CSE 331

More Loop Reasoning

James Wilcox and Kevin Zatloukal

Facts About Sublists

- "With great power, comes great responsibility"
  - since we can easily access any $L[j]$, may need facts about it

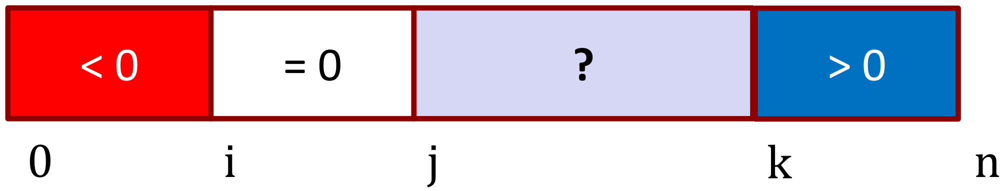
- We can write facts about several elements at once:
  - this says that elements at indexes 0 .. $j\text{-}1$ are not y

$$S[i] \neq y \qquad \text{for any } 0 \leq i < j$$

- These facts get hard to write down!
  - we will need to find ways to make this <u>easier</u>
  - a common trick is to draw pictures instead…

Example: Sorting Negative, Zero, Positive

Our Invariant:



$A[\ell] < 0$ for any $0 \leq \ell < i$

$A[\ell] = 0$ for any $i \leq \ell < j$

(no constraints on $A[\ell]$ for $j \leq \ell < k$)

$A[\ell] > 0$ for any $k \leq \ell < n$

Most humans write invariants in pictures (on their whiteboard) rather than in algebra (in the code)

Visualizing Array Algorithms

Linear Search of an Array

- **Let's check the correctness of this loop (w/ pictures)**

```
public static boolean contains(int[] S, int y) {
  int j = 0;
  // Inv: S[i] /= y for any 0 <= i <= j-1
  while (j != S.length && S[j] != y) {
    j = j + 1;
  }
  return j != S.length;
};
```

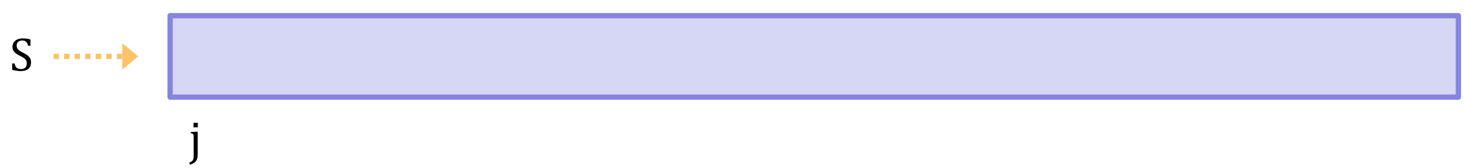
<u>Inv</u>: gold part contains no y

Linear Search of an Array


```
public static boolean contains(int[] S, int y) {
  int j = 0;
  {{ j = 0 }}
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j != S.length && S[j] != y) {
    j = j + 1;
  }
  return j != S.length;
};
```

What is the picture when j = 0?

Inv holds because there is no gold part.



Linear Search of an Array


```
public static boolean contains(int[] S, int y) {
  int j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j != S.length && S[j] != y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j != S.length;
};
```

Linear Search of an Array


```
public static boolean contains(int[] S, int y) {
  int j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j != S.length && S[j] != y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j != S.length;
};
```

Is this valid?

