Sorting

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Why sorting

Most computational problems involve *data processing*.

Processing data is typically much simpler if that data is *sorted*.

Example

Finding values: LowerBound versus LinearSearch.

Why sorting

Most computational problems involve *data processing*.

Processing data is typically much simpler if that data is *sorted*.

Example

Finding values: LowerBound versus LinearSearch.

The analysis of *sorting* will require universal tools and techniques.
Sort algorithms utilize common design strategies for algorithms.

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \ldots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0\dots N)$ *of distinct weights and a target weight* w*, find all distinct values* $v_1, v_2 \in L$ *with* $w = v_1 + v_2$*.*

L:

1	3	7	9	8	4	10	5

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \ldots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L:

1	3	7	9	8	4	10	5

Target weight: $w = 11$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0\ldots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L:							
1	3	7	9	8	4	10	5

Target weight: $w = 11$.

Algorithm SimpleTwoSum(L, w):

Input: List $L[0\ldots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    for j := i + 1 to N − 1 do
        if L[i] + L[j] = w then
            add (L[i], L[j]) to result.
return result.
```

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \ldots N)$ of N distinct weights, target weight w.

1: *result* := empty bag.
2: **for** $i := 0$ **to** $N - 2$ **do**
3:     **for** $j := i + 1$ **to** $N - 1$ **do**
4:         **if** $L[i] + L[j] = w$ **then**
5:             add $(L[i], L[j])$ to *result*.
6: **return** *result*.

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \ldots N)$ of N distinct weights, target weight w.

```
1: result := empty bag.
2: for i := 0 to N − 2 do
3:     for j := i + 1 to N − 1 do
4:         if L[i] + L[j] = w then
5:             add (L[i], L[j]) to result.
6: return result.
```

$$\sum_{j=i+1}^{N-1} 1$$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \ldots N)$ of N distinct weights, target weight w.

1: *result* := empty bag.
2: **for** $i := 0$ **to** $N-2$ **do**
3:     **for** $j := i+1$ **to** $N-1$ **do**
4:         **if** $L[i] + L[j] = w$ **then**
5:             add $(L[i], L[j])$ to *result*.
6: **return** *result*.

(Lines 3–5:) $$\sum_{j=i+1}^{N-1} 1 = N - (i+1)$$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \dots N)$ of N distinct weights, target weight w.

```
1: result := empty bag.
2: for i := 0 to N − 2 do
3:     for j := i + 1 to N − 1 do
4:         if L[i] + L[j] = w then
5:             add (L[i], L[j]) to result.
6: return result.
```

Lines 3–5: $\sum_{j=i+1}^{N-1} 1 = N - (i + 1)$

Lines 2–5: $\sum_{i=0}^{N-2} (N - (i + 1))$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \ldots N)$ of N distinct weights, target weight w.

```
1: result := empty bag.
2: for i := 0 to N − 2 do
3:     for j := i + 1 to N − 1 do
4:         if L[i] + L[j] = w then
5:             add (L[i], L[j]) to result.
6: return result.
```

Lines 3–5: $\sum_{j=i+1}^{N-1} 1 = N - (i+1)$

Lines 2–5: $\sum_{i=0}^{N-2} (N - (i+1))$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

$$\sum_{i=0}^{N-2} (N - (i+1)) = \sum_{i=0}^{N-2} (N-1) - \sum_{i=0}^{N-2} i$$

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0 \ldots N)$ of N distinct weights, target weight w.

```
1: result := empty bag.
2: for i := 0 to N − 2 do
3:     for j := i + 1 to N − 1 do
4:         if L[i] + L[j] = w then
5:             add (L[i], L[j]) to result.
6: return result.
```

$$\sum_{j=i+1}^{N-1} 1 = N - (i+1)$$

$$\sum_{i=0}^{N-2} (N - (i+1))$$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

$$\sum_{i=0}^{N-2} (N - (i+1)) = \sum_{i=0}^{N-2} (N-1) - \sum_{i=0}^{N-2} i = (N-1)^2 - \frac{(N-2)(N-1)}{2}$$

The power of sorting: The two-sum problem

Algorithm SIMPLETWOSUM(L, w):

Input: List $L[0\ldots N)$ of N distinct weights, target weight w.

1: result := empty bag.
2: **for** $i := 0$ **to** $N-2$ **do**
3:     **for** $j := i+1$ **to** $N-1$ **do**
4:         **if** $L[i] + L[j] = w$ **then**
5:             add $(L[i], L[j])$ to result.
6: **return** result.

Lines 3–5: $\sum_{j=i+1}^{N-1} 1 = N - (i+1)$

Lines 2–5: $\sum_{i=0}^{N-2} (N - (i+1))$

Complexity of SIMPLETWOSUM

For a rough estimate, we can count the number of times Line 4 is executed.

$$\sum_{i=0}^{N-2}(N-(i+1)) = \sum_{i=0}^{N-2}(N-1) - \sum_{i=0}^{N-2} i = (N-1)^2 - \frac{(N-2)(N-1)}{2} = \frac{N(N-1)}{2}$$

The power of sorting: The two-sum problem

Algorithm SimpleTwoSum(L, w):

Input: List $L[0\ldots N)$ of N distinct weights, target weight w.

```
1: result := empty bag.
2: for i := 0 to N − 2 do
3:     for j := i + 1 to N − 1 do
4:         if L[i] + L[j] = w then
5:             add (L[i], L[j]) to result.
6: return result.
```

$$\sum_{j=i+1}^{N-1} 1 = N - (i+1)$$

$$\sum_{i=0}^{N-2} (N - (i+1))$$

Complexity of SimpleTwoSum

For a rough estimate, we can count the number of times Line 4 is executed.

$$\sum_{i=0}^{N-2} (N - (i+1)) = \sum_{i=0}^{N-2} (N-1) - \sum_{i=0}^{N-2} i = (N-1)^2 - \frac{(N-2)(N-1)}{2} = \frac{N(N-1)}{2} = \Theta(N^2).$$

The power of sorting: The two-sum problem

Problem

Given a list $L[0\ldots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):

1	3	4	5	7	8	9	10

Target weight: $w = 11$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \dots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):

1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

The power of sorting: The two-sum problem

Algorithm BETTERTWOSUM(L, w):
Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Complexity of BETTERTWOSUM

For a rough estimate, we can count the cost of each LOWERBOUND call.

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0\dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).   } log₂(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each LowerBound call.

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Cost of line 3 (LowerBound call): $\log_2(N - (i+1))$

Cost of the whole for-loop: $\sum_{i=0}^{N-2} \log_2(N - (i+1))$

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each LowerBound call.

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0\ldots N)$ of N distinct weights, target weight w.

result := empty bag.
for $i := 0$ **to** $N-2$ **do**
    $j :=$ LowerBound(L, $i+1$, N, $w - L[i]$). $\}\ \log_2(N-(i+1))$
    if $j \neq$ 'not found' **then**
        add $(L[i], L[j])$ to *result*.
return *result*.

$$\left.\right\}\ \sum_{i=0}^{N-2} \log_2(N-(i+1))$$

Complexity of BetterTwoSum

For a rough *upper bound* estimate, we can count the cost of each LowerBound call.

$$\sum_{i=0}^{N-2} \log_2(N-(i+1)) \leq \sum_{i=0}^{N-2} \log_2(N)$$

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0\dots N)$ of N distinct weights, target weight w.

1: *result* := empty bag.
2: **for** $i := 0$ **to** $N - 2$ **do**
3: $j :=$ LowerBound(L, $i+1$, N, $w - L[i]$). $\Big\}\ \log_2(N-(i+1))$
4: **if** $j \neq$ 'not found' **then**
5: add $(L[i], L[j])$ to *result*.
6: **return** *result*.

(lines 2–5) $\Bigg\}\ \sum_{i=0}^{N-2} \log_2(N-(i+1))$

Complexity of BetterTwoSum

For a rough *upper bound* estimate, we can count the cost of each LowerBound call.

$$\sum_{i=0}^{N-2} \log_2(N-(i+1)) \leq \sum_{i=0}^{N-2} \log_2(N) = (N-1)\log_2(N).$$

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).      } log_2(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

(Lines 2–5 together: $\sum_{i=0}^{N-2} \log_2(N-(i+1))$)

Complexity of BetterTwoSum

For a rough *lower bound* estimate, we can count the cost of each LowerBound call.

$$\sum_{i=0}^{N-2} \log_2(N-(i+1)) \geq \sum_{i=0}^{\frac{N}{2}-1} \log_2\left(\frac{N}{2}\right)$$

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).      } log_2(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Loop cost (lines 2–5): $\sum_{i=0}^{N-2} \log_2(N - (i+1))$

Complexity of BetterTwoSum

For a rough *lower bound* estimate, we can count the cost of each LowerBound call.

$$\sum_{i=0}^{N-2} \log_2(N - (i+1)) \geq \sum_{i=0}^{\frac{N}{2}-1} \log_2\left(\frac{N}{2}\right) = \frac{N}{2} \log_2\left(\frac{N}{2}\right)$$

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \ldots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).     } log_2(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Cost of line 3 (LowerBound call): $\log_2(N - (i+1))$; total over the loop: $\sum_{i=0}^{N-2} \log_2(N - (i+1))$

Complexity of BetterTwoSum

For a rough *lower bound* estimate, we can count the cost of each LowerBound call.

$$\sum_{i=0}^{N-2} \log_2(N-(i+1)) \geq \sum_{i=0}^{\frac{N}{2}-1} \log_2\left(\frac{N}{2}\right) = \frac{N}{2}\log_2\left(\frac{N}{2}\right) = \frac{N}{2}(\log_2(N) - 1).$$

The power of sorting: The two-sum problem

Algorithm BetterTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).     } log2(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Cost of line 3: $\log_2(N-(i+1))$; cost of the loop: $\sum_{i=0}^{N-2} \log_2(N-(i+1))$

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each LowerBound call.

$$\frac{N}{2}(\log_2(N)-1) \leq \sum_{i=0}^{N-2} \log_2(N-(i+1)) \leq (N-1)\log_2(N).$$

The power of sorting: The two-sum problem

Algorithm $\textsc{BetterTwoSum}(L, w)$:

Input: *Ordered* list $L[0 \ldots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := LowerBound(L, i + 1, N, w − L[i]).      } log2(N − (i + 1))
    if j ≠ 'not found' then
        add (L[i], L[j]) to result.
return result.
```

Cost of line 3: $\log_2(N - (i+1))$. Cost of the whole loop (lines 2–5): $\sum_{i=0}^{N-2} \log_2(N - (i+1))$

Complexity of $\textsc{BetterTwoSum}$

For a rough estimate, we can count the cost of each $\textsc{LowerBound}$ call.

$$\frac{N}{2}(\log_2(N)-1) \leq \sum_{i=0}^{N-2} \log_2(N-(i+1)) \leq (N-1)\log_2(N). \qquad \sum_{i=0}^{N-2} \log_2(N-(i+1)) = \Theta(N \log_2(N)).$$

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \ldots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):

1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \dots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):

1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

Assume that $(L[i], L[j])$ and $(L[k], L[m])$ are in the output with $i < k$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0\dots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):	1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

Assume that $(L[i], L[j])$ and $(L[k], L[m])$ are in the output with $i < k$.

- $L[i] + L[j] = w = L[k] + L[m]$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \dots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):	1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

Assume that $(L[i], L[j])$ and $(L[k], L[m])$ are in the output with $i < k$.

- $L[i] + L[j] = w = L[k] + L[m]$.
- $L[i] < L[k]$ implies $L[j] > L[m]$.

The power of sorting: The two-sum problem

Problem

Given a list $L[0 \dots N)$ of distinct weights and a target weight w, find all distinct values $v_1, v_2 \in L$ with $w = v_1 + v_2$.

L (sorted):							
1	3	4	5	7	8	9	10

Target weight: $w = 11$.

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

Assume that $(L[i], L[j])$ and $(L[k], L[m])$ are in the output with $i < k$.

- $L[i] + L[j] = w = L[k] + L[m]$.
- $L[i] < L[k]$ implies $L[j] > L[m]$.

We can search from both ends in L: position i as a *lower bound* and j as an *upper bound*.

The power of sorting: The two-sum problem

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

We can search from both ends in L: position i as a *lower bound* and j as an *upper bound*.

Algorithm BestTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

result := empty bag.
$i, j := 0, N - 1$.
while $i < j$ **do**
    if $L[i] + L[j] = w$ **then**
        add $(L[i], L[j])$ to *result*.
        $i, j := i + 1, j - 1$.
    else if $L[i] + L[j] < w$ **then**
        $i := i + 1$.
    else
        $j := j - 1$.
return *result*.

The power of sorting: The two-sum problem

Can we do better than $\Theta(N \log_2(N))$ if L is ordered?

We can search from both ends in L: position i as a *lower bound* and j as an *upper bound*.

Algorithm BestTwoSum(L, w):

Input: *Ordered* list $L[0 \dots N)$ of N distinct weights, target weight w.

```
result := empty bag.
i, j := 0, N − 1.
while i < j do
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
return result.
```

Lines 3–10 (the while loop): $\Theta(N)$

Intermezzo: Correctness of BestTwoSum

Warning

Proving the correctness of BestTwoSum in all details is *tricky*!

Intermezzo: Correctness of BestTwoSum

High-level proof steps

```
result := empty bag.
i, j := 0, N − 1.
while i < j do
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
return result.
```

Intermezzo: Correctness of BestTwoSum

High-level proof steps

1. Specify what the *result* should be.

```
result := empty bag.
i, j := 0, N − 1.
while i < j do
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
return result.
```

Intermezzo: Correctness of BestTwoSum

High-level proof steps

1. Specify what the *result* should be.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.
while $i < j$ **do**
    if $L[i] + L[j] = w$ **then**
        add $(L[i], L[j])$ to $result$.
        $i, j := i + 1, j - 1$.
    else if $L[i] + L[j] < w$ **then**
        $i := i + 1$.
    else
        $j := j - 1$.
return $result$. /* $result = \mathsf{TS}(0, N - 1)$. */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

1. Specify what the *result* should be. The *invariant* must establish this result!

Let $\text{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

```
result := empty bag.
i, j := 0, N − 1.
while i < j do
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
return result. /* result = TS(0, N − 1). */
```

Intermezzo: Correctness of BestTwoSum

High-level proof steps

2. Specify the *invariant*.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

```
result := empty bag.
i, j := 0, N − 1.
while i < j do
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
return result. /* result = TS(0, N − 1). */
```

Intermezzo: Correctness of BestTwoSum

High-level proof steps

2. Specify the *invariant*. Look at what you need *after* the loop!

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.
while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ */
    if $L[i] + L[j] = w$ **then**
        add $(L[i], L[j])$ to $result$.
        $i, j := i + 1, j - 1$.
    else if $L[i] + L[j] < w$ **then**
        $i := i + 1$.
    else
        $j := j - 1$.
return $result$. /* $result = \mathsf{TS}(0, N-1)$. */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

3. Prove the *invariant* right *before the loop*.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.

Base case: prove that the invariant holds before the loop.

while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

3. Prove the *invariant* right *before the loop*. Use facts established *before* the loop.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.
Known: we have $i = 0$, $j = N - 1$, and $result = \emptyset$ (due to assignments).

Base case: prove that the invariant holds before the loop.
while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

3. Prove the *invariant* right *before the loop*. Use facts established *before* the loop.

Let $\mathsf{TS}(\mathit{start}, \mathit{end}) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (\mathit{start} \le i < j \le \mathit{end})\}$.

result := empty bag.

$i, j := 0, N - 1$.

Known: we have $i = 0$, $j = N - 1$, and $\mathit{result} = \emptyset$ (due to assignments).

Hence, $\mathsf{TS}(0, N - 1) \setminus \mathsf{TS}(i, j) = \emptyset = \mathit{result}$.

