6. Searching

Linear Search, Binary Search [Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems 2.1-3,2.2-3,2.3-5]

The Search Problem

Provided

- A set of data sets

  telephone book, dictionary, symbol table

- Each dataset has a key k.
- Keys are comparable: unique answer to the question $k_1 \leq k_2$ for keys k_1, k_2.

Task: find data set by key k.

Search in Array

Provided

- Array A with n elements $(A[1], \dots, A[n])$.
- Key b

Wanted: index k, $1 \leq k \leq n$ with $A[k] = b$ or "not found".

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

Linear Search

Traverse the array from $A[1]$ to $A[n]$.

Linear Search

Traverse the array from $A[1]$ to $A[n]$.

- **Best case:** 1 comparison.

Linear Search

Traverse the array from $A[1]$ to $A[n]$.

- **Best case:** 1 comparison.
- **Worst case:** n comparisons.

Search in a Sorted Array

Provided

- Sorted array A with n elements $(A[1], \dots, A[n])$ with $A[1] \leq A[2] \leq \cdots \leq A[n]$.
- Key b

Wanted: index k, $1 \leq k \leq n$ with $A[k] = b$ or "not found".

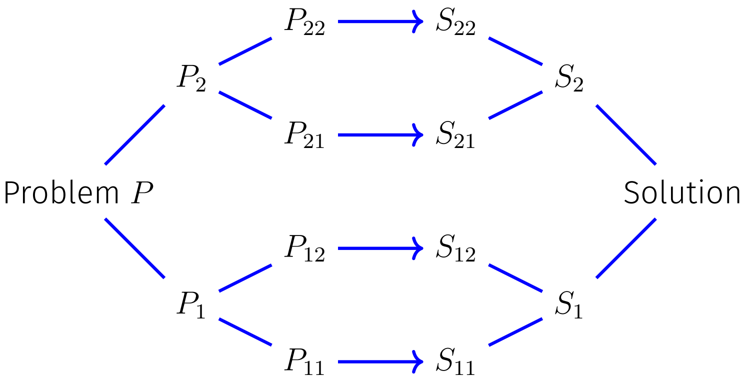
10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

divide et impera

Divide and Conquer

Divide the problem into subproblems that contribute to the simplified computation of the overal problem.

divide et impera



Divide and Conquer!

Search $b = 23$.

Divide and Conquer!

Search $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

Divide and Conquer!

Search $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

$b < 28$

Divide and Conquer!

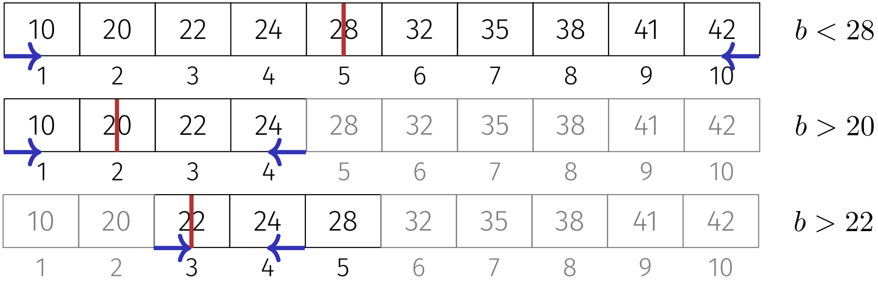
Search $b = 23$.

10	20	22	24	28	32	35	38	41	42	$b < 28$
1	2	3	4	5	6	7	8	9	10	

10	20	22	24	28	32	35	38	41	42	$b > 20$
1	2	3	4	5	6	7	8	9	10	

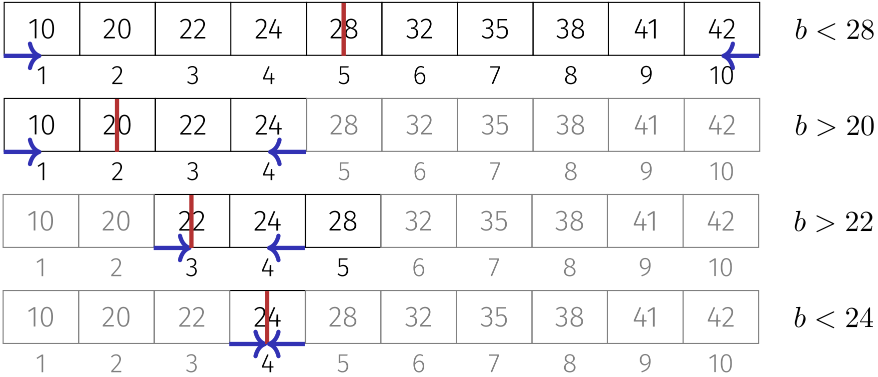
Divide and Conquer!

Search $b = 23$.



