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: BinarySearch versus LinearSearch.

Why sorting

Most computational problems involve *data processing*.

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

Example

Finding values: BinarySearch 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 $L[0\ldots N)$ 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

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 $L[0 \dots N)$ 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.
```

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** $\left.\right\} \sum_{j=i+1}^{N-1} 1$
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 $\textsc{SimpleTwoSum}(L, w)$:

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*.

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

Complexity of $\textsc{SimpleTwoSum}$

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

The power of sorting: The two-sum problem

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

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.

The power of sorting: The two-sum problem

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

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

1: $\mathit{result} :=$ 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.

$$\left.\vphantom{\sum}\right\} \sum_{j=i+1}^{N-1} 1 = N - (i + 1)$$

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

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 $\textsc{SimpleTwoSum}(L, w)$**:**

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*.

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

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

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 $L[0 \dots N)$ 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.
```

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

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

Complexity of SimpleTwoSum

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

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

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 $L[0\dots N)$ 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: $w = 11$.

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: $w = 11$.

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 := BinarySearch(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 $L[0 \ldots N)$ of N distinct weights, target weight w.

```
result := empty bag.
for i := 0 to N − 2 do
    j := BinarySearch(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 BinarySearch 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 := BinarySearch(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.
```

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each BinarySearch 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 := BinarySearch(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.
```

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

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each BinarySearch 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**
  $j :=$ BinarySearch($L, i+1, N, w - L[i]$). $\Big\}\ \log_2(N-(i+1))$
  if $j \neq$ 'not found' **then**
    add $(L[i], L[j])$ to *result*.
return *result*.

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

Complexity of BetterTwoSum

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

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

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

Complexity of BetterTwoSum

For a rough *upper bound* estimate, we can count the cost of each BinarySearch 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 $L[0 \ldots N)$ of N distinct weights, target weight w.

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

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

Complexity of BetterTwoSum

For a rough *lower bound* estimate, we can count the cost of each BinarySearch 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**
  $j :=$ BINARYSEARCH(L, $i + 1$, N, $w - L[i]$). $\Big\}\ \log_2(N - (i + 1))$
  if $j \neq$ 'not found' **then**
    add $(L[i], L[j])$ to *result*.
return *result*.

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

Complexity of BETTERTWOSUM

For a rough *lower bound* estimate, we can count the cost of each BINARYSEARCH 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**
  $j :=$ BinarySearch(L, $i+1$, N, $w - L[i]$). $\Big\}\ \log_2(N-(i+1))$
  if $j \neq$ 'not found' **then**
    add $(L[i], L[j])$ to *result*.
return *result*.

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

Complexity of BetterTwoSum

For a rough *lower bound* estimate, we can count the cost of each BinarySearch 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**
    $j :=$ BinarySearch(L, $i + 1$, N, $w - L[i]$). $\Big\}\ \log_2(N - (i + 1))$
    if $j \neq$ 'not found' **then**
        add $(L[i], L[j])$ to *result*.
return *result*.

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

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each BinarySearch 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 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 := BinarySearch(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.
```

$$\left.\begin{array}{l}\text{for } i := 0 \text{ to } N-2 \text{ do} \\ \quad j := \text{BinarySearch}(L, i+1, N, w - L[i]). \quad \} \log_2(N-(i+1)) \\ \quad \text{if } j \neq \text{`not found' then} \\ \quad\quad \text{add } (L[i], L[j]) \text{ to } result.\end{array}\right\} \sum_{i=0}^{N-2} \log_2(N-(i+1))$$

Complexity of BetterTwoSum

For a rough estimate, we can count the cost of each BinarySearch 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 $L[0 \dots N)$ 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?

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?

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 $L[0\ldots N)$ 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?

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 $L[0\ldots N)$ 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?

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 $L[0 \ldots N)$ 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?

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 $\textsc{BestTwoSum}(L, w)$**:**

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 $L[0 \ldots N)$ 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*.

$\Theta(N)$

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

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

Intermezzo: Correctness of BestTwoSum

High-level proof steps

2. Specify the *invariant*.

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

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

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

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

1: result := empty bag.
2: $i, j := 0, N - 1$.
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.
3: **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 $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

$result :=$ empty bag.
$i, j := 0, N - 1$.
Known: we have $i = 0$, $j = N - 1$, and $result = \emptyset$ (due to assignments).
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 $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

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

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 $L[i] + L[j] = w$.

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

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*.

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

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

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

4: **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: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

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

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

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

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

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

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

Given: $\mathit{result} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i, j)$, $i < j$, 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: $\mathit{result}_{\text{new}} = \mathit{result}_{\text{old}} \cup \{(L[i_{\text{old}}], L[j_{\text{old}}])\}$, $i_{\text{new}} = i_{\text{old}} + 1$, $j_{\text{new}} = j_{\text{old}} - 1$, $\mathit{result}_{\text{old}} = \mathrm{TS}(0, N-1) \setminus \mathrm{TS}(i_{\text{old}}, j_{\text{old}})$, and $(L[i_{\text{old}}], L[j_{\text{old}}]) \in \mathrm{TS}(i_{\text{old}}, j_{\text{old}})$.

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

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

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

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

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

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.

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

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

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

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

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

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

4: **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: $(L[i], L[j]) \in \mathsf{TS}(i, j)$.

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

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

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

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

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

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

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

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

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 $L[i] + L[j] = w$.

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

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

$i := i + 1$.

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

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

$i := i + 1$.

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 $L[i] + L[j] = w$.

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 $(L[i_{\text{old}}], v) \notin \mathsf{TS}(i_{\text{old}}, j)$.

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 $i < j \to result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$ 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 $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \land (start \leq i < j \leq end)\}$.

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

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

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 $L[i] + L[j] = w$.

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

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 $L[i] + L[j] = w$.

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

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

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

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

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

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

$result$:= empty bag.
$i, j := 0, N - 1$.
while $i < j$ **do** /* inv: $result = \mathsf{TS}(0, N-1) \setminus \mathsf{TS}(i, j)$; 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 $\mathsf{TS}(start, end) = \{(L[i], L[j]) \mid (L[i] + L[j] = w) \wedge (start \leq i < j \leq end)\}$.

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

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

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

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

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

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

Intermezzo: Correctness of BestTwoSum

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

A first sort algorithm: SelectionSort

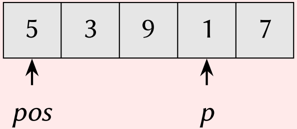
5	3	9	1	7

Algorithm SelectionSort(L):

Input: List $L[0 \dots N)$ 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

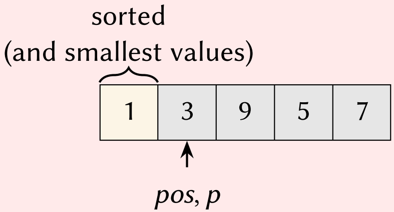
$$\overbrace{\boxed{1}}^{\substack{\text{sorted} \\ \text{(and smallest values)}}}\boxed{3}\boxed{9}\boxed{5}\boxed{7}$$

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 ... 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

$$\overbrace{\boxed{1}\,\boxed{3}}^{\substack{\text{sorted}\\ \text{(and smallest values)}}}\,\boxed{9}\,\boxed{5}\,\boxed{7}$$

Algorithm SelectionSort(L):

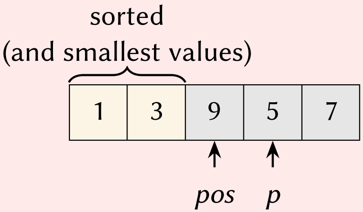
Input: List $L[0 \ldots N)$ of N values.

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 $L[0 \ldots N)$ 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

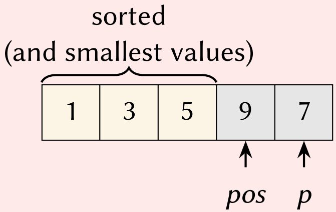
$$\overbrace{\boxed{1}\,\boxed{3}\,\boxed{5}}^{\substack{\text{sorted} \\ \text{(and smallest values)}}}\,\boxed{9}\,\boxed{7}$$

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 $L[0 \ldots N)$ 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].
```

A first sort algorithm: SelectionSort

$$\overbrace{\boxed{1}\boxed{3}\boxed{5}\boxed{7}\boxed{9}}^{\substack{\text{sorted}\\ \text{(and smallest values)}}}$$

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

$$\overbrace{\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{7}\,\boxed{9}}^{\substack{\text{sorted}\\ \text{(and smallest values)}}}$$

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 $L[0 \ldots N)$ 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: $N - 1 - pos$.
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 $L[0 \dots N)$ 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].
```

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

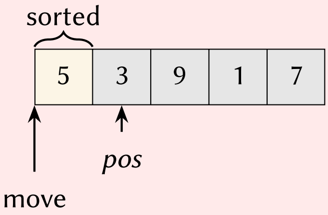
$$\overbrace{\boxed{5}}^{\text{sorted}}\boxed{3}\boxed{9}\boxed{1}\boxed{7}$$

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.

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

A second sort algorithm: InsertionSort

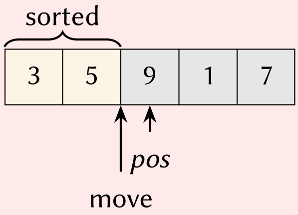
$$\overbrace{\boxed{3}\,\boxed{5}}^{\text{sorted}}\,\boxed{9}\,\boxed{1}\,\boxed{7}$$

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 ∈ L[0 ... pos) with v < w one to the right.
    L[p] := v.
```

A second sort algorithm: InsertionSort

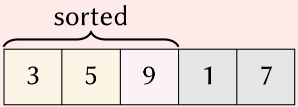


Algorithm InsertionSort(L):

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

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

A second sort algorithm: InsertionSort

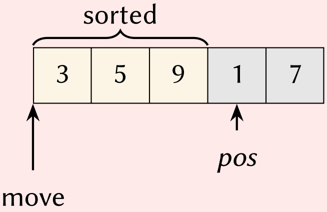


Algorithm InsertionSort(L):

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

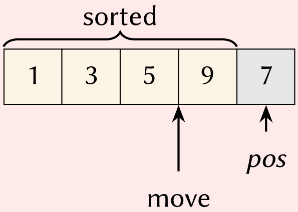
$$\overbrace{\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{9}}^{\text{sorted}}\,\boxed{7}$$

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

$$\overbrace{\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{7}\,\boxed{9}}^{\text{sorted}}$$

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

$$\overbrace{\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{7}\,\boxed{9}}^{\text{sorted}}$$

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 ∈ L[0 . . . 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.

1: **for** $pos := 1$ **to** $N - 1$ **do**
2: $\quad v := L[pos]$.
$\quad$ Move all values $w \in L[0 \dots pos)$ with $v < w$ one to the right.
3: $\quad p := pos$.
4: $\quad$ **while** $p > 0$ **and** $v < L[p - 1]$ **do**
5: $\quad\quad L[p] := L[p - 1]$.
6: $\quad\quad p := p - 1$.
7: $\quad L[p] := v$.

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**
        $L[p] := L[p - 1]$.
        $p := p - 1$.
    $L[p] := v$.

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 $L[0 \dots N)$ 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: $\leq \sum_{pos=1}^{N-1} pos$.

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 $L[0 \dots N)$ 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.
```