Base case: prove that the invariant holds before the loop.

while $i < j$ **do** /* inv: $\mathit{result} = \mathsf{TS}(0, N - 1) \setminus \mathsf{TS}(i, j)$ */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

3. Prove the *invariant* right *before the loop*. Use facts established *before* the loop.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.
Known: we have $i = 0$, $j = N - 1$, and $result = \emptyset$ (due to assignments).
Hence, $\mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j) = \emptyset = result$.
Base case: the invariant holds before the loop.
while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

4. Prove that the *invariant* is maintaned by the loop.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

```
if L[i] + L[j] = w then
    add (L[i], L[j]) to result.
    i, j := i + 1, j − 1.
else if L[i] + L[j] < w then
    i := i + 1.
else
    j := j − 1.
```

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

5. An if-statement introduces a case distinction: prove each branch separately.

Let $\mathsf{TS}(\mathit{start}, \mathit{end}) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (\mathit{start} \leq i < j \leq \mathit{end})\}$.

Given: invariant and $i < j \rightarrow \mathit{result} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

```
if L[i] + L[j] = w then
    Given: result = TS(0, N − 1) \ TS(i, j), i < j, and L[i] + L[j] = w.

    add (L[i], L[j]) to result.
    i, j := i + 1, j − 1.
```

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

5. An if-statement introduces a case distinction: prove each branch separately.

Let $\mathsf{TS}(\mathit{start}, \mathit{end}) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (\mathit{start} \leq i < j \leq \mathit{end})\}$.

Given: invariant and $i < j \rightarrow \mathit{result} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $\mathit{result} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

add $(L[i], L[j])$ to result.

$i, j := i + 1, j - 1$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

6. Carry over all facts obtained via the assignments.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

add $(L[i], L[j])$ to *result*.

$i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$,
$result_{old} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathsf{TS}(i_{old}, j_{old})$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

7. Complete the proof for this case using all provided facts.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

add $(L[i], L[j])$ to *result*.

$i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$, $result_{old} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathsf{TS}(i_{old}, j_{old})$.

Need to prove: $result_{new} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{new}, j_{new})$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

7. Complete the proof for this case using all provided facts.

Let $\mathrm{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$ and $i < j$.

4: **if** $L[i] + L[j] = w$ **then**

Given: $result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathrm{TS}(i, j)$.

5: add $(L[i], L[j])$ to $result$.

6: $i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$, $result_{old} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathrm{TS}(i_{old}, j_{old})$.

Need to prove: $result_{new} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{new}, j_{new})$.

$$result_{new} = (\mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{old}, j_{old})) \cup \{(L[i_{old}], L[j_{old}])\}.$$

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

7. Complete the proof for this case using all provided facts.

Let $\mathrm{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathrm{TS}(i, j)$.

add $(L[i], L[j])$ to $result$.

$i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$, $result_{old} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathrm{TS}(i_{old}, j_{old})$.

Need to prove: $result_{new} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{new}, j_{new})$.

$$result_{new} = \mathrm{TS}(0, N-1) \setminus (\mathrm{TS}(i_{old}, j_{old}) \setminus \{(L[i_{old}], L[j_{old}])\}).$$

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

7. Complete the proof for this case using all provided facts.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \le i < j \le end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

add $(L[i], L[j])$ to *result*.

$i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$,
$result_{old} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathsf{TS}(i_{old}, j_{old})$.

Need to prove: $result_{new} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{new}, j_{new})$.

$result_{new} = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{new}, j_{new})$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

7. Complete the proof for this case using all provided facts.

Let $\mathrm{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$ and $i < j$.

if $L[i] + L[j] = w$ **then**

Given: $result = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

By $L[i] + L[j] = w$ and the Definition of TS, we have: $(L[i], L[j]) \in \mathrm{TS}(i, j)$.

add $(L[i], L[j])$ to *result*.

$i, j := i + 1, j - 1$.

Known: $result_{new} = result_{old} \cup \{(L[i_{old}], L[j_{old}])\}$, $i_{new} = i_{old} + 1$, $j_{new} = j_{old} - 1$,
$result_{old} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{old}, j_{old})$, and $(L[i_{old}], L[j_{old}]) \in \mathrm{TS}(i_{old}, j_{old})$.

Need to prove: $result_{new} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{new}, j_{new})$.

$result_{new} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{new}, j_{new})$.

Induction step: the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

7: **else if** $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

8: $i := i + 1$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

```
else if L[i] + L[j] < w then
    Given: result = TS(0, N − 1) \ TS(i, j), i < j, and L[i] + L[j] < w.
    By L[i] + L[j] < w and the Definition of TS, we have: (L[i], v) ∉ TS(i, j), ∀v.
    i := i + 1.
```

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

else if $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

By $L[i] + L[j] < w$ and the Definition of TS, we have: $(L[i], v) \notin \mathsf{TS}(i, j), \forall v$.

$i := i + 1$.

Known: $i_{\text{new}} = i_{\text{old}} + 1$,

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{old}}, j)$, and $(L[i_{\text{old}}], v) \notin \mathsf{TS}(i_{\text{old}}, j)$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

else if $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

By $L[i] + L[j] < w$ and the Definition of TS, we have: $(L[i], v) \notin \mathsf{TS}(i, j), \forall v$.

$i := i + 1$.

Known: $i_{\text{new}} = i_{\text{old}} + 1$,
$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{old}}, j)$, and $(L[i_{\text{old}}], v) \notin \mathsf{TS}(i_{\text{old}}, j)$.

Need to prove: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{new}}, j)$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

else if $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

By $L[i] + L[j] < w$ and the Definition of TS, we have: $(L[i], v) \notin \mathsf{TS}(i, j), \forall v$.

$i := i + 1$.

Known: $i_{new} = i_{old} + 1$,

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{old}, j)$, and $(L[i_{old}], v) \notin \mathsf{TS}(i_{old}, j)$.

Need to prove: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{new}, j)$.

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{old}, j)$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \le i < j \le end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

7: **else if** $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

By $L[i] + L[j] < w$ and the Definition of TS, we have: $(L[i], v) \notin \mathsf{TS}(i, j), \forall v$.

8: $i := i + 1$.

Known: $i_{\text{new}} = i_{\text{old}} + 1$,

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{old}}, j)$, and $(L[i_{\text{old}}], v) \notin \mathsf{TS}(i_{\text{old}}, j)$.

Need to prove: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{new}}, j)$.

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{new}}, j)$.

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

8. Next, the *else if* case of the case distinction.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] = w$.

else if $L[i] + L[j] < w$ **then**

Given: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$, $i < j$, and $L[i] + L[j] < w$.

By $L[i] + L[j] < w$ and the Definition of TS, we have: $(L[i], v) \notin \mathsf{TS}(i, j), \forall v$.

$i := i + 1$.

Known: $i_{\text{new}} = i_{\text{old}} + 1$,

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{old}}, j)$, and $(L[i_{\text{old}}], v) \notin \mathsf{TS}(i_{\text{old}}, j)$.

Need to prove: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{new}}, j)$.

$result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i_{\text{new}}, j)$.

Induction step: the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

9. Finally, the *else* case of the case distinction (analogous).

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Given: invariant and $i < j \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i < j$.

```
if L[i] + L[j] = w then
    add (L[i], L[j]) to result.
    i, j := i + 1, j − 1.
else if L[i] + L[j] < w then
    i := i + 1.
else
    j := j − 1.
```

Induction step: prove that the invariant holds after each step of the loop.

Intermezzo: Correctness of BestTwoSum

High-level proof steps

10. The invariant holds!

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

```
result := empty bag.
i, j := 0, N − 1.
while i < j do /* inv: result = TS(0, N − 1) \ TS(i, j) */
    if L[i] + L[j] = w then
        add (L[i], L[j]) to result.
        i, j := i + 1, j − 1.
    else if L[i] + L[j] < w then
        i := i + 1.
    else
        j := j − 1.
Known: invariant and ¬(i < j) → result = TS(0, N − 1) \ TS(i, j) and i ≥ j.
return result. /* result = TS(0, N − 1). */
```

Intermezzo: Correctness of BestTwoSum

High-level proof steps

10. The invariant holds! Do not forget termination of the while-loop.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

$result$:= empty bag.
$i, j := 0, N - 1$.
while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$; bf: $j - i$ */
  if $L[i] + L[j] = w$ **then**
    add $(L[i], L[j])$ to $result$.
    $i, j := i + 1, j - 1$.
  else if $L[i] + L[j] < w$ **then**
    $i := i + 1$.
  else
    $j := j - 1$.
Known: invariant and $\neg(i < j) \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i \geq j$.
return $result$. /* $result = \mathsf{TS}(0, N-1)$. */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

11. Prove the post-condition.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

Known: invariant and $\neg(i < j) \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i \geq j$.

11: **return** $result$. /* $result = \mathsf{TS}(0, N-1)$. */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

11. Prove the post-condition.

Let $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

Known: invariant and $\neg(i < j) \rightarrow result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ and $i \geq j$.

By $i \geq j$ and the Definition of TS, we have $\mathsf{TS}(i, j) = \emptyset$.

11: **return** *result*. /* $result = \mathsf{TS}(0, N-1)$. */

Intermezzo: Correctness of BestTwoSum

High-level proof steps

11. Prove the post-condition.

Let $\text{TS}(\mathit{start}, \mathit{end}) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (\mathit{start} \le i < j \le \mathit{end})\}$.

Known: invariant and $\neg(i < j) \to \mathit{result} = \text{TS}(0, N-1) \setminus \text{TS}(i, j)$ and $i \ge j$.

By $i \ge j$ and the Definition of TS, we have $\text{TS}(i, j) = \emptyset$.

Hence, $\mathit{result} = \text{TS}(0, N-1) \setminus \text{TS}(i, j) = \text{TS}(0, N-1) \setminus \emptyset = \text{TS}(0, N-1)$.

11: **return** result. /* $\mathit{result} = \text{TS}(0, N-1)$. */

Intermezzo: Correctness of BestTwoSum

Warning

You cannot learn correctness proofs from slides: practice on simple algorithms yourself!

A first sort algorithm: SelectionSort

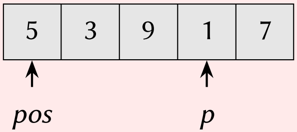
5	3	9	1	7

Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 0$ **to** $N - 2$ **do**
    Find the position p of the *minimum value* in $L[pos \ldots N)$.
    Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SELECTIONSORT

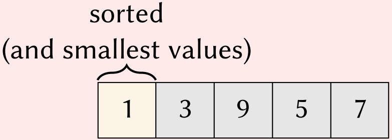


Algorithm SELECTIONSORT(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 0$ **to** $N - 2$ **do**
    Find the position p of the *minimum value* in $L[pos \dots N)$.
    Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SELECTIONSORT

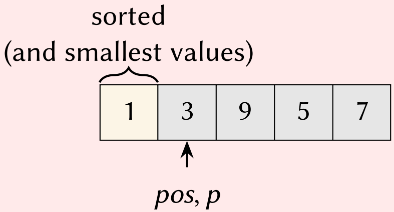


Algorithm SELECTIONSORT(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 0$ **to** $N - 2$ **do**
    Find the position p of the *minimum value* in $L[pos \dots N)$.
    Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SelectionSort



Algorithm SelectionSort(L):

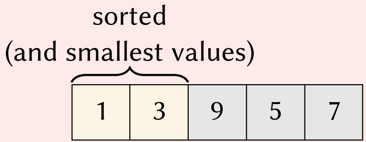
Input: List $L[0 \dots N)$ of N values.

1: **for** $pos := 0$ **to** $N - 2$ **do**

2: Find the position p of the *minimum value* in $L[pos \dots N)$.

3: Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SELECTIONSORT



Algorithm SELECTIONSORT(L):

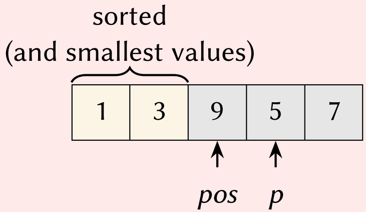
Input: List $L[0 \ldots N)$ of N values.

1: **for** $pos := 0$ **to** $N - 2$ **do**

2: Find the position p of the *minimum value* in $L[pos \ldots N)$.

3: Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SELECTIONSORT



Algorithm SELECTIONSORT(L):

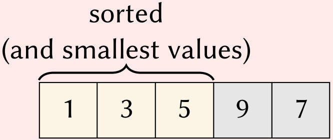
Input: List $L[0 \dots N)$ of N values.

1: **for** $pos := 0$ **to** $N - 2$ **do**

2: Find the position p of the *minimum value* in $L[pos \dots N)$.

3: Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SELECTIONSORT

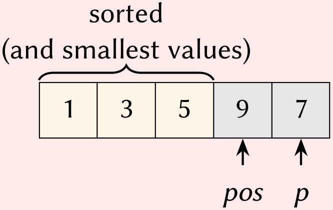


Algorithm SELECTIONSORT(L)**:**

Input: List $L[0 \dots N)$ of N values.

```
for pos := 0 to N − 2 do
    Find the position p of the minimum value in L[pos ... N).
    Exchange L[pos] and L[p].
```

A first sort algorithm: SelectionSort

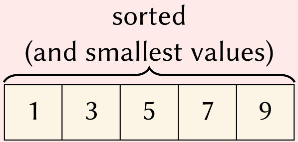


Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 0$ **to** $N - 2$ **do**
    Find the position p of the *minimum value* in $L[pos \ldots N)$.
    Exchange $L[pos]$ and $L[p]$.

A first sort algorithm: SelectionSort



Algorithm SelectionSort(L):

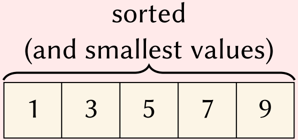
Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    Find the position p of the minimum value in L[pos ... N).
    Exchange L[pos] and L[p].
```

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

A first sort algorithm: SelectionSort



Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    Find the position p of the minimum value in L[pos ... N).    ⟵ ?
    Exchange L[pos] and L[p].
```

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0 \dots N)$ of N values.

```
for pos := 0 to N − 2 do
    Find the position p of the minimum value in L[pos ... N).
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 0$ **to** $N - 2$ **do**
    $p := pos$.
    for $i := pos + 1$ **to** $N - 1$ **do**
        if $L[i] < L[p]$ **then**
            $p := i$.
    Exchange $L[pos]$ and $L[p]$.

(Lines 3–5:) Comparisons: $N - 1 - pos$. Changes: 0.

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Comparisons: $\sum_{pos=0}^{N-2} (N - 1 - pos)$.

Changes: $2(N - 1)$.

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Comparisons: $\sum_{pos=0}^{N-2} (N - 1 - pos)$.

Changes: $2(N - 1)$.

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

$$\text{Comparisons: } \sum_{pos=0}^{N-2} (N - 1 - pos)$$

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0\ldots N)$ of N values.

for $pos := 0$ **to** $N-2$ **do**
  $p := pos$.
  for $i := pos + 1$ **to** $N-1$ **do**
    if $L[i] < L[p]$ **then**
      $p := i$.
  Exchange $L[pos]$ and $L[p]$.

Comparisons: $\sum_{pos=0}^{N-2}(N-1-pos)$.

Changes: $2(N-1)$.

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

$$\text{Comparisons: } \sum_{pos=0}^{N-2}(N-1-pos) = \sum_{j=1}^{N-1} j$$

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Comparisons: $\sum_{pos=0}^{N-2} (N - 1 - pos)$.

Changes: $2(N-1)$.

Runtime complexity of SelectionSort

A good estimate: number of comparisons and changes to list values.

$$\text{Comparisons: } \sum_{pos=0}^{N-2} (N - 1 - pos) = \sum_{j=1}^{N-1} j = \frac{N(N-1)}{2}$$

A first sort algorithm: SELECTIONSORT

Algorithm SELECTIONSORT(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 0 to N − 2 do
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Comparisons: $\sum_{pos=0}^{N-2} (N - 1 - pos) = \Theta(N^2)$.

Changes: $2(N - 1) = \Theta(N)$.

Runtime complexity of SELECTIONSORT

A good estimate: number of comparisons and changes to list values.

$$\text{Comparisons: } \sum_{pos=0}^{N-2} (N - 1 - pos) = \sum_{j=1}^{N-1} j = \frac{N(N-1)}{2} = \Theta(N^2).$$

A first sort algorithm: SelectionSort

Algorithm SelectionSort(L):

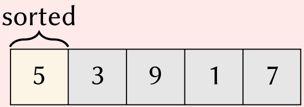
Input: List $L[0 \dots N)$ of N values.

