Fundamentals

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

- Analysis of algorithms and data structures: *correctness* and *complexity*.
- Common design strategies for algorithms and data structures.
- A useful toolbox of standard fundamental algorithms and data structures.
- Graph representations and fundamental graph algorithms.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

- Analysis of algorithms and data structures: *correctness* and *complexity*.
- Common design strategies for algorithms and data structures.
- A useful toolbox of standard fundamental algorithms and data structures.
- Graph representations and fundamental graph algorithms.

This course is *not* about learning how to program (basic programming is prior knowledge).

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

An algorithm takes one-or-more values as input
and produces an output via a *well-defined computational procedure*.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

An algorithm takes one-or-more values as input
and produces an output via a *well-defined computational procedure*.

Definition (Data structure)

Scheme to store and organize data in order to facilitate *efficient* access and modification.

About programming languages

We all have our own favorites.

About programming languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

About programming languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

For *optimal* implementations, we sometimes need a lower-level toolbox.
E.g., references or pointers when implementing data structures.

About programming languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

For *optimal* implementations, we sometimes need a lower-level toolbox.
E.g., references or pointers when implementing data structures.

Many programming languages suffice, e.g.,

- the book has many examples in Java;
- I will provide some examples in C++.

Feel free to experiment in your programming language of choice.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0,$ `false`.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

```
i, r := 0, false.
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
return r.
```

Is Contains correct?

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0, \texttt{false}$.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

Result: return `true` if $v \in L$ and `false` otherwise.

Is CONTAINS correct?

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

Input: L is an *array*, v a value.

$i, r := 0,$ `false`.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

Result: return `true` if $v \in L$ and `false` otherwise.

Is CONTAINS correct?

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm EVILCONTAINS(L, v):

Input: L is an *array*, v a value.

1: $L := []$.

2: **return** `false`.

Result: return `true` if $v \in L$ and `false` otherwise.

Is EVILCONTAINS correct?

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

$i, r := 0,$ `false`.

while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.

return r.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0,$ `false`.
/* L is an *array*, v a value, $i = 0$, and $r =$ `false`. */

while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.
/* r is `true` if $v \in L$ and `false` otherwise. */
return r.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

$i, r := 0, \texttt{false}$.
/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */
/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */
return r.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Proof by induction

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Proof by induction

Base case Prove invariant holds before the loop.

Hypothesis The invariant holds after the j-th, $j < m$, repetition of the loop.

Step Assume invariant holds when we start the m-th repetition of the loop.
Prove invariant holds again when we reach the end of the m-th repetition.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.
/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */
2: **while** ....

Argument

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.

/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */

2: **while** ....

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

```
i, r := 0, false.
   /* L is an array, v a value, i = 0, and r = false. */
while ....
```

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.
2. $L[0, 0)$ is empty, hence $v \notin L[0, 0)$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.
/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */
2: **while** ....

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.
2. $L[0, 0)$ is empty, hence $v \notin L[0, 0)$.
3. Hence, $r = \texttt{false}$ must hold (which is the case).

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the *m*-th repetition.

```
while i ≠ |L| do
    /* Invariant and i ≠ |L|. */
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
    /* Invariant. */
```

Argument

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the *m*-th repetition.

while $i \neq |L|$ **do**
    /* Invariant and $i \neq |L|$. */
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
    /* Invariant. */

Argument

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the m-th repetition.

```
while i ≠ |L| do
    /* Invariant and i ≠ |L|. */
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
    /* Invariant. */
```

Argument

If-statement: Case distinction.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r =$ `true`, $v \notin L[0, i)$ implies $r =$ `false`. */

Case distinction: If-case ($L[i] = v$ holds).

3: **if** $L[i] = v$ **then**
    /* Invariant, $i \neq |L|$, and $L[i] = v$ */
4:     $r :=$ `true`.
5:     $i := i + 1$.
    /* Invariant. */

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
       /* Invariant, i ≠ |L|, and L[i] = v */
4:     r := true.
5:     i := i + 1.
       /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
      /* Invariant, i ≠ |L|, and L[i] = v */
4:    r := true.
5:    i := i + 1.
      /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
     /* Invariant, i ≠ |L|, and L[i] = v */
4:   r := true.
5:   i := i + 1.
     /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.
3. Hence, $r_{\text{new}} = \texttt{true}$ must hold (which is the case).

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

3: **if** $L[i] = v$ **then**
  /* Invariant, $i \neq |L|$, and $L[i] = v$ */
4:   $r := \texttt{true}$.
5:   $i := i + 1$.
  /* Invariant. */

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
      /* Invariant, i ≠ |L|, and L[i] = v */
4:    r := true.
5:    i := i + 1.
      /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $0 \leq i \leq |L|$ and $i \neq |L|$ implies $0 \leq i < |L|$.
2. $i_{\text{new}} = i + 1$, hence, $0 < i_{\text{new}} \leq |L|$.
3. $0 < i_{\text{new}} \leq |L|$ implies $0 \leq i_{\text{new}} \leq |L|$.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** …**else**

    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */

7: $i := i + 1$.

    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

```
6: if L[i] = v then ... else
       /* Invariant, i ≠ |L|, and L[i] ≠ v */
7:     i := i + 1.
       /* Invariant. */
```

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** … **else**
    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7:     $i := i + 1$.
    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** … **else**
 /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7: $i := i + 1$.
 /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.
3. Hence, $r = \texttt{true}$ must hold (which is the case).

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

```
6: if L[i] = v then ... else
       /* Invariant, i ≠ |L|, and L[i] ≠ v */
7:     i := i + 1.
       /* Invariant. */
```

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{false}$. Hence, $v \notin L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$ and $L[i] \neq v$, hence, $v \notin L[0, i_{\text{new}})$.
3. Hence, $r = \texttt{false}$ must hold (which is the case).

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

6: **while** $i \neq |L|$ **do** … **end while**

/* Invariant and $\neg(i \neq |L|)$. */

/* r is `true` if $v \in L$ and `false` otherwise. */

7: **return** r.

Are we done?

Intermezzo: The correctness of Contains

We have proven the invariant holds

/* inv: $0 \le i \le |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

6: **while** $i \neq |L|$ **do** … **end while**

/* Invariant and $\neg(i \neq |L|)$. */

/* r is `true` if $v \in L$ and `false` otherwise. */

7: **return** r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is `true` if $v \in L$ and `false` otherwise */?

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ **do** … **end while**
/* Invariant and $\neg(i \neq |L|)$. */
/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */
return r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise */?