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

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

Runtime complexity of InsertionSort

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

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.
```

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

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

Runtime complexity of InsertionSort

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

When does InsertionSort have N^2 comparisons and changes?

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?

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

A second sort algorithm: InsertionSort

Algorithm InsertionSort(L):

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

When does InsertionSort have less than 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 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 $L[0 \ldots N)$ of N values, $L = \mathcal{L}$.

$pos := 1$.
while $pos \neq N$ **do**
  /* inv: $L[0 \ldots pos)$ 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: $F = L[0 \ldots p)$ is ordered, $S = L[p+1 \ldots pos+1)$ 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 $L[pos+1 \ldots, N)$ 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 $L[0 \ldots N)$ of N values, $L = \mathcal{L}$.

$pos := 1$.

while $pos \neq N$ **do**

  /* inv: $L[0 \ldots pos)$ 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: $F = L[0 \ldots p)$ is ordered, $S = L[p+1 \ldots pos+1)$ 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 $L[pos+1 \ldots, N)$ are exactly the values in $\mathcal{L}$, bf: p. */

    $L[p] := L[p-1]$.

    $p := p - 1$.

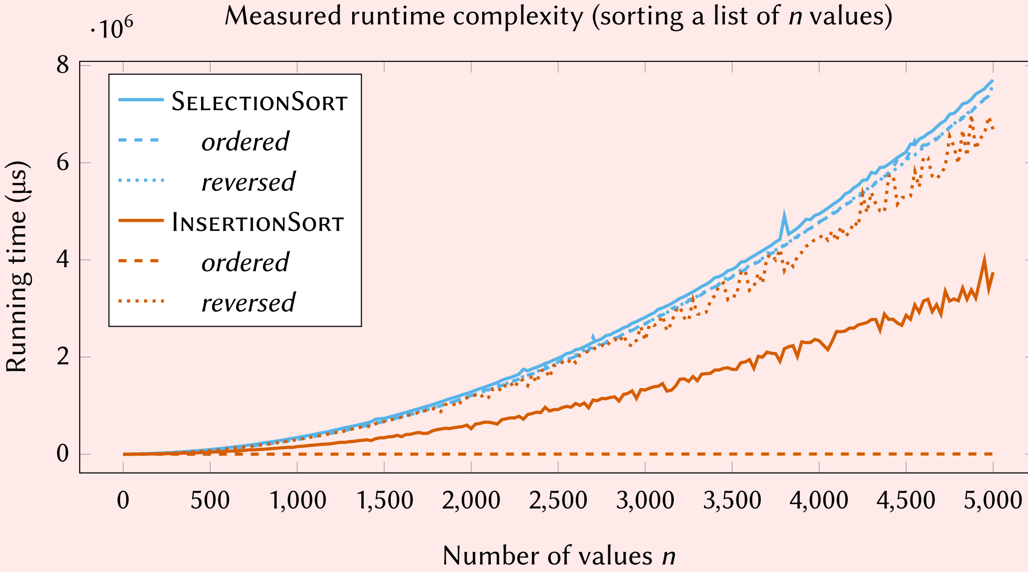
  $L[p] := v$.

  $pos := pos + 1$.

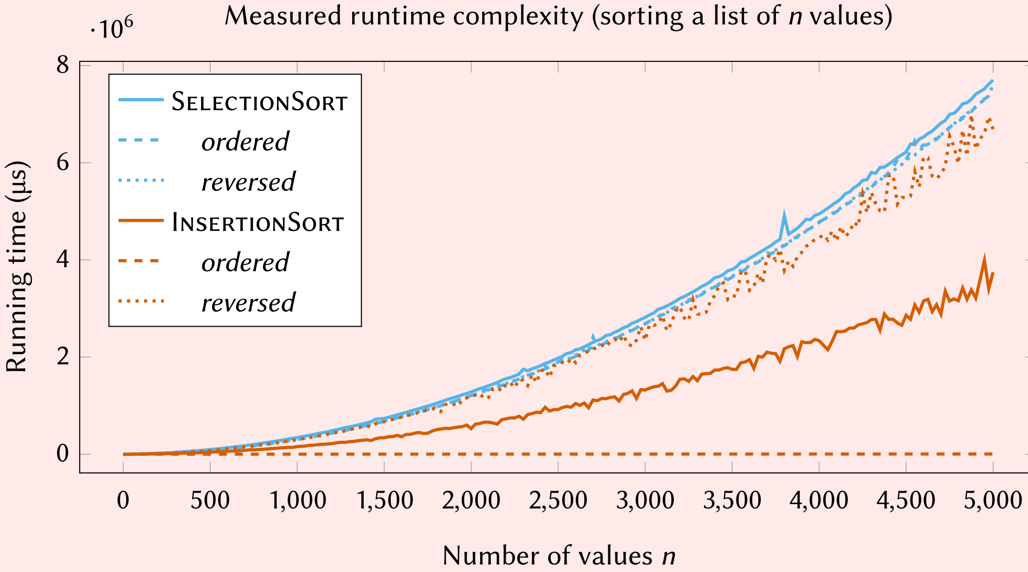
A summary of basic sorting

	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.

BinarySearchR 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

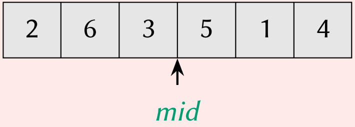
MergeSortR: High-level overview

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

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

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

else return L.



MergeSortR: High-level overview

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

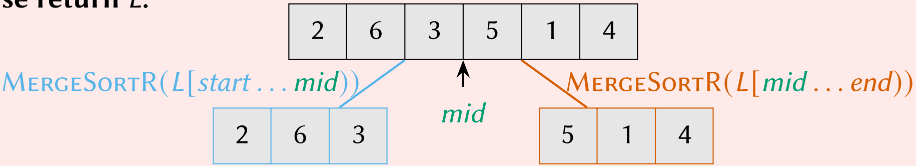
if $end - start > 1$ **then**

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

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

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

else return L.

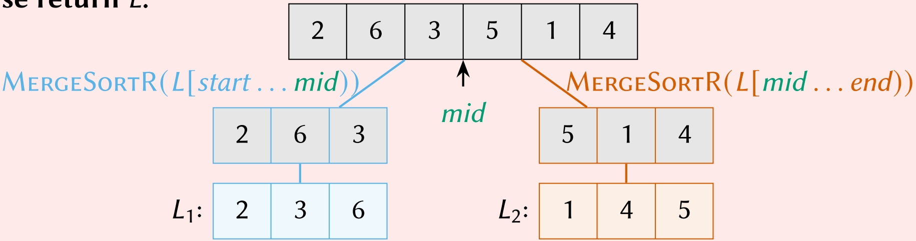


MergeSortR: High-level overview

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

if $end - start > 1$ **then**
    $mid := (end - start) \operatorname{div} 2.$
    $L_1 :=$ MergeSortR($L[start \ldots mid]$).
    $L_2 :=$ MergeSortR($L[mid \ldots end]$).

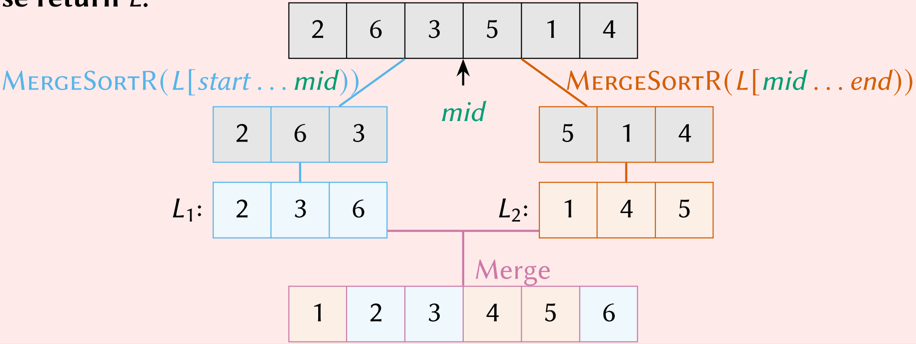
else return L.



MergeSortR: High-level overview

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

if $end - start > 1$ **then**
  $mid := (end - start) \operatorname{div} 2.$
  $L_1 :=$ 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($L[start \dots end]$) sorts

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

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.

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($L[start \ldots end]$) 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.

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

Proof of correctness: $\textsc{MergeSortR}(L[\mathit{start} \ldots \mathit{end}])$ sorts

Base case $\textsc{MergeSortR}$ sorts $0 \leq \mathit{end} - \mathit{start} \leq 1$ values.

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

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

start | $\mathit{mid} := (\mathit{end} - \mathit{start}) \operatorname{div} 2$ | end

$\downarrow \textsc{MergeSortR}(L[\mathit{start} \ldots \mathit{mid}])$ $\downarrow \textsc{MergeSortR}(L[\mathit{mid} \ldots \mathit{end}])$

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.

$start$	$mid := (end - start)$ div 2	end
$\downarrow$ MergeSortR($L[start \dots mid)$)	$\downarrow$ MergeSortR($L[mid \dots end)$)	

$\boxed{start < mid < end}$

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.

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

$\downarrow$ MergeSortR($L[start \dots mid)$) $\downarrow$ MergeSortR($L[mid \dots end)$)

IH IH

L_1: $L[start \dots mid)$, but sorted

L_2: $L[mid \dots end)$, but sorted

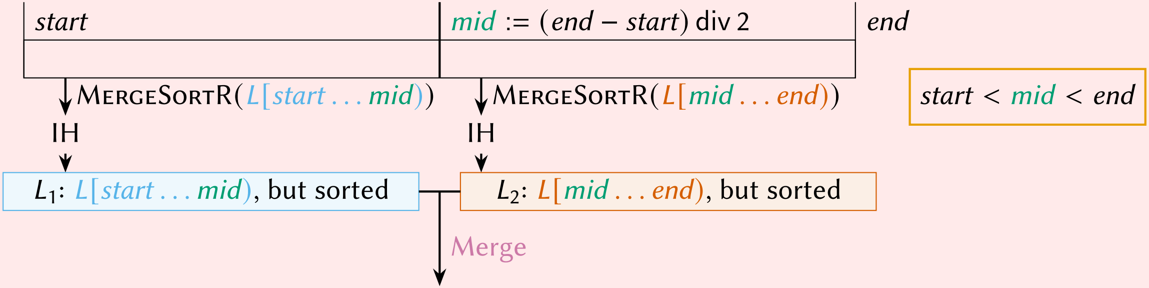
$start < mid < end$

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.

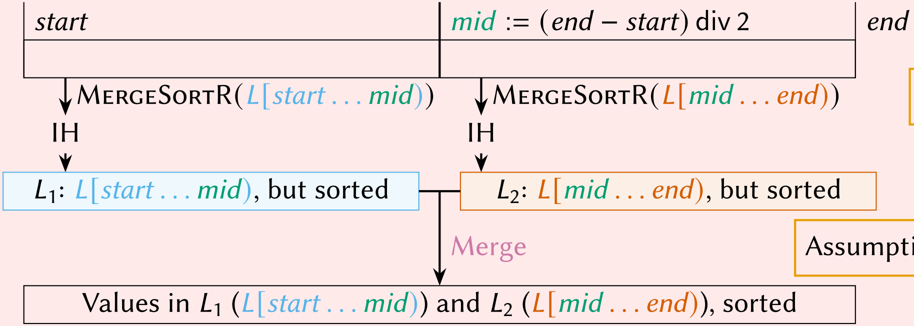


Proof of correctness: MERGESORTR($L[start \ldots end)$) sorts

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.



$start < mid < end$

Assumption: Merge is correct

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

Input: L_1 and L_2 are sorted.

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

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**

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:

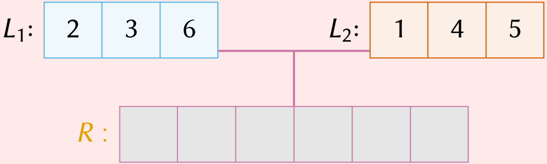
1	4	5

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**

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

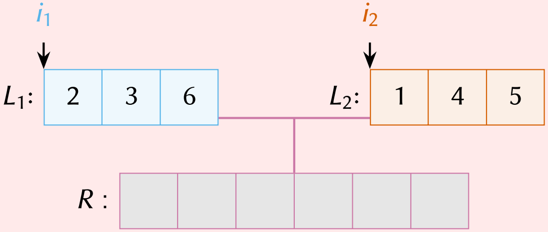
    $i_1 := i_1 + 1$.

  else

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

    $i_2 := i_2 + 1$.

return R.

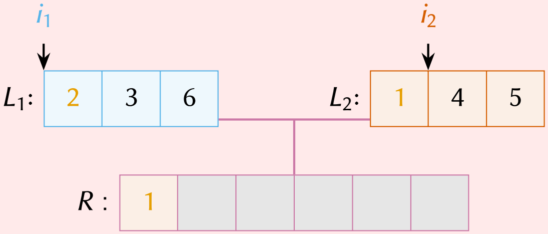


Assumption: Merge is correct

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.

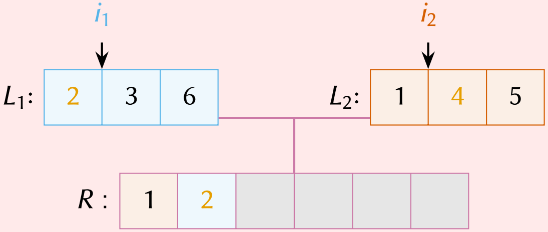


Assumption: Merge is correct

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

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 (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.
```