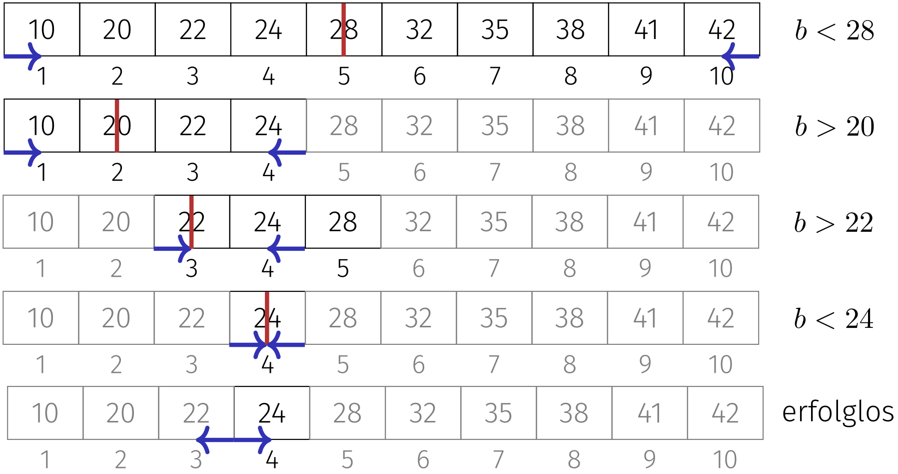
Divide and Conquer!

Search $b = 23$.



Divide and Conquer!

Search $b = 23$.



Binary Search Algorithm BSearch(A, l, r, b)

Input: Sorted array A of n keys. Key b. Bounds $1 \leq l, r \leq n$ mit $l \leq r$ or $l = r + 1$.

Output: Index $m \in [l, \dots, r + 1]$, such that $A[i] \leq b$ for all $l \leq i < m$ and $A[i] \geq b$ for all $m < i \leq r$.

```
m ← ⌊(l + r)/2⌋
if l > r then // Unsuccessful search
    return l
else if b = A[m] then// found
    return m
else if b < A[m] then// element to the left
    return BSearch(A, l, m − 1, b)
else // b > A[m]: element to the right
    return BSearch(A, m + 1, r, b)
```

Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute: [2]

$$T(n) = T\left(\frac{n}{2}\right) + c$$

[2]Try to find a closed form of T by applying the recurrence repeatedly (starting with $T(n)$).

Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute: [2]

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$

[2] Try to find a closed form of T by applying the recurrence repeatedly (starting with $T(n)$).

Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute: [2]

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \ldots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \end{aligned}$$

[2] Try to find a closed form of T by applying the recurrence repeatedly (starting with $T(n)$).

Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute: [2]

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \dots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

---

[2] Try to find a closed form of T by applying the recurrence repeatedly (starting with $T(n)$).

Result

Theorem 3

The binary sorted search algorithm requires $\Theta(\log n)$ fundamental operations.

7. Sorting

Simple Sorting, Quicksort, Mergesort

Problem

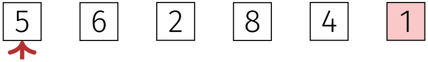
Input: An array $A = (A[1], ..., A[n])$ with length n.
Output: a permutation A' of A, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

Selection Sort

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

- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.

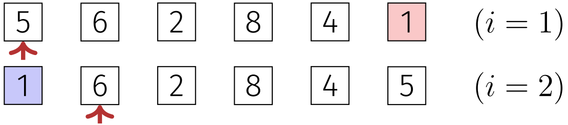
Selection Sort



$(i = 1)$

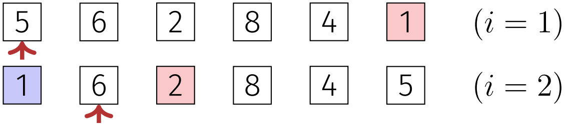
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.

Selection Sort



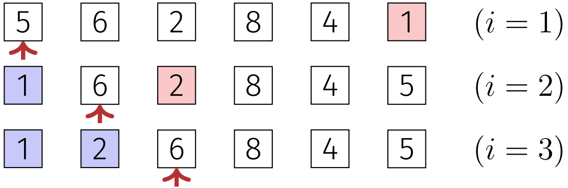
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



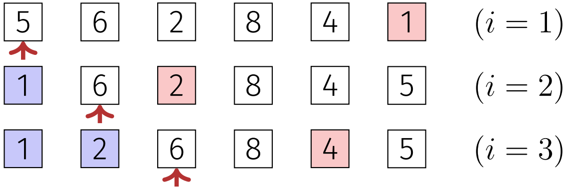
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \rightarrow i+1$). Repeat until all is sorted. ($i = n$)

Selection Sort



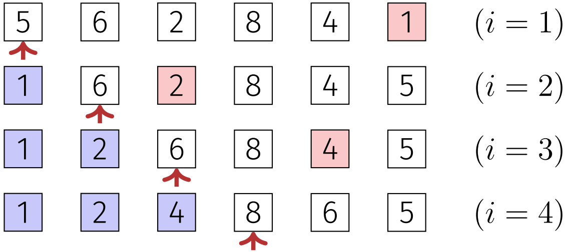
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i+1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



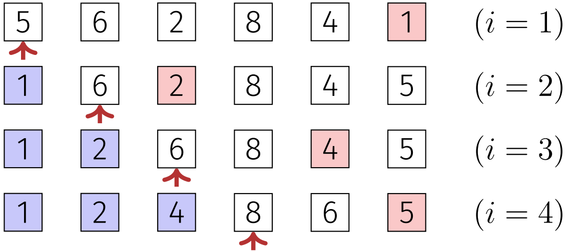
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \rightarrow i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