```
for pos := 0 to N − 2 do
    p := pos.
    for i := pos + 1 to N − 1 do
        if L[i] < L[p] then
            p := i.
    Exchange L[pos] and L[p].
```

Correctness of SelectionSort: Some tips

- Rework the for-loops into while loops.
- The inner loop only changes p: prove whatever that loop does separately.
- Include as much information into the invariant of the outer loop.
  What exactly do we know about the values in $L[0 \dots pos)$.
- A complete proof guarantees that list L keeps all original values!

A second sort algorithm: InsertionSort

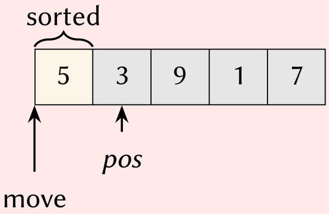


Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \ldots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

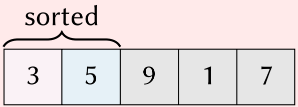


Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

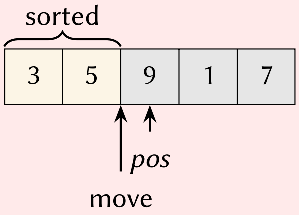


Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \ldots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

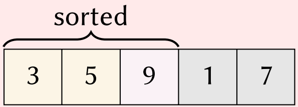


Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
  $v := L[pos]$.
  Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
  $L[p] := v$.

A second sort algorithm: InsertionSort

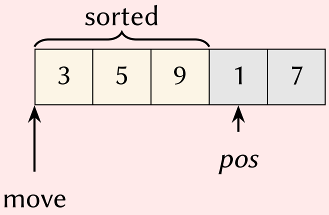


Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

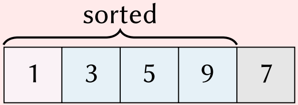


Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \ldots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

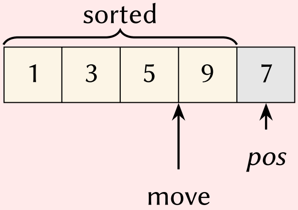


Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

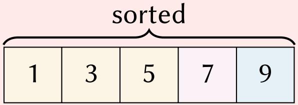


Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \ldots pos)$ with $v < w$ one to the right.
    $L[p] := v$.

A second sort algorithm: InsertionSort

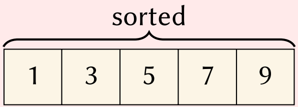


Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
  $v := L[pos]$.
  Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
  $L[p] := v$.

A second sort algorithm: InsertionSort



Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right. $\longleftarrow$?
    $L[p] := v$.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    Move all values $w \in L[0 \ldots pos)$ with $v < w$ one to the right.
    $p := pos$.
    while $p > 0$ **and** $v < L[p - 1]$ **do**
        $L[p] := L[p - 1]$.
        $p := p - 1$.
    $L[p] := v$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    $p := pos$.
    while $p > 0$ **and** $v < L[p - 1]$ **do**
        $L[p] := L[p - 1]$.
        $p := p - 1$.
    $L[p] := v$.

(while loop:) Comparisons: $\leq pos$. Changes: $\leq pos$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N - 1$ **do**
    $v := L[pos]$.
    $p := pos$.
    while $p > 0$ **and** $v < L[p-1]$ **do**
        $L[p] := L[p-1]$.
        $p := p - 1$.
    $L[p] := v$.

Comparisons: $\leq \sum_{pos=1}^{N-1} pos$.

Changes: $\leq \sum_{pos=1}^{N-1} (1 + pos)$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

```
for pos := 1 to N − 1 do
    v := L[pos].
    p := pos.
    while p > 0 and v < L[p − 1] do
        L[p] := L[p − 1].
        p := p − 1.
    L[p] := v.
```

Comparisons: $\leq \sum_{pos=1}^{N-1} pos = \frac{N(N-1)}{2}$.

Changes: $\leq \sum_{pos=1}^{N-1} (1 + pos) = \frac{N(N-1)}{2} + N - 1$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0\dots N)$ of N values.

for $pos := 1$ **to** $N-1$ **do**
    $v := L[pos]$.
    $p := pos$.
    while $p > 0$ **and** $v < L[p-1]$ **do**
        $L[p] := L[p-1]$.
        $p := p-1$.
    $L[p] := v$.

Comparisons: $\leq \sum_{pos=1}^{N-1} pos = \frac{N(N-1)}{2}$.

Changes: $\leq \sum_{pos=1}^{N-1} (1+pos) = \frac{N(N-1)}{2} + N - 1$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

When does InsertionSort have N^2 comparisons and changes?

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \ldots N)$ of N values.

for $pos := 1$ **to** $N-1$ **do**
    $v := L[pos]$.
    $p := pos$.
    while $p > 0$ **and** $v < L[p-1]$ **do**
        $L[p] := L[p-1]$.
        $p := p - 1$.
    $L[p] := v$.

$$\text{Comparisons: } \leq \sum_{pos=1}^{N-1} pos = \frac{N(N-1)}{2}.$$

$$\text{Changes: } \leq \sum_{pos=1}^{N-1} (1 + pos) = \frac{N(N-1)}{2} + N - 1.$$

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

When does InsertionSort have N^2 comparisons and changes?

Reverse-ordered array: every next array is moved to the start of the list.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

for $pos := 1$ **to** $N-1$ **do**
  $v := L[pos]$.
  $p := pos$.
  while $p > 0$ **and** $v < L[p-1]$ **do**
    $L[p] := L[p-1]$.
    $p := p - 1$.
  $L[p] := v$.

Comparisons: $\leq \sum_{pos=1}^{N-1} pos = \frac{N(N-1)}{2}$.

Changes: $\leq \sum_{pos=1}^{N-1} (1 + pos) = \frac{N(N-1)}{2} + N - 1$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

When does InsertionSort have less than N^2 comparisons and changes?

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values.

```
for pos := 1 to N − 1 do
    v := L[pos].
    p := pos.
    while p > 0 and v < L[p − 1] do
        L[p] := L[p − 1].
        p := p − 1.
    L[p] := v.
```

Comparisons: $\leq \sum_{pos=1}^{N-1} pos = \frac{N(N-1)}{2}$.

Changes: $\leq \sum_{pos=1}^{N-1} (1 + pos) = \frac{N(N-1)}{2} + N - 1$.

Runtime complexity of InsertionSort

A good estimate: number of comparisons and exchanges of list values.

When does InsertionSort have less than N^2 comparisons and changes?

Ordered array: N comparisons and changes as every value stays in place.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

Input: List $L[0 \dots N)$ of N values, $L = \mathcal{L}$.

$pos := 1$.
while $pos \neq N$ **do**
    /* inv: $L[0 \dots pos)$ is ordered and L holds the same values as $\mathcal{L}$, bf: $N - pos$. */
    $v := L[pos]$.
    $p := pos$.
    while $p > 0$ **and** $v < L[p-1]$ **do**
        /* inv: $F = L[0 \dots p)$ is ordered, $S = L[p+1 \dots pos+1)$ is ordered, all values in F are smaller than the values in S, all values in S are larger than v, and the values in F, S, $[v]$, and $L[pos+1 \dots, N)$ are exactly the values in $\mathcal{L}$, bf: p. */
        $L[p] := L[p-1]$.
        $p := p - 1$.
    $L[p] := v$.
    $pos := pos + 1$.

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

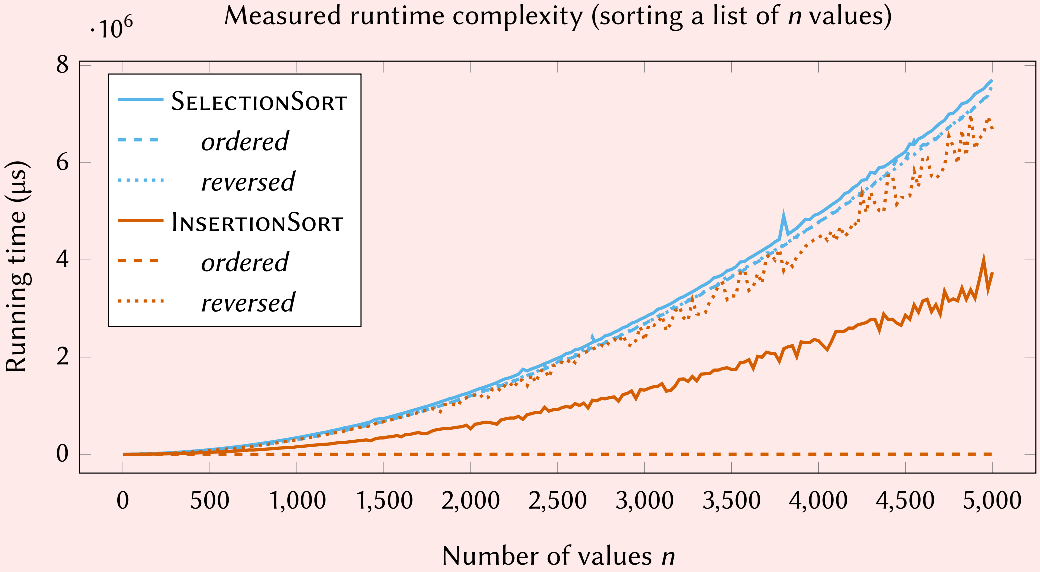
Input: List $L[0 \dots N)$ of N values, $L = \mathcal{L}$.

$pos := 1$.
while $pos \neq N$ **do**
  /* inv: $L[0 \dots pos)$ is ordered and L holds the same values as $\mathcal{L}$, bf: $N - pos$. */
  $v := L[pos]$.
  $p := pos$.
  while $p > 0$ **and** $v < L[p-1]$ **do**
    /* inv: $F = L[0 \dots p)$ is ordered, $S = L[p+1 \dots pos+1)$ is ordered, all values in F are smaller than the values in S, all values in S are larger than v, and the values in F, S, $[v]$, and $L[pos+1 \dots, N)$ are exactly the values in $\mathcal{L}$, bf: p. */
    $L[p] := L[p-1]$.
    $p := p - 1$.
  $L[p] := v$.
  $pos := pos + 1$.

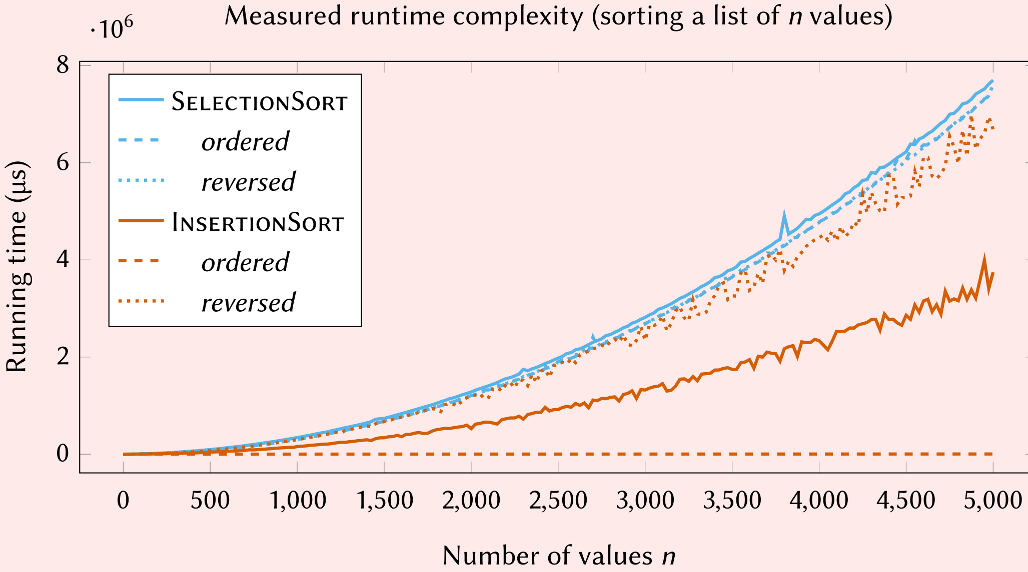
A summary of basic sorting

	Comparisons	Changes	Memory
SelectionSort	$\Theta(N^2)$	$\Theta(N)$	$\Theta(1)$
InsertionSort	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\Theta(1)$

A summary of basic sorting



A summary of basic sorting



Toward faster sorting

The issue with SELECTIONSORT and INSERTIONSORT

- The algorithms do not perform "global reorderings".
- The algorithms sort one element at a time.
  E.g., small elements at the end of the list are moved to the front one at a time.

MergeSortR: Sorting using divide-and-conquer

Divide-and-conquer

Divide Turn problem into smaller subproblems.

Conquer Solve the smaller subproblems using *recursion*.

Combine Combine the subproblem solutions into a final solution.

MergeSortR: Sorting using divide-and-conquer

Divide-and-conquer

Divide Turn problem into smaller subproblems.

Conquer Solve the smaller subproblems using *recursion*.

Combine Combine the subproblem solutions into a final solution.

LowerBoundRec is a divide-and-conquer algorithm.

MergeSortR: Sorting using divide-and-conquer

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Break the list in two halves.

Conquer Solve the smaller subproblems using *recursion*.

Combine Combine the subproblem solutions into a final solution.

MergeSortR: Sorting using divide-and-conquer

Divide-and-conquer

Divide
: Turn problem into smaller subproblems.
Break the list in two halves.

Conquer
: Solve the smaller subproblems using *recursion*.
Sort both halves separately: by recursion, we reach lists with one element.

Combine
: Combine the subproblem solutions into a final solution.

MergeSortR: Sorting using divide-and-conquer

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Break the list in two halves.

Conquer Solve the smaller subproblems using *recursion*.
Sort both halves separately: by recursion, we reach lists with one element.

Combine Combine the subproblem solutions into a final solution.
Merge two sorted halves together to obtain the result.

MergeSortR: High-level overview

Algorithm MergeSortR($L[start \ldots end]$):

2	6	3	5	1	4

MergeSortR: High-level overview

Algorithm MergeSortR($L[start \ldots end]$):

if $end - start > 1$ **then**

else return L.

2	6	3	5	1	4

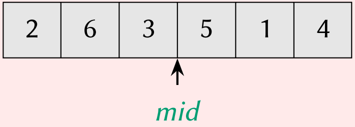
MergeSortR: High-level overview

Algorithm MergeSortR($L[start \ldots end]$):

if $end - start > 1$ **then**

  $mid := (end - start) \text{ div } 2.$

else return L.



MergeSortR: High-level overview

Algorithm MergeSortR($L[start \ldots end]$):

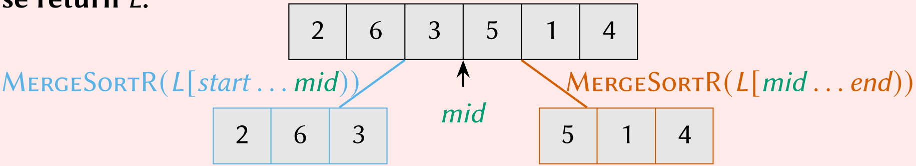
if $end - start > 1$ **then**

  $mid := (end - start) \text{ div } 2.$

  $L_1 :=$ MergeSortR($L[start \ldots mid]$).

  $L_2 :=$ MergeSortR($L[mid \ldots end]$).

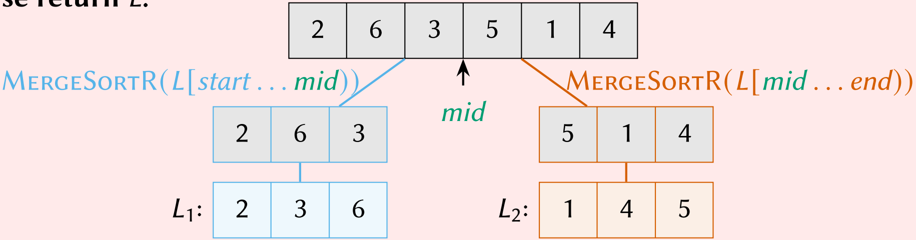
else return L.



MergeSortR: High-level overview

Algorithm MergeSortR($L[start \dots end]$):

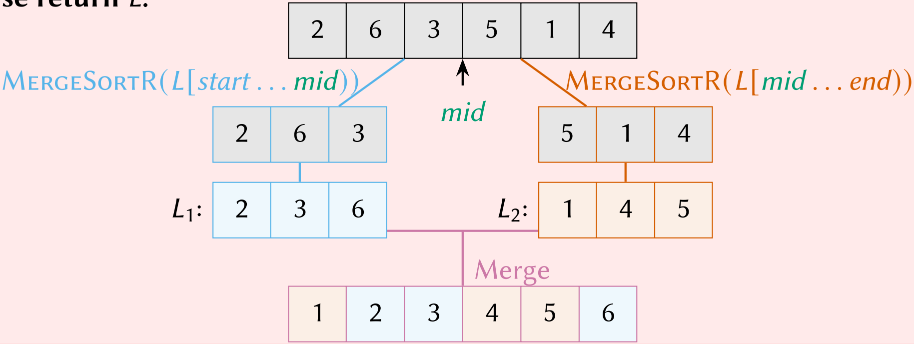
```
if end − start > 1 then
    mid := (end − start) div 2.
    L1 := MergeSortR(L[start ... mid]).
    L2 := MergeSortR(L[mid ... end]).