Linear Search of an Array

$\{\{\ (S[i] \neq y \text{ for any } 0 \leq i \leq j-1) \text{ and } j \neq \text{len}(S) \text{ and } S[j] \neq y\ \}\}$

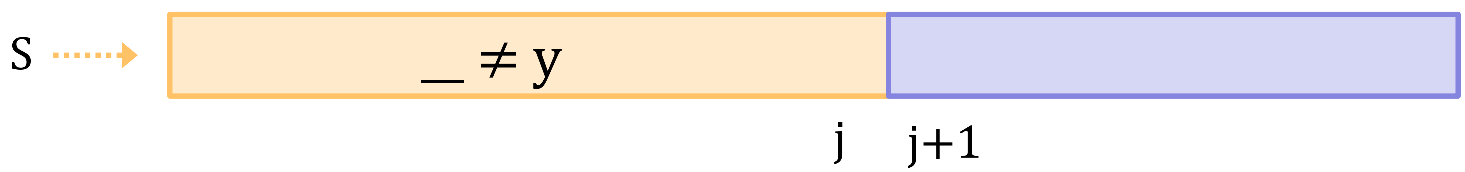
$\{\{\ S[i] \neq y \text{ for any } 0 \leq i \leq j\ \}\}$

- **What does the top assertion say about $S[j]$?**
  - **it is not y**

Linear Search of an Array

{{ (S[i] ≠ y for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
{{ S[i] ≠ y for any 0 ≤ i ≤ j }}

- **What is the picture for the bottom assertion?**

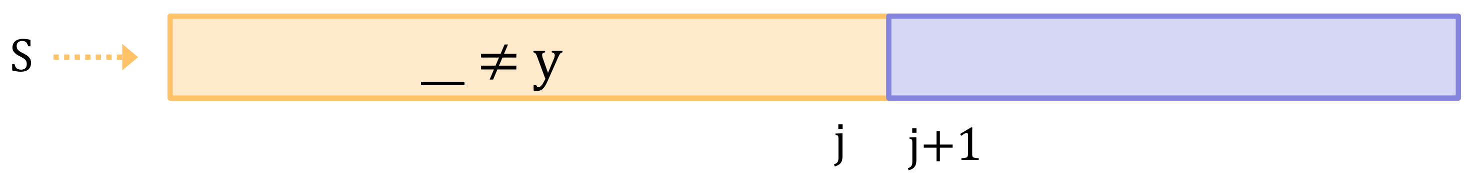


- **Do the facts above imply this holds?**
  – Yes! It's the same picture

Linear Search of an Array

{{ (S[i] ≠ y for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
{{ S[i] ≠ y for any 0 ≤ i ≤ j }}

- **What is the picture for the bottom assertion?**



- **Most likely bug is an off-by-one error**
  - **must check** S[j], **not** S[j-1] **or** S[j+1]

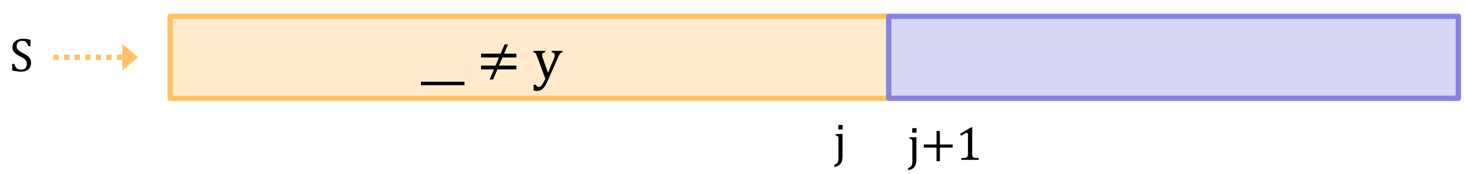
Linear Search of an Array


```
while (j != S.length && S[j+1] != y) {
```

$\{\{\ (S[i] \neq y \text{ for any } 0 \leq i \leq j - 1) \text{ and } j \neq \text{len}(S) \text{ and } S[j+1] \neq y\ \}\}$

$\{\{\ S[i] \neq y \text{ for any } 0 \leq i \leq j\ \}\}$

- **What is the picture for the bottom assertion?**



- **Reasoning would verify that this is not correct**

Linear Search of an Array


```
public static boolean contains(int[] S, int y) {
  int j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j != S.length && S[j] != y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j != S.length;
};
```

