Fundamentals

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Problem: What if we search often?

LinearSearch(L, v, o) can read all of array L: *potentially-high cost*.

Can we do better?

Problem: What if we search often?

$\textsc{LinearSearch}(L, v, o)$ can read all of array L: *potentially-high cost*.

Can we do better?

No: We do not know anything about L to help us!

$\longrightarrow$ we have to look at all elements in L.

Problem: What if we search often?

LinearSearch(L, v, o) can read all of array L: *potentially-high cost*.

Can we do better?

No: We do not know anything about L to help us!

$\longrightarrow$ we have to look at all elements in L.

Maybe: If we know more about L.

An example of a list

Consider a list *enrolled* with schema

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

that models a list of all students enrolled for a course.

What if…

We add enrollment data to *the end of the list*.

Question: What do we know about enrolled?

An example of a list

Consider a list *enrolled* with schema

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

that models a list of all students enrolled for a course.

What if...

We add enrollment data to *the end of the list*.

Question: What do we know about enrolled? $\longrightarrow$ enrolled is *ordered* on *date*!

Searching in an ordered list

3	14	16	18	30	37	44	49	66	76	84	90

v

Conclusion of comparing $L[i]$ and v?

Searching in an ordered list

3	14	16	18	30	37	44	49	66	76	84	90

$\uparrow$

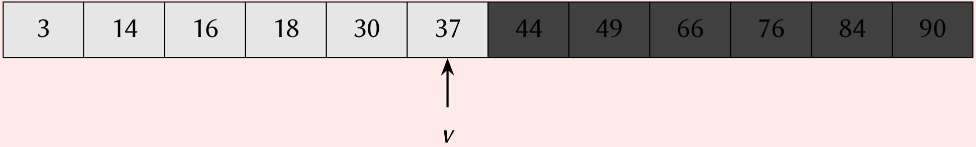
v

Conclusion of comparing $L[i]$ and v?

$L[i] < v$ As the list L is ordered, every value in $L[0, i]$ is smaller than v.

$\longrightarrow v \in L$ if and only if $v \in L[i+1, |L|)$.

Searching in an ordered list



Conclusion of comparing $L[i]$ and v?

$L[i] < v$ As the list L is ordered, every value in $L[0, i]$ is smaller than v.
$\longrightarrow v \in L$ if and only if $v \in L[i+1, |L|)$.

$L[i] > v$ As the list L is ordered, every value in $L[i, |L|)$ is larger than v.
$\longrightarrow v \in L$ if and only if $v \in L[0, i)$.

Searching in an ordered list

3	14	16	18	30	37	44	49	66	76	84	90

v

Conclusion of comparing $L[i]$ and v?

$L[i] < v$ As the list L is ordered, every value in $L[0, i]$ is smaller than v.
$\longrightarrow v \in L$ if and only if $v \in L[i + 1, |L|)$.

$L[i] > v$ As the list L is ordered, every value in $L[i, |L|)$ is larger than v.
$\longrightarrow v \in L$ if and only if $v \in L[0, i)$.

$L[i] = v$ We found v!

Searching in an ordered list

3	14	16	18	30	37	44	49	66	76	84	90

v

Conclusion of comparing $L[i]$ and v?

$L[i] < v$ As the list L is ordered, every value in $L[0, i]$ is smaller than v.
$\longrightarrow v \in L$ if and only if $v \in L[i+1, |L|)$.

$L[i] > v$ As the list L is ordered, every value in $L[i, |L|)$ is larger than v.
$\longrightarrow v \in L$ if and only if $v \in L[0, i)$.

$L[i] = v$ We found v!

One comparison can remove a large portion of the array.

Searching in an ordered list

3	14	16	18	30	37	44	49	66	76	84	90

v

Conclusion of comparing $L[i]$ and v?

$L[i] < v$ As the list L is ordered, every value in $L[0, i]$ is smaller than v.
$\longrightarrow v \in L$ if and only if $v \in L[i+1, |L|)$.