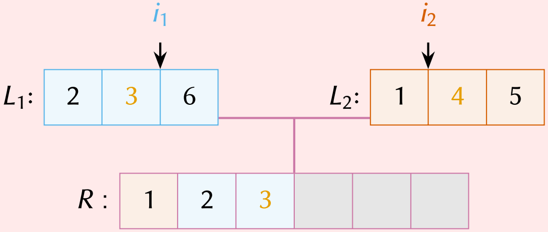


Assumption: Merge is correct

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 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.
```

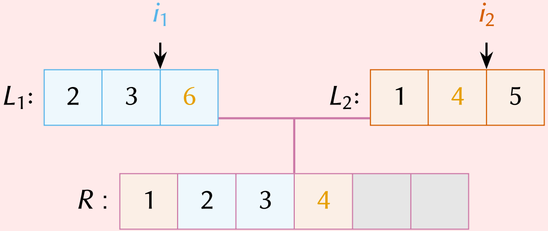


Assumption: Merge is correct

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 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.
```

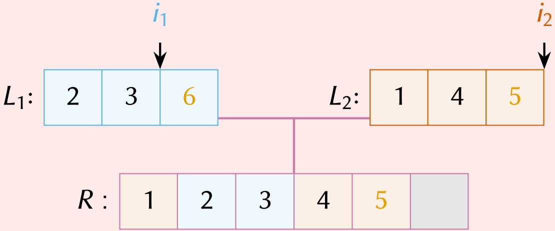


Assumption: Merge is correct

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 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.
```

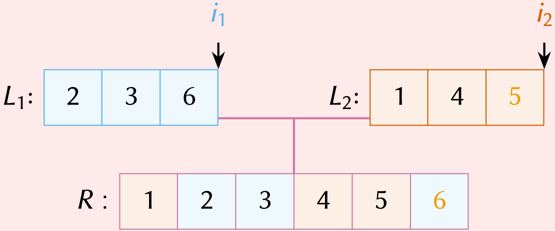


Assumption: Merge is correct

Algorithm 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.
```

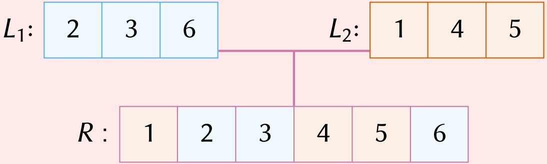


Assumption: Merge is correct

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

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**

  /* inv: $R[0\ldots i_1 + i_2)$ has all values from $L_1[0\ldots i_1)$ and $L_2[0\ldots i_2)$, 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