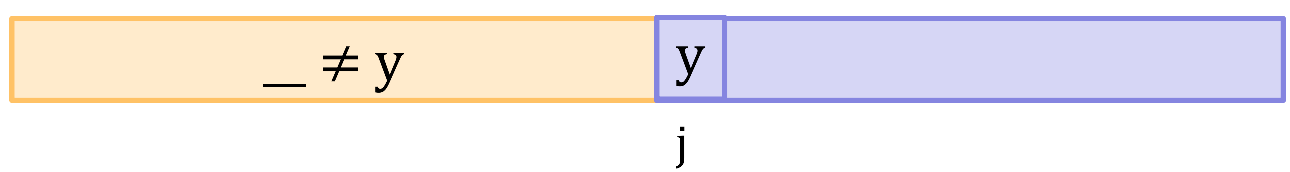
"or" means cases...

Case $j \neq \text{len}(S)$**:**

Must have $S[j] = y$**.**

What is the picture now?

Code should and does return true.



Linear Search of an Array


```
public static boolean contains(int[] S, int y) {
  int j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j != S.length && S[j] != y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j != S.length;
};
```

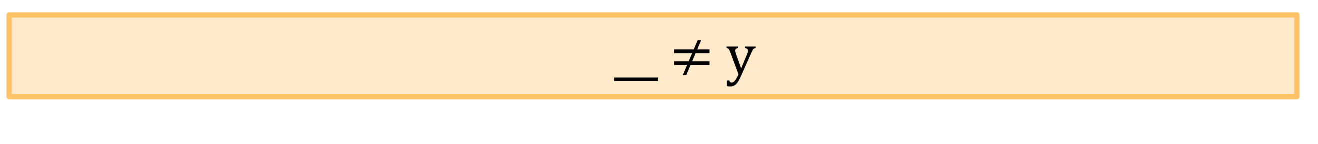
"or" means cases...

Case j = len(S):

What does Inv say now?

Says y is not in the array!

Code should and does return false.



Finding an Element in an Array

- Can search for an element in an array as follows

$$
\begin{array}{lll}
\text{contains}(\text{nil}, y) & := \text{false} & \\
\text{contains}(x :: L, y) & := \text{true} & \textbf{if } x = y \\
\text{contains}(x :: L, y) & := \text{contains}(L, y) & \textbf{if } x \neq y
\end{array}
$$

- Searches through the array in linear time
  - did the same on lists
- Can be done more quickly if the list is sorted
  - binary search!

Finding an Element in a Sorted Array

- Can search more quickly if the list is sorted
  - precondition is $A[0] \leq A[1] \leq \ldots \leq A[n\text{-}1]$ (informal)
  - write this formally as

$$A[j] \leq A[j+1] \text{ for any } 0 \leq j \leq n-2$$

- Not easy to describe this visually…
  - how about a gradient?

S 

Binary Search of an Array


```
boolean bsearch(int[] S, int y) {
  int j = 0, k = S.length;
```

{{ **Inv**: ($S[i] < y$ for any $0 \le i < j$) **and** ($y \le S[i]$ for any $k \le i < n$) }}

```
  while (j != k) {
    int m = (j + k) / 2;
    if (S[m] < y) {
      j = m + 1;
    } else {
      k = m;
    }
  }
  return (S[k] == y);
};
```

Inv includes facts about two regions.

Let's check that this is right...