$L[i] > v$ As the list L is ordered, every value in $L[i, |L|)$ is larger than v.
$\longrightarrow v \in L$ if and only if $v \in L[0, i)$.

$L[i] = v$ We found v!

One comparison can remove a large portion of the array.

Binary Search: *Maximize potential* by comparing v with the middle of L.

The *recursive* Binary Search algorithm

Algorithm LowerBoundRec(L, v, $begin$, end):

Input: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.

```
if begin = end then
    return begin.
else
    mid := (begin + end) div 2.
    if L[mid] < v then
        return LowerBoundRec(L, v, mid + 1, end).
    else /* L[mid] ≥ v */
        return LowerBoundRec(L, v, begin, mid).
```

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

- Is LowerBoundRec correct?
- What is the runtime and memory complexity of LowerBoundRec?

Correctness of LowerBoundRec

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
    return $begin$.
else
    $mid := (begin + end) \operatorname{div} 2$.
    if $L[mid] < v$ **then**
        return LowerBoundRec($L, v, mid + 1, end$).
    else /* $L[mid] \geq v$ */
        return LowerBoundRec($L, v, begin, mid$).

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

```
Algorithm LowerBoundRec(L, v, begin, end):
  if begin = end then
      return begin.
  else
      mid := (begin + end) div 2.
      if L[mid] < v then
          return LowerBoundRec(L, v, mid + 1, end).
      else /* L[mid] ≥ v */
          return LowerBoundRec(L, v, begin, mid).
```

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
    return $begin$.
else
    $mid := (begin + end)$ div 2.
    if $L[mid] < v$ **then**
        return LowerBoundRec($L, v, mid + 1, end$).
    else /* $L[mid] \geq v$ */
        return LowerBoundRec($L, v, begin, mid$).

Induction Hypothesis

For any L', v', and $0 \leq begin' \leq end' \leq |L'|$ with $0 \leq end' - begin' < m$, LowerBoundRec(L', v', $begin'$, end') returns the correct result.

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
  return $begin$.
else
  $mid := (begin + end)$ div 2.
  if $L[mid] < v$ **then**
    return LowerBoundRec($L, v, mid + 1, end$).
  else /* $L[mid] \geq v$ */
    return LowerBoundRec($L, v, begin, mid$).

Base case:
Inspecting $end - begin = 0$ elements.

Recursive case:
Inspecting $end - begin > 0$ elements.

Induction Hypothesis

For any L', v', and $0 \leq begin' \leq end' \leq |L'|$ with $0 \leq end' - begin' < m$,
LowerBoundRec(L', v', $begin'$, end') returns the correct result.

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
    return $begin$.
else
    $mid := (begin + end) \text{ div } 2$.
    if $L[mid] < v$ **then**
        return LowerBoundRec$(L, v, mid + 1, end)$.
    else /* $L[mid] \geq v$ */
        return LowerBoundRec$(L, v, begin, mid)$.

$begin \leq mid < end$

Induction Hypothesis

For any L', v', and $0 \leq begin' \leq end' \leq |L'|$ with $0 \leq end' - begin' < m$, LowerBoundRec(L', v', $begin'$, end') returns the correct result.

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

Algorithm LowerBoundRec(L, v, $begin$, end):

```
if begin = end then
    return begin.
else
    mid := (begin + end) div 2.
    if L[mid] < v then
        return LowerBoundRec(L, v, mid + 1, end).
    else /* L[mid] ≥ v */
        return LowerBoundRec(L, v, begin, mid).
```

$begin \leq mid < end$

$\text{LowerBoundRec}(L, v, \overbrace{mid + 1, end}^{\text{Induction Hypothesis}})$

$\text{LowerBoundRec}(L, v, \underbrace{begin, mid}_{\text{Induction Hypothesis}})$

Induction Hypothesis

For any L', v', and $0 \leq begin' \leq end' \leq |L'|$ with $0 \leq end' - begin' < m$, LowerBoundRec(L', v', $begin'$, end') returns the correct result.