Argument

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r =$ `true`, $v \notin L[0, i)$ implies $r =$ `false`. */

6: **while** $i \neq |L|$ **do** … **end while**

/* Invariant and $\neg(i \neq |L|)$. */

/* r is `true` if $v \in L$ and `false` otherwise. */

7: **return** r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is `true` if $v \in L$ and `false` otherwise */?

Argument

1. $\neg(i \neq |L|)$ implies $i = |L|$.

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r =$ `true`, $v \notin L[0, i)$ implies $r =$ `false`. */

6: **while** $i \neq |L|$ **do** … **end while**

/* Invariant and $\neg(i \neq |L|)$. */

/* r is `true` if $v \in L$ and `false` otherwise. */

7: **return** r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is `true` if $v \in L$ and `false` otherwise */?

Argument

1. $\neg(i \neq |L|)$ implies $i = |L|$.
2. $L[0, i)$ with $i = |L|$ is equivalent to L.

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0,i)$ implies $r = \texttt{true}$, $v \notin L[0,i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ **do** … **end while**
/* Invariant and $\neg(i \neq |L|)$. */
/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */
return r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise */?

Argument

1. $\neg(i \neq |L|)$ implies $i = |L|$.
2. $L[0,i)$ with $i = |L|$ is equivalent to L.
3. Hence, $v \in L$ implies $r = \texttt{true}$, $v \notin L$ implies $r = \texttt{false}$.

Intermezzo: The correctness of Contains

We have proven the invariant holds

/* inv: $0 \le i \le |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
6: **while** $i \neq |L|$ **do** … **end while**
/* Invariant and $\neg(i \neq |L|)$. */
/* r is true if $v \in L$ and false otherwise. */
7: **return** r.

Are we done?

- *Assuming* /* Invariant and $\neg(i \neq |L|)$ */,
  Do we have /* r is true if $v \in L$ and false otherwise */? $\longrightarrow$ *Yes!*
- Do we reach the end of the loop?

Intermezzo: The correctness of CONTAINS

Are we done?

- Do we reach the end of the loop?

$i, r := 0, \texttt{false}$.
while $i \neq |L|$ **do**
  if $L[i] = v$ **then**
    $r := \texttt{true}$.
    $i := i + 1$.
  else
    $i := i + 1$.

Intermezzo: The correctness of Contains

Are we done?

- Do we reach the end of the loop? $\longrightarrow$ *Yes—obviously i will only be $0, \ldots, |L|$.*

```
i, r := 0, false.
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
```

Intermezzo: The correctness of CONTAINS

Are we done?

- Do we reach the end of the loop? $\longrightarrow$ *Yes—obviously i will only be $0, \ldots, |L|$.*

$i, r := 0,$ `false`.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.

Formal argument: prove a bound function

Define a *bound function* f on the state of the algorithm such that the output of f:

- is a *natural number* $(0, 1, 2, \ldots)$.
- *strictly decreases* after each iteration of the loop body.

Intermezzo: The correctness of Contains

Are we done?

- Do we reach the end of the loop? $\longrightarrow$ *Yes—obviously i will only be $0, \ldots, |L|$.*

```
i, r := 0, false.
while i ≠ |L| do /* bound function: |L| − i */
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
```

Formal argument: prove a bound function

Define a *bound function* f on the state of the algorithm such that the output of f:

- is a *natural number* $(0, 1, 2, \ldots)$.
- *strictly decreases* after each iteration of the loop body.

Intermezzo: The correctness of CONTAINS

Are we done?

- Do we reach the end of the loop? $\longrightarrow$ *Yes—obviously i will only be $0, \dots, |L|$.*

$i, r := 0, \texttt{false}.$ $\longleftarrow$ *$|L| - i$ starts at $|L|$, $|L| \geq 0$.*
while $i \neq |L|$ **do** /* bound function: $|L| - i$ */ $\longleftarrow$ *$|L| - i$ stops at 0.*
  if $L[i] = v$ **then**
    $r := \texttt{true}.$
    $i := i + 1.$ $\longleftarrow$ *$|L| - i$ strictly decreases.*
  else
    $i := i + 1.$ $\longleftarrow$ *$|L| - i$ strictly decreases.*

Formal argument: prove a bound function

Define a *bound function* f on the state of the algorithm such that the output of f:

- is a *natural number* $(0, 1, 2, \dots)$.
- *strictly decreases* after each iteration of the loop body.

Intermezzo: How to prove correctness

Summary

- Define a *pre-condition*: What restrictions do we require on the input?
- Define a *post-condition*: What should the output be?
- Prove that *running the program* turns the pre-condition into the post-condition.

Intermezzo: How to prove correctness

Summary

- Define a *pre-condition*: What restrictions do we require on the input?
- Define a *post-condition*: What should the output be?
- Prove that *running the program* turns the pre-condition into the post-condition.

Hard parts: loops $\longrightarrow$ invariants (induction proofs) and bound functions.

Intermezzo: How to prove correctness

Hard parts: loops $\longrightarrow$ invariants (induction proofs) and bound functions.

On finding invariants

Most induction proofs are *easy* if you have the correct *induction hypothesis*.

Finding the induction hypothesis (invariant) is the *hard part* $\longrightarrow$ trial and error.

Intermezzo: How to prove correctness

Hard parts: loops $\longrightarrow$ invariants (induction proofs) and bound functions.

On finding invariants

Most induction proofs are *easy* if you have the correct *induction hypothesis*.
Finding the induction hypothesis (invariant) is the *hard part* $\longrightarrow$ trial and error.

Take inspiration from what should hold *after the loop* and *what is changed during the loop*.

Example: Contains

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L| \ldots$ **end while**
/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */

Intermezzo: How to prove correctness

Hard parts: loops $\longrightarrow$ invariants (induction proofs) and bound functions.

On finding invariants

Most induction proofs are *easy* if you have the correct *induction hypothesis*.
Finding the induction hypothesis (invariant) is the *hard part* $\longrightarrow$ trial and error.

Take inspiration from what should hold *after the loop* and *what is changed during the loop*.

Example: Contains

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

3: **while** $i \neq |L| \ldots$ **end while**

/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */

Intermezzo: How to prove correctness

Hard parts: loops $\longrightarrow$ invariants (induction proofs) and bound functions.

On finding invariants

Most induction proofs are *easy* if you have the correct *induction hypothesis*.
Finding the induction hypothesis (invariant) is the *hard part* $\longrightarrow$ trial and error.

