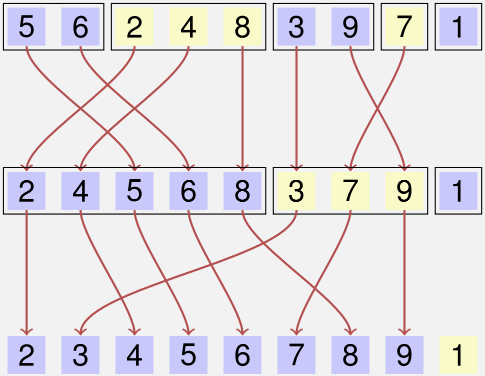
Natural 2-way mergesort



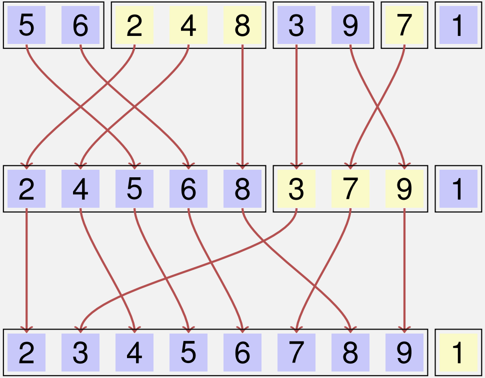
Natural 2-way mergesort



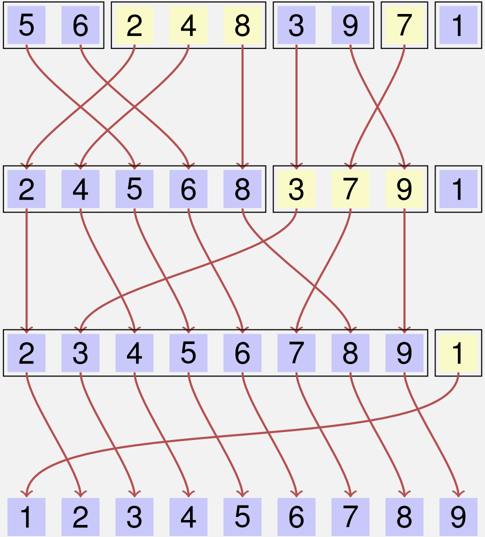
Natural 2-way mergesort



Natural 2-way mergesort



Natural 2-way mergesort



Algorithm NaturalMergesort(A)

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

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m + 1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r + 1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```

Analysis

In the best case, natural merge sort requires only $n - 1$ comparisons.

In the worst case and on average, natural merge sort requires $\Theta(n \log n)$ comparisons and memory movements.