Correctness of LowerBoundRec

Recursion is repetition $\longrightarrow$ induction.

Algorithm LowerBoundRec(L, v, *begin*, *end*):

if $\mathit{begin} = \mathit{end}$ **then**
    return *begin*.
else
    $\mathit{mid} := (\mathit{begin} + \mathit{end})$ div 2.
    if $L[\mathit{mid}] < v$ **then**
        return LowerBoundRec($L, v, \mathit{mid} + 1, \mathit{end}$).
    else /* $L[\mathit{mid}] \geq v$ */
        return LowerBoundRec($L, v, \mathit{begin}, \mathit{mid}$).

Termination

Bound function: $\mathit{end} - \mathit{begin}$.

Intermezzo: Runtime complexity of LowerBoundRec

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
  return $begin$.
else
  $mid := (begin + end)$ div 2.
  if $L[mid] < v$ **then**
    return LowerBoundRec$(L, v, mid + 1, end)$.
  else /* $L[mid] \geq v$ */
    return LowerBoundRec$(L, v, begin, mid)$.

Complexity of LowerBoundRec with $n = end - begin$

$$T(n) =$$

Intermezzo: Runtime complexity of LowerBoundRec

Algorithm $\textsc{LowerBoundRec}(L, v, \mathit{begin}, \mathit{end})$:

```
if begin = end then
    return begin.
else
    mid := (begin + end) div 2.
    if L[mid] < v then
        return LowerBoundRec(L, v, mid + 1, end).
    else /* L[mid] ≥ v */
        return LowerBoundRec(L, v, begin, mid).
```

Base case:
1 operation.

Complexity of LowerBoundRec with $n = \mathit{end} - \mathit{begin}$

$$T(n) = \begin{cases} 1 & \text{if } n = 0; \\ & \end{cases}$$

Intermezzo: Runtime complexity of LowerBoundRec

Algorithm LowerBoundRec(L, v, $begin$, end):

```
if begin = end then
    return begin.
else
    mid := (begin + end) div 2.
    if L[mid] < v then
        return LowerBoundRec(L, v, mid + 1, end).
    else /* L[mid] ≥ v */
        return LowerBoundRec(L, v, begin, mid).
```

Base case: *1 operation.*

Recursive case: *1 operation and 1 recursive call.*

Complexity of LowerBoundRec with $n = end - begin$

$$T(n) = \begin{cases} 1 & \text{if } n = 0; \\ 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 & \text{if } n \geq 1. \end{cases}$$

Intermezzo: Runtime complexity of LowerBoundRec

Algorithm LowerBoundRec(L, v, $begin$, end):

if $begin = end$ **then**
  return $begin$.
else
  $mid := (begin + end)$ div 2.
  if $L[mid] < v$ **then**
    return LowerBoundRec(L, v, $mid + 1$, end).
  else /* $L[mid] \geq v$ */
    return LowerBoundRec(L, v, $begin$, mid).

Complexity of LowerBoundRec with $n = end - begin$

$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = end - begin$ (assume: $n = 2^x$)

$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

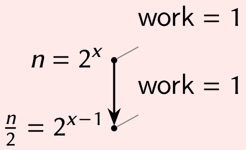
work = 1

$n = 2^x$ •

Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = end - begin$ (assume: $n = 2^x$)

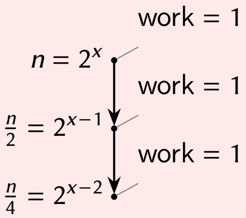
$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$



Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = end - begin$ (assume: $n = 2^x$)

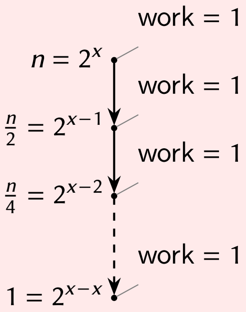
$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$



Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = end - begin$ (assume: $n = 2^x$)

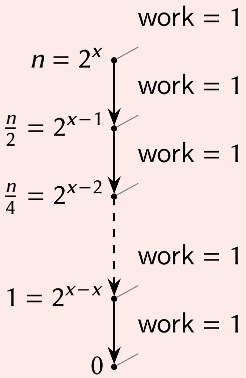
$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$



Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = end - begin$ (assume: $n = 2^x$)

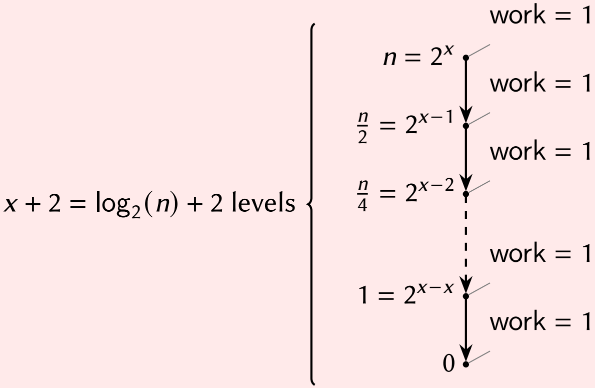
$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$



Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = \mathit{end} - \mathit{begin}$ (assume: $n = 2^x$)

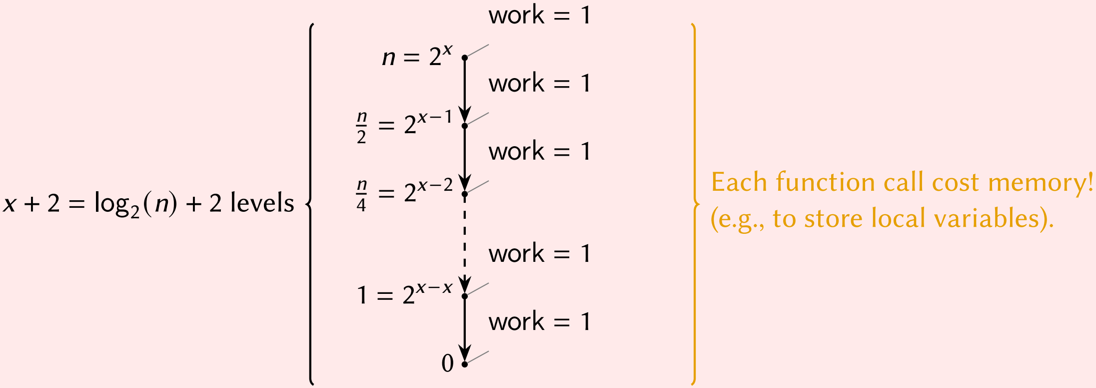
$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$



Intermezzo: Runtime complexity of LowerBoundRec

Complexity of LowerBoundRec with $n = \mathit{end} - \mathit{begin}$ (assume: $n = 2^x$)

$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 = \Theta(\log_2(n)).$$



The *recursive* Binary Search algorithm

Algorithm $\text{LowerBoundRec}(L, v, \mathit{begin}, \mathit{end})$:

Input: L is an ordered *array*, v a value, and $0 \leq \mathit{begin} \leq \mathit{end} \leq |L|$.

```
if begin = end then
    return begin.
else
    mid := (begin + end) div 2.
    if L[mid] < v then
        return LowerBoundRec(L, v, mid + 1, end).
    else /* L[mid] ≥ v */
        return LowerBoundRec(L, v, begin, mid).
```

Result: return the first offset r, $\mathit{begin} \leq r < \mathit{end}$, with $L[r] = v$ or,
if no such offset exists, $r = \mathit{end}$.

Theorem

LowerBoundRec is correct and has a runtime and memory complexity of $\Theta(\log_2(|L|))$.