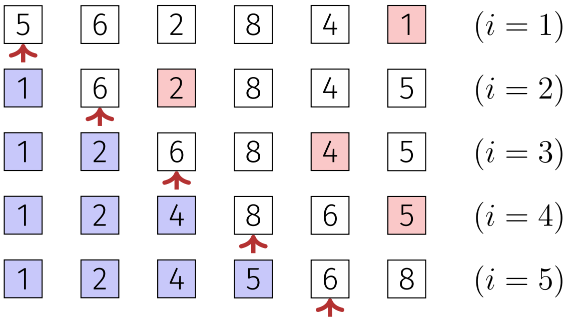
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



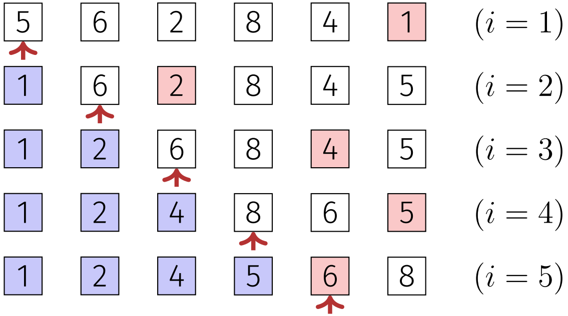
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



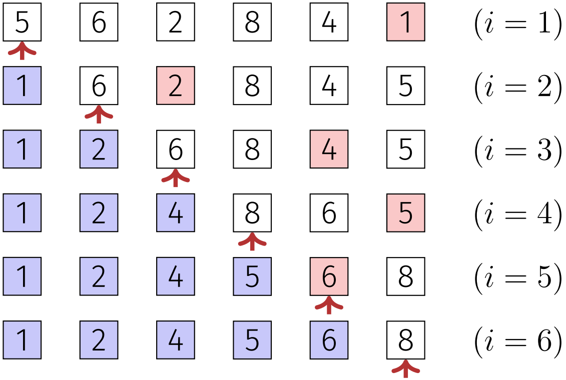
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



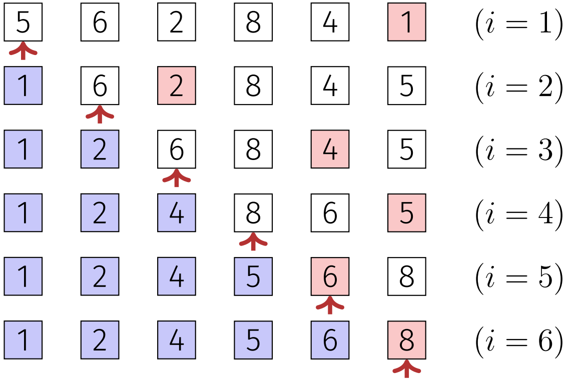
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



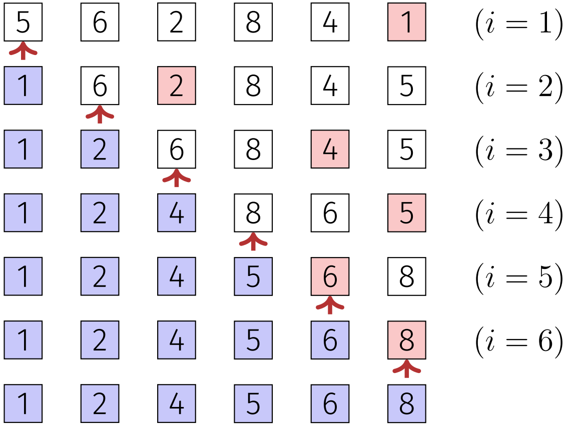
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \rightarrow i+1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i+1)$. Repeat until all is sorted. $(i = n)$

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: $n - 1 = \Theta(n)$

Insertion Sort

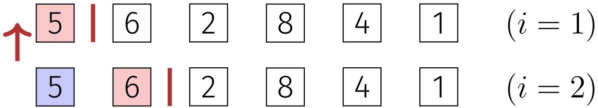
5 |	6	2	8	4	1	$(i = 1)$

Insertion Sort

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

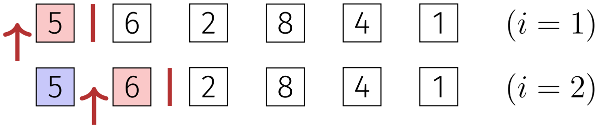
- Iterative procedure:
  $i = 1 \ldots n$

Insertion Sort



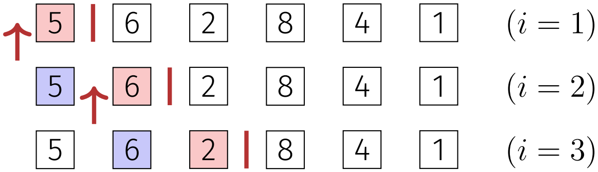
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.