4	6	5	3	2	1	7	9	8

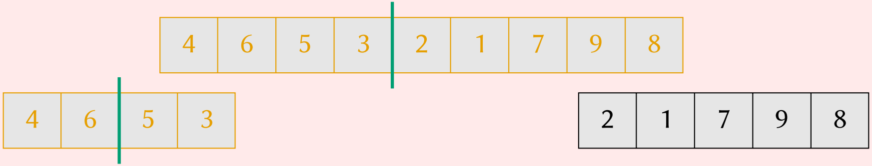
MergeSortR: A complete example

4	6	5	3	2	1	7	9	8

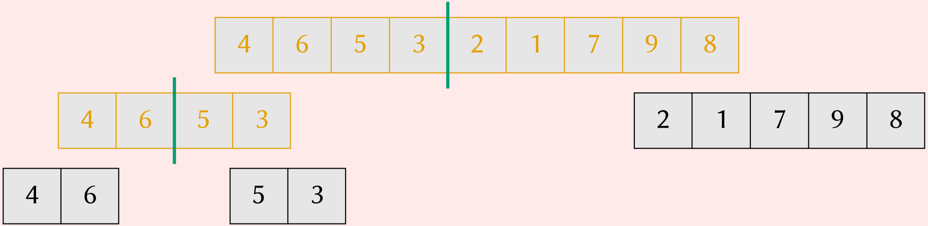
4	6	5	3

2	1	7	9	8

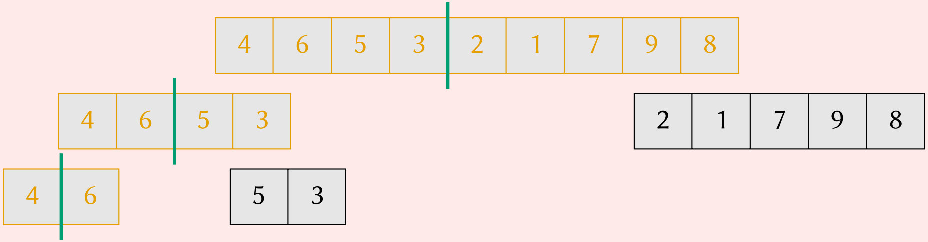
MergeSortR: A complete example



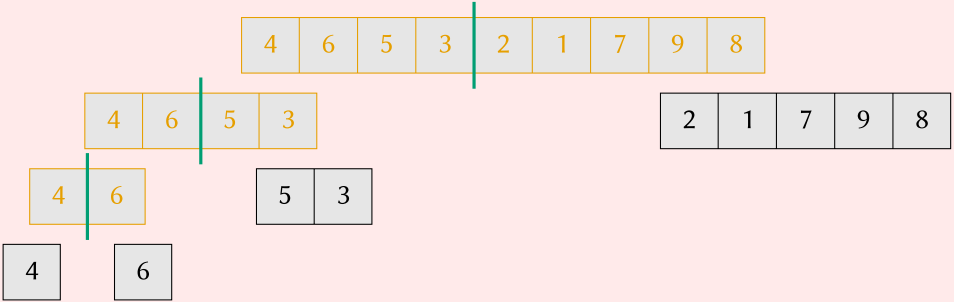
MergeSortR: A complete example



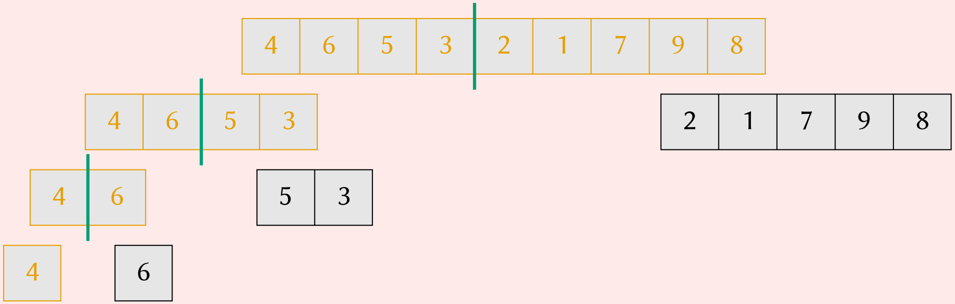
MergeSortR: A complete example



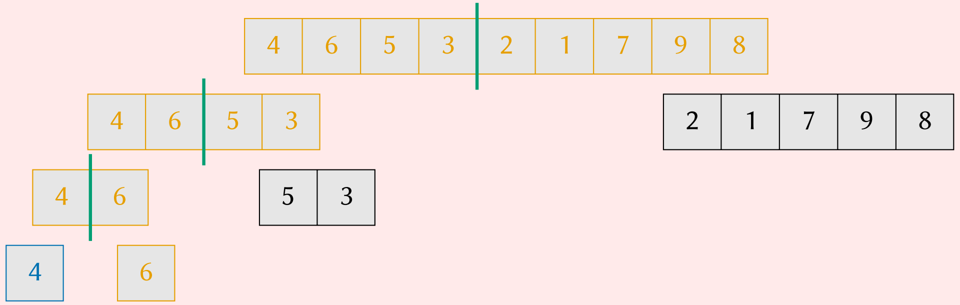
MergeSortR: A complete example



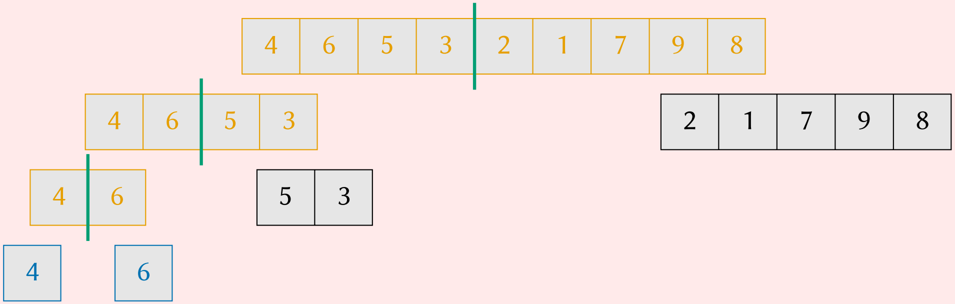
MergeSortR: A complete example



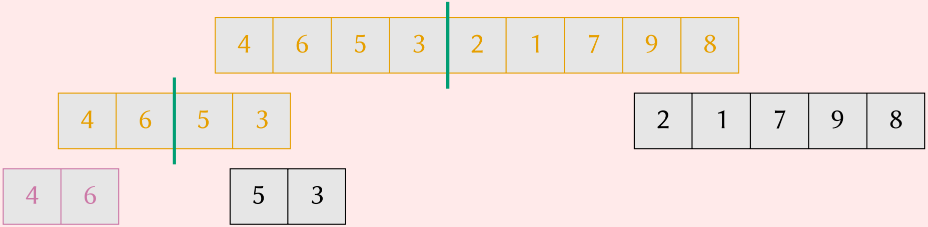
MergeSortR: A complete example



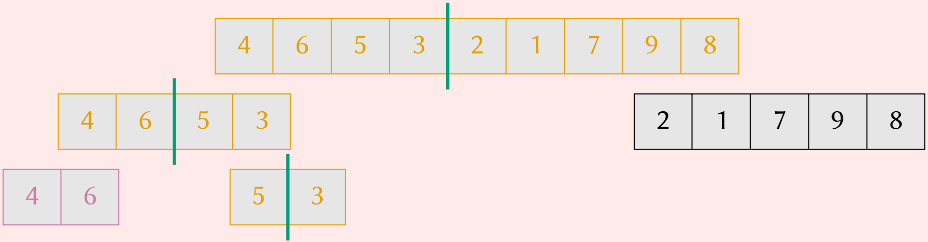
MergeSortR: A complete example



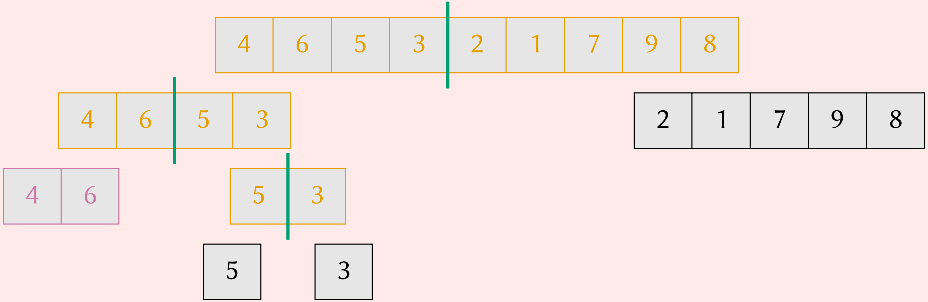
MergeSortR: A complete example



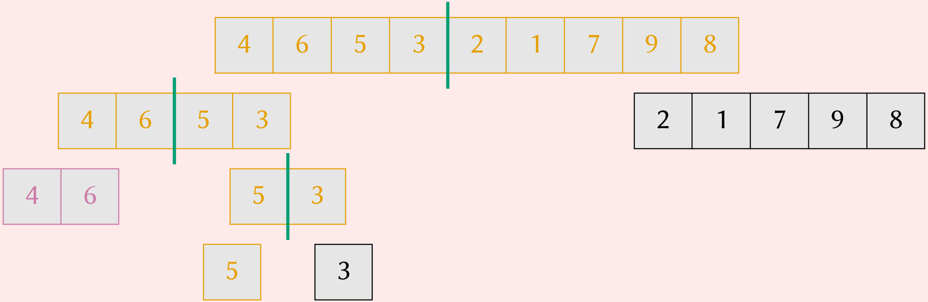
MERGESORTR: A complete example



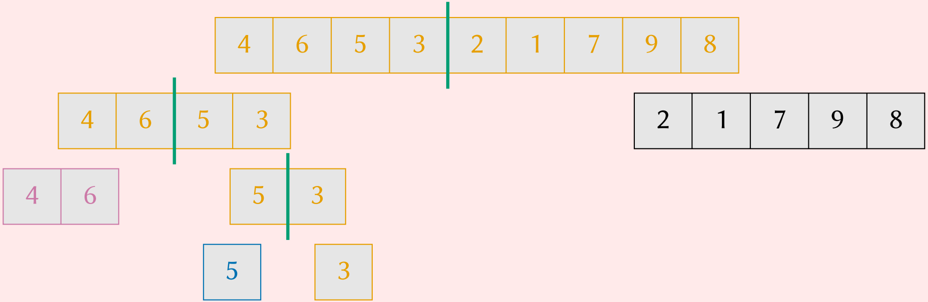
MergeSortR: A complete example



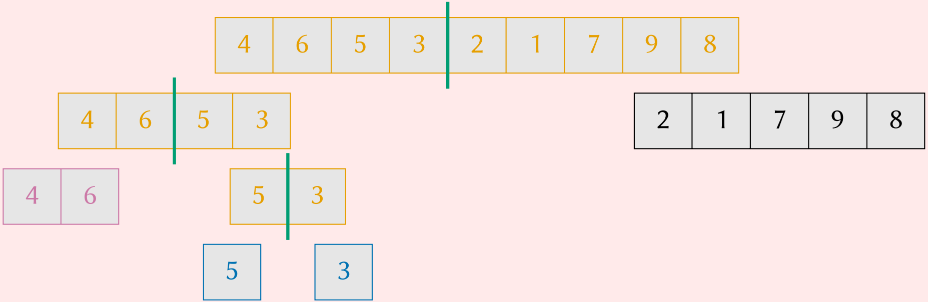
MergeSortR: A complete example



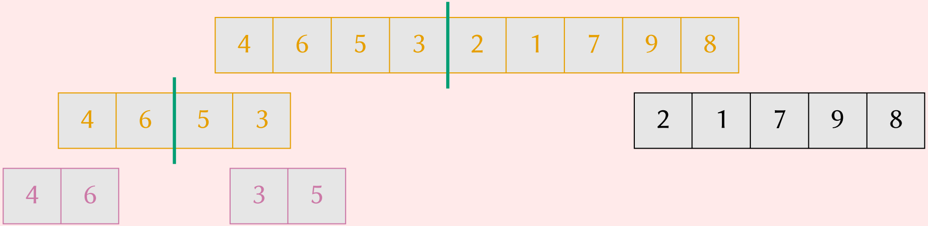
MergeSortR: A complete example



MergeSortR: A complete example



MergeSortR: A complete example



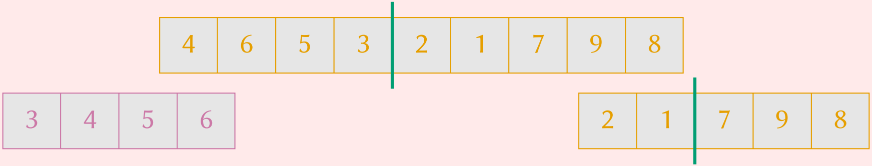
MergeSortR: A complete example

4	6	5	3	2	1	7	9	8

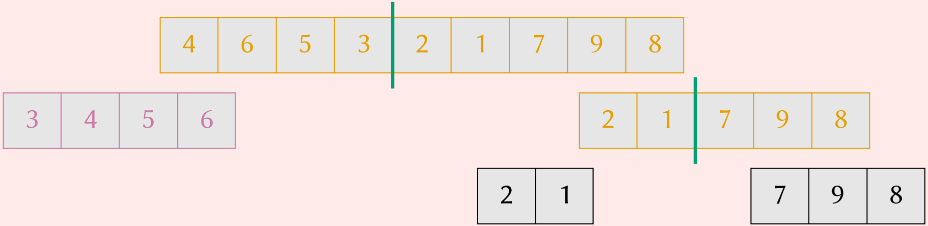
3	4	5	6

2	1	7	9	8

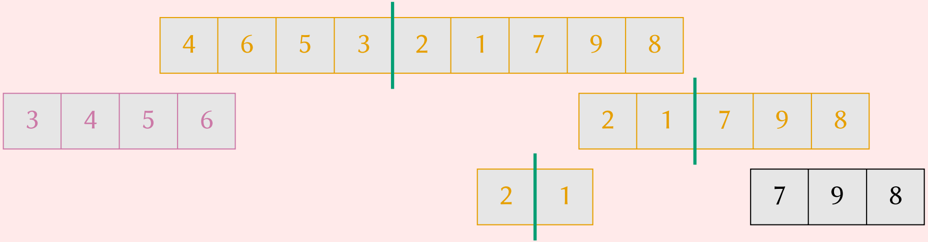
MergeSortR: A complete example



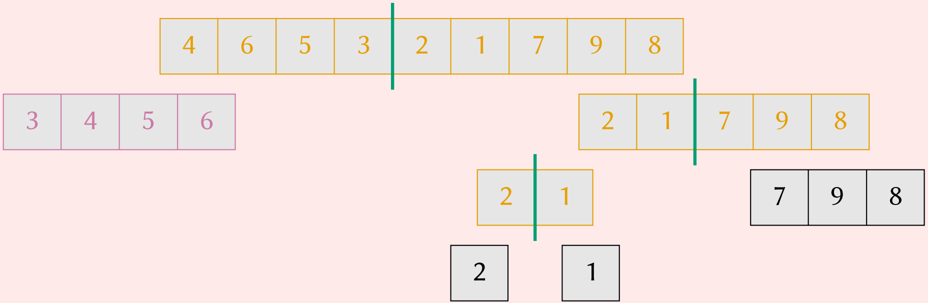
MergeSortR: A complete example



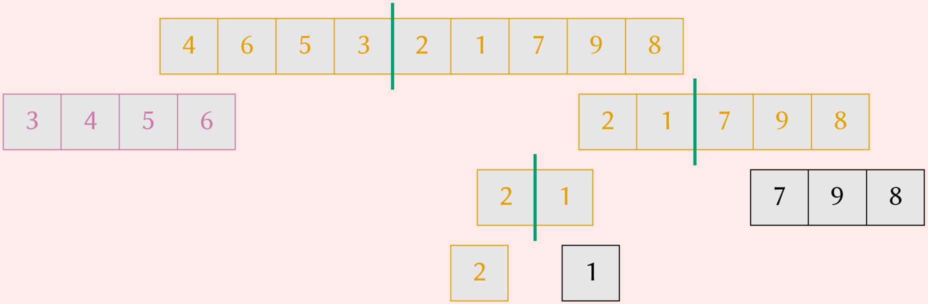
MergeSortR: A complete example



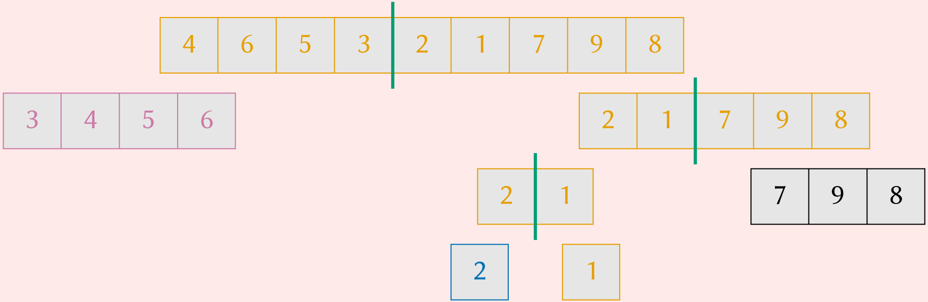
MergeSortR: A complete example



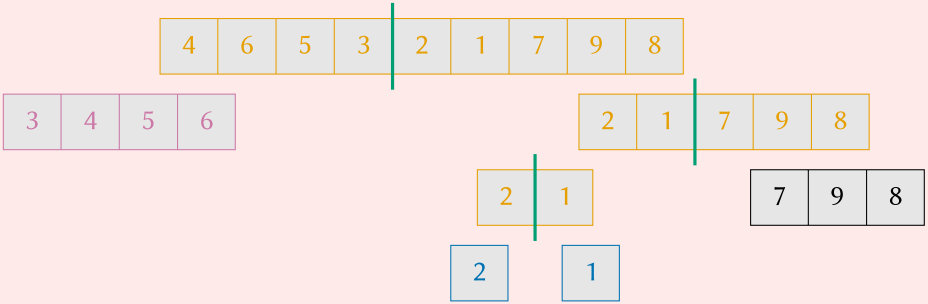
MergeSortR: A complete example



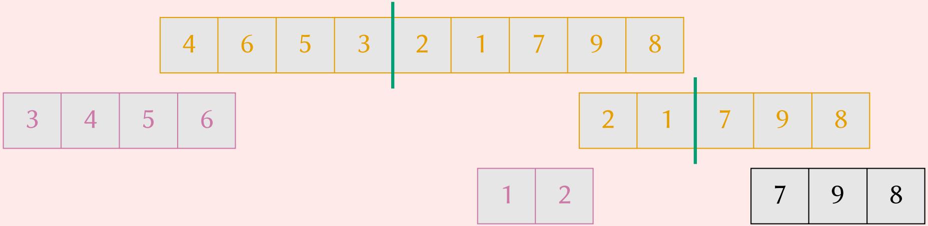