The *non-recursive* Binary Search algorithm

Algorithm LowerBound(L, v, $begin$, end):

Input: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.

```
while begin ≠ end do
    mid := (begin + end) div 2.
    if L[mid] < v then
        begin := mid + 1.
    else
        end := mid.
return begin.
```

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

The *non-recursive* Binary Search algorithm

Algorithm LowerBound(L, v, $begin$, end):

Input: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.

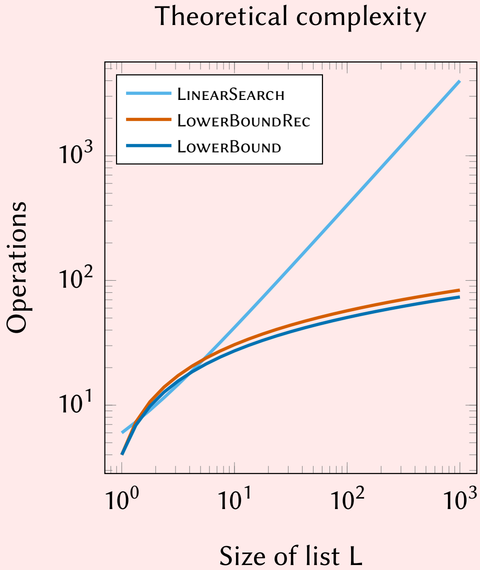
```
while begin ≠ end do
    mid := (begin + end) div 2.
    if L[mid] < v then
        begin := mid + 1.
    else
        end := mid.
return begin.
```

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

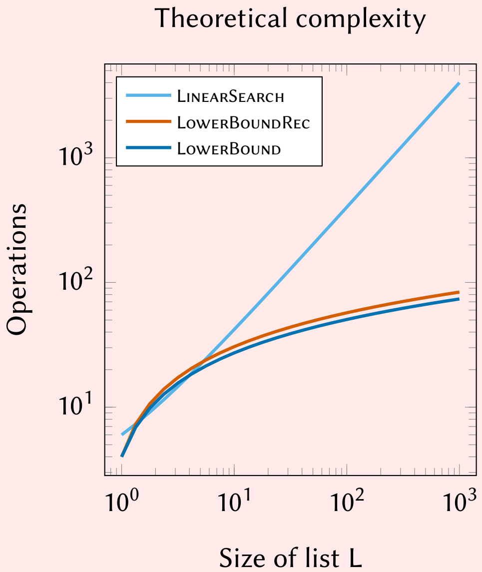
Theorem

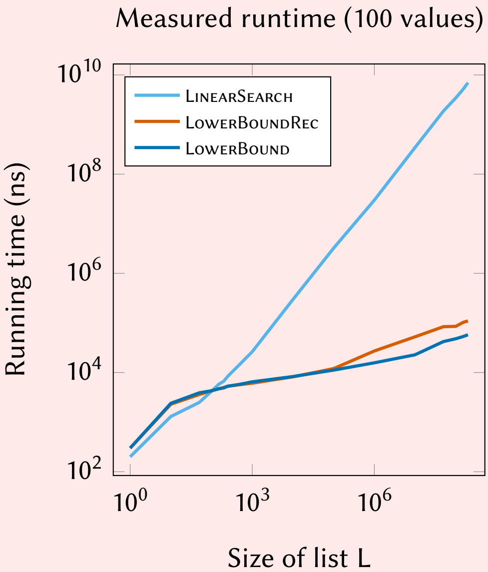
LowerBound is correct, has a runtime complexity of $\Theta(\log_2(|L|))$, and a memory complexity of $\Theta(1)$.