else return L.
```



MergeSortR: High-level overview

Algorithm MergeSortR($L[start \ldots end)$):

if $end - start > 1$ **then**
    $mid := (end - start) \text{ div } 2.$
    $L_1 :=$ MergeSortR($L[start \ldots mid)$).
    $L_2 :=$ MergeSortR($L[mid \ldots end)$).
    return Merge(L_1, L_2)   (maintain sorted order).
else return L.



Proof of correctness: MERGESORTR($L[start \ldots end]$) sorts

Proof of correctness: MergeSortR($L[start \ldots end)$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Proof of correctness: MergeSortR($L[start \dots end]$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Proof of correctness: MERGESORTR($L[start \ldots end]$) sorts

Base case MERGESORTR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MERGESORTR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MERGESORTR with $2 \leq end - start = n$ values.

Proof of correctness: MergeSortR($L[start \dots end]$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \leq end - start = n$ values.

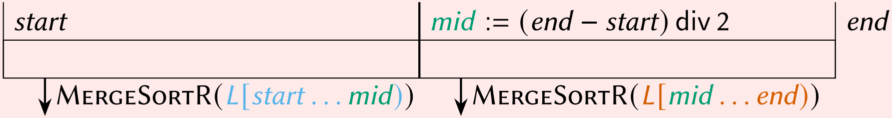
start	*mid* := ($end - start$) div 2	*end*

Proof of correctness: MergeSortR($L[start \dots end)$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \leq end - start = n$ values.

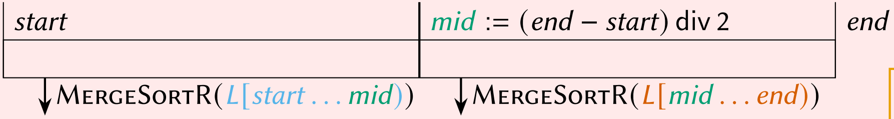


Proof of correctness: MergeSortR($L[start \ldots end)$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \leq end - start = n$ values.



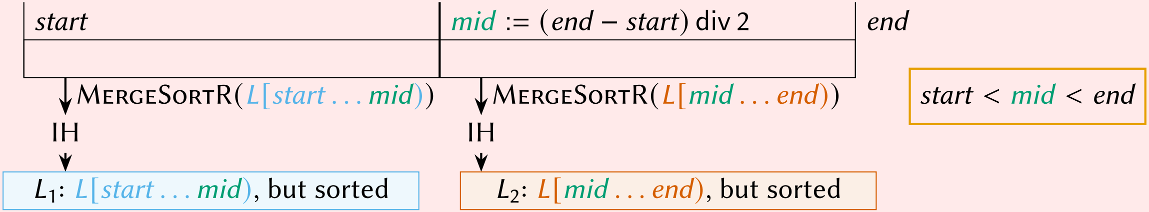
$start < mid < end$

Proof of correctness: MergeSortR($L[start \ldots end)$) sorts

Base case MergeSortR sorts $0 \le end - start \le 1$ values.

Induction hypothesis MergeSortR sorts $0 \le end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \le end - start = n$ values.

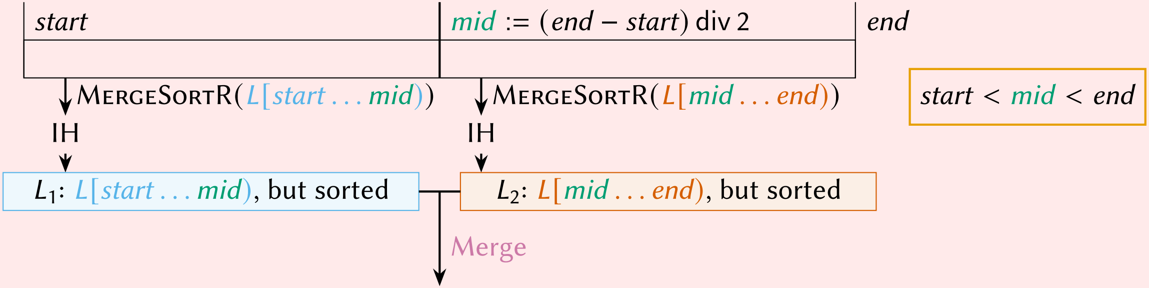


Proof of correctness: MergeSortR($L[start \dots end)$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \leq end - start = n$ values.

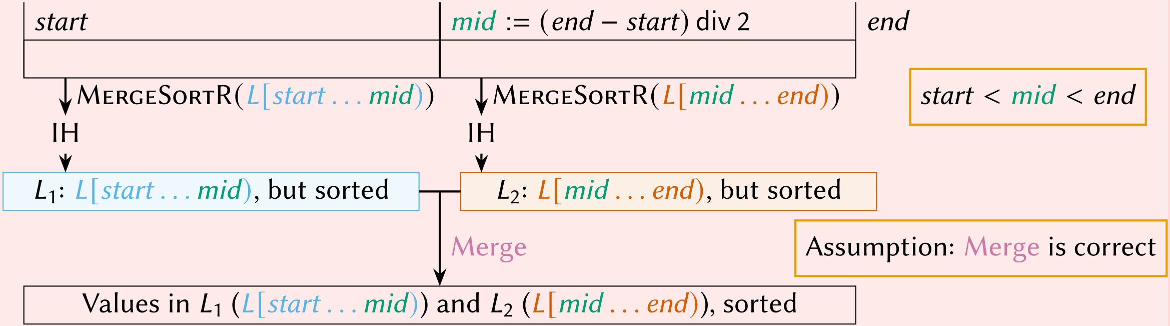


Proof of correctness: MergeSortR($L[start \dots end)$) sorts

Base case MergeSortR sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis MergeSortR sorts $0 \leq end - start < n$ values correctly.

Induction step Consider MergeSortR with $2 \leq end - start = n$ values.



Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \ldots N_1), L_2[0 \ldots N_2))$**:**

Input: L_1 and L_2 are sorted.

Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \dots N_1), L_2[0 \dots N_2))$**:**

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.

return R.

Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$)**:**

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.

$i_1, i_2 := 0, 0.$

while $i_1 < N_1$ **or** $i_2 < N_2$ **do**

return R.

Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \ldots N_1), L_2[0 \ldots N_2))$:

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.
$i_1, i_2 := 0, 0$.
while $i_1 < N_1$ **or** $i_2 < N_2$ **do**
    if $i_2 = N_2$ **or** ($i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2]$) **then**
        $R[i_1 + i_2] := L_1[i_1]$.
        $i_1 := i_1 + 1$.
    else
        $R[i_1 + i_2] := L_2[i_2]$.
        $i_2 := i_2 + 1$.
return R.

Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \ldots N_1), L_2[0 \ldots N_2))$:

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.

$i_1, i_2 := 0, 0$.

while $i_1 < N_1$ **or** $i_2 < N_2$ **do**

 if $i_2 = N_2$ **or** $(i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2])$ **then**

  $R[i_1 + i_2] := L_1[i_1]$.

  $i_1 := i_1 + 1$.

 else

  $R[i_1 + i_2] := L_2[i_2]$.

  $i_2 := i_2 + 1$.

return R.

L_1:

2	3	6

L_2:

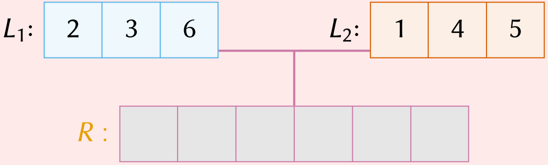
1	4	5

Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or (i1 < N1 and L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```

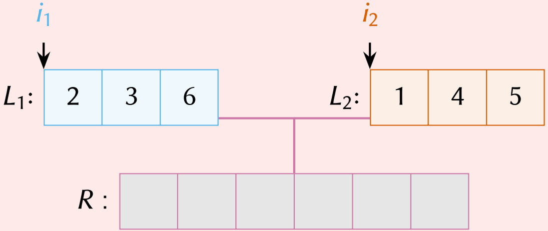


Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \ldots N_1), L_2[0 \ldots N_2))$:

Input: L_1 and L_2 are sorted.

```
R is a new array for N_1 + N_2 values.
i_1, i_2 := 0, 0.
while i_1 < N_1 or i_2 < N_2 do
    if i_2 = N_2 or (i_1 < N_1 and L_1[i_1] < L_2[i_2]) then
        R[i_1 + i_2] := L_1[i_1].
        i_1 := i_1 + 1.
    else
        R[i_1 + i_2] := L_2[i_2].
        i_2 := i_2 + 1.
return R.
```

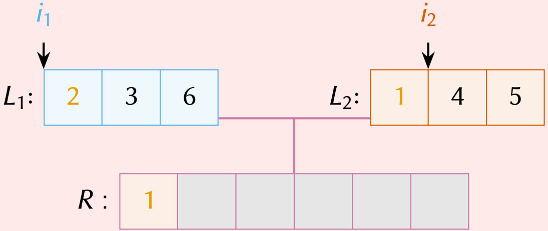


Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0\ldots N_1), L_2[0\ldots N_2))$**:**

Input: L_1 and L_2 are sorted.