Take inspiration from what should hold *after the loop* and *what is changed during the loop*.

Example: Contains

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ ... **end while**
/* r is `true` if $v \in L$ and `false` otherwise. */

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

```
i, r := 0, false.
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
return r.
```

What is the complexity of Contains?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

Intermezzo: The complexity of Contains

What is the complexity of Contains?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

Algorithm Contains(L, v):

```
i, r := 0, false.
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
return r.
```

We need a *scientific model* of the work done by Contains

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINS?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

Algorithm CONTAINS(L, v):

$i, r := 0,$ `false`.	$\longleftarrow$ *2 instruction(s).*
while $i \neq \lvert L \rvert$ **do**	$\longleftarrow$ *2 instruction(s).*
if $L[i] = v$ **then**	$\longleftarrow$ *3 instruction(s).*
$r :=$ `true`.	$\longleftarrow$ *1 instruction(s).*
$i := i + 1$.	$\longleftarrow$ *2 instruction(s).*
else	
$i := i + 1$.	$\longleftarrow$ *2 instruction(s).*
return r.	$\longleftarrow$ *1 instruction(s).*

We need a *scientific model* of the work done by CONTAINS

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

Algorithm CONTAINS(L, v):

```
i, r := 0, false.                ⟵ 2 instruction(s).
while i ≠ |L| do                 ⟵ 2 instruction(s).
    if L[i] = v then             ⟵ 3 instruction(s).
        r := true.               ⟵ 1 instruction(s).
        i := i + 1.              ⟵ 2 instruction(s).
    else
        i := i + 1.              ⟵ 2 instruction(s).
return r.                        ⟵ 1 instruction(s).
```

We need a *scientific model* of the work done by CONTAINS

Intermezzo: The complexity of Contains

What is the complexity of Containsif $v \notin L$?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

Algorithm Contains(L, v):

$i, r := 0,$ `false`. $\longleftarrow$ *2 instruction(s).*

while $i \neq |L|$ **do** $\longleftarrow$ *2 instruction(s).* $|L| + 1$ *times.*

  if $L[i] = v$ **then** $\longleftarrow$ *3 instruction(s).*

    $r :=$ `true`. $\longleftarrow$ *1 instruction(s).*

    $i := i + 1.$ $\longleftarrow$ *2 instruction(s).* $|L|$ *times.*

  else

    $i := i + 1.$ $\longleftarrow$ *2 instruction(s).*

return r. $\longleftarrow$ *1 instruction(s).*

We need a *scientific model* of the work done by Contains

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

Algorithm CONTAINS(L, v):

$i, r := 0, \texttt{false}$. ⟵ *2 instruction(s).*
while $i \neq |L|$ **do** ⟵ *2 instruction(s).* } $|L| + 1$ *times.*
  if $L[i] = v$ **then** ⟵ *3 instruction(s).*
    $r := \texttt{true}$. ⟵ *1 instruction(s).*
    $i := i + 1$. ⟵ *2 instruction(s).* } $|L|$ *times.*
  else
    $i := i + 1$. ⟵ *2 instruction(s).*
return r. ⟵ *1 instruction(s).*

We need a *scientific model* of the work done by CONTAINS

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

Intermezzo: The complexity of Contains

What is the complexity of Contains?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

Algorithm Contains(L, v):

$i, r := 0, \texttt{false}.$ $\longleftarrow$ *2 instruction(s).*
while $i \neq |L|$ **do** $\longleftarrow$ *2 instruction(s).*
  if $L[i] = v$ **then** $\longleftarrow$ *3 instruction(s).*
    $r := \texttt{true}.$ $\longleftarrow$ *1 instruction(s).*
    $i := i + 1.$ $\longleftarrow$ *2 instruction(s).* } *m times.*
  else
    $i := i + 1.$ $\longleftarrow$ *2 instruction(s).*
return r. $\longleftarrow$ *1 instruction(s).*

We need a *scientific model* of the work done by Contains

$$\text{NumInstr}(N) = 5 + 7N + m \text{ with } N = |L|.$$

Intermezzo: The complexity of Contains

What is the complexity of Containsif $v \notin L$?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

We need a *scientific model* of the work done by Contains

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: Contains with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does Contains take with a list of 2000 values?

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

We need a *scientific model* of the work done by CONTAINS

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: CONTAINS with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does CONTAINS take with a list of 2000 values?

Argument

1. $\text{NumInstrOnlyElse}(1000) = 7005$ instructions $\longrightarrow$ 12 µs.
2. $\text{NumInstrOnlyElse}(2000) = 14\,005$ instructions $\longrightarrow$

$$\frac{14005}{7005} \cdot 12\,\mu\text{s} \approx 2 \cdot 12\,\mu\text{s} = 24\,\mu\text{s}.$$

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

We need a *scientific model* of the work done by CONTAINS

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: CONTAINS with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does CONTAINS take with a list of 2000 values? $\longrightarrow$ 24 µs.

Intermezzo: The complexity of Contains

What is the complexity of Containsif $v \notin L$?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

We need a *scientific model* of the work done by Contains

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: Contains with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does Contains take with a list of 2000 values? $\longrightarrow$ 24 µs.

Useful models are *simple* and make *correct* predictions

- Are our predictions correct?
- Is our model simple?

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

We need a *scientific model* of the work done by CONTAINS

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: CONTAINS with a list L, $|L| = 1000$, takes 12 µs.

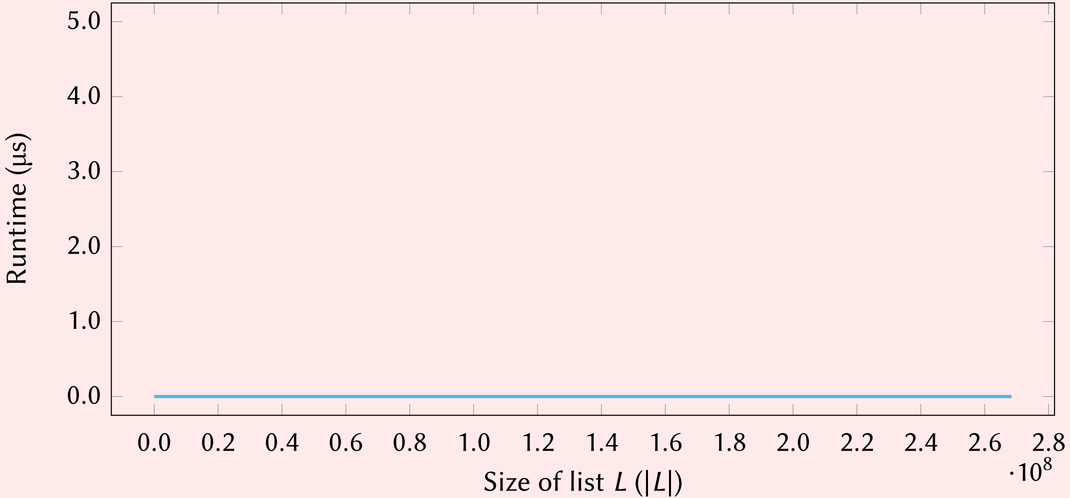
Predict: How long does CONTAINS take with a list of 2000 values? $\longrightarrow$ 24 µs.