Insertion Sort



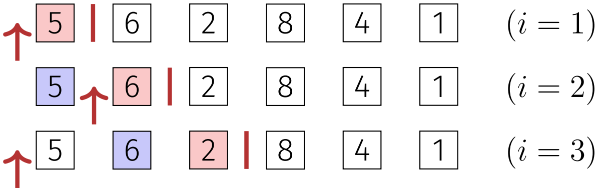
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



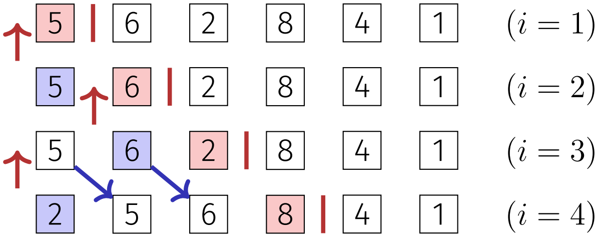
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



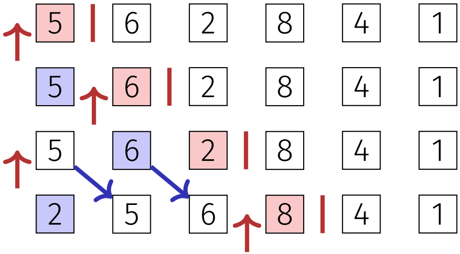
- Iterative procedure: $i = 1 ... n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



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

Insertion Sort



$(i = 1)$

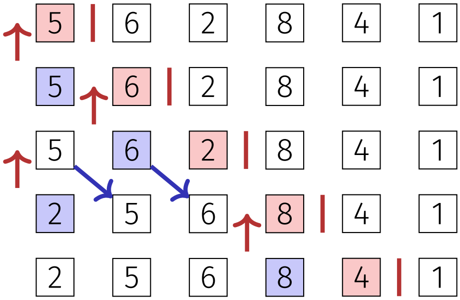
$(i = 2)$

$(i = 3)$

$(i = 4)$

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

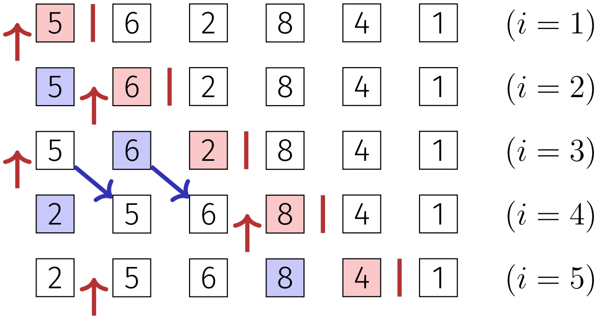
Insertion Sort



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

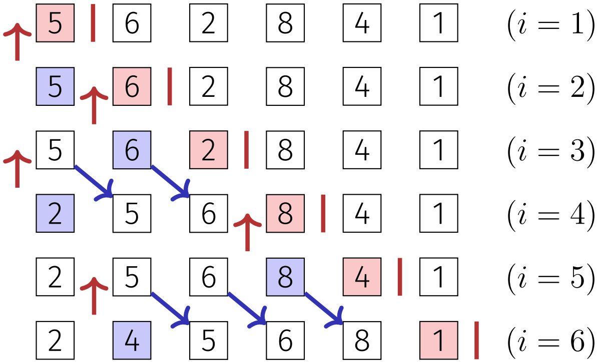
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



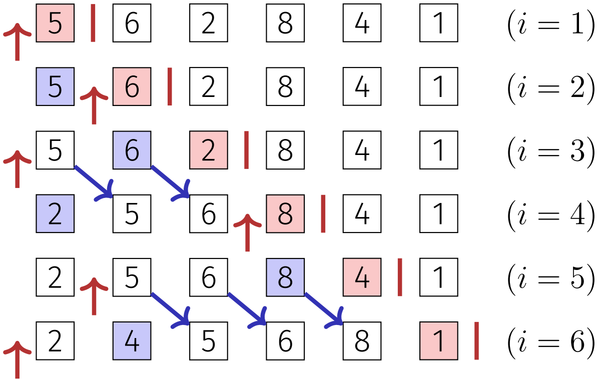
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



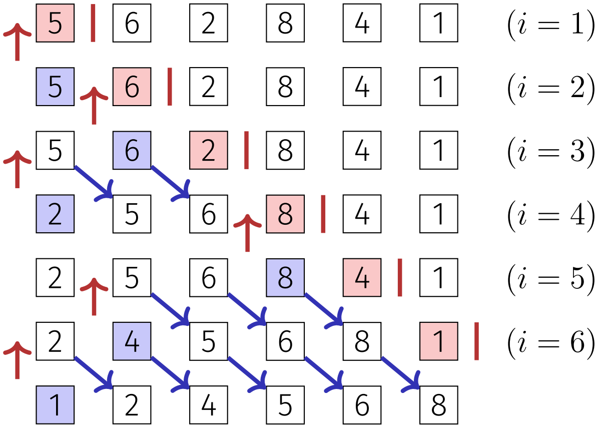
- Iterative procedure: $i = 1 \ldots n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

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

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 ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A, 1, i − 1, x); // Smallest p ∈ [1, i] with A[p] ≥ x
    for j ← i − 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```

7.1 Mergesort

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

Mergesort

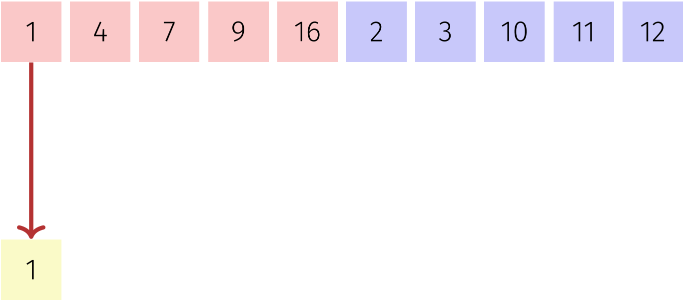
Divide and Conquer!