Comparing the complexity of searching

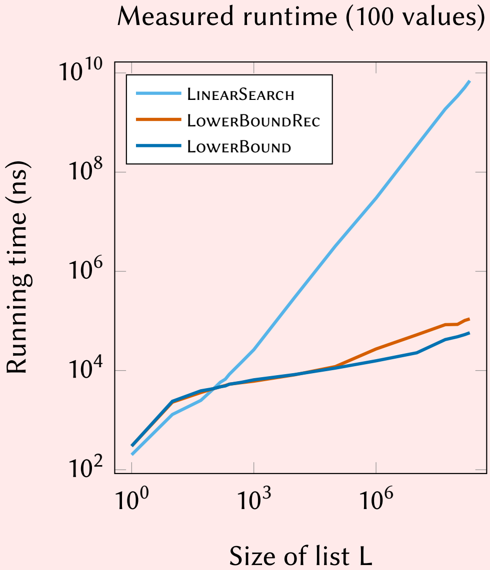


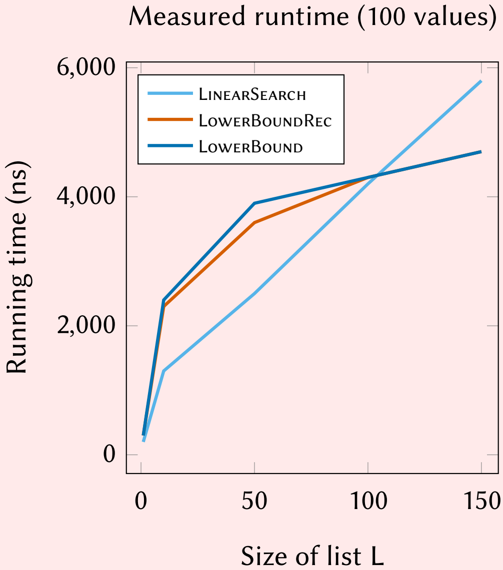
Comparing the complexity of searching





Comparing the complexity of searching





Using Binary Search as a *building block*

Problem

Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the range query problem for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Example

Consider a list *enrolled* with schema enrolled(*dept*, *code*, *sid*, *date*).

Query: All students enrolled in 2023

Range query on *enrolled* with $[(\text{`'}, \text{`'}, -1, 2023), (\text{`'}, \text{`'}, -1, 2024)]$.

Using Binary Search as a *building block*

Problem

Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the range query problem for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Example

Consider a list *enrolled* with schema enrolled(*dept*, *code*, *sid*, *date*).

Query: All students enrolled in 2023

Range query on *enrolled* with $[(``, ``, -1, 2023), (``, ``, -1, 2024)]$.

We add enrollment data to *the end of the list* $\longrightarrow$ enrolled is *ordered* on *date*!

Using Binary Search as a *building block*

Problem

Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the range query problem for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Algorithm RangeQuery$L$, $[v, w]$:

Input: L is an ordered *array*, v, w are values, and $v \leq w$.

```
i := LowerBound(L, v, 0, |L|).
j := i.
while j ≤ |L| and also L[j] ≤ w do
    j := j + 1.
return L[i, j).
```

Result: return the list $L[m, n)$, $0 \leq m \leq n \leq |L|$, such that $L[m, n)$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Using Binary Search as a *building block*

Problem

Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the range query problem for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Algorithm RangeQuery$L$, $[v, w]$**:**

Input: L is an ordered *array*, v, w are values, and $v \leq w$.

$i := \textsc{LowerBound}(L, v, 0, |L|)$.
$j := i$.
while $j \leq |L|$ **and also** $L[j] \leq w$ **do**
    $j := j + 1$.
return $L[i, j)$.

Result: return the list $L[m, n)$, $0 \leq m \leq n \leq |L|$, such that
$L[m, n)$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Theorem

RangeQuery is correct and has worst case runtime complexity $\Theta(|L|)$.