Useful models are *simple* and make *correct* predictions

- Are our predictions correct? $\longrightarrow$ *Lets implement CONTAINS and measure.*
- Is our model simple?

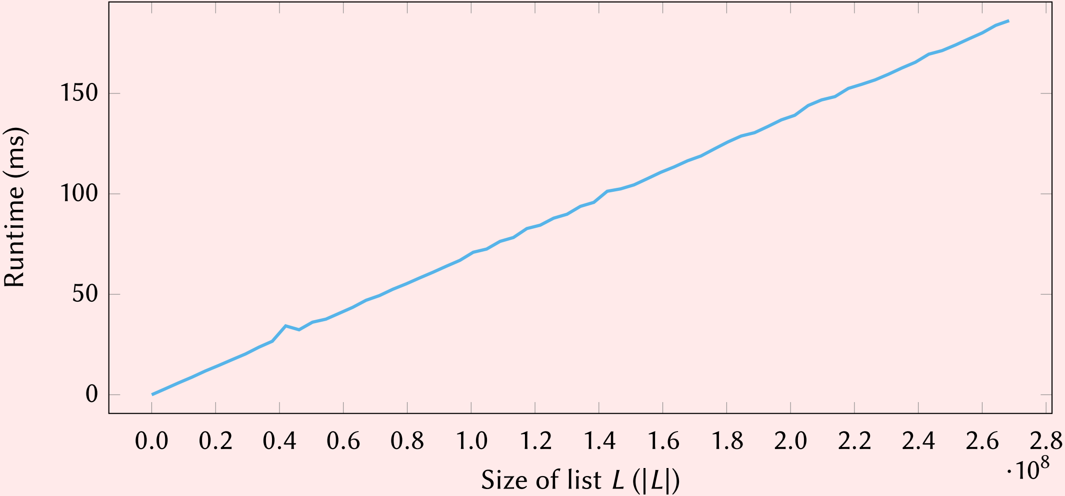
Intermezzo: The complexity of Contains

First Attempt



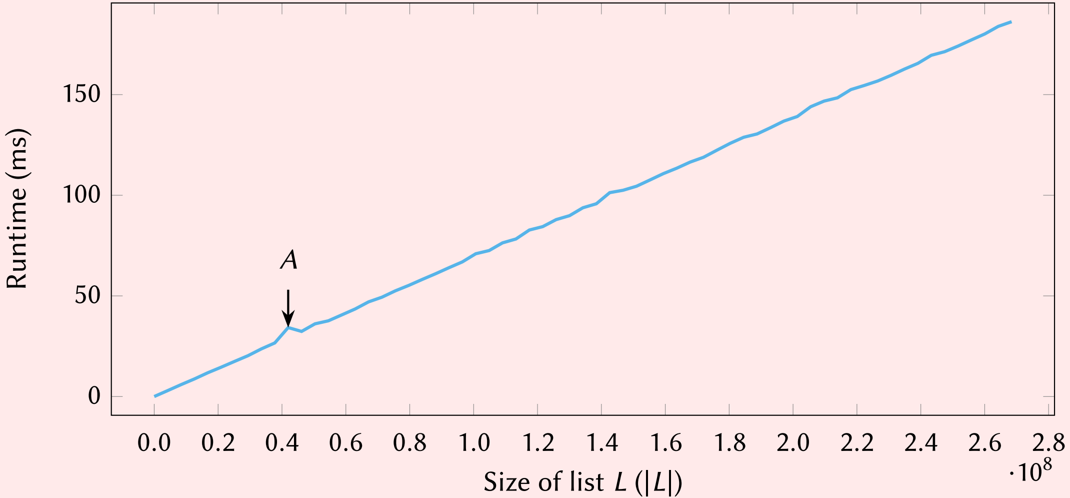
Intermezzo: The complexity of CONTAINS

Second Attempt



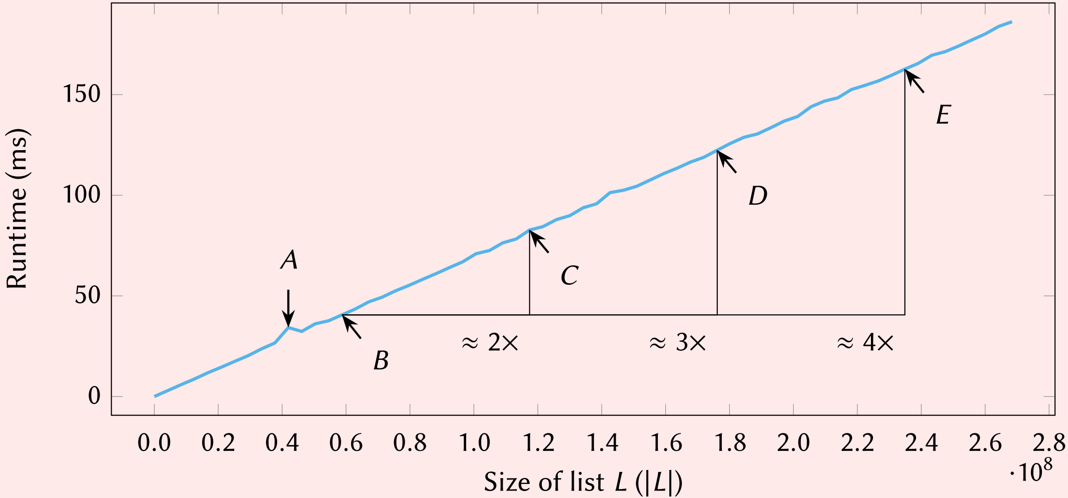
Intermezzo: The complexity of Contains

Second Attempt



Intermezzo: The complexity of CONTAINS

Second Attempt



Intermezzo: The complexity of Contains

What is the complexity of Containsif $v \notin L$?