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

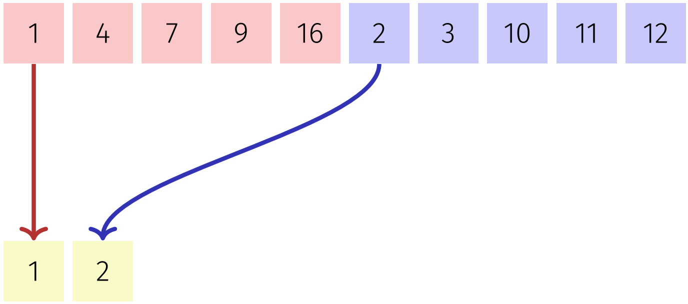
Merge

1	4	7	9	16	2	3	10	11	12

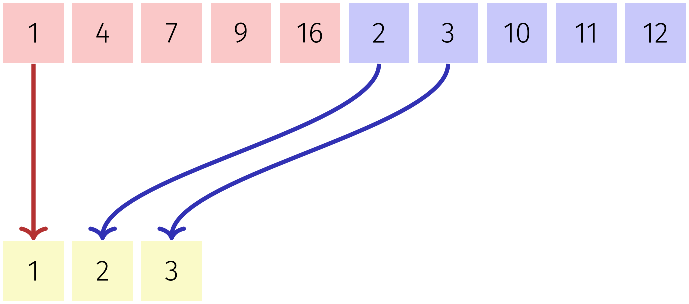
Merge



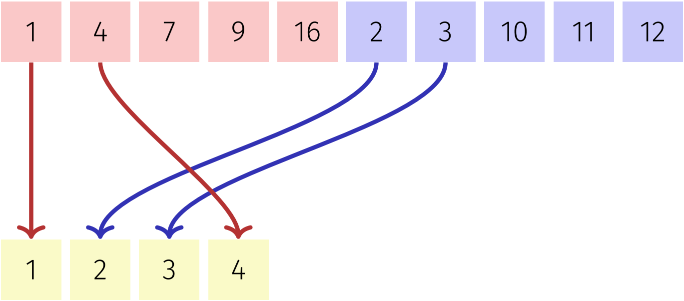
Merge



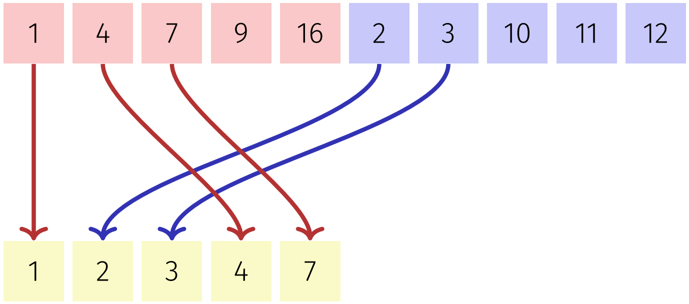
Merge



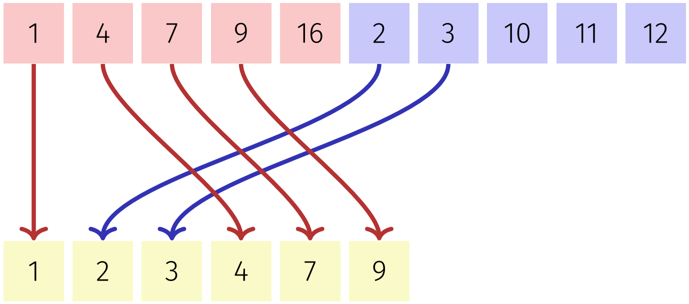
Merge



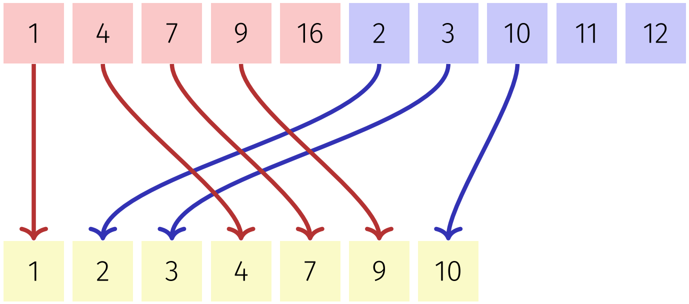
Merge



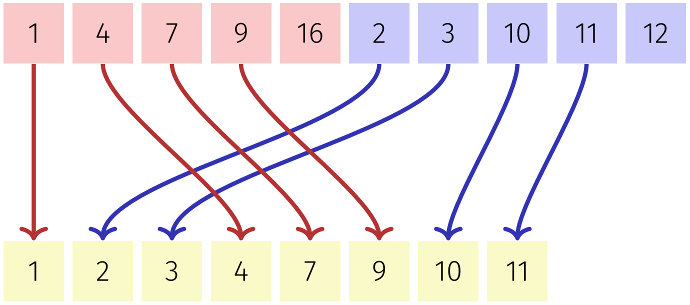
Merge



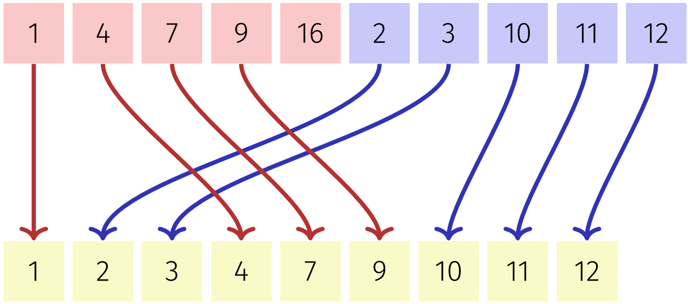
Merge



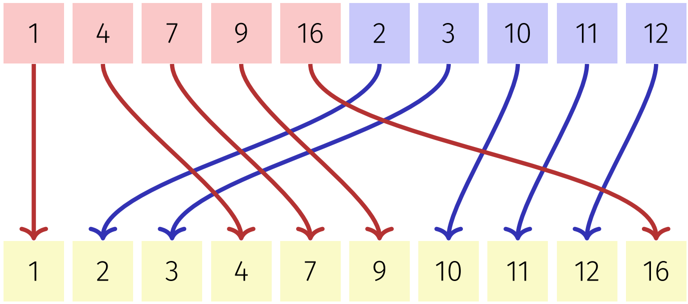
Merge



Merge



Merge



Algorithm Merge(A, l, m, r)

Input: Array A with length n, indexes $1 \leq l \leq m \leq r \leq n$.
$A[l, \dots, m]$, $A[m+1, \dots, r]$ 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 $i \leq m$ and $j \leq r$ **do**
  if $A[i] \leq A[j]$ **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 $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