Using Binary Search as a *building block*

Problem

Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the range query problem for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Algorithm $\textsc{RangeQuery}L, [v, w]$**:**

Input: L is an ordered *array*, v, w are values, and $v \leq w$.

```
i := LowerBound(L, v, 0, |L|).
j := i.
while j ≤ |L| and also L[j] ≤ w do
    j := j + 1.
return L[i, j).
```

Result: return the list $L[m, n)$, $0 \leq m \leq n \leq |L|$, such that $L[m, n)$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Theorem

$\textsc{RangeQuery}$ is correct and has all case runtime complexity $\Theta(\log_2(|L|) + |result|)$.

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function INSPECT(L, i) *that returns* `true` *if the list has an* i*-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length *and assume we have a function* INSPECT(*L*, *i*) *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem *is the problem of finding the length of list L.*

Example

'apple'	'pear'	'orange'

$$\textsc{Inspect}(L, 0) = \texttt{true} \qquad \textsc{Inspect}(L, 1) = \texttt{true}$$
$$\textsc{Inspect}(L, 2) = \texttt{true} \qquad \textsc{Inspect}(L, 3) = \texttt{false}.$$

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function $\textsc{Inspect}(L, i)$ *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Solving the list-length problem

We have ordered list $0, 1, \ldots$ of possible values for $|L|$.

Conclusion of $\textsc{Inspect}(L, i)$?

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function INSPECT(*L*, *i*) *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Solving the list-length problem

We have ordered list $0, 1, \ldots$ of possible values for $|L|$.

Conclusion of $\textsc{Inspect}(L, i)$?

$\textsc{Inspect}(L, i) = \texttt{true}$ $|L| > i$ (list L has more than i values).

$\textsc{Inspect}(L, i) = \texttt{false}$ $|L| \leq i$ (list L has at-most i values).

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function $\textsc{Inspect}(L, i)$ *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Solving the list-length problem

We have ordered list $0, 1, \ldots$ of possible values for $|L|$.

Conclusion of $\textsc{Inspect}(L, i)$?

$\textsc{Inspect}(L, i) = \texttt{true}$ $|L| > i$ (list L has more than i values).

$\textsc{Inspect}(L, i) = \texttt{false}$ $|L| \leq i$ (list L has at-most i values).

Issue: no upper bound on the ordered list $0, 1, \ldots$ of possible values for $|L|$

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function $\textsc{Inspect}(L, i)$ *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Issue: no upper bound on the ordered list $0, 1, \ldots$ of possible values for $|L|$

Guess repeatedly with exponentially-growing guesses.

Algorithm $\textsc{ListLength}\text{UB}(L)$:

Input: L is an *array* of unknown length.

$n := 1$.
while $\textsc{Inspect}(L, n)$ **do**
    $n := 2 \cdot n$.
return n.

Result: return N, $|L| \leq N = 1$ or $|L| \leq N < 2|L|$.

Using the *idea* of Binary Search

Problem

Let L be list of values of unknown length and assume we have a function $\textsc{Inspect}(L, i)$ *that returns* `true` *if the list has an i-th value and returns* `false` *otherwise.*

The list-length problem is the problem of finding the length of list L.

Algorithm $\text{LB}\textsc{ListLength}(L, N)$ with $N := \textsc{ListLength}\text{UB}(L)$:

$begin, end := 0, N.$
while $begin \neq end$ **do**
    $mid := (begin + end) \operatorname{div} 2.$
    if $\textsc{Inspect}(L, mid)$ **then**
        $begin := mid + 1.$
    else
        $end := mid.$
return $begin.$

Result: return the length $|L|$ of array L.

Optimizing joins using range queries

Definition

A join of two lists L and M results in a list A in which each list value is computed from a combination of values $u \in L$ and $v \in M$ according to some *join condition*.

Example (Return pairs (p, r) of product name p and related category r)

products	
name	*category*
Apple	Fruit
Bok choy	Vegetable
Canelé	Pastry
Donut	Pastry

categories	
category	*related*
Fruit	Food
Fruit	Produce
Pastry	Food
Vegetable	Food
Vegetable	Produce

Optimizing joins using range queries

Definition

A join of two lists L and M results in a list A in which each list value is computed from a combination of values $u \in L$ and $v \in M$ according to some *join condition*.