Interested in *scalability*: How do the costs of Contains *increase* when increasing $|L|$?

We need a *scientific model* of the work done by Contains

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: Contains with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does Contains take with a list of 2000 values? ⟶ 24 µs.

Useful models are *simple* and make *correct* predictions

- Are our predictions correct? ⟶ *Yes.*
- Is our model simple? ⟶ *No: Runtime$(N) = N$ predicts the same!*

Intermezzo: The complexity of CONTAINS

What is the complexity of CONTAINSif $v \notin L$?

Interested in *scalability*: How do the costs of CONTAINS *increase* when increasing $|L|$?

We need a *scientific model* of the work done by CONTAINS

$$\text{NumInstrOnlyElse}(N) = 5 + 7N \text{ with } N = |L|.$$

A *scientific model* allows predictions

Assume: CONTAINS with a list L, $|L| = 1000$, takes 12 µs.

Predict: How long does CONTAINS take with a list of 2000 values? $\longrightarrow$ 24 µs.

Useful models are *simple* and make *correct* predictions

- Are our predictions correct? $\longrightarrow$ *Yes.*
- Is our model simple? $\longrightarrow$ *No: Runtime$(N) = N$ predicts the same!*
  Also: Our instruction counting is mostly fiction!

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0,$ `false`.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

Theorem

CONTAINS is correct, its runtime complexity is modelled by ContainsRuntime$(|L|) = |L|$, and its memory complexity is modelled by ContainsMemory$(|L|) = 1$.

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime($|L|$) $= |L|$.
- ALTC with AltCRuntime($|L|$) $= |L|^2$.

Which one is *faster*?

Can we conclude that CONTAINS is always fastest, ALTC is slowest?

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime$(|L|) = |L|$.
- ALTC with AltCRuntime$(|L|) = |L|^2$.

Which one is *faster*?

Input size	1000	
Runtime CONTAINS	12 µs	
Runtime ALTC	3 µs	
Speed up of ALTC	4×	

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime($|L|$) = $|L|$.
- ALTC with AltCRuntime($|L|$) = $|L|^2$.

Which one is *faster*?

Input size	1000	2000
Runtime CONTAINS	12 µs	24 µs
Runtime ALTC	3 µs	12 µs
Speed up of ALTC	4×	2×

Argument

- $\text{C.Runtime}(2000) = 2000 = 2 \cdot 1000 = 2 \cdot \text{C.Runtime}(1000)$.
- $\text{AltCRuntime}(2000) = 2000^2 = 2^2 \cdot 1000^2 = 2^2 \cdot \text{AltCRuntime}(1000)$.

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with $\text{C.Runtime}(|L|) = |L|$.
- ALTC with $\text{AltCRuntime}(|L|) = |L|^2$.

Which one is *faster*?

Input size	1000	2000	4000
Runtime CONTAINS	12 µs	24 µs	48 µs
Runtime ALTC	3 µs	12 µs	48 µs
Speed up of ALTC	4×	2×	1×

Argument

- $\text{C.Runtime}(4000) = 4000 = 4 \cdot 1000 = 4 \cdot \text{C.Runtime}(1000)$.
- $\text{AltCRuntime}(4000) = 4000^2 = 4^2 \cdot 1000^2 = 4^2 \cdot \text{AltCRuntime}(1000)$.

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with $\text{C.Runtime}(|L|) = |L|$.
- ALTC with $\text{AltCRuntime}(|L|) = |L|^2$.

Which one is *faster*?

Input size	1000	2000	4000	8000
Runtime CONTAINS	12 µs	24 µs	48 µs	96 µs
Runtime ALTC	3 µs	12 µs	48 µs	192 µs
Speed up of ALTC	4×	2×	1×	0.5×

Argument

- $\text{C.Runtime}(8000) = 8000 = 8 \cdot 1000 = 8 \cdot \text{C.Runtime}(1000)$.
- $\text{AltCRuntime}(8000) = 8000^2 = 8^2 \cdot 1000^2 = 8^2 \cdot \text{AltCRuntime}(1000)$.

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with $\text{C.Runtime}(|L|) = |L|$.
- ALTC with $\text{AltCRuntime}(|L|) = |L|^2$.

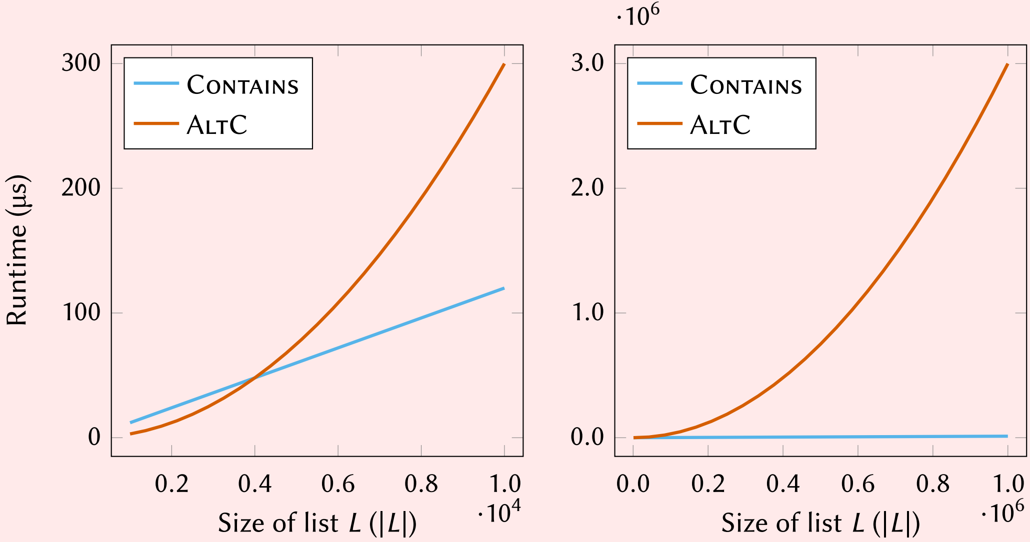
Which one is *faster*?