Binary Search of an Array


```
boolean bsearch(int[] S, int y) {
  int j = 0, k = S.length;
```

{{ $j = 0$ and $k = n$ }}

{{ **Inv**: ($S[i] < y$ for any $0 \leq i < j$) **and** ($y \leq S[i]$ for any $k \leq i < n$) }}

- **What does the picture look like with $j = 0$ and $k = n$?**

- **Does this hold?**
  - Yes! It's vacuously true

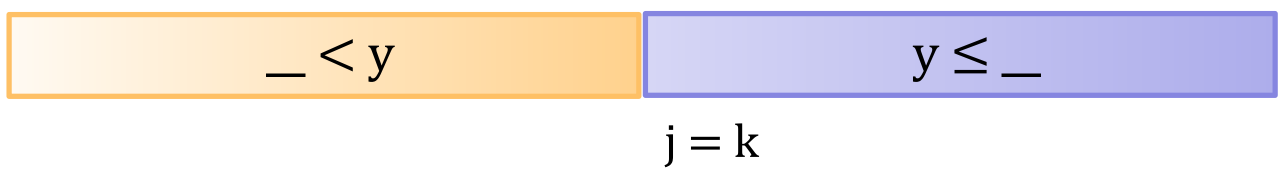
Binary Search of an Array


```
boolean bsearch(int[] S, int y) {
  int j = 0, k = S.length;
  {{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
  while (j != k) {
   …
  }
  {{ Inv and (j = k) }}
  {{ contains(S, y) = (S[y] = y) }}
  return (S[k] == y);
};
```

Binary Search of an Array


```
    {{ Inv and (j = k) }}
    {{ contains(S, y) = (S[y] = y) }}
    return (S[k] == y);
};
```

- **What does the picture look like with $j = k$?**



- **Does S contain y iff $S[k] = y$?**
  - **If** $S[k] = y$**, then** $contains(S, y) = true$
  - **If** $S[k] \neq y$**, then** $S[k] < y$ **and** $S[i] < y$ **for every** $k < i$**, so** $contains(S, y) = false$

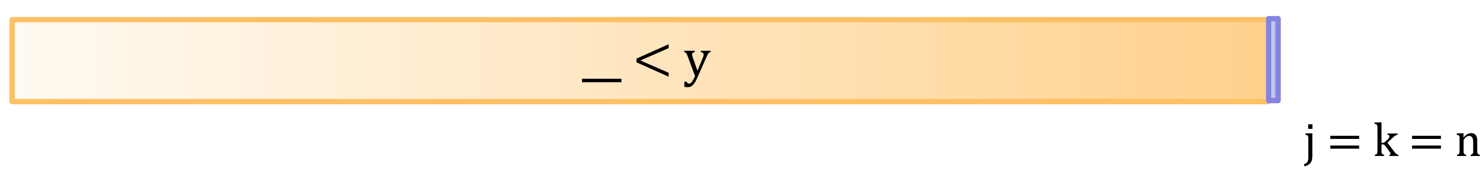
What case are we missing?

Binary Search of an Array

{{ Inv and (j = k) }}
{{ contains(S, y) = (S[y] = y) }}

```
    return (k < S.length) && (S[k] == y);
};
```

- What does the picture look like with $j = k = n$?



- In this case...
  - we see that contains(S, y) = false
  - and the code returns false because "`k < S.length`" is false

Binary Search of an Array

{{ **Inv**: (S[i] < y for any 0 ≤ i < j) **and** (y ≤ S[i] for any k ≤ i < n) }}

```
while (j != k) {
  {{ Inv and (j < k) }}
  int m = (j + k) / 2;
  if (S[m] < y) {
    j = m + 1;
  } else {
    k = m;
  }
  {{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
}
```

Reason through both paths...

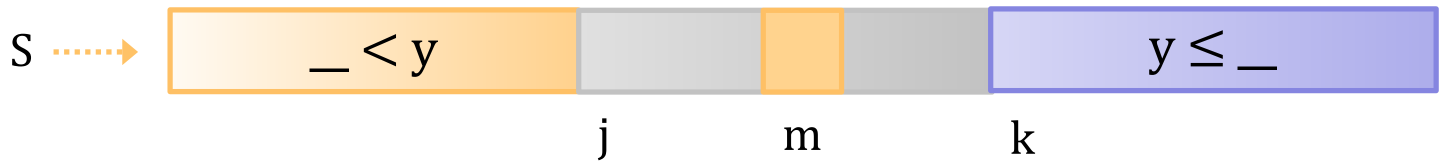
Binary Search of an Array


```
{{ Inv and (j < k) }}
int m = (j + k) / 2;
if (S[m] < y) {
  {{ Inv and (j < k) and (S[m] < y) }}
  j = m + 1;
} else {
  {{ Inv and (j < k) and (S[m] ≥ y) }}
  k = m;
}
{{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
}
```