```
R is a new array for N_1 + N_2 values.
i_1, i_2 := 0, 0.
while i_1 < N_1 or i_2 < N_2 do
    if i_2 = N_2 or (i_1 < N_1 and L_1[i_1] < L_2[i_2]) then
        R[i_1 + i_2] := L_1[i_1].
        i_1 := i_1 + 1.
    else
        R[i_1 + i_2] := L_2[i_2].
        i_2 := i_2 + 1.
return R.
```

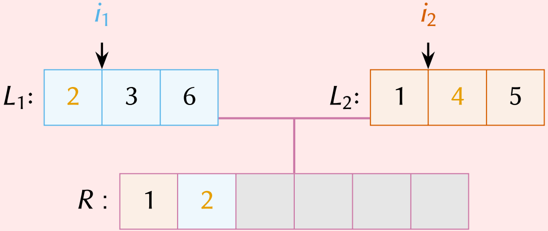


Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or (i1 < N1 and L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```

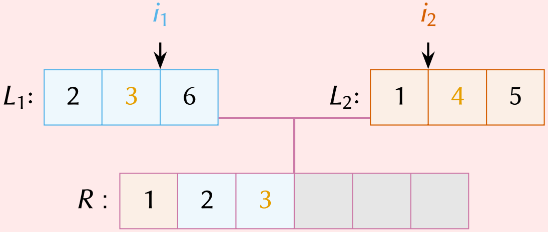


Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \dots N_1), L_2[0 \dots N_2))$:

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.
$i_1, i_2 := 0, 0$.
while $i_1 < N_1$ **or** $i_2 < N_2$ **do**
    if $i_2 = N_2$ **or** $(i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2])$ **then**
        $R[i_1 + i_2] := L_1[i_1]$.
        $i_1 := i_1 + 1$.
    else
        $R[i_1 + i_2] := L_2[i_2]$.
        $i_2 := i_2 + 1$.
return R.

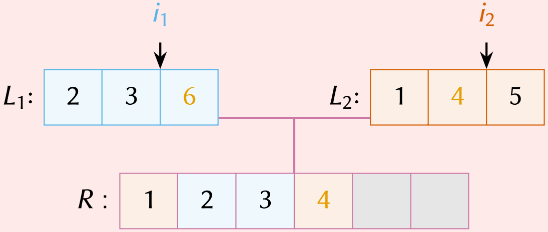


Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.
$i_1, i_2 := 0, 0$.
while $i_1 < N_1$ **or** $i_2 < N_2$ **do**
    if $i_2 = N_2$ **or** ($i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2]$) **then**
        $R[i_1 + i_2] := L_1[i_1]$.
        $i_1 := i_1 + 1$.
    else
        $R[i_1 + i_2] := L_2[i_2]$.
        $i_2 := i_2 + 1$.
return R.

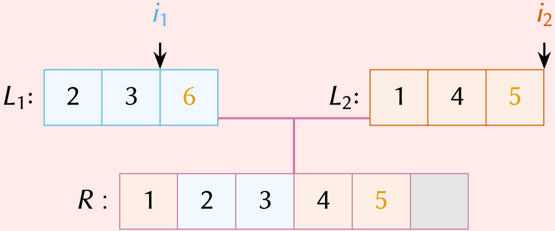


Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or (i1 < N1 and L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```

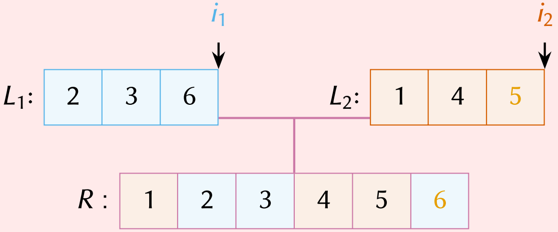


Assumption: Merge is correct

Algorithm $\text{Merge}(L_1[0 \ldots N_1), L_2[0 \ldots N_2))$:

Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.
$i_1, i_2 := 0, 0.$
while $i_1 < N_1$ **or** $i_2 < N_2$ **do**
    if $i_2 = N_2$ **or** $(i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2])$ **then**
        $R[i_1 + i_2] := L_1[i_1].$
        $i_1 := i_1 + 1.$
    else
        $R[i_1 + i_2] := L_2[i_2].$
        $i_2 := i_2 + 1.$
return R.

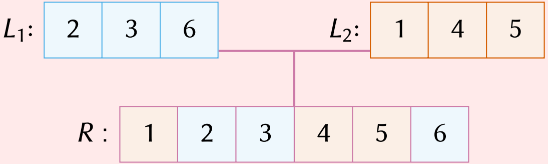


Assumption: Merge is correct

Algorithm MERGE($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or (i1 < N1 and L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```



Assumption: Merge is correct

Algorithm $\textsc{Merge}(L_1[0 \dots N_1), L_2[0 \dots N_2))$:

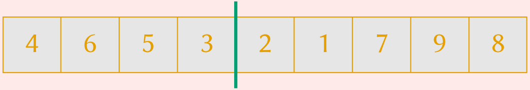
Input: L_1 and L_2 are sorted.

R is a new array for $N_1 + N_2$ values.
$i_1, i_2 := 0, 0$.
while $i_1 < N_1$ **or** $i_2 < N_2$ **do**
  /* inv: $R[0 \dots i_1 + i_2)$ has all values from $L_1[0 \dots i_1)$ and $L_2[0 \dots i_2)$, sorted. */
  /* bf: $(N_1 + N_2) - (i_1 + i_2)$. */
  if $i_2 = N_2$ **or** $(i_1 < N_1$ **and** $L_1[i_1] < L_2[i_2])$ **then**
    $R[i_1 + i_2] := L_1[i_1]$.
    $i_1 := i_1 + 1$.
  else
    $R[i_1 + i_2] := L_2[i_2]$.
    $i_2 := i_2 + 1$.
return R.

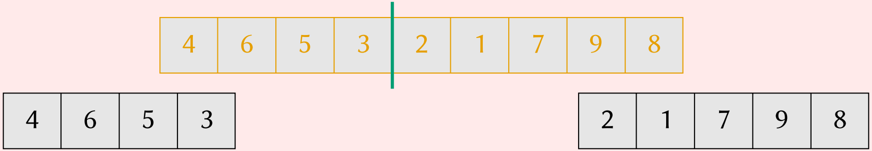
MergeSortR: A complete example

4	6	5	3	2	1	7	9	8

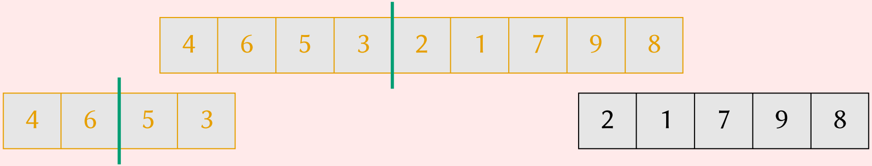
MergeSortR: A complete example