Input size	1000	2000	4000	8000	1 000 000
Runtime CONTAINS	12 µs	24 µs	48 µs	96 µs	12 ms
Runtime ALTC	3 µs	12 µs	48 µs	192 µs	3000 ms
Speed up of ALTC	4×	2×	1×	0.5×	0.004×

Argument

- $\text{C.Runtime}(1\,000\,000) = 1000000 = 1000 \cdot 1000 = 1000 \cdot \text{C.Runtime}(1000)$.
- $\text{AltCRuntime}(1\,000\,000) = 1000000^2 = 1000^2 \cdot 1000^2 = 1000^2 \cdot \text{AltCRuntime}(1000)$.

Comparing algorithm: Runtime complexity



Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime($|L|$) $= |L|$.
- ALTC with AltCRuntime($|L|$) $= |L|^2$.

Which one is *faster*?

Can we conclude that CONTAINS is always fastest, ALTC is slowest? $\longrightarrow$ *No!*

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- Contains with C.Runtime$(|L|) = |L|$.
- AltC with AltCRuntime$(|L|) = |L|^2$.

Which one is *faster*?

Can we conclude that Contains is always fastest, AltC is slowest? $\longrightarrow$ *No!*

Our models are simplifications!

Exact performance influenced by details of the compiler, memory, CPU architecture, ...

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime$(|L|) = |L|$.
- ALTC with AltCRuntime$(|L|) = |L|^2$.

Which one is *faster*?

Can we conclude that CONTAINS is always fastest, ALTC is slowest? $\longrightarrow$ *No!*

Our models are simplifications!

Exact performance influenced by details of the compiler, memory, CPU architecture, ...

Are our models meaningless?

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime($|L|$) = $|L|$.
- ALTC with AltCRuntime($|L|$) = $|L|^2$.

Which one is *faster*?

Can we conclude that CONTAINS is always fastest, ALTC is slowest? $\longrightarrow$ *No!*

Our models are simplifications!

Exact performance influenced by details of the compiler, memory, CPU architecture, ...

Are our models meaningless?

No: our comparisons shows *differences in growth rates*: $|L|$ versus $|L|^2$ $\longrightarrow$
for large-enough inputs, ALTC should always be *much slower* than CONTAINS.

Comparing algorithm: Runtime complexity

Say we have two algorithms for the *contains* problem

- CONTAINS with C.Runtime($|L|$) = $|L|$.
- ALTC with AltCRuntime($|L|$) = $|L|^2$.

Which one is *faster*?

Can we conclude that CONTAINS is always fastest, ALTC is slowest? $\longrightarrow$ *No!*

Our models are simplifications!

Exact performance influenced by details of the compiler, memory, CPU architecture, ...

Remember: We are interested in *scalability* of algorithms

For large-enough inputs, CONTAINS will always be much faster than ALTC *because* the order of growth of C.Runtime is *lower* than the order of growth of AltCRuntime.

Comparing algorithm: Runtime complexity

Remember: We are interested in *scalability* of algorithms

For large-enough inputs, CONTAINS will always be much faster than ALTC *because* the order of growth of C.Runtime is *lower* than the order of growth of AltCRuntime.

Runtime complexity (size of input: N)		Which is faster? (for large-enough N)
ALGORITHM 1	ALGORITHM 2	
$5 + 7N$	$3N + 100$	
$5 + 7N$	$100\log_2(N) + 2$	
$5 + 7N$	$N(N-1)/2$	
$5 + 7N$	$1000N^{\frac{1}{2}} - 120$	
$2N^3 + 1000$	$2^N - 1$	

Comparing algorithm: Runtime complexity

Remember: We are interested in *scalability* of algorithms

For large-enough inputs, CONTAINS will always be much faster than ALTC *because* the order of growth of C.Runtime is *lower* than the order of growth of AltCRuntime.

Runtime complexity (size of input: N)		Which is faster? (for large-enough N)
ALGORITHM 1	ALGORITHM 2	
$5 + 7N$	$3N + 100$	Similar
$5 + 7N$	$100 \log_2(N) + 2$	Algorithm 2
$5 + 7N$	$N(N-1)/2$	Algorithm 1
$5 + 7N$	$1000N^{\frac{1}{2}} - 120$	Algorithm 2
$2N^3 + 1000$	$2^N - 1$	Algorithm 1

Comparing algorithm: Runtime complexity

Remember: We are interested in *scalability* of algorithms

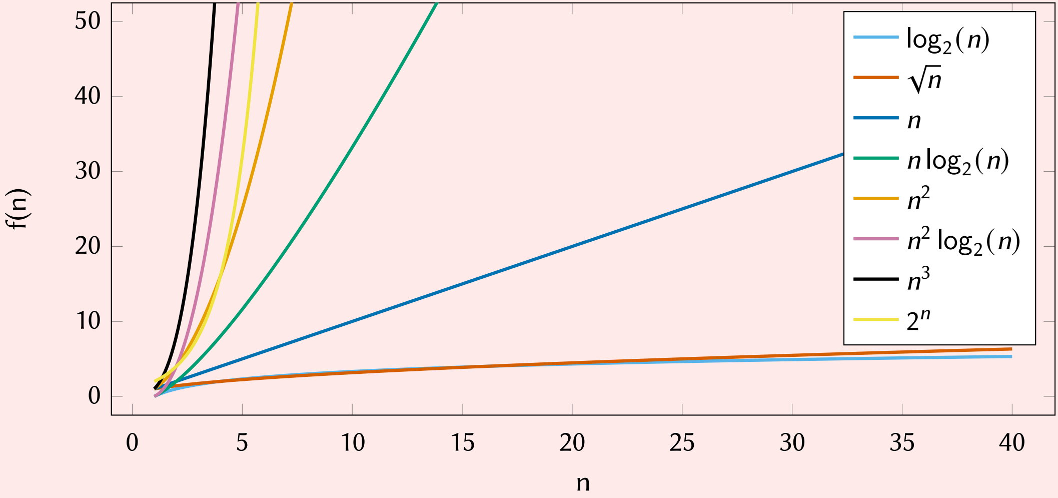
For large-enough inputs, Contains will always be much faster than AltC *because* the order of growth of C.Runtime is *lower* than the order of growth of AltCRuntime.

Runtime complexity (size of input: N)		Which is faster? (for large-enough N)
Algorithm 1	Algorithm 2	
N	N	Similar
N	$\ln(N)$	Algorithm 2
N	N^2	Algorithm 1
N	$\sqrt{N}$	Algorithm 2
N^3	2^N	Algorithm 1

Simpler models are easier to compare!