Binary Search of an Array


```
int m = (j + k) / 2;
if (S[m] < y) {
    {{ Inv and (j < k) and (S[m] < y) }}
    {{ (S[i] < y  for any 0 ≤ i < m+1) and (y ≤ S[i] for any k ≤ i < n) }}
    j = m + 1;
} else {
    {{ Inv and (j < k) and (S[m] ≥ y) }}
    {{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any m ≤ i < n) }}
    k = m;
}
{{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
```

Binary Search of an Array



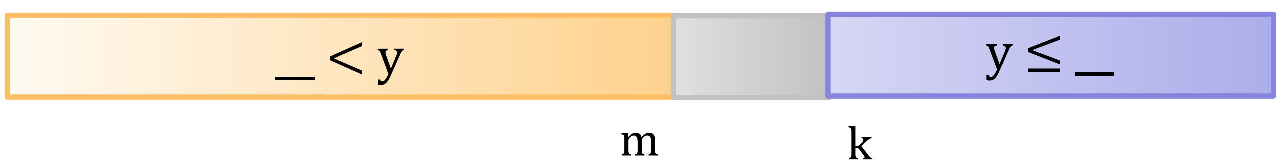

```
int m = (j + k) / 2;
if (S[m] < y) {
```

    {{ Inv and (j < k) and (S[m] < y) }}

    {{ (S[i] < y for any $0 \leq i < m+1$) **and** (y ≤ S[i] for any $k \leq i < n$) }}

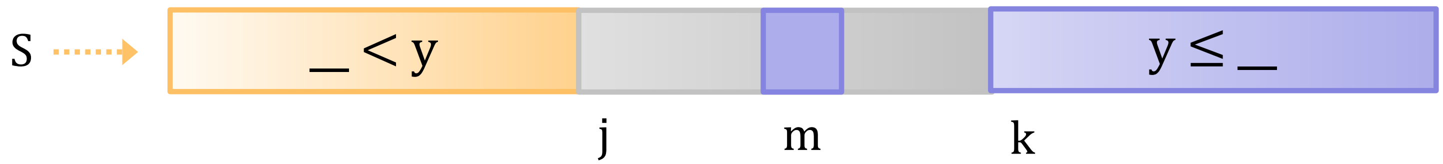
```
  j = m + 1;
} ...
```

- **What does the picture look like in the bottom assertion?**



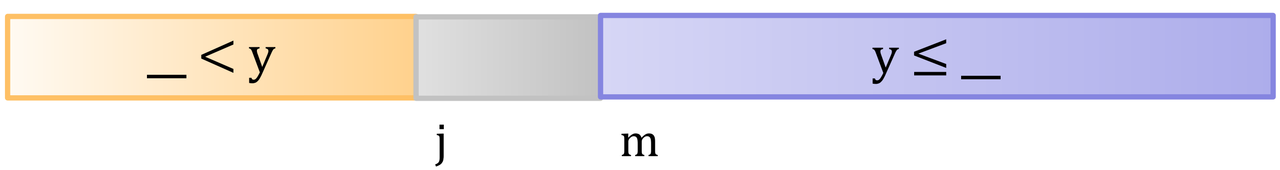
- **Does this hold?**
  - **Yes! Because the array is sorted (everything before S[m] is even smaller)**

Binary Search of an Array




```
int m = (j + k) / 2;
… else {
    {{ Inv and (j < k) and (S[m] ≥ y) }}
    {{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any m ≤ i < n) }}
    k = m;
}
```