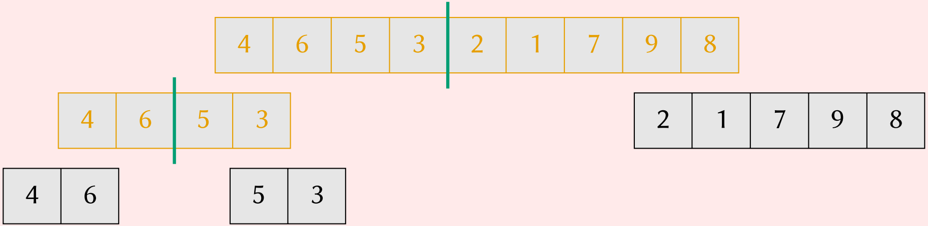
MergeSortR: A complete example



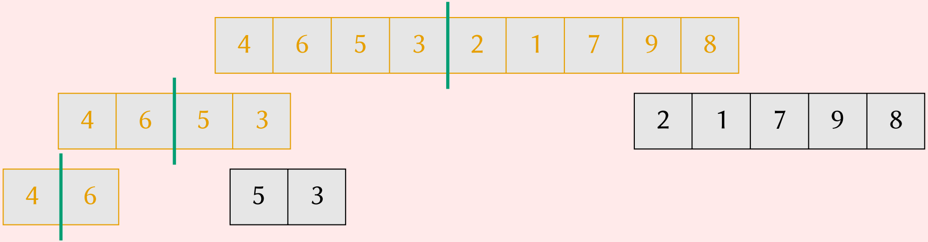
MergeSortR: A complete example



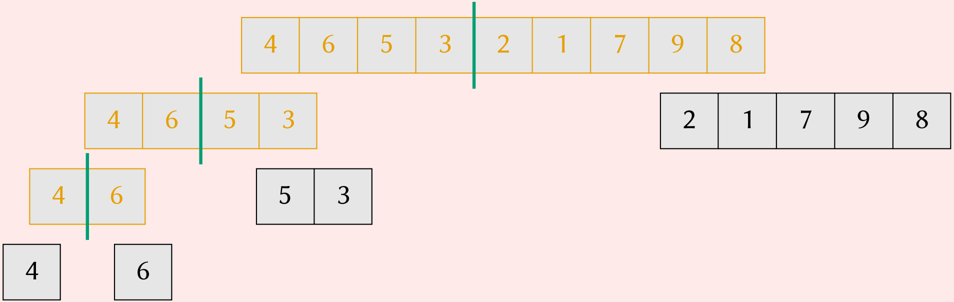
MergeSortR: A complete example



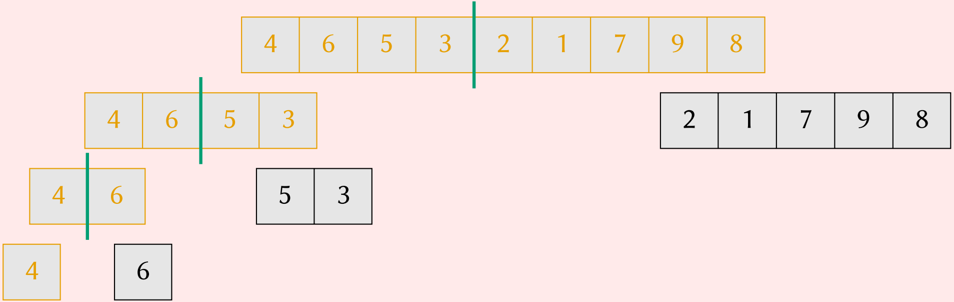
MergeSortR: A complete example



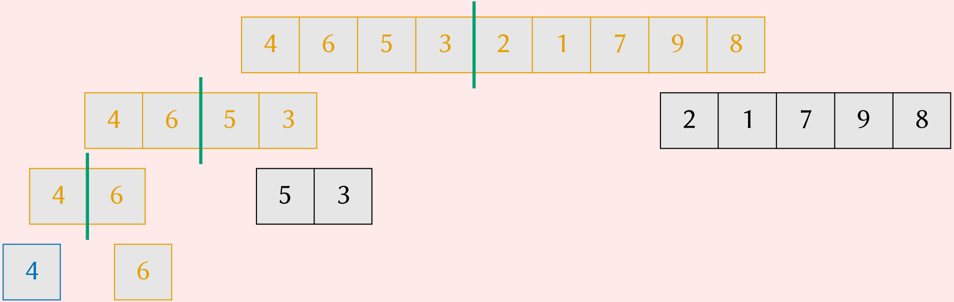
MergeSortR: A complete example



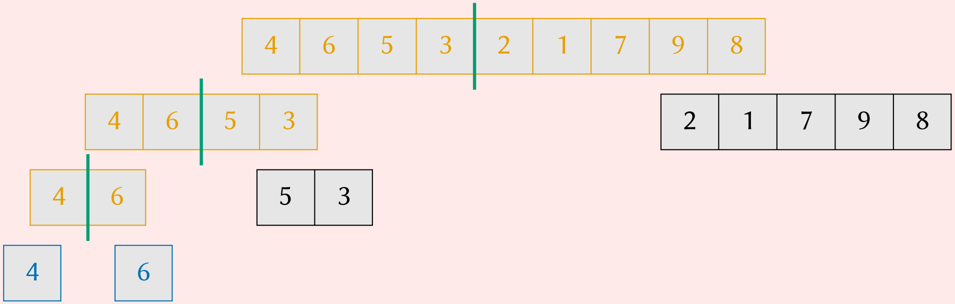
MergeSortR: A complete example



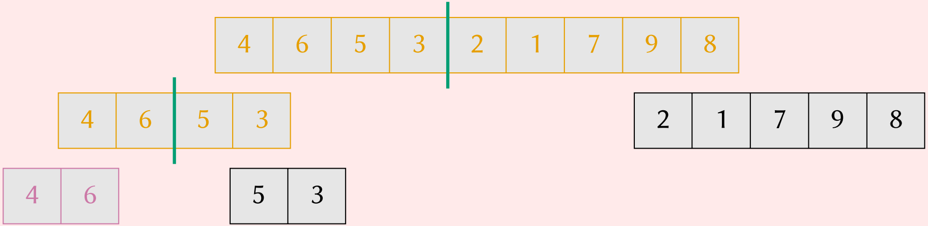
MergeSortR: A complete example



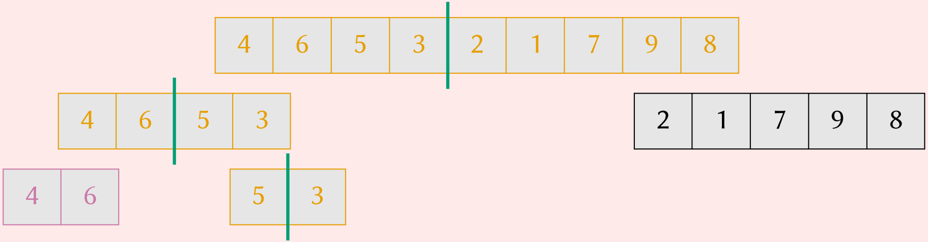
MergeSortR: A complete example



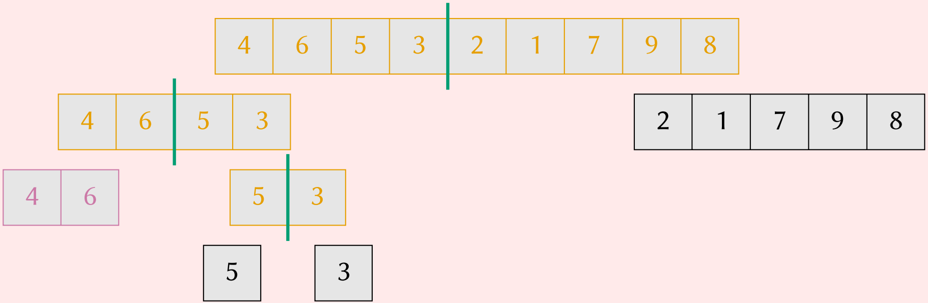
MergeSortR: A complete example



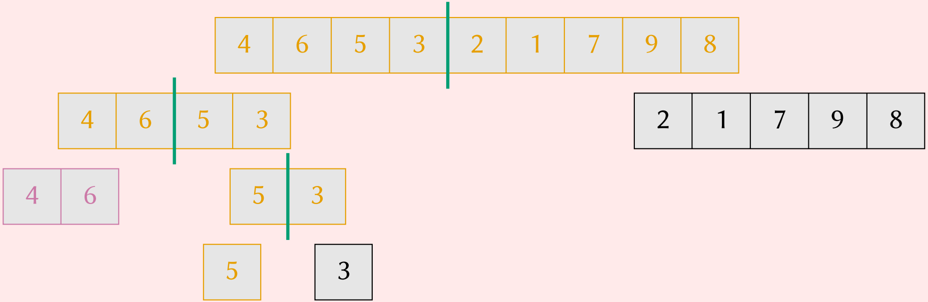
MergeSortR: A complete example



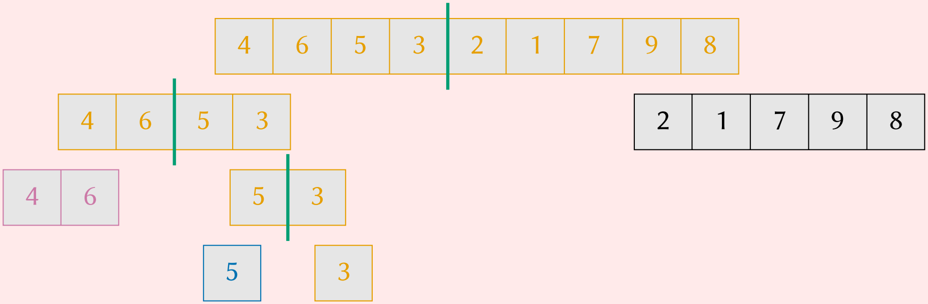
MergeSortR: A complete example



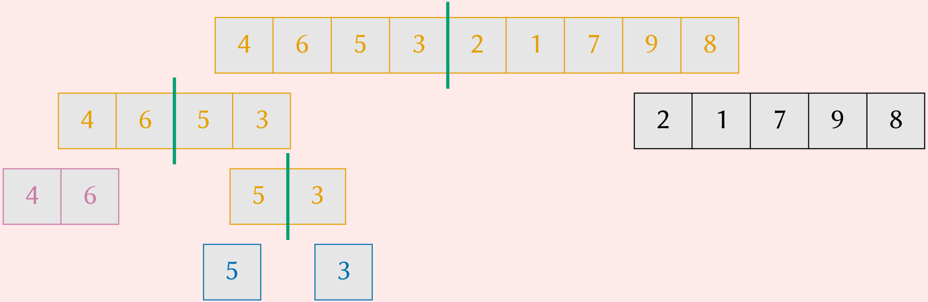
MergeSortR: A complete example



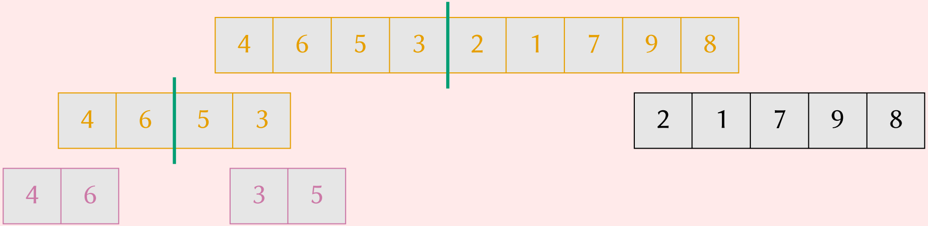
MergeSortR: A complete example



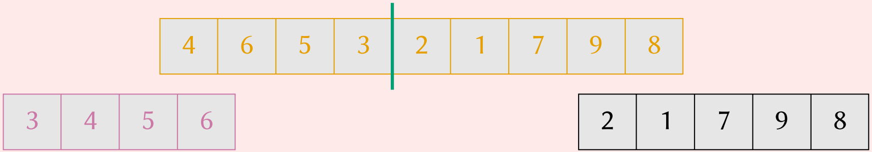
MergeSortR: A complete example



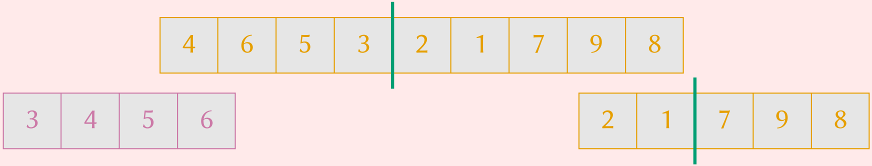
MergeSortR: A complete example



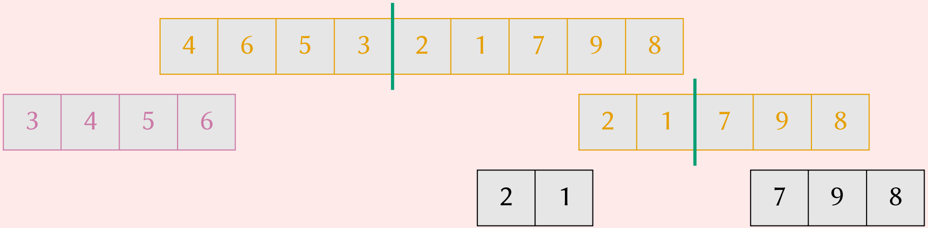
MergeSortR: A complete example



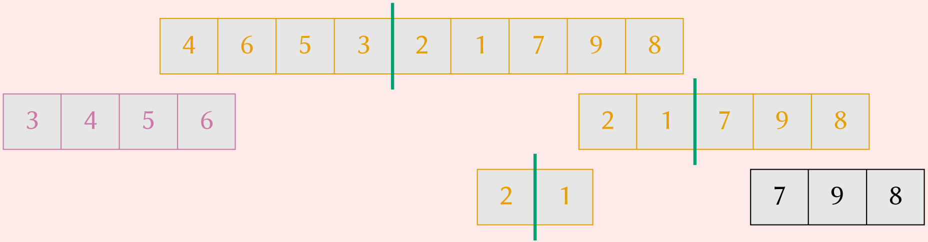
MergeSortR: A complete example



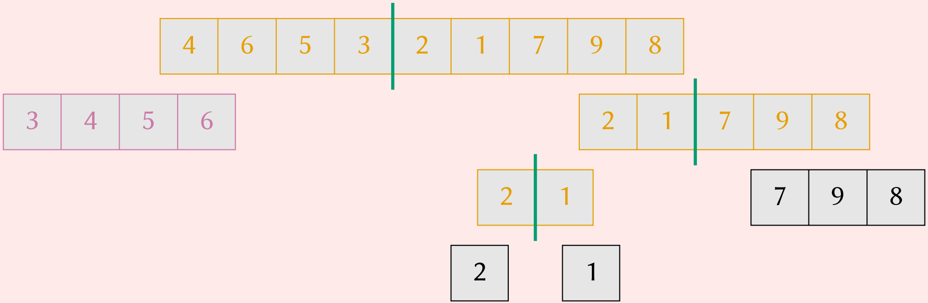
MergeSortR: A complete example



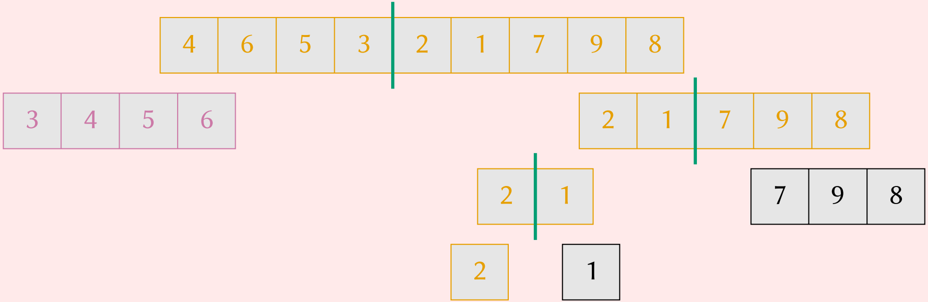
MergeSortR: A complete example



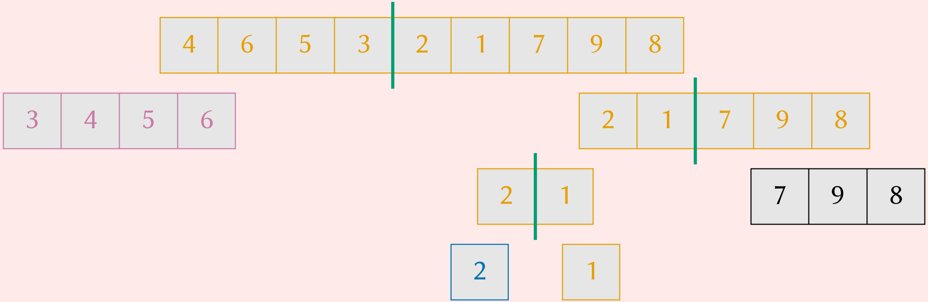
MergeSortR: A complete example



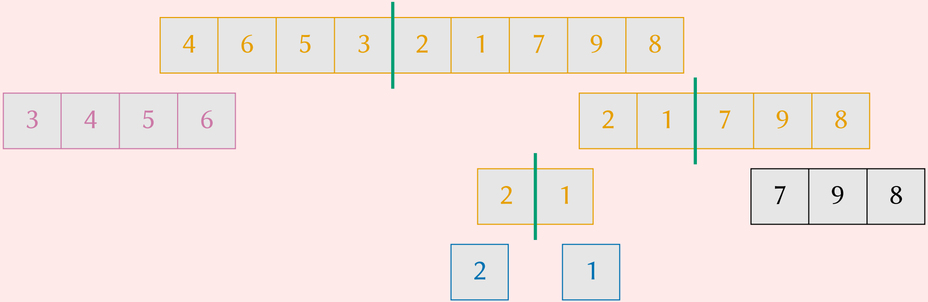
MergeSortR: A complete example



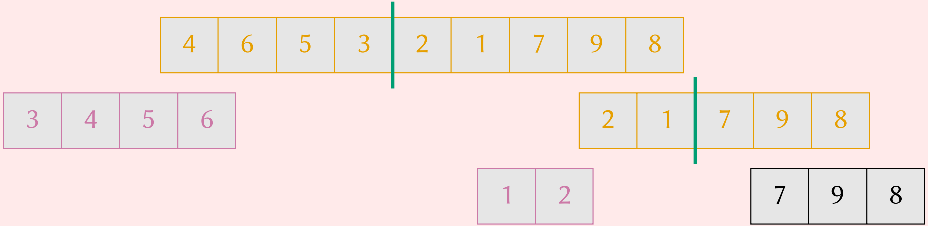
MergeSortR: A complete example



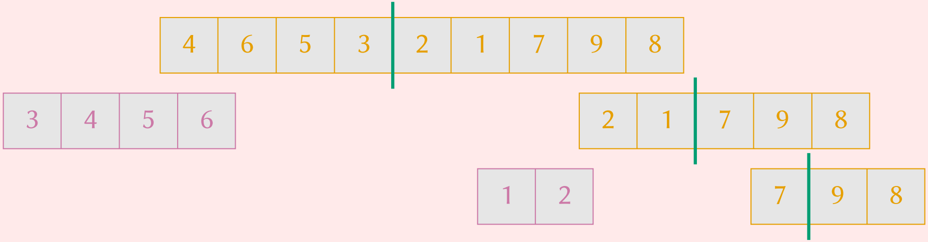
MergeSortR: A complete example



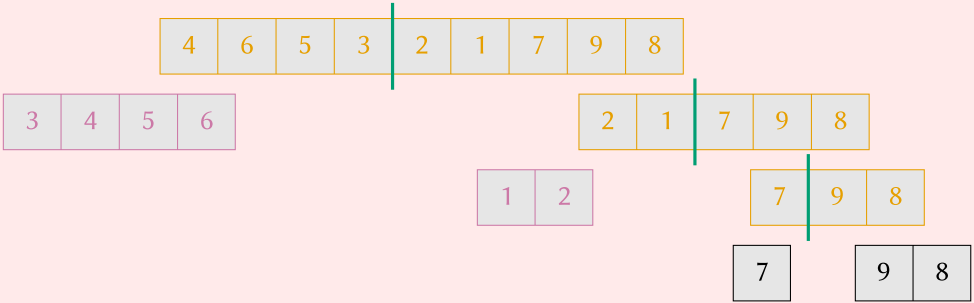
MergeSortR: A complete example



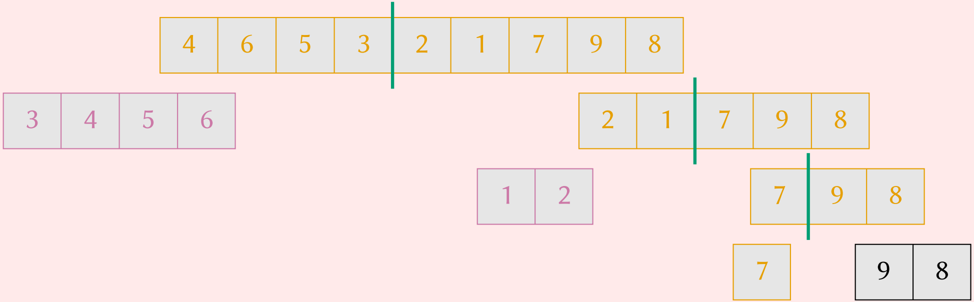
MergeSortR: A complete example



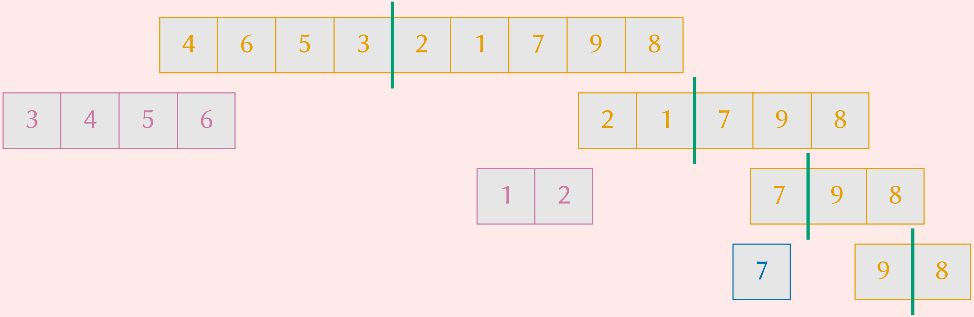
MergeSortR: A complete example



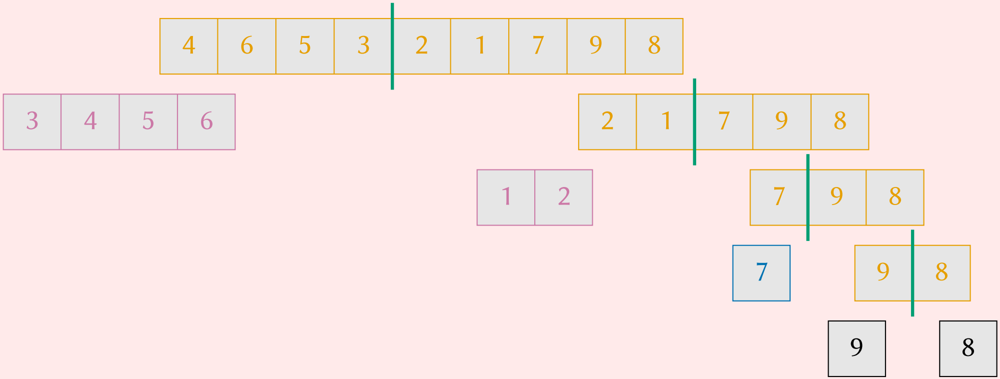
MergeSortR: A complete example



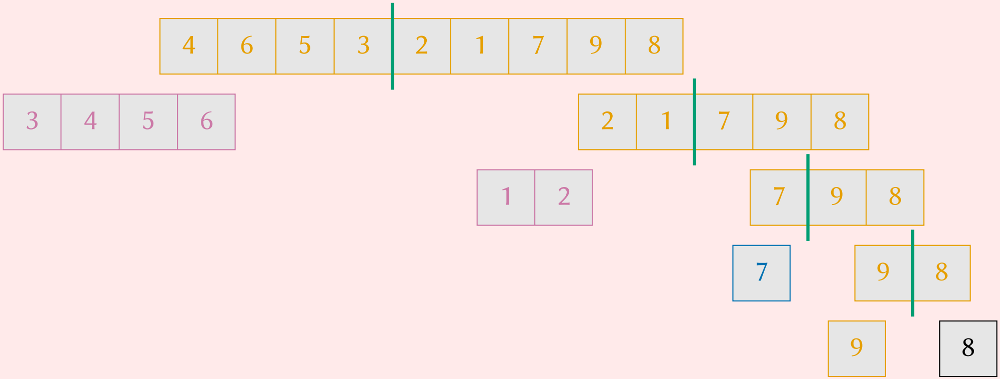
MergeSortR: A complete example



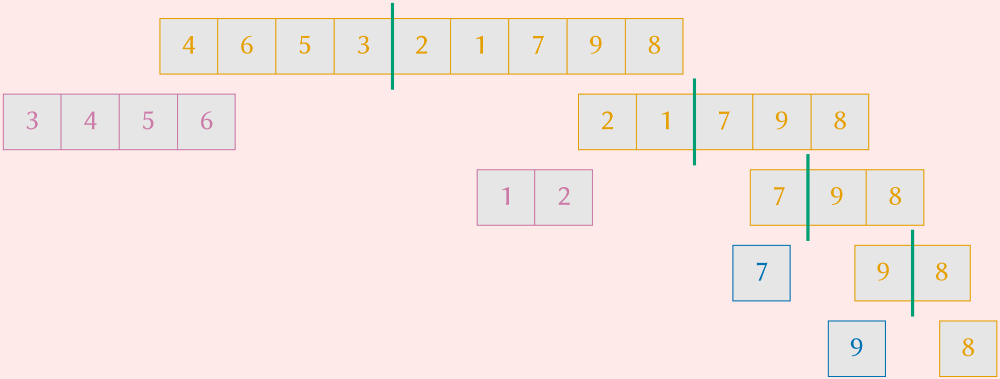
MergeSortR: A complete example



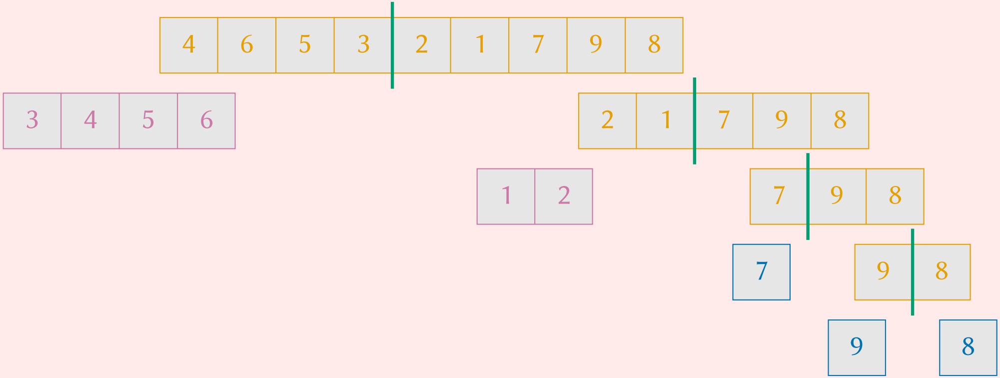
MergeSortR: A complete example



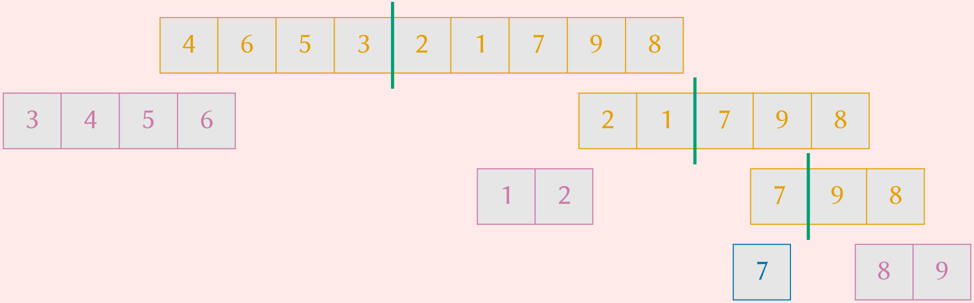
MergeSortR: A complete example



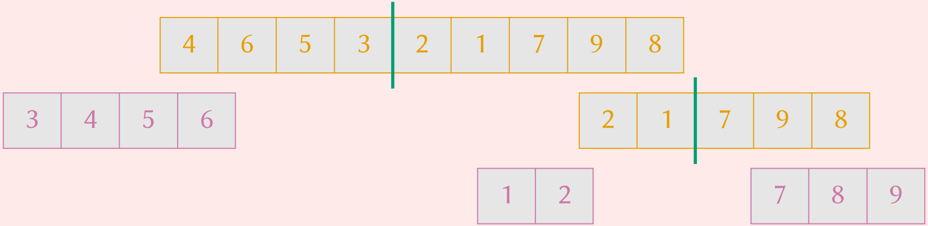
MergeSortR: A complete example



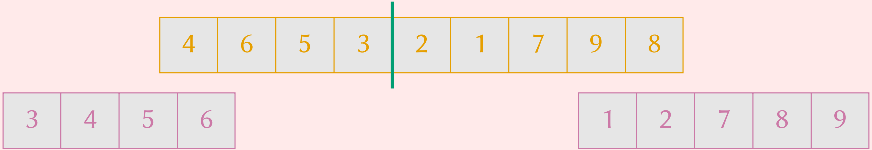
MergeSortR: A complete example



MergeSortR: A complete example



MergeSortR: A complete example



MergeSortR: A complete example

1	2	3	4	5	6	7	8	9

The complexity of MergeSortR

Plan

1. First, determine the complexity of a Merge call.
2. Then we can look at MergeSortR.

The complexity of MergeSortR

Algorithm Merge($L_1[0 \dots N_1)$, $L_2[0 \dots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or else(i1 < N1 and also L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```

The complexity of MergeSortR

Algorithm Merge($L_1[0 \ldots N_1)$, $L_2[0 \ldots N_2)$):

Input: L_1 and L_2 are sorted.