Comparing algorithm: Runtime complexity

Some very common functions $f(n)$—(increasing order of growth)



Comparing functions: Order of growth

Definition (informal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ denotes f "scales better" than $g(n)$.
   The order of growth of f is *upper bounded* by g: any increase in the runtime predicted by f as a consequence of increasing n is *at-most* the increase predicted by $g(n)$.
2. $f(n) = \Omega(g(n))$ denotes f "scales worse" than $g(n)$.
   The order of growth of f is *lower bounded* by g: any increase in the runtime predicted by f as a consequence of increasing n is *at-least* the increase predicted by $g(n)$.
3. $f(n) = \Theta(g(n))$ denotes f "scales the same" as $g(n)$.
   The order of growth of f is *equivalent* to g: any increase in the runtime predicted by f as a consequence of increasing n is *equivalent to* the increase predicted by $g(n)$.
   In this case, we also say that $f(n)$ is *strictly bounded by* $g(n)$.

The book uses the notation $f(n) {\sim} (g(n))$ instead of $f(n) = \Theta(g(n))$.

Comparing functions: Order of growth

Definition (formal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

2. $f(n) = \Omega(g(n))$ if there exists constants $n_0, c > 0$ such that,

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0.$$

3. $f(n) = \Theta(g(n))$ if there exists constants $n_0, c_{\text{lb}}, c_{\text{ub}} > 0$ such that,

$$0 \leq c_{\text{lb}} \cdot g(n) \leq f(n) \leq c_{\text{ub}} \cdot g(n) \text{ for all } n \geq n_0.$$

Comparing functions: Order of growth

Definition (formal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

Constants n_0? c?

Comparing functions: Order of growth

Definition (formal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

Constants n_0? c?

- *Constant n_0* allows us to *only* look at large inputs (larger than n_0).
  Example, $n^2 > n$ *only when* inputs are large enough!

Comparing functions: Order of growth

Definition (formal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

Constants n_0? c?

- *Constant n_0* allows us to *only* look at large inputs (larger than n_0).
  Example, $n^2 > n$ *only when* inputs are large enough!
- *Constant c* hides “irrelevant details”.
  Example, $3 + 7 \cdot n$ and n *model* the same behavior!

Comparing functions: Order of growth

Definition (formal)

Let f and g be functions of size of input n:

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

Show that $3 + 7 \cdot n = \mathcal{O}(n)$

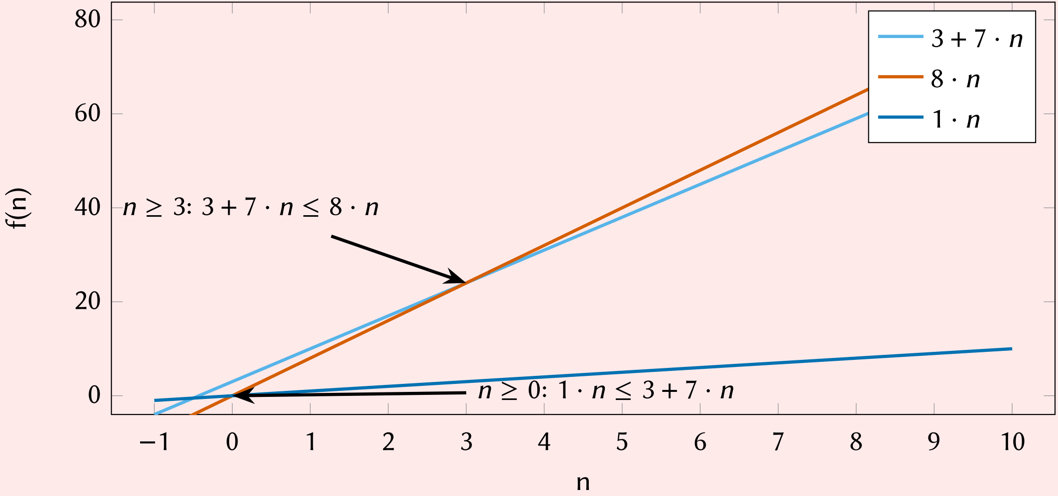
- $3 + 7 \cdot n = \mathcal{O}(n)$. Choose $n_0 = 3$ and $c = 8$. The statement

$$\text{for all } n \geq 3,\ 0 \leq 3 + 7 \cdot n \leq 8 \cdot n$$

is true, completing the proof.

Comparing functions: Order of growth

Show that $3 + 7 \cdot n = \Theta(n)$



Comparing functions: Order of growth

Theorem

- *The runtime complexity of* CONTAINS *is* $\Theta(|L|)$.
- *The memory complexity of* CONTAINS *is* $\Theta(1)$.

How to compare the order of growth of functions?

How to compare the order of growth of functions?

Limits: A mathematical power tool

Let f and g be functions of n with non-negative ranges. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

How to compare the order of growth of functions?

Limits: A mathematical power tool

Let f and g be functions of n with non-negative ranges. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

Example

$$\lim_{n\to\infty} \frac{c \cdot f(n)}{f(n)} = c \cdot \left(\lim_{n\to\infty} \frac{f(n)}{f(n)}\right) = c$$

How to compare the order of growth of functions?

Limits: A mathematical power tool

Let f and g be functions of n with non-negative ranges. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

Example

$$\lim_{n\to\infty} \frac{c \cdot f(n)}{f(n)} = c \cdot \left(\lim_{n\to\infty} \frac{f(n)}{f(n)}\right) = c \qquad \longrightarrow \boxed{c \cdot f(n) = \Theta(f(n))}$$

How to compare the order of growth of functions?

Limits: A mathematical power tool

Let f and g be functions of n with non-negative ranges. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

Example

$$\lim_{n\to\infty} \frac{c \cdot f(n)}{f(n)} = c \cdot \left(\lim_{n\to\infty} \frac{f(n)}{f(n)} \right) = c \qquad \longrightarrow \quad c \cdot f(n) = \Theta(f(n))$$

$$\lim_{n\to\infty} \frac{n^c}{n^{c+d}} = \lim_{n\to\infty} \frac{1}{n^d} = 0 \qquad \longrightarrow \quad n^c = \mathcal{O}(n^{c+d})$$

How to compare the order of growth of functions?

Limits: A mathematical power tool

Let f and g be functions of n with non-negative ranges. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

Example

$$\lim_{n\to\infty} \frac{c \cdot f(n)}{f(n)} = c \cdot \left(\lim_{n\to\infty} \frac{f(n)}{f(n)}\right) = c \qquad \longrightarrow \boxed{c \cdot f(n) = \Theta(f(n))}$$

$$\lim_{n\to\infty} \frac{n^c}{n^{c+d}} = \lim_{n\to\infty} \frac{1}{n^d} = 0 \qquad \longrightarrow \boxed{n^c = \mathcal{O}(n^{c+d})}$$

$$\lim_{n\to\infty} \frac{n^c}{d^n} = 0 \qquad \longrightarrow \boxed{n^c = \mathcal{O}(d^n)}$$

How to compare the order of growth of functions?