- **What does the picture look like in the bottom assertion?**



- **Does this hold?**
  – **Yes! Because the array is sorted (everything after S[m] is even larger)**

Binary Search of an Array


```
boolean bsearch(int[] S, int y) {
  int j = 0, k = S.length;
```

{{ **Inv**: (S[i] < y for any $0 \leq i < j$) **and** ($y \leq$ S[i] for any $k \leq i < n$) }}

```
  while (j != k) {
    int m = (j + k) / 2;
    if (S[m] < y) {
      j = m + 1;
    } else {
      k = m;
    }
  }
  return (S[k] == y);
};
```

Does this terminate?

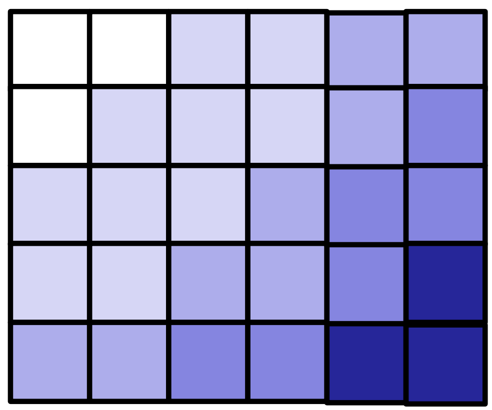
Need to check that $k - j$ *decreases*

Can see that $j \leq m \leq k$, so the "then" branch is fine.

Can see that $j < k$ implies $m < k$ (integer division rounds <u>down</u>), so the "else" branch is also fine

Sorted Matrix Search

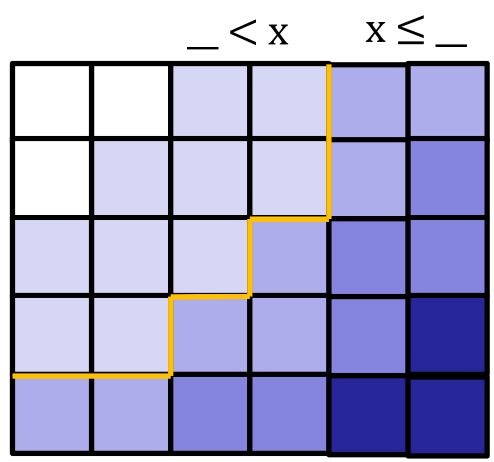
Given a sorted matrix M, with m rows and n cols,
where every row and every column is sorted,
find out whether a given number x is in the matrix



(darker color means larger)

Sorted Matrix Search

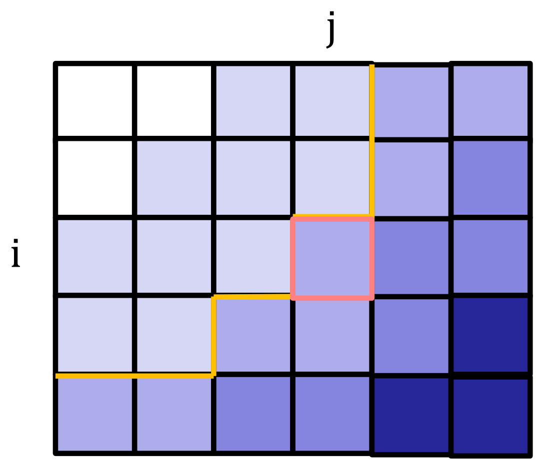
Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Idea: Trace the contour between the numbers ≤ x and > x in each row to see if x appears.