```
R is a new array for N1 + N2 values.
i1, i2 := 0, 0.
while i1 < N1 or i2 < N2 do
    if i2 = N2 or else(i1 < N1 and also L1[i1] < L2[i2]) then
        R[i1 + i2] := L1[i1].
        i1 := i1 + 1.
    else
        R[i1 + i2] := L2[i2].
        i2 := i2 + 1.
return R.
```

Comparisons: $< N_1 + N_2$.
Changes: $N_1 + N_2$.

The complexity of MergeSortR

Algorithm MergeSortR($L[start \ldots end]$):

```
if end − start > 1 then
    mid := (end − start) div 2.
    L1 := MergeSortR(L[start ... mid]).
    L2 := MergeSortR(L[mid ... end]).
    return Merge(L1, L2). N comparisons and changes.
else return L.
```

The complexity of MergeSortR

Algorithm MergeSortR($L[start \ldots end]$):

if $end - start > 1$ **then**
  $mid := (end - start)$ div 2.
  $L_1 :=$ MergeSortR($L[start \ldots mid]$).
  $L_2 :=$ MergeSortR($L[mid \ldots end]$).
  return Merge(L_1, L_2). N comparisons and changes.
else return L.

} Recursive case. (lines 1–5)

} Base case. (line 6)

The complexity of MergeSortR

Algorithm MergeSortR($L[start \dots end]$):

if $end - start > 1$ **then**
  $mid := (end - start) \operatorname{div} 2$.
  $L_1 :=$ MergeSortR($L[start \dots mid]$).
  $L_2 :=$ MergeSortR($L[mid \dots end]$).
  return Merge(L_1, L_2). N comparisons and changes.
else return L.

} Recursive case. (lines 1–5)
} Base case. (line 6)

The runtime complexity of MergeSortR(L, $start$, end) with $N = end - start$ is

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

The complexity of MergeSortR

Algorithm MergeSortR($L[start \dots end]$):

```
if end − start > 1 then
    mid := (end − start) div 2.
    L1 := MergeSortR(L[start ... mid]).
    L2 := MergeSortR(L[mid ... end]).
    return Merge(L1, L2).  N comparisons and changes.
else return L.
```

Lines 1–5: Recursive case.
Line 6: Base case.

The runtime complexity of MergeSortR(L, $start$, end) with $N = end - start$ is

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

How can we determine that $T(N) = f(N)$ for a closed-form $f(N)$?

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

How can we determine that $T(N) = f(N)$ for a closed-form $f(N)$?

We can use induction!

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

How can we determine that $T(N) = f(N)$ for a closed-form $f(N)$?

We can use induction!?

We need to know $f(N)$ to formalize an induction hypothesis!

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$

N

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$

N

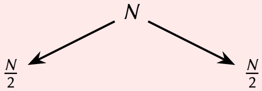
Number	Cost
$1 = 2^0$	N

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



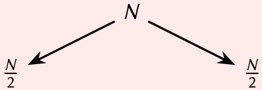
Number	Cost
$1 = 2^0$	N

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



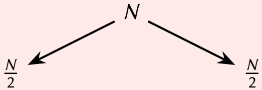
Number	Cost
$1 = 2^0$	N
$2 = 2^1$	$\frac{N}{2}$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



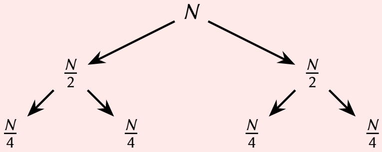
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



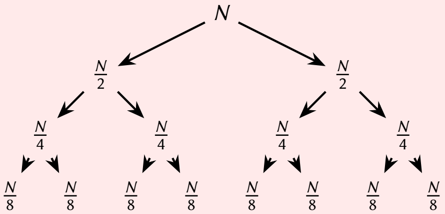
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



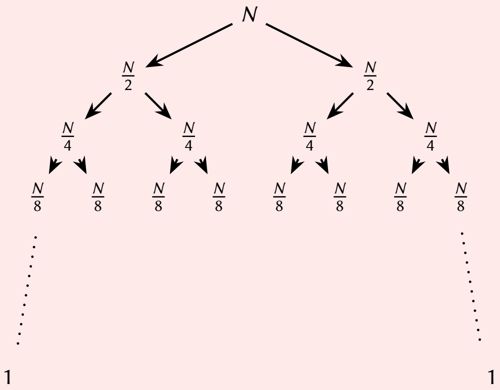
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



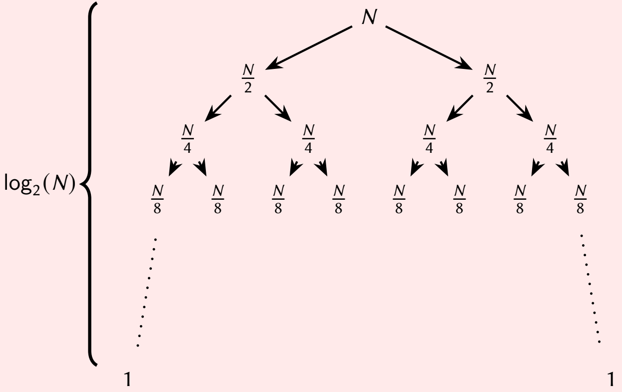
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$
$\vdots$	$\vdots$	$\vdots$
2^i	$\frac{N}{2^i}$	$2^i\frac{N}{2^i} = N$
$\vdots$	$\vdots$	$\vdots$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



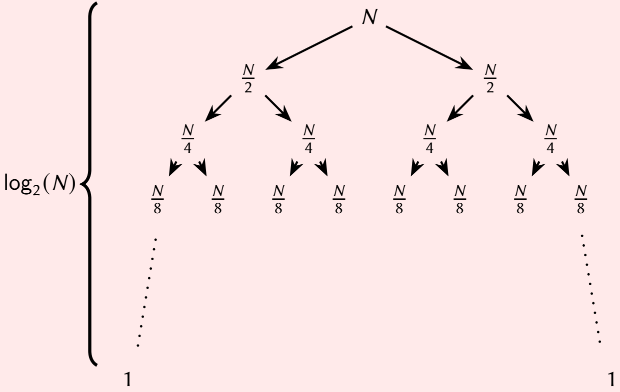
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$
$\vdots$	$\vdots$	$\vdots$
2^i	$\frac{N}{2^i}$	$2^i\frac{N}{2^i} = N$
$\vdots$	$\vdots$	$\vdots$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Recurrence tree for $T(N)$



Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$
$\vdots$	$\vdots$	$\vdots$
2^i	$\frac{N}{2^i}$	$2^i\frac{N}{2^i} = N$
$\vdots$	$\vdots$	$\vdots$ +
		$\Theta(N\log_2(N))$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Can do without a power-of-two assumption?

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Can do without a power-of-two assumption?

For any N, we have $2^{\lfloor \log_2(N) \rfloor} \leq N \leq 2^{\lceil \log_2(N) \rceil}$.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T\left(\frac{N}{2}\right) + N & \text{if } N > 1. \end{cases}$$

Assumption: N is a power-of-two.

Can do without a power-of-two assumption?

For any N, we have $2^{\lfloor \log_2(N) \rfloor} \leq N \leq 2^{\lceil \log_2(N) \rceil}$.

The assumption provides lower and upper bounds that are off by a small factor

$\longrightarrow$ Typically good enough to understand the complexity of your code.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction is the answer.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction is the answer.

This induction becomes messy due to terms $\left\lfloor \frac{N}{2} \right\rfloor$ and $\left\lceil \frac{N}{2} \right\rceil$.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

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

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

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

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$T(i) = T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2\left(c_2 \left\lceil \frac{i}{2} \right\rceil \log_2\left(\left\lceil \frac{i}{2} \right\rceil\right) + d_2\right) + i$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$\begin{aligned} T(i) &= T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2\left(c_2 \left\lceil \frac{i}{2} \right\rceil \log_2\left(\left\lceil \frac{i}{2} \right\rceil\right) + d_2\right) + i \\ &\leq 2\left(c_2 \frac{i+1}{2} \log_2\left(\frac{i+1}{2}\right) + d_2\right) + i \end{aligned}$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$\begin{aligned} T(i) &= T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2\left(c_2 \left\lceil \frac{i}{2} \right\rceil \log_2\left(\left\lceil \frac{i}{2} \right\rceil\right) + d_2\right) + i \\ &\leq 2\left(c_2 \frac{i+1}{2} \log_2\left(\frac{i+1}{2}\right) + d_2\right) + i = c_2(i+1)(\log_2(i+1) - 1) + 2d_2 + i \end{aligned}$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$\begin{aligned} T(i) &= T\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{i}{2} \right\rceil\right) + i \leq 2T\left(\left\lceil \tfrac{i}{2} \right\rceil\right) + i \leq 2\left(c_2 \left\lceil \tfrac{i}{2} \right\rceil \log_2\left(\left\lceil \tfrac{i}{2} \right\rceil\right) + d_2\right) + i \\ &\leq 2\left(c_2 \tfrac{i+1}{2} \log_2\left(\tfrac{i+1}{2}\right) + d_2\right) + i = c_2(i+1)(\log_2(i+1) - 1) + 2d_2 + i \\ &\leq c_2(i+1)(\log_2(i) - 0.4) + 2d_2 + i \end{aligned}$$

$$\log_2(2+1) - 1 = \log_2(2) + (\log_2(3) - \log_2(2)) - 1 \approx 1 + (1.6 - 1) - 1) = \log_2(2) - 0.4.$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$\begin{aligned} T(i) &= T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2T\left(\left\lceil \frac{i}{2} \right\rceil\right) + i \leq 2\left(c_2 \left\lceil \frac{i}{2} \right\rceil \log_2\left(\left\lceil \frac{i}{2} \right\rceil\right) + d_2\right) + i \\ &\leq 2\left(c_2 \frac{i+1}{2} \log_2\left(\frac{i+1}{2}\right) + d_2\right) + i = c_2(i+1)(\log_2(i+1) - 1) + 2d_2 + i \\ &\leq c_2(i+1)(\log_2(i) - 0.4) + 2d_2 + i \\ &= (c_2 i \log_2(i) + d_2) + (c_2 \log_2(i) + d_2 + i) - 0.4c_2(i+1). \end{aligned}$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$T(i) \leq (c_2 i \log_2(i) + d_2) + (c_2 \log_2(i) + d_2 + i) - 0.4c_2(i + 1).$$

We have $T(i) \leq (c_2 i \log_2(i) + d_2)$ if

$$(c_2 \log_2(i) + d_2 + i) \leq 0.4c_2(i + 1).$$

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$T(i) \leq (c_2 i \log_2(i) + d_2) + (c_2 \log_2(i) + d_2 + i) - 0.4c_2(i + 1).$$

We have $T(i) \leq (c_2 i \log_2(i) + d_2)$ if

$$(c_2 \log_2(i) + d_2 + i) \leq 0.4c_2(i + 1).$$

For *big enough* values of i and c_2, $i > B$, this is certainly true!

The complexity of MergeSortR

$$T(N) = \begin{cases} X & \text{if } N \leq B; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Due to Θ-notation, we need to prove: $c_1 N \log_2(N) + d_1 \leq T(N) \leq c_2 N \log_2(N) + d_2$.

Induction hypothesis For some c_2, d_2, $T(N) \leq c_2 N \log_2(N) + d_2$ for all $1 \leq N < i$.

Induction step Prove $T(i) \leq c_2 i \log_2(i) + d_2$.

$$T(i) \leq (c_2 i \log_2(i) + d_2) + (c_2 \log_2(i) + d_2 + i) - 0.4c_2(i+1).$$

We have $T(i) \leq (c_2 i \log_2(i) + d_2)$ if

$$(c_2 \log_2(i) + d_2 + i) \leq 0.4c_2(i+1).$$

For *big enough* values of i and c_2, $i > B$, this is certainly true!

Trick: make sure we always have big values of i.

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Yes we can—but is is very tedious!

The complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?

Yes we can—but is is very tedious!

Properly work out *recurrence trees* when possible: often easier and clearer!

The complexity of MergeSortR

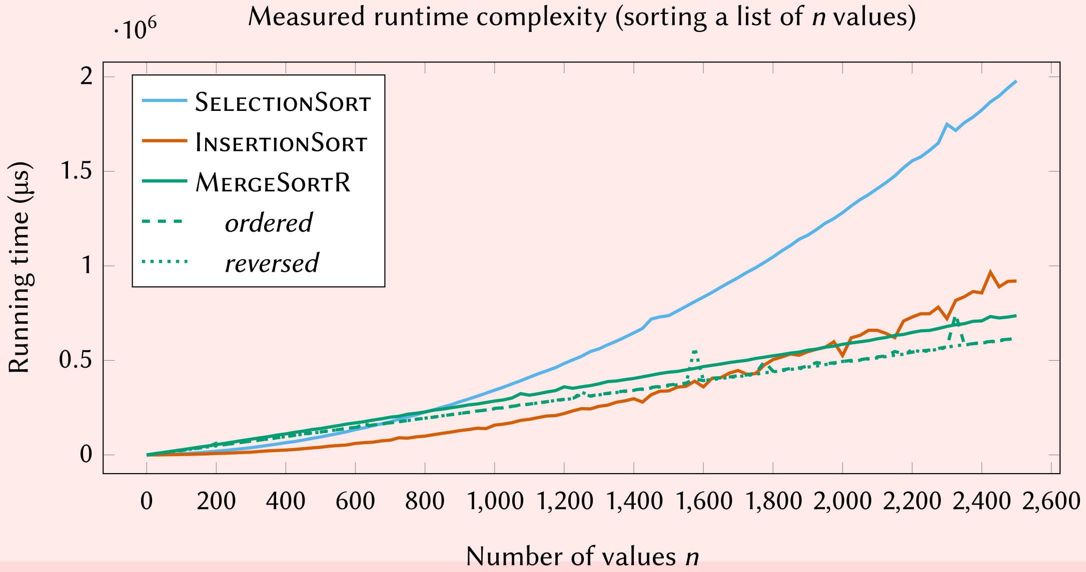
$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Can we prove $T(N) = \Theta(N \log_2(N))$ *exactly*?
Yes we can—but is is very tedious!

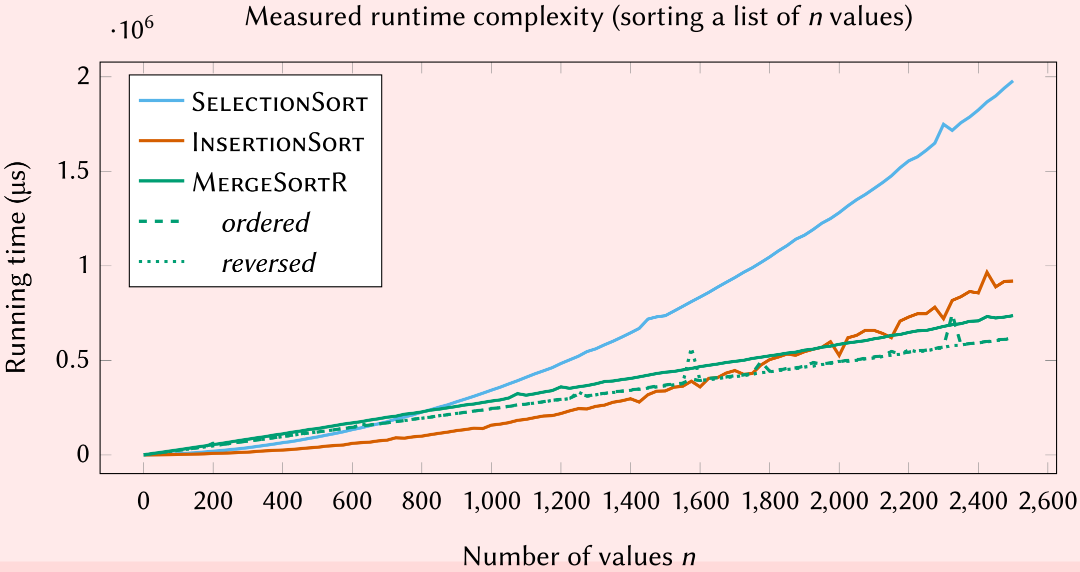
Properly work out *recurrence trees* when possible: often easier and clearer!

There are also *standard solutions* that you can use: the Master Theorem.

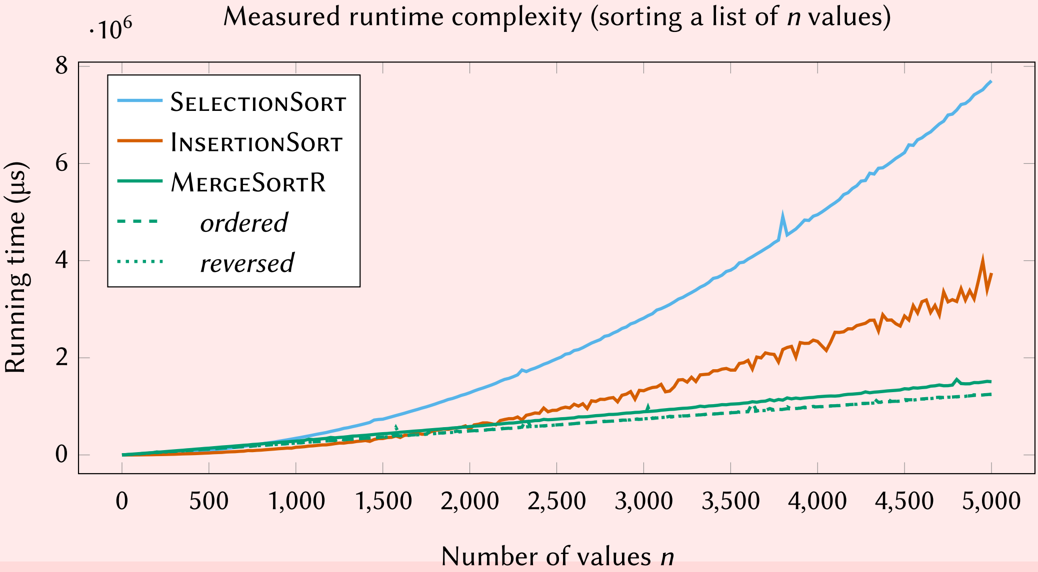
The performance of MergeSortR



The performance of MergeSortR



The performance of MergeSortR



The performance of MergeSortR

MergeSortR should be much better than SelectionSort and InsertionSort:
Especially on big lists.

Concern: MergeSortR has big constants.

- Each Merge makes new arrays.
- A lot of recursive calls that only get us to arrays of size one.

The performance of MergeSortR

MergeSortR should be much better than SelectionSort and InsertionSort:
Especially on big lists.

Concern: MergeSortR has big constants.
Can we finetune MergeSortR to reduce these constants?

- Each Merge makes new arrays.
- A lot of recursive calls that only get us to arrays of size one.

The performance of MergeSortR

MergeSortR should be much better than SelectionSort and InsertionSort:
Especially on big lists.

Concern: MergeSortR has big constants.

Can we finetune MergeSortR to reduce these constants?

- Each Merge makes new arrays.
  Idea: make a single target array to merge into.
- A lot of recursive calls that only get us to arrays of size one.
  Idea: switch from top-down (big-to-small arrays) to bottom-up (small-to-big arrays),
  we can do so using a loop instead of recursion!

MergeSort: A better MergeSortR

Algorithm MergeSort($L[0 \dots N)$):

R is a new array for N values.
$sl := 1$. The current *sorted length* of blocks in L.
while $sl \leq N$ **do**
    $i := 0$.
    while $i < N$ **do**
        Conceptually: Merge $L[i \dots i + sl)$ and $L[i + sl \dots i + 2sl)$ into $R[i \dots i + 2sl)$.
        $i := i + 2sl$.
    $sl := 2sl$.
    Switch the role of L and R.

MergeSort: A better MergeSortR

Algorithm MergeSort($L[0 \dots N)$):