Example (See Example 3.26 in the course notes for details)

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = \frac{1}{\log_b(a)} \cdot \log_b(n) \longrightarrow \boxed{\log_a(n) = \Theta(\log_b(n))}$$

$$\lim_{n\to\infty} \frac{\log_2(n)^c}{n^d} = 0 \longrightarrow \boxed{\log_2(n)^c = \mathcal{O}(n^d)}$$

$$\lim_{n\to\infty} \frac{d^{n/u}}{c^{n/v}} = 0 \text{ (if } c \geq d \geq 1,\ u \geq v \geq 1) \longrightarrow \boxed{d^{n/u} = \mathcal{O}(c^{n/v})}$$

$$\lim_{n\to\infty} \frac{c_1 n^{d_1} + \cdots + c_m n^{d_m}}{n^{d_i}} = c_i \ (d_i = \max(d_1, \ldots, d_m)) \longrightarrow \boxed{c_1 n^{d_1} + \cdots + c_m n^{d_m} = \Theta(n^{d_i})}$$

$$\lim_{n\to\infty} \frac{f(n) + g(n)}{g(n)} = 1 \text{ (if } f(n) = \mathcal{O}(g(n))) \longrightarrow \boxed{f(n) + g(n) = \Theta(g(n))}$$

$$\lim_{n\to\infty} \frac{h(n) \cdot f(n)}{h(n) \cdot g(n)} = 0 \text{ (if } f(n) = \mathcal{O}(g(n))) \longrightarrow \boxed{h(n) \cdot f(n) = \mathcal{O}(h(n) \cdot g(n))}$$

Reflection on CONTAINS

Is CONTAINS a *good* algorithm?

CONTAINS is correct and has a runtime complexity of $\Theta(|L|) \longrightarrow$ Sounds good to me!

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

1: $r := o$.
/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
2: **while** $r \neq |L|$ **and also** $L[r] \neq v$ **do**
3: $r := r + 1$.
4: **return** r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

Reflection on CONTAINS

Is CONTAINS a *good* algorithm?

CONTAINS is correct and has a runtime complexity of $\Theta(|L|) \longrightarrow$ Sounds good to me!

Critique: CONTAINS is *too specialized* $\longrightarrow$.

We cannot use CONTAINS for anything else than the contains problem!

Example

- Searching in only *part* of the list?
- Finding where v is in the list?

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

1: $r := o$.

/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */

2: **while** $r \neq |L|$ **and also** $L[r] \neq v$ **do**

Reflection on CONTAINS

Critique: CONTAINS is *too specialized* $\longrightarrow$.

We cannot use CONTAINS for anything else than the contains problem!

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

1: $r := o$.
/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
2: **while** $r \neq |L|$ **and also** $L[r] \neq v$ **do**
3: $\quad r := r + 1$.
4: **return** r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

Reflection on CONTAINS

Critique: CONTAINS is *too specialized* $\longrightarrow$.

We cannot use CONTAINS for anything else than the contains problem!

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

1: $r := o$.
/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
2: **while** $r \neq |L|$ **and also** $L[r] \neq v$ **do**
3: $r := r + 1$.
4: **return** r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

Algorithm LSCONTAINS(L, v):

1: **return** LINEARSEARCH$(L, v, 0) \neq |L|$.

Reflection on Contains

Algorithm LinearSearch(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

$r := o$.
/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
while $r \neq |L|$ **and also** $L[r] \neq v$ **do**
    $r := r + 1$.
return r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

What is the runtime complexity of LinearSearch?

Reflection on CONTAINS

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

```
r := o.
/* invariant: "o ≤ r ≤ |L| and v ∉ L[o, r)", bound function: |L| − r */
while r ≠ |L| and also L[r] ≠ v do
    r := r + 1.
return r.
```

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or,
if no such offset exists, $r = |L|$.

What is the runtime complexity of LINEARSEARCH?

- With respect to worst-case inputs ($v \notin L$): $\Theta(|L|)$.

Reflection on CONTAINS

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

$r := o$.
/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
while $r \neq |L|$ **and also** $L[r] \neq v$ **do**
    $r := r + 1$.
return r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or,
if no such offset exists, $r = |L|$.

What is the runtime complexity of LINEARSEARCH?

- With respect to worst-case inputs ($v \notin L$): $\Theta(|L|)$.
- With respect to best-case inputs ($v = L[o]$): $\Theta(1)$.

Reflection on CONTAINS

Algorithm LINEARSEARCH(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

$r := o$.

/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */

while $r \neq |L|$ **and also** $L[r] \neq v$ **do**

  $r := r + 1$.

return r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or,
if no such offset exists, $r = |L|$.

What is the runtime complexity of LINEARSEARCH?

- With respect to worst-case inputs ($v \notin L$): $\Theta(|L|)$.
- With respect to best-case inputs ($v = L[o]$): $\Theta(1)$.

Problem: Modeling runtime complexity in terms of *only the input* limits us!

Reflection on Contains

Algorithm LinearSearch(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

$r := o$.

/* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */

while $r \neq |L|$ **and also** $L[r] \neq v$ **do**

  $r := r + 1$.

return r.

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

What is the runtime complexity of LinearSearch?

Problem: Modeling runtime complexity in terms of *only the input* limits us!

Assume: $L[i] = v$ and i is the *first* offset after o equivalent to v.
The runtime complexity of LinearSearch is $\Theta(i - o)$.

Reflection on Contains

Algorithm LinearSearch(L, v, o):

Input: L is an *array*, v a value, $0 \leq o \leq |L|$.

```
r := o.
/* invariant: "o ≤ r ≤ |L| and v ∉ L[o, r)", bound function: |L| − r */
while r ≠ |L| and also L[r] ≠ v do
    r := r + 1.
return r.
```

Result: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or,
if no such offset exists, $r = |L|$.

What is the runtime complexity of LinearSearch?

Problem: Modeling runtime complexity in terms of *only the input* limits us!

Assume: $L[i] = v$ and i is the *first* offset after o equivalent to v.
The runtime complexity of LinearSearch is $\Theta(i - o)$ with $i =$ LinearSearch(L, v, o).