MergeSortR: A complete example



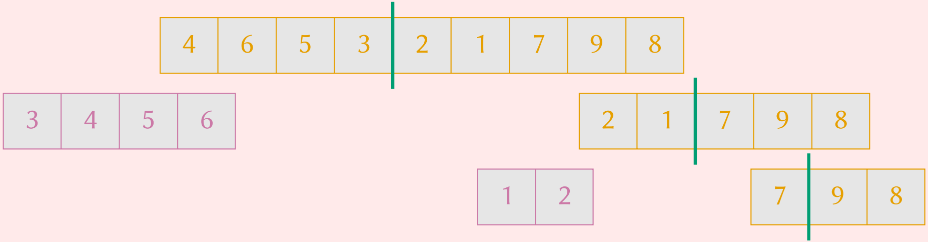
MergeSortR: A complete example



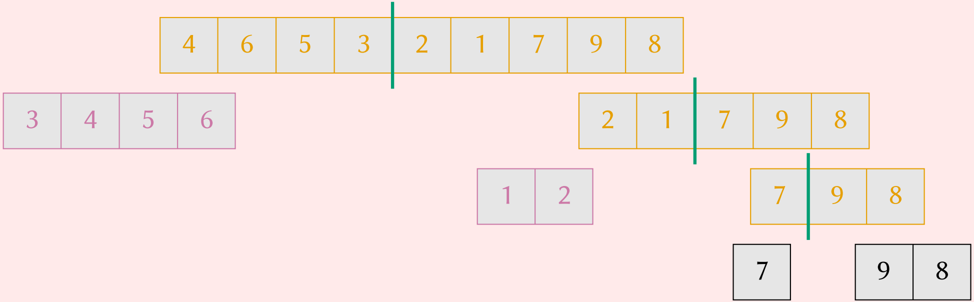
MergeSortR: A complete example



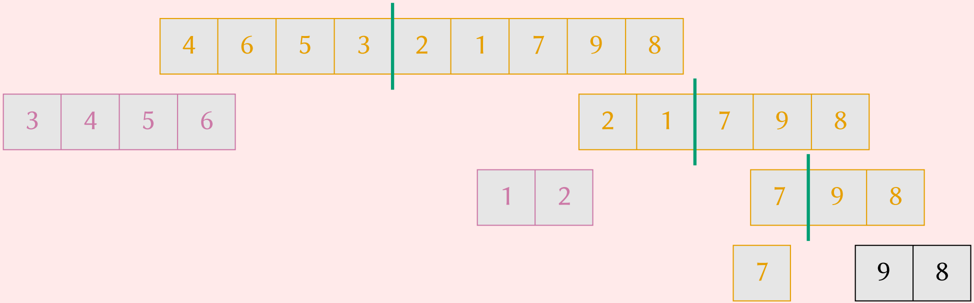
MergeSortR: A complete example



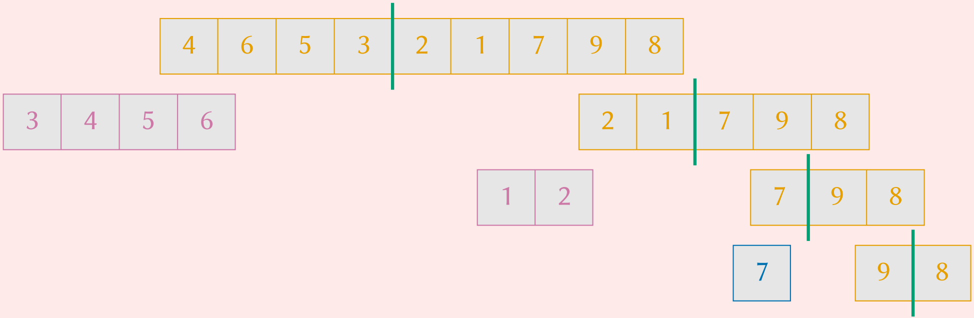
MergeSortR: A complete example



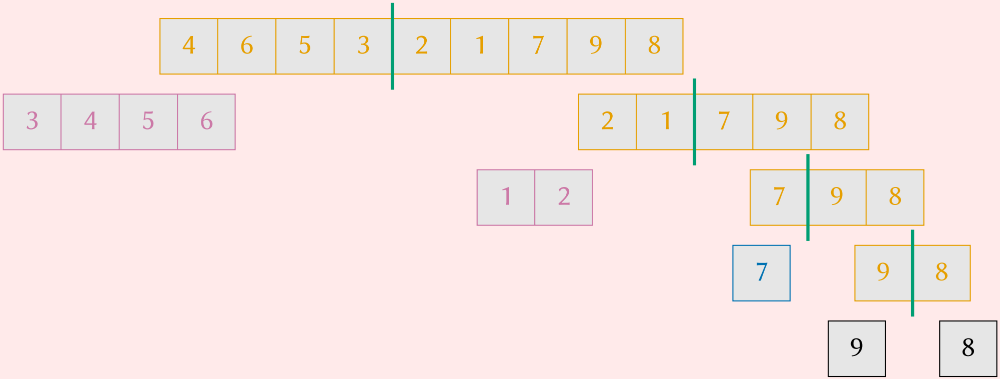
MergeSortR: A complete example



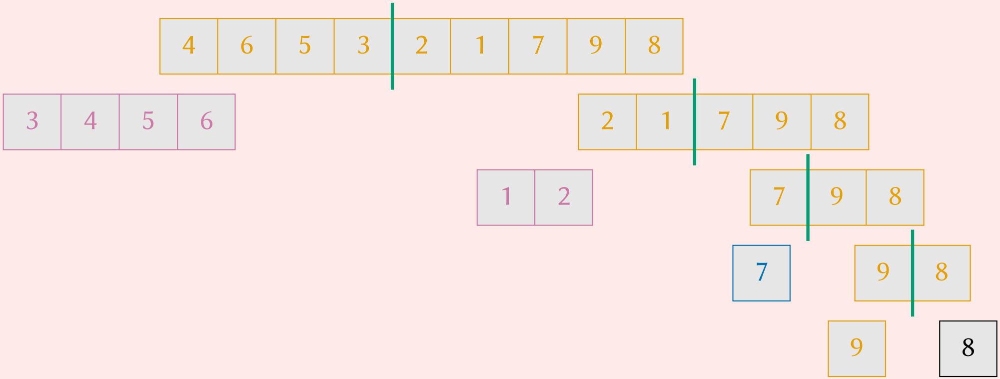
MergeSortR: A complete example



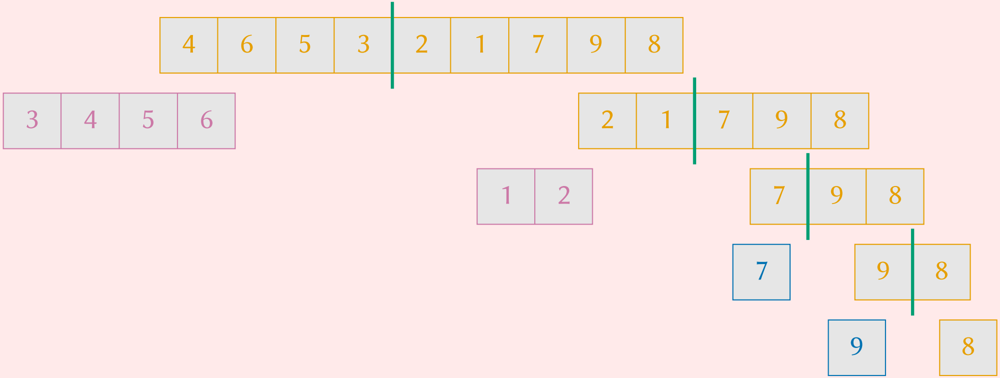
MergeSortR: A complete example



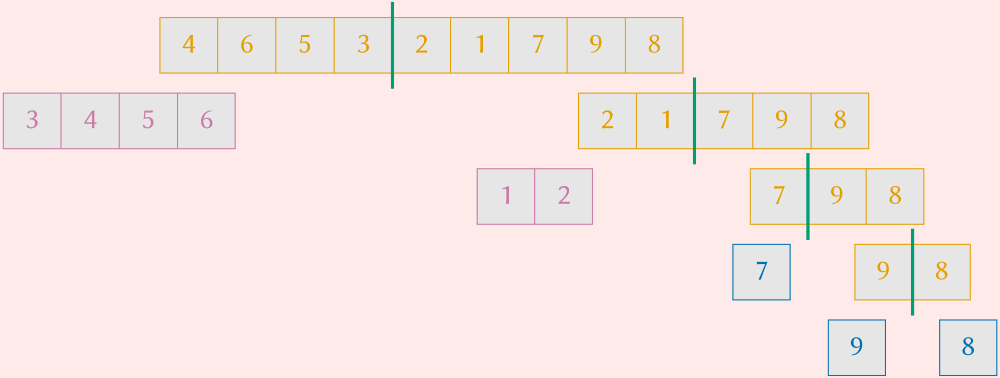
MergeSortR: A complete example



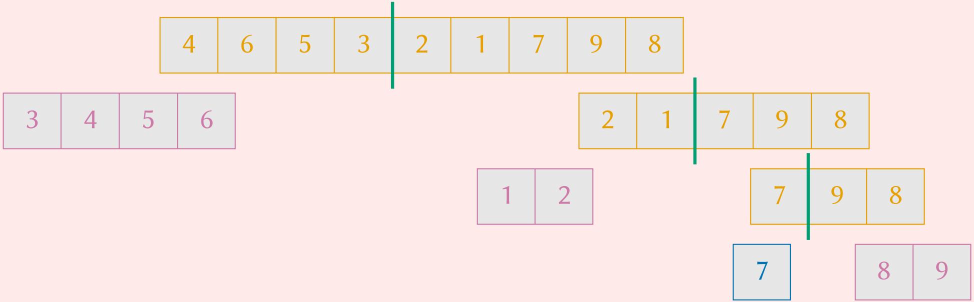
MergeSortR: A complete example



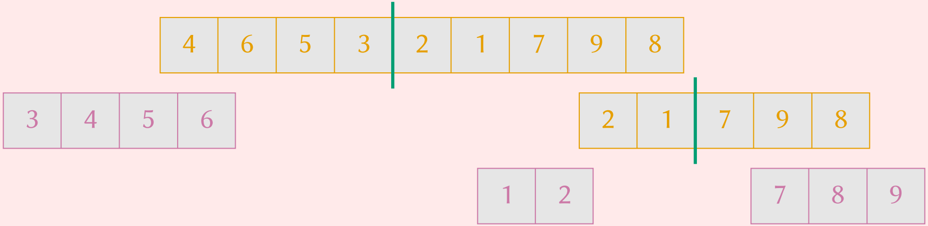
MergeSortR: A complete example



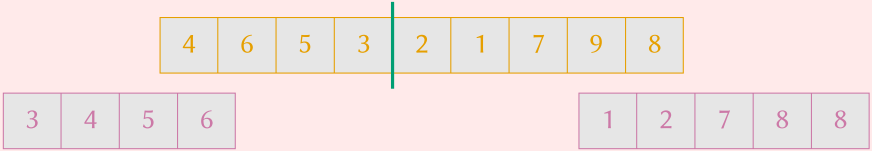
MergeSortR: A complete example



MergeSortR: A complete example



MergeSortR: A complete example



MergeSortR: A complete example

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

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.
```

The complexity of MergeSortR

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 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.
```

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**

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

  $L_1 :=$ 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.

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.

Base case.

The complexity of MergeSortR

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

if $end - start > 1$ **then**
  $mid := (end - start) \operatorname{div} 2$.
  $L_1 :=$ 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.

Base case.