Example (Return pairs (p, r) of product name p and related category r)

products	
name	*category*
Apple	Fruit
Bok choy	Vegetable
Canelé	Pastry
Donut	Pastry

categories	
category	*related*
Fruit	Food
Fruit	Produce
Pastry	Food
Vegetable	Food
Vegetable	Produce

Join Result	
name	*related*
Apple	Food
Apple	Produce
Bok choy	Food
Bok choy	Produce
Canelé	Food
Donut	Food

Optimizing joins using range queries

Algorithm NestedLoopPC(products, categories):

Input: relations products(*name*, *category*) and categories(*category*, *related*).

$output := \emptyset$.
for $(p.n, p.c) \in$ products **do**
  for $(c.c, c.r) \in$ categories **do**
    if $p.c = c.c$ **then**
      add $(p.n, c.r)$ to *output*.

Result: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \wedge ((c.c, c.r) \in \text{categories})\}$.

Optimizing joins using range queries

Algorithm NestedLoopPC(products, categories):

Input: relations products(*name*, *category*) and categories(*category*, *related*).

$output := \emptyset$.
for $(p.n, p.c) \in$ products **do**
  for $(c.c, c.r) \in$ categories **do**
    if $p.c = c.c$ **then**
      add $(p.n, c.r)$ to $output$.

$\Theta(|categories|)$.

Result: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \wedge ((c.c, c.r) \in \text{categories})\}$.

Optimizing joins using range queries

Algorithm NestedLoopPC(products, categories):

Input: relations products(*name*, *category*) and categories(*category*, *related*).

$output := \emptyset$.
for $(p.n, p.c) \in$ products **do**
  for $(c.c, c.r) \in$ categories **do**
    if $p.c = c.c$ **then**
      add $(p.n, c.r)$ to $output$.

$\Theta(|categories|)$.

$|products|$ *times.*

Result: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \wedge ((c.c, c.r) \in \text{categories})\}$.

Theorem

The NestedLoopPC *algorithm is correct and has a runtime complexity of* $\Theta(|product| \cdot |categories|)$.

Optimizing joins using range queries

Algorithm NestedBinaryPC(products, categories):

Input: relations products(*name*, *category*) and categories(*category*, *related*), relation categories ordered.

output := $\emptyset$.
for $(p.n, p.c) \in$ products **do**
    $i :=$ LowerBound(categories, $(p.c, \text{‘’})$, 0, |categories|).
    while $i <$ |categories| **and also** categories$[i]$.*category* $= p.c$ **do**
        add $(p.n,$ categories$[i]$.*related*$)$ to *output*.
        $i := i + 1$.

Result: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \land ((c.c, c.r) \in \text{categories})\}$.

Theorem

The NestedBinaryPC algorithm is correct and has a runtime complexity of $\Theta(|product| \cdot \log_2(|categories|) + |result|)$.

CONTAINS, LINEARSEARCH, and LOWERBOUND in practice

Algorithm	C++	Java
CONTAINS	`std::ranges::contains`	*collection*`.contains`[a]
LINEARSEARCH LINEARPREDSEARCH	`std::find` `std::find_if`	*collection*`.indexOf`[a] `java.util.stream::filter`[b]
LOWERBOUND	`std::lower_bound` `std::upper_bound`[d]	`java.util.Arrays::`[c] `binarySearch`
Related libraries	`<algorithm>`, `<ranges>`	`java.util.Arrays`, `java.util.ArrayList`, …

[a] Here, *collection* is a standard Java data collection such as `java.util.ArrayList`.

[b] Using the stream library supported by standard Java data collections.

[c] Does not guarantee to return the offset of the *first* occurrence of a value.

[d] Returns the offset of the first element in the list that is strictly larger than the searched-for value.