```
R is a new array for N values.
sl := 1. The current sorted length of blocks in L.
while sl ≤ N do
    i := 0.
    while i < N do
        Conceptually: Merge L[i ... i + sl) and L[i + sl ... i + 2sl) into R[i ... i + 2sl).
        Careful: N does not have to be a multiple of 2sl.

        i := i + 2sl.
    sl := 2sl.
    Switch the role of L and R.
```

MergeSort: A better MergeSortR

Algorithm MergeSort($L[0 \ldots N)$):

R is a new array for N values.
$sl := 1$. The current *sorted length* of blocks in L.
while $sl \leq N$ **do**
    $i := 0$.
    while $i < N$ **do**
        Conceptually: Merge $L[i \ldots i + sl)$ and $L[i + sl \ldots i + 2sl)$ into $R[i \ldots i + 2sl)$.
        Careful: N does not have to be a multiple of $2sl$.
        MergeInto(L, i, $\min(i + sl, N)$, $\min(i + 2sl, N)$, R).
        $i := i + 2sl$.
    $sl := 2sl$.
    Switch the role of L and R.

MergeSort: A better MergeSortR

Algorithm MergeInto($S[0\ldots N)$, $start$, mid, end, $T[0\ldots N)$):

Input: $0 \le start \le mid \le end \le N$ and
$S[start\ldots mid)$ and $S[mid\ldots end)$ are sorted.

$i_1, i_2 := start, mid$.
while $i_1 < mid$ **or** $i_2 < end$ **do**
  if $i_2 = end$ **or** ($i_1 < mid$ **and** $S[i_1] < S[i_2]$) **then**
    $T[i_1 + i_2] := S[i_1]$.
    $i_1 := i_1 + 1$.
  else
    $T[i_1 + i_2] := S[i_2]$.
    $i_2 := i_2 + 1$.

MergeSort: A better MergeSortR

4	6	5	3	2	1	7	9	8

MergeSort: A better MergeSortR

$sl = 1$, L:

4	6	5	3	2	1	7	9	8

Conceptually: [4] [6] [5] [3] [2] [1] [7] [9] [8]

MergeSort: A better MergeSortR

$sl = 2$, L:

4	6	3	5	1	2	7	9	8

Conceptually: [4 | 6] [3 | 5] [1 | 2] [7 | 9] [8]

MergeSort: A better MergeSortR

$sl = 4$, L:

3	4	5	6	1	2	7	9	8

Conceptually:

3	4	5	6

1	2	7	9

8

MergeSort: A better MergeSortR

$sl = 8$, L:

1	2	3	4	5	6	7	9	8

Conceptually:

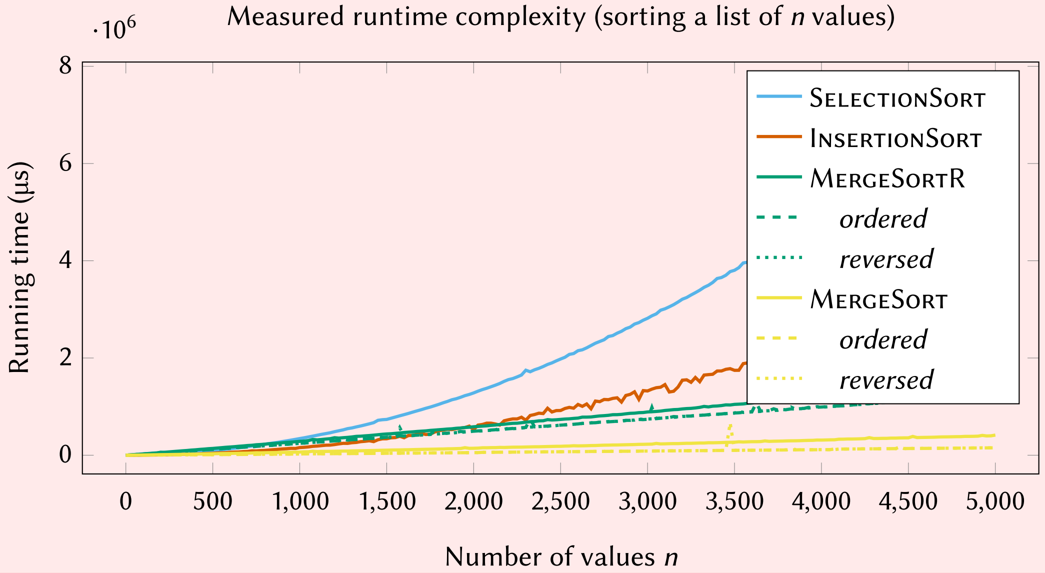
1	2	3	4	5	6	7	9		8

MergeSort: A better MergeSortR

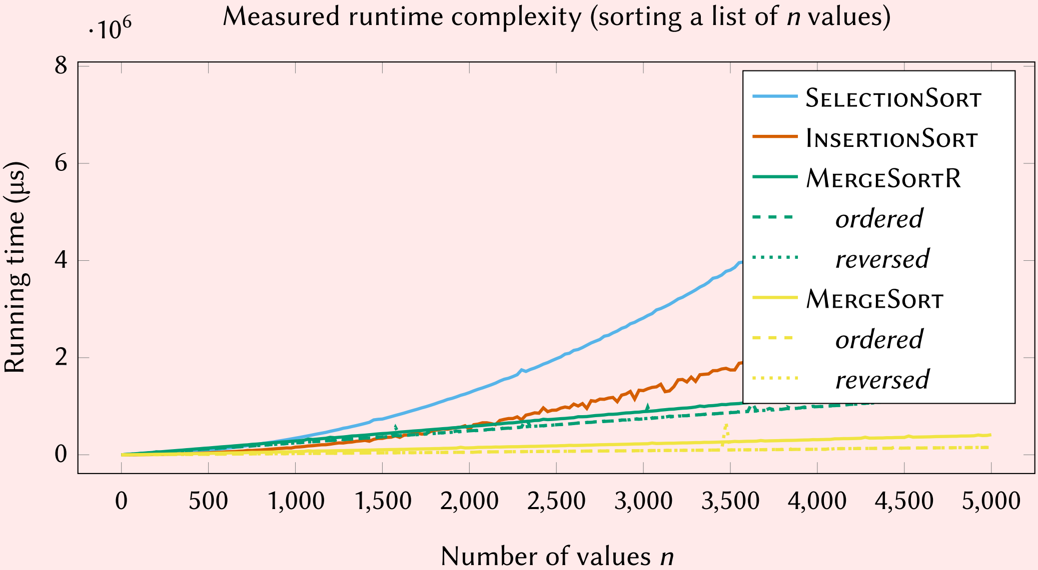
$sl = 16$, L:

1	2	3	4	5	6	7	8	9

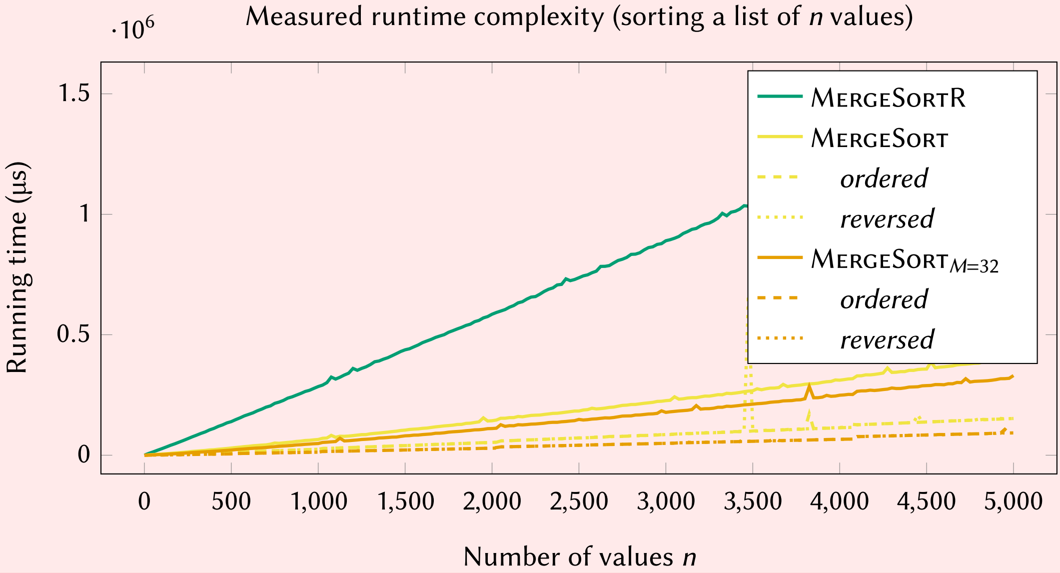
MergeSort: A better MergeSortR



MergeSort: A better MergeSortR



MergeSort: A better MergeSortR



Final notes on MergeSort

- Runtime complexity: $\Theta(N \log_2(N))$ comparisons and changes;
- Memory complexity: $\Theta(N)$ (for merging).

Final notes on MergeSort

- Runtime complexity: $\Theta(N \log_2(N))$ comparisons and changes;
- Memory complexity: $\Theta(N)$ (for merging).

The power of MergeSort

The MergeSort algorithm is at the basis of many large-scale sort algortihms:

- multi-threaded sorting (GiB),
- sorting data on external memory (GiB–TiB),
- sorting data in a cluster (TiB–PiB).

Final notes on MergeSort

- Runtime complexity: $\Theta(N \log_2(N))$ comparisons and changes;
- Memory complexity: $\Theta(N)$ (for merging).

The power of MergeSort

The MergeSort algorithm is at the basis of many large-scale sort algortihms:

- multi-threaded sorting (GiB),
- sorting data on external memory (GiB–TiB),
- sorting data in a cluster (TiB–PiB).

The power of Merge

The Merge algorithm is flexible: you can easily change it to

- compute the *union* (without duplicates) of two sorted list;
- compute the *intersection* of two sorted list;
- compute the *difference* of two sorted list;
- compute a *join* of two tables (if sorted on the join attributes).

Final notes on MergeSort

	C++	Java
MergeSort	`std::stable_sort`	`java.util.Arrays.sort` (usually)
Merge	`std::merge`	
Merge-like	`std::set_union` `std::set_intersection` `std::set_difference` `std::set_symmetric_difference`	
(related)	`std::inplace_merge`	