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

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

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.
```

Recursive case.

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 $T(N)$

$$\begin{tikzcd} & N \arrow[dl] \arrow[dr] & \\ \frac{N}{2} & & \frac{N}{2} \end{tikzcd}$$

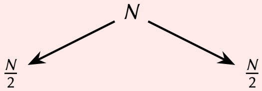
Number	Cost
$1 = 2^0$	N

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 $T(N)$

$$\begin{tikzcd} & N \arrow[dl] \arrow[dr] & \\ \frac{N}{2} & & \frac{N}{2} \end{tikzcd}$$

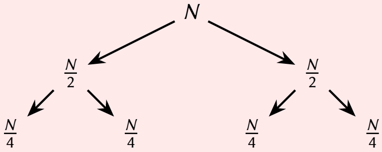
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$

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 $T(N)$



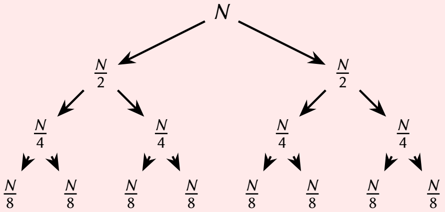
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$

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 $T(N)$



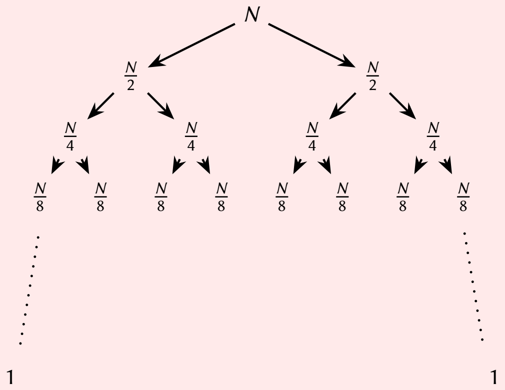
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$

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 $T(N)$



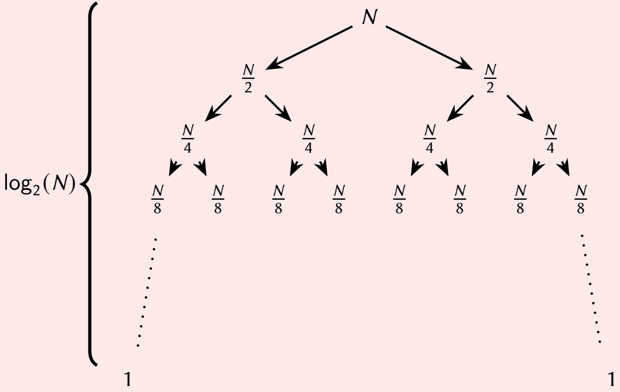
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$
$\vdots$	$\vdots$	$\vdots$
2^i	$\frac{N}{2^i}$	$2^i\frac{N}{2^i} = N$
$\vdots$	$\vdots$	$\vdots$

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 $T(N)$



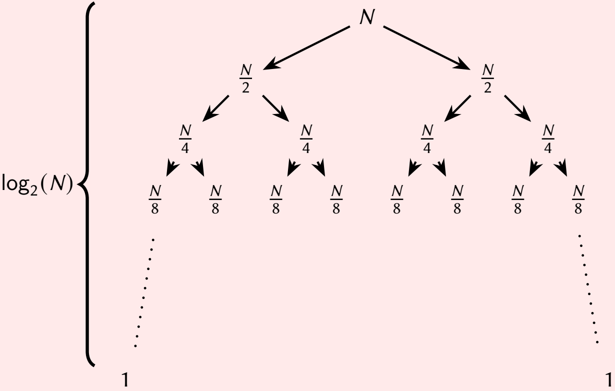
Number	Cost	Total
$1 = 2^0$	N	$1N = N$
$2 = 2^1$	$\frac{N}{2}$	$2\frac{N}{2} = N$
$4 = 2^2$	$\frac{N}{4}$	$4\frac{N}{4} = N$
$8 = 2^3$	$\frac{N}{8}$	$8\frac{N}{8} = N$
$\vdots$	$\vdots$	$\vdots$
2^i	$\frac{N}{2^i}$	$2^i\frac{N}{2^i} = N$
$\vdots$	$\vdots$	$\vdots$

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

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

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

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

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

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

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

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

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

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

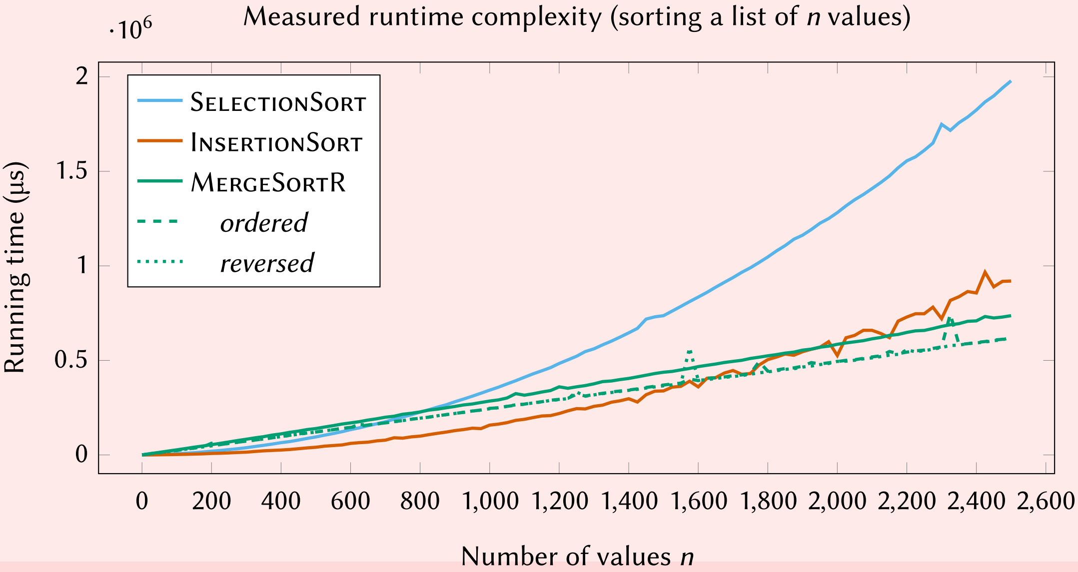
$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

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

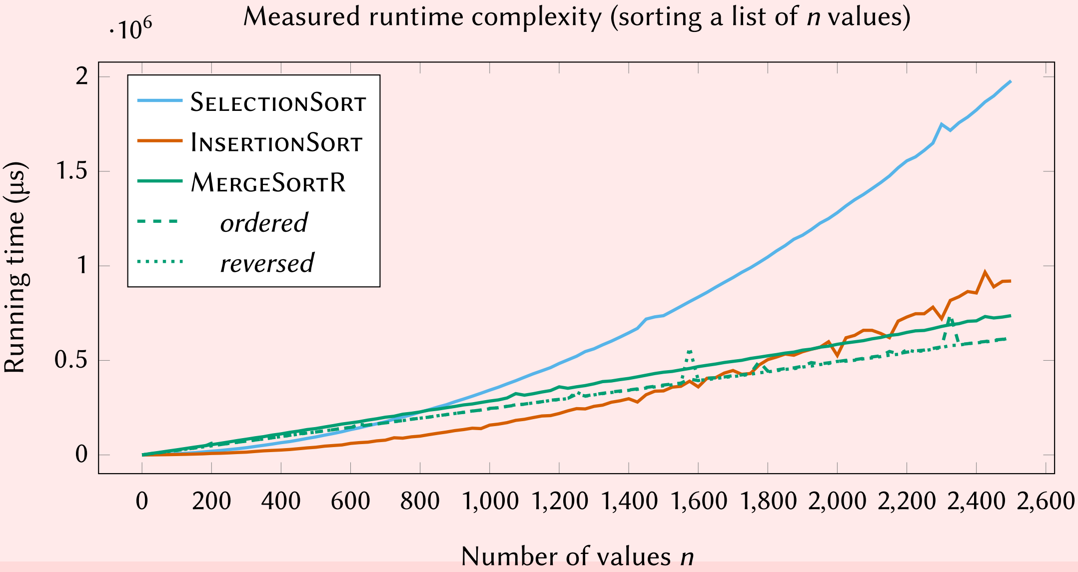
Properly work out *recurrence trees* when possible: often easier and clearer!

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

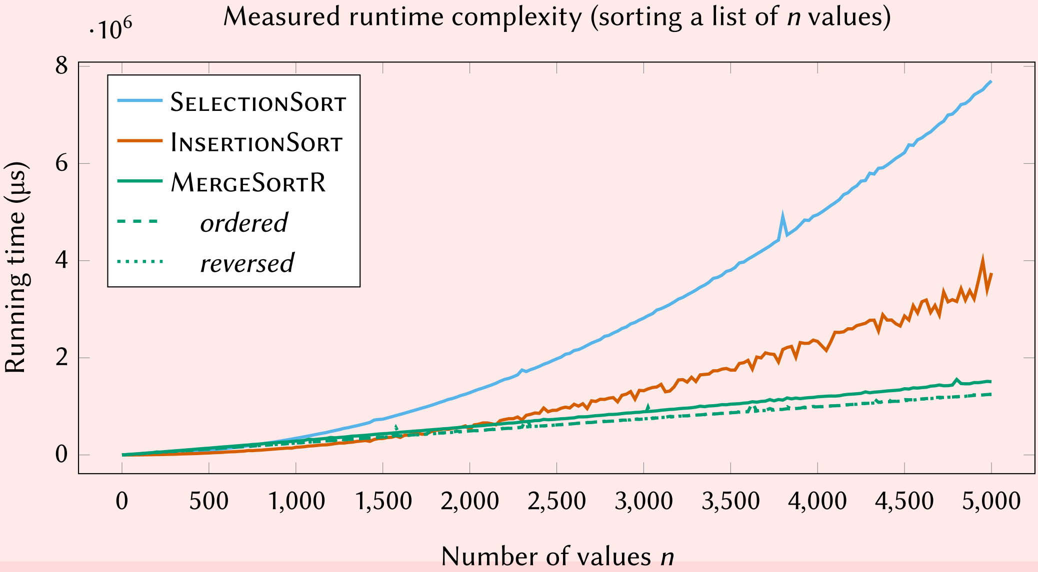
The performance of MergeSortR



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

```
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).

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

MergeSort: A better MergeSortR

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

1: R is a new array for N values.
2: $sl := 1$. The current *sorted length* of blocks in L.
3: **while** $sl \leq N$ **do**
4: $i := 0$.
5: **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 $2sl$.
6:
7: $i := i + 2sl$.
8: $sl := 2sl$.
9: 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($S[0\dots N)$, $start$, mid, end, $T[0\dots N)$):

Input: $0 \leq start \leq mid \leq end \leq N$ and
$S[start \dots mid)$ and $S[mid \dots end)$ 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:

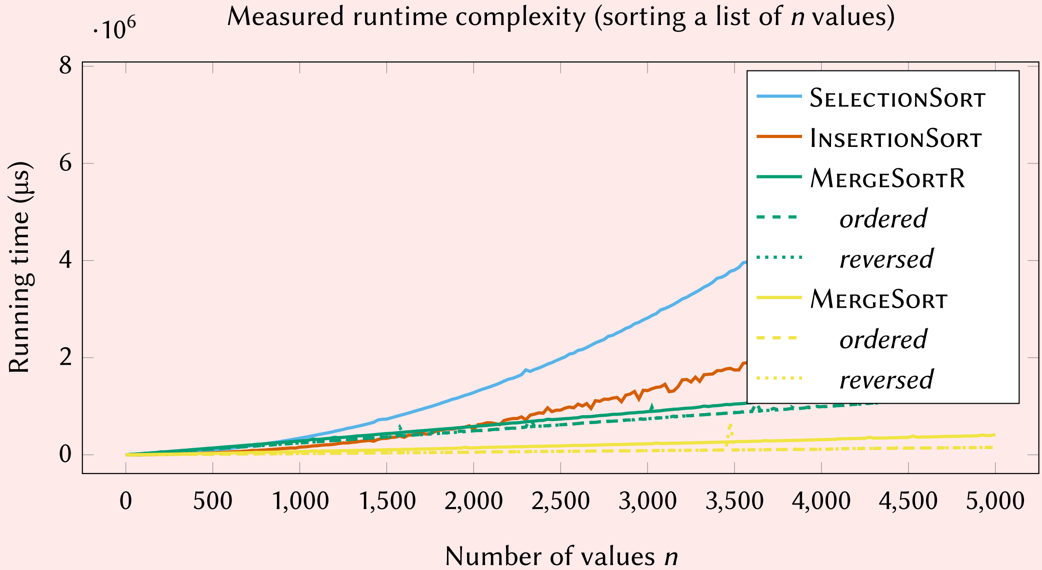
1	2	3	4	5	6	7	9		8

MergeSort: A better MergeSortR

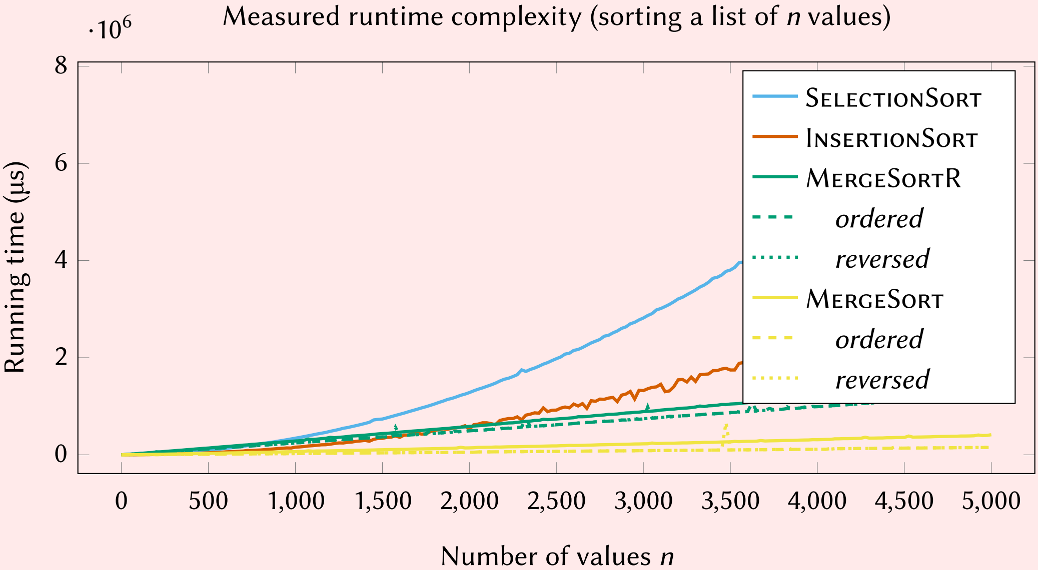
$sl = 16$, L:

1	2	3	4	5	6	7	8	9

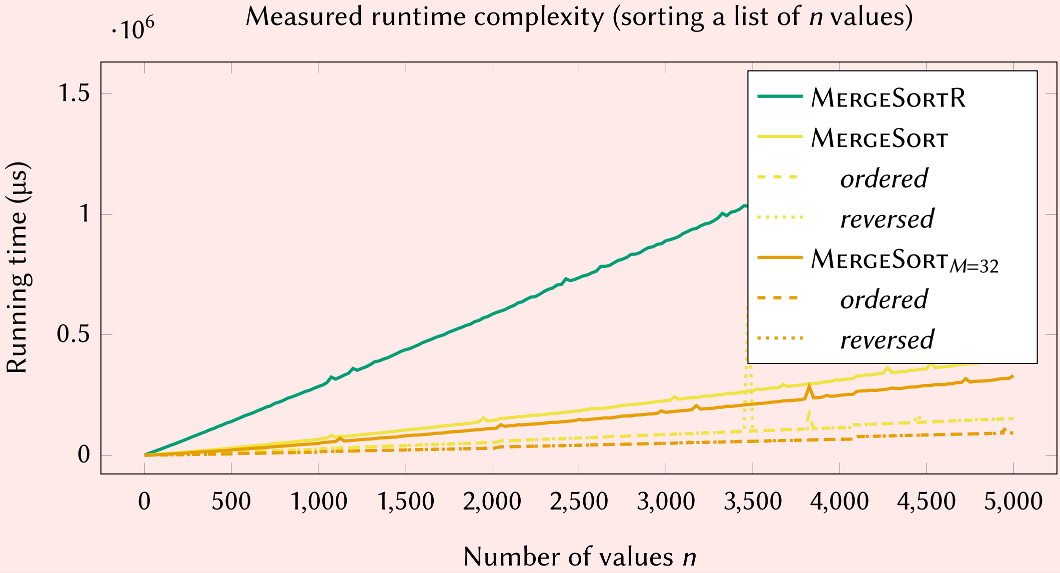
MergeSort: A better MergeSortR



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`	

Intermezzo: Recurrence trees

In a recurrence tree

- nodes labeled N represent a *function call* with "input size N";
- the children of a node represent *recursive calls*;
- per node, we can determine *the work* within that call (besides recursion);
- per depth, we can determine the *total work for that depth*;
- by *summing over all depths*: the total complexity.

Intermezzo: Recurrence trees

In a recurrence tree

- nodes labeled N represent a *function call* with "input size N";
- the children of a node represent *recursive calls*;
- per node, we can determine *the work* within that call (besides recursion);
- per depth, we can determine the *total work for that depth*;
- by *summing over all depths*: the total complexity.

We already saw two examples: BinarySearchR and MergeSortR.

Intermezzo: Recurrence trees

Example: the *Fibonacci numbers*

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

	Number	Cost	Total
N	$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

$$\begin{tikzcd} & N \arrow[dl] \arrow[dr] & \\ N-1 & & N-2 \end{tikzcd}$$

Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

$$\begin{tikzcd} & N \arrow[dl] \arrow[dr] & \\ N-1 & & N-1 \end{tikzcd}$$

Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

$$\begin{tikzcd} & N \arrow[dl] \arrow[dr] & \\ N-1 & & N-1 \end{tikzcd}$$

Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

N

$N-1$ $N-1$

$N-2$ $N-2$ $N-2$ $N-2$

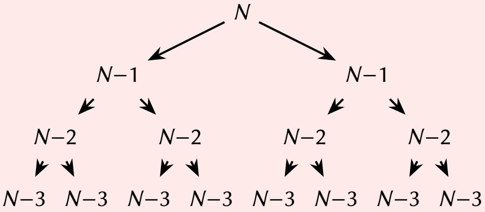
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



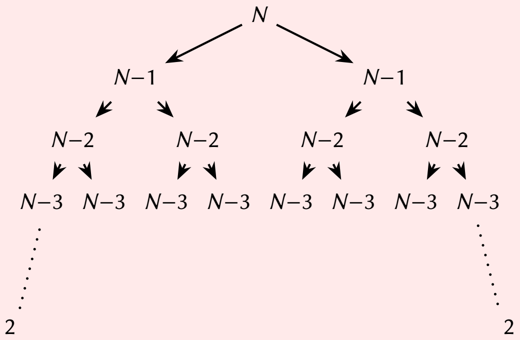
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



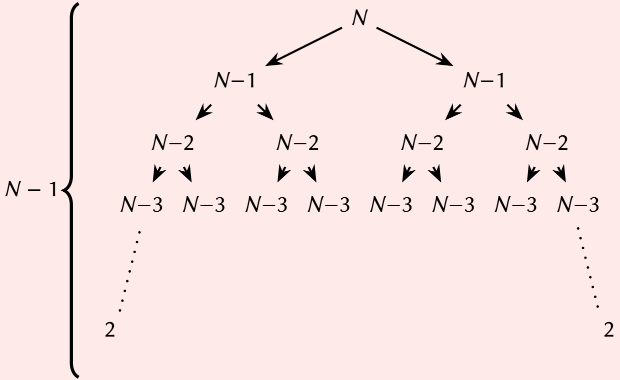
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



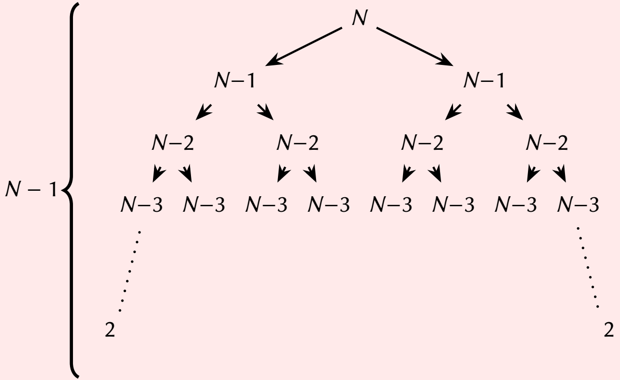
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



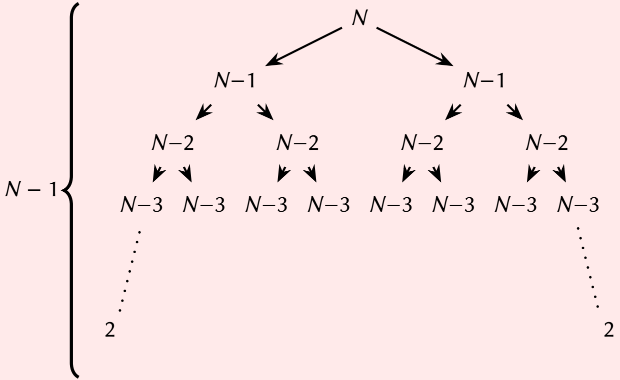
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$
		+
		$\sum_{i=0}^{N-2} 2^i$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



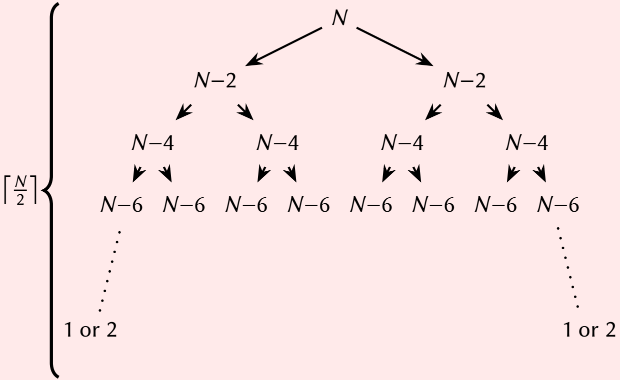
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$
		+
		$\sum_{i=0}^{N-2} 2^i = 2^{N-1} - 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $2^{\lceil \frac{N}{2} \rceil} \leq fib(N)$

Simplication: $fib(i-1) \geq fib(i-2)$.



Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$

$+$

$$\sum_{i=0}^{\lceil \frac{N}{2} \rceil} 2^i = 2^{\lceil \frac{N}{2} \rceil + 1} - 1$$

Intermezzo: Recurrence trees

Example: the *Fibonacci numbers*

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Via recurrence trees, we have proven that:

$$2^{\lceil \frac{N}{2} \rceil} \leq fib(N) \leq 2^N.$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Someone else has already proved this—so we can reuse the result!

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of BinarySearchR

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of BinarySearchR

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

We have $a = 1$, $b = 2$, $f(N) = 8 = \Theta(1) = N^{\log_2(1)}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of BinarySearchR

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

We have $a = 1$, $b = 2$, $f(N) = 8 = \Theta(1) = N^{\log_2(1)}$.

Case 2 yields: $T(N) = \Theta(N^{\log_2(1)} \log^1(N)) = \log(N)$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

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

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

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

We have $a = 2$, $b = 2$, $f(N) = N = \Theta(N) = N^{\log_2(2)}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case};\\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

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

We have $a = 2$, $b = 2$, $f(N) = N = \Theta(N) = N^{\log_2(2)}$.

Case 2 yields: $T(N) = \Theta(N^{\log_2(2)} \log^1(N)) = \Theta(N \log(N))$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 7T\left(\left\lfloor \frac{N}{4} \right\rfloor\right) + N & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 7T\left(\left\lfloor \frac{N}{4} \right\rfloor\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 7$, $b = 4$, $f(N) = N = \mathcal{O}N^{\log_4(7)-\epsilon}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case};\\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil\frac{N}{b}\right\rceil$ or $\left\lfloor\frac{N}{b}\right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)}\log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)}\log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1;\\ 7T\left(\left\lfloor\frac{N}{4}\right\rfloor\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 7$, $b = 4$, $f(N) = N = \mathcal{O}N^{\log_4(7)-\epsilon}$.

Case 1 yields: $T(N) = \Theta(N^{\log_4(7)}) \approx \Theta(N^{1.40367\ldots})$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case};\\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1;\\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N^3 = \Omega N^{\log_2(2)+\epsilon}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N^3 = \Omega N^{\log_2(2)+\epsilon}$.

Case 3 yields: $T(N) = \Theta(N^3)$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Feel free to use the Master Theorem, we will provide a copy during the final exam.

Can we do better than MergeSort?

Can we do better than MergeSort?

Algorithm CountSort($L[0 \ldots N]$):

Input: Each value in L is either 0 or 1.

```
count_0 := 0
for all v ∈ L do              Count number of 0's
    if v = 0 then
        count_0 := count_0 + 1.
for i := 0 to count_0 − 1 do  Write the counted number of 0's
    L[i] := 0.
for i := count_0 to N − 1 do  Write the remaining 1's
    L[i] := 1.
```

Can we do better than MergeSort?

Algorithm CountSort($L[0 \dots N]$):

Input: Each value in L is either 0 or 1.

```
count_0 := 0
for all v ∈ L do                          Count number of 0's
    if v = 0 then
        count_0 := count_0 + 1.
for i := 0 to count_0 − 1 do             Write the counted number of 0's
    L[i] := 0.
for i := count_0 to N − 1 do             Write the remaining 1's
    L[i] := 1.
```

Complexity: Linear ($\sim N$ comparisons, $\sim N$ changes)

Can we do better than MergeSort?

Algorithm CountSort($L[0 \ldots N]$):

Input: Each value in L is either 0 or 1.

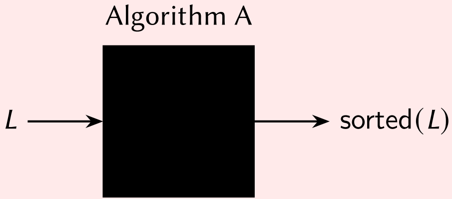
$count_0 := 0$
for all $v \in L$ **do** Count number of 0's
    if $v = 0$ **then**
        $count_0 := count_0 + 1.$
for $i := 0$ to $count_0 - 1$ **do** Write the counted number of 0's
    $L[i] := 0.$
for $i := count_0$ to $N - 1$ **do** Write the remaining 1's
    $L[i] := 1.$

Complexity: Linear ($\sim N$ comparisons, $\sim N$ changes)

CountSort does *not* solve general-purpose sorting!