Mergesort

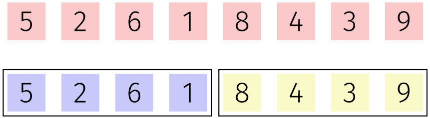
5	2	6	1	8	4	3	9

Mergesort

5	2	6	1	8	4	3	9

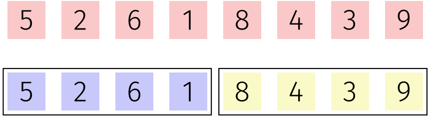
Split

Mergesort



Split

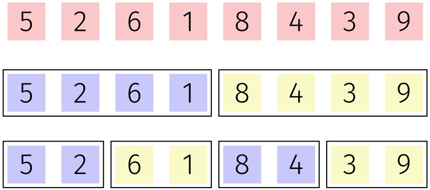
Mergesort



Split

Split

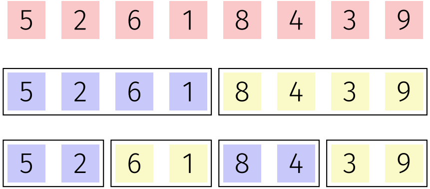
Mergesort



Split

Split

Mergesort

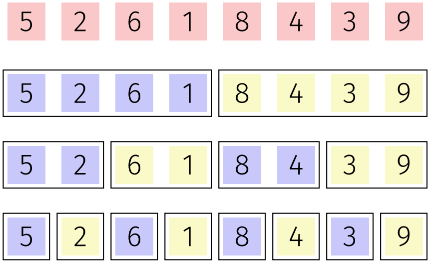


Split

Split

Split

Mergesort

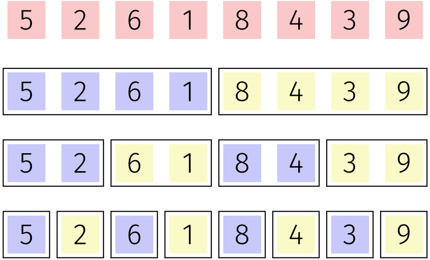


Split

Split

Split

Mergesort



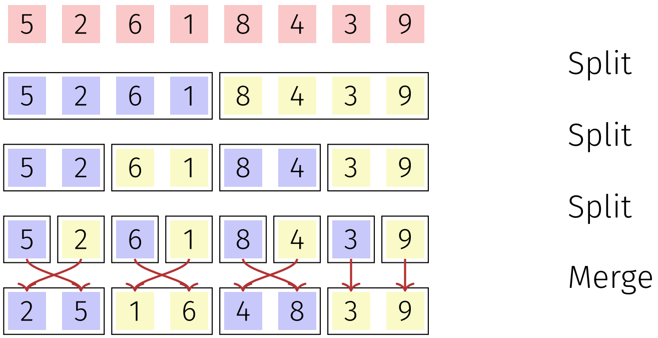
Split

Split

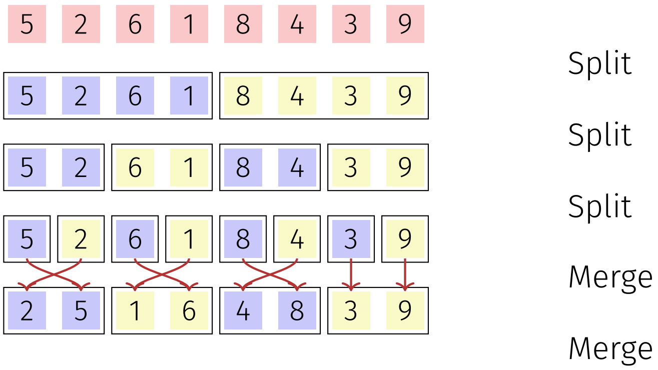
Split

Merge

Mergesort

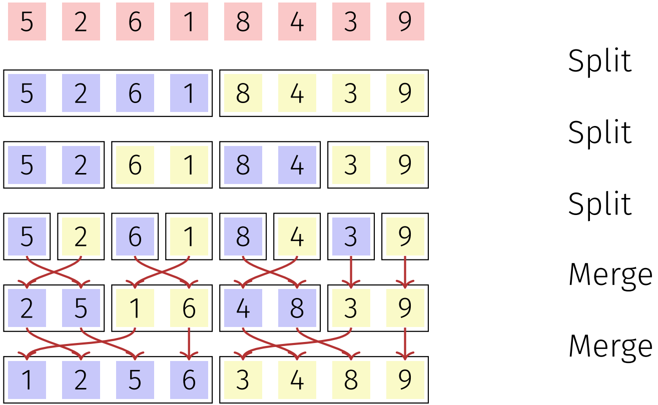

5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
Split
Split
Split
Merge

Mergesort

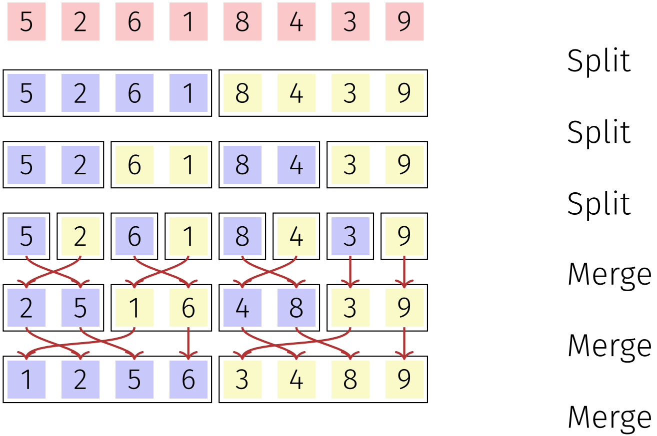


5 2 6 1 8 4 3 9
5 2 6 1 | 8 4 3 9
5 2 | 6 1 | 8 4 | 3 9
5 | 2 | 6 | 1 | 8 | 4 | 3 | 9
2 5 | 1 6 | 4 8 | 3 9
Split
Split
Split
Merge
Merge

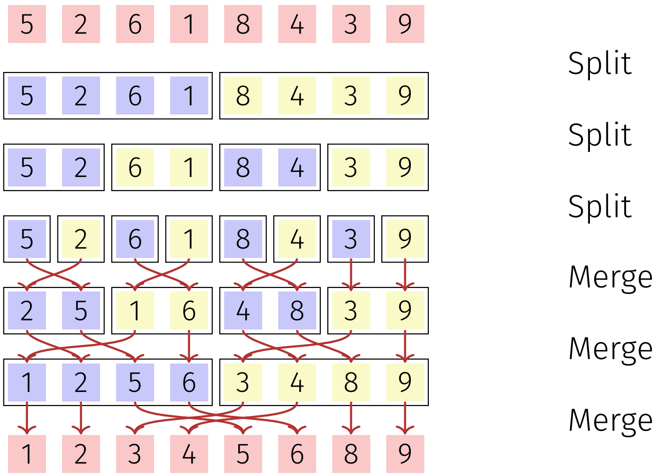
Mergesort


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

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


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
1 2 3 4 5 6 8 9
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**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$ // 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:

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

Analysis

Recursion equation for the number of comparisons and key movements:

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

Derivation for $n = 2^k$

Let $n = 2^k$, $k > 0$. Recurrence

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Apply recursively