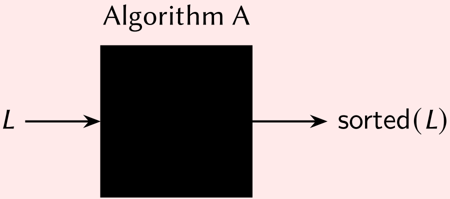
A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values



A lower bound for general-purpose sorting

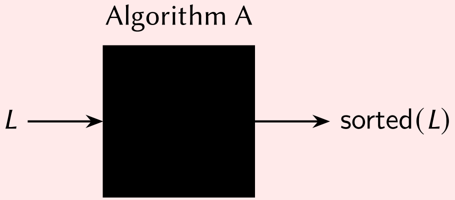
Assume: We have a list $L[0 \dots N)$ of N distinct values



When is Algorithm A *general-purpose*?

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values

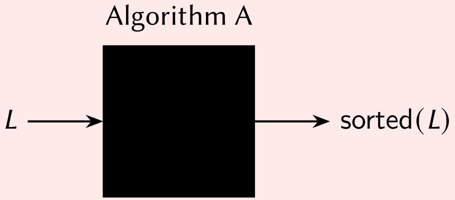


When is Algorithm A *general-purpose*?

- A uses *comparisons* to determine sorted order;
- A does *not require assumptions* on the value distribution in L.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values

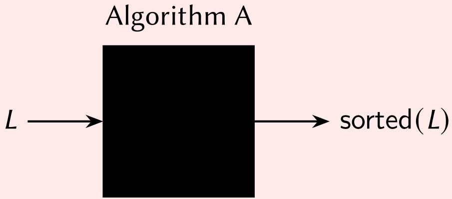


What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



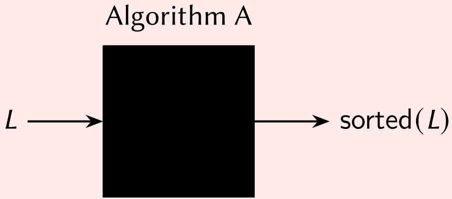
What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



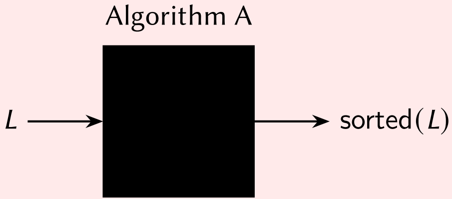
What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.
- Algorithm A uses *comparisons* to decide which operations to perform.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values



What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.
- Algorithm A uses *comparisons* to decide which operations to perform.

There must be a *distinguishing comparison* after which A behaves *differently*.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are
treated the same by Algorithm A up till this point

$\boxed{C\colon L[i] < L[j]}$

All lists in $\mathcal{S}$ for
which C *did not* hold

All lists in $\mathcal{S}$ for
which C *did* hold

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are
treated the same by Algorithm A up till this point

$\boxed{C\colon L[i] < L[j]}$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are
treated the same by Algorithm A up till this point

$$\boxed{C\colon L[i] < L[j]}$$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L.

- in $\mathcal{T}$, each leaf of $\mathcal{T}$ must represent *one* list;
- in $\mathcal{T}$, there must be a leaf for *every possible* list L.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are
treated the same by Algorithm A up till this point

$\boxed{C\colon L[i] < L[j]}$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L:

- in $\mathcal{T}$, each leaf of $\mathcal{T}$ must represent *one* list;
- in $\mathcal{T}$, there must be a leaf for *every possible* list L.

Otherwise not all distinct lists L are processed in a different way.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0 \ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

What is the worst-case length of path π?

The lengths of paths in $\mathcal{T}$ depend on the *height of $\mathcal{T}$*,

$\longrightarrow$ which depends on the *number of leaves* in $\mathcal{T}$.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1,\ldots,N$ in an unknown order?

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1, \ldots, N$ in an unknown order?

- N possible values for the first value,
- $N - 1$ possible values for the second value,
- $\ldots$
- 1 possible value for the last value.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1,\ldots,N$ in an unknown order?

- N possible values for the first value,
- $N-1$ possible values for the second value,
- $\ldots$
- 1 possible value for the last value.

$$\prod_{i=1}^{N} i = N! \text{ leaves} \qquad \text{(all possible permutations).}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

What is the worst-case length of path π?

The lengths of paths in $\mathcal{T}$ depend on the *height of* $\mathcal{T}$,

$\rightarrow$ which depends on the *number of leaves* $N!$ in $\mathcal{T}$.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

Consider a node n from which we can reach M leaves.

How do we make the distance from n to all its leaves minimal?

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

Consider a node n from which we can reach M leaves.

How do we make the distance from n to all its leaves minimal?

The left and right child of n each can reach $\frac{M}{2}$ leaves:

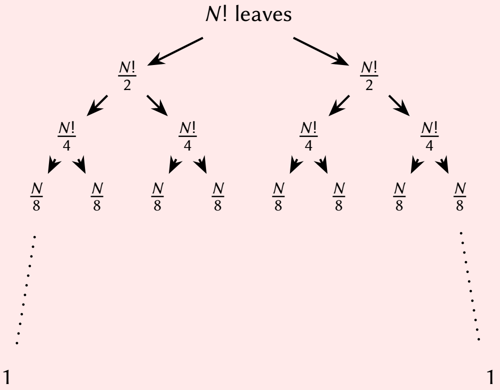
$\longrightarrow$ minimize the size of the tree rooted at *both children*.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

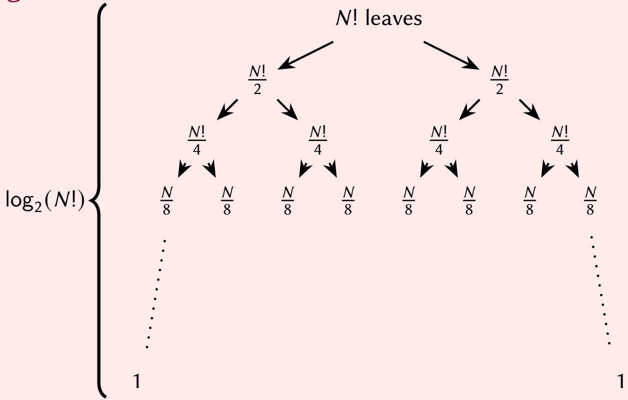


A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves



A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\log_2(N!) = \log_2(N\cdot(N-1)\cdot\cdots\cdot 1)$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\cdots\cdot 1)\\ &= \log_2(N)+\log_2(N-1)+\cdots+\log_2(1)\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\dots\cdot 1)\\
&= \log_2(N)+\log_2(N-1)+\dots+\log_2(1)\\
&\geq \log_2(N)+\log_2(N-1)+\dots+\log_2(\lceil \tfrac{N}{2}\rceil)
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\dots\cdot 1)\\
&= \log_2(N)+\log_2(N-1)+\dots+\log_2(1)\\
&\geq \log_2(N)+\log_2(N-1)+\dots+\log_2(\lceil\tfrac{N}{2}\rceil)\\
&\geq \tfrac{N}{2}\log_2(\tfrac{N}{2})
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\cdots\cdot 1)\\
&= \log_2(N) + \log_2(N-1) + \cdots + \log_2(1)\\
&\geq \log_2(N) + \log_2(N-1) + \cdots + \log_2(\lceil \tfrac{N}{2} \rceil)\\
&\geq \tfrac{N}{2}\log_2(\tfrac{N}{2})\\
&= \tfrac{N}{2}(\log_2(N) - 1)
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

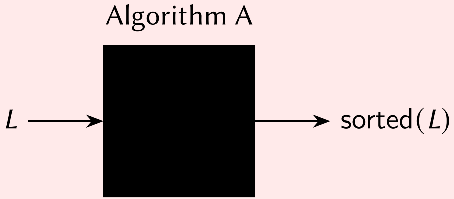
The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N \cdot (N-1) \cdot \dots \cdot 1) \\
&= \log_2(N) + \log_2(N-1) + \dots + \log_2(1) \\
&\geq \log_2(N) + \log_2(N-1) + \dots + \log_2(\lceil \tfrac{N}{2} \rceil) \\
&\geq \tfrac{N}{2} \log_2(\tfrac{N}{2}) \\
&= \tfrac{N}{2}(\log_2(N) - 1) \\
&= \tfrac{N}{2} \log_2(N) - \tfrac{N}{2} \sim N \log_2(N).
\end{aligned}$$

A lower bound for general-purpose sorting

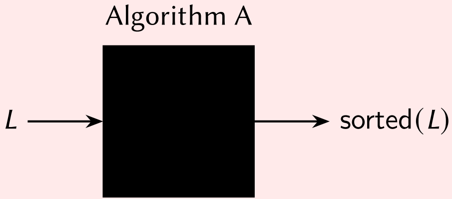
Assume: We have a list $L[0 \ldots N)$ of N distinct values



If Algorithm A is general-purpose, then A will perform
at-least $\sim N \log_2(N)$ *comparisons* for some inputs of N values.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values

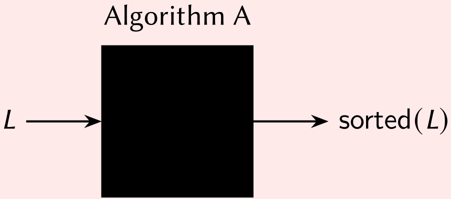


If Algorithm A is general-purpose, then A will perform *at-least* $\sim N \log_2(N)$ *comparisons* for some inputs of N values.

If Algorithm A performs less comparisons for *some* inputs, then A will perform more comparisons for *other* inputs.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



General-purpose sorting algorithms such as MergeSort are *optimal*:
their worst-case complexity matches the lower bound of $\sim N \log_2(N)$.

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\textit{dept}, \textit{code}, \textit{sid}, \textit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on (*dept*, *code*) and within each group sort enrollments on *date*.

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on $(\mathit{dept}, \mathit{code})$ and within each group sort enrollments on *date*.

Brute-force solution: Lexicographical sorting on $(\mathit{dept}, \mathit{code}, \mathit{date})$

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in \text{enrolled}$. We use the comparison

$$(d_1, c_1, s_1, t_1) \textit{ before } (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2)) \lor \\ ((d_1 = d_2) \land (c_1 = c_2) \land (t_1 < t_2)).$$

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on $(\mathit{dept}, \mathit{code})$ and within each group sort enrollments on *date*.

Brute-force solution: Lexicographical sorting on $(\mathit{dept}, \mathit{code}, \mathit{date})$

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in$ enrolled. We use the comparison

$$(d_1, c_1, s_1, t_1) \mathbin{\mathit{before}} (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2)) \lor \\ ((d_1 = d_2) \land (c_1 = c_2) \land (t_1 < t_2)).$$

Downside: During sorting, we end up throwing away the existing ordering on *date*, and then we rebuild that order from scratch!

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on $(\mathit{dept}, \mathit{code})$ and within each group sort enrollments on *date*.

Better solution: Use a *stable sort algorithm*

A *stable sort algorithm* maintains the relative order of “equal values”.

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in$ enrolled. If we sort enrolled using a *stable sort algorithm* using the comparison

$$(d_1, c_1, s_1, t_1) \textit{ before } (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2))$$

then within each $(\mathit{dept}, \mathit{code})$-group, enrollments remain ordered on *date* for free!

Stable sorting

Definition

Let L be a list that is already ordered with respect to some attributes $a_1, \ldots, a_n$.
Consider a sort step S that re-orders L based on other attributes $b_1, \ldots, b_m$.

We say that the sort step S is *stable* if, for every value $r_1 \in L$ and $r_2 \in L$ such that r_1 originally came before r_2 and r_1 and r_2 agreee on attributes $b_1, \ldots, b_m$, the resulting re-ordered list will still have r_1 come before r_2.

Stable sorting

Definition

Let L be a list that is already ordered with respect to some attributes $a_1, \dots, a_n$.
Consider a sort step S that re-orders L based on other attributes $b_1, \dots, b_m$.

We say that the sort step S is *stable* if, for every value $r_1 \in L$ and $r_2 \in L$ such that r_1 originally came before r_2 and r_1 and r_2 agreee on attributes $b_1, \dots, b_m$, the resulting re-ordered list will still have r_1 come before r_2.

Question: Have we already seen stable sort algorithms?

Yes: SelectionSort, InsertionSort, and MergeSort.

Note: even minor changes to these algorithms will make them non-stable!
(e.g., changing $<$ into $\leq$).