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(T(n/8) + cn/4) + cn/2) + cn = \ldots \\ &= 2(2(\ldots(2(2T(n/2^k) + cn/2^{k-1})\ldots) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + \ldots + 2^{k-k}cn/2^{k-k}}_{k\text{terms}} \\ &= nd + cnk = nd + cn\log_2 n \in \Theta(n \log n). \end{aligned}$$

7.2 Quicksort

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

Quicksort

What is the disadvantage of Mergesort?

Quicksort

What is the disadvantage of Mergesort?

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

Quicksort

What is the disadvantage of Mergesort?

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

How could we reduce the merge costs?

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?

Quicksort

What is the disadvantage of Mergesort?

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

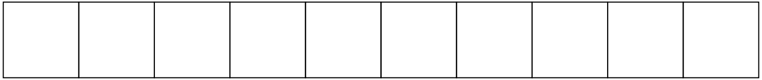
How could we reduce the merge costs?

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

How?

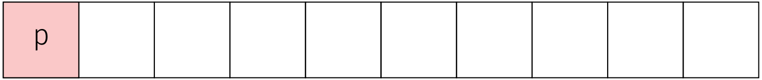
Pivot and Partition!

Use a pivot



Use a pivot

1. Choose a (an arbitrary) **pivot** p



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 $A[i] > p$

p	$>$	$\leq$	$\leq$	$>$	$>$	$\leq$	$\leq$	$>$	$\leq$

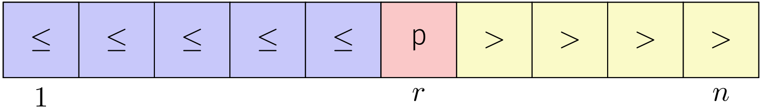
Use a pivot

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

p	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$>$	$>$	$>$	$>$

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 $A[l, \dots, r]$ at least once.
Output: Array A partitioned in $[l, \dots, r]$ 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

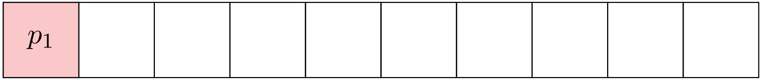
Algorithm `Quicksort`(A, l, r)

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

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

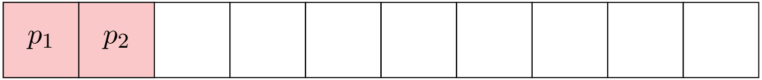
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



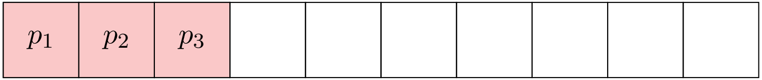
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



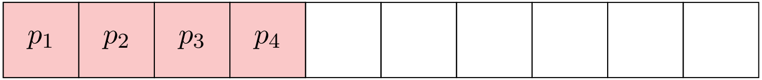
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



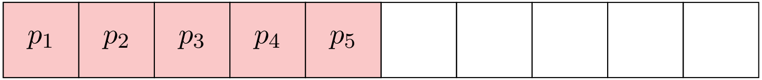
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



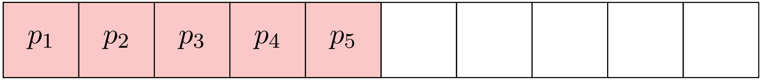
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(n^2)$

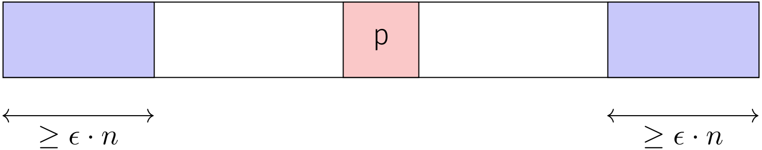


Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(n^2)$

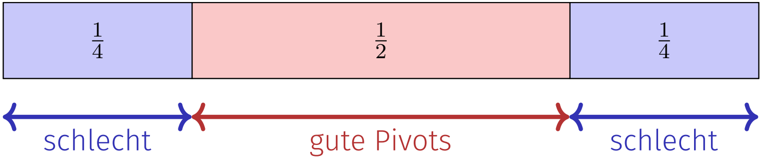


A good pivot has a linear number of elements on both sides.



Choice of the Pivot?

Randomness to our rescue (Tony Hoare, 1961). In each step choose a random pivot.



Probability for a good pivot in one trial: $\frac{1}{2} =: \rho$.

Probability for a good pivot after k trials: $(1-\rho)^{k-1} \cdot \rho$.

Expected number of trials[3]: $1/\rho = 2$

[3] Expected value of the geometric distribution:

Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1

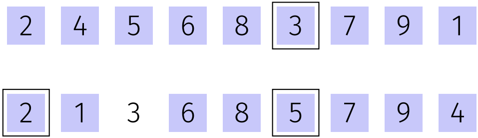
Quicksort (arbitrary pivot)

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

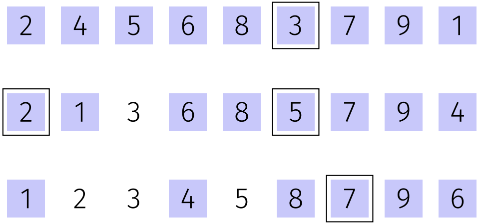
Quicksort (arbitrary pivot)



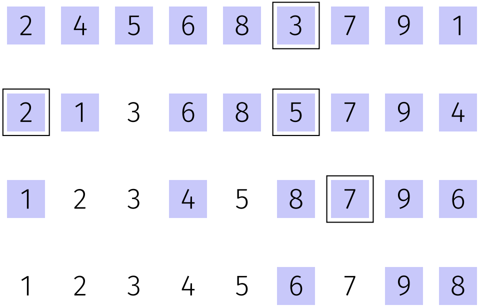
Quicksort (arbitrary pivot)

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

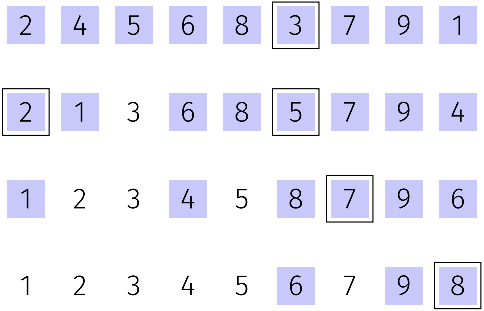
Quicksort (arbitrary pivot)



Quicksort (arbitrary pivot)



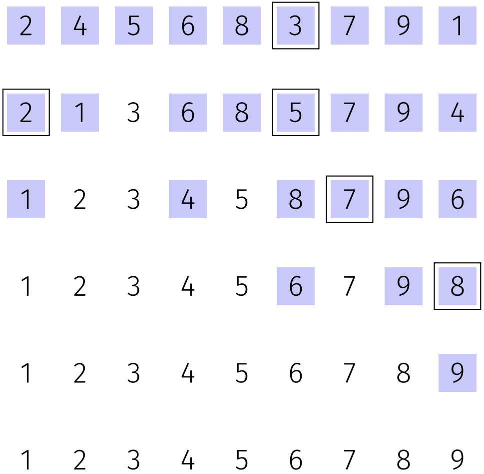
Quicksort (arbitrary pivot)



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)



Analysis: number comparisons

Worst case.

Analysis: number comparisons

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 4

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

(without proof.)

Practical Considerations.

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