Sorted Matrix Search

Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Invariant: at the left-most entry with $x \leq \_$ in the row

– for each row i, this holds for exactly one column j

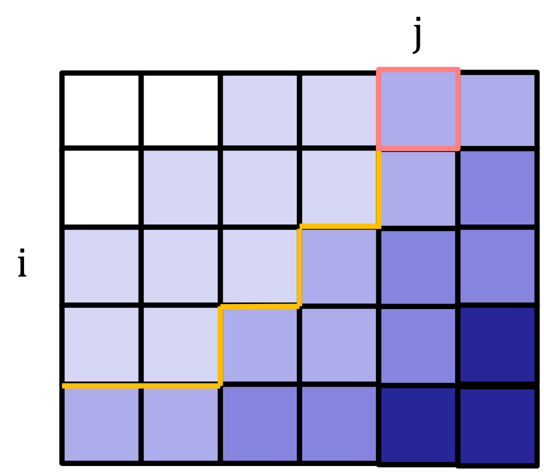
Sorted Matrix Search

Invariant: at the left-most entry with $x \leq$ _ in the row

- for each row i, this holds for exactly one column j

Initialization: how do we get this to hold for $i = 0$?

- could be anywhere in the first row

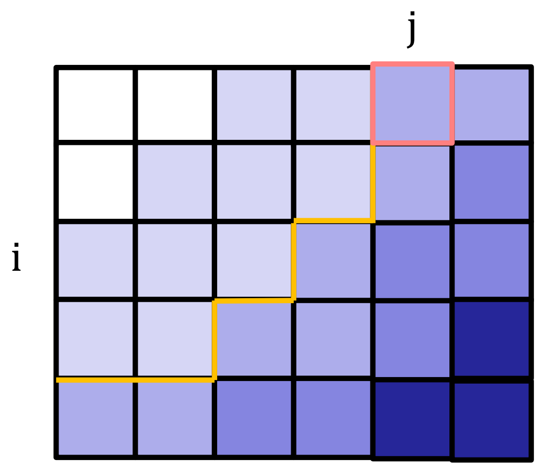


Need to *search* to find this location

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

– will need a loop…



How do we find an invariant for that loop?

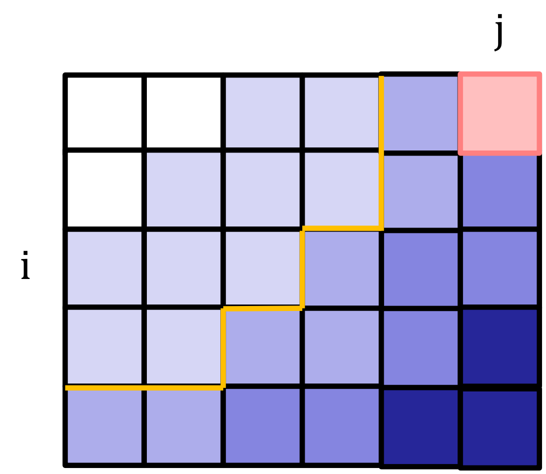
– try weakening this assertion (allow any j, not just smallest)

– decrease j until $x \leq M[0, j\text{-}1]$ does not hold

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

```
int i = 0;
int j = ??
```
{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}
```
while (??)
  ??
```
{{ **Post**: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ }}



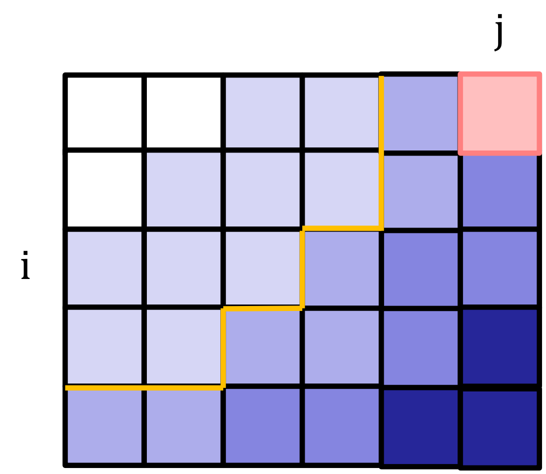
How do we set j to make Inv hold initially?

– range is empty when $j = n$

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

```
int i = 0;
int j = n;
{{ Inv: x ≤ M[0, k] for any j ≤ k < n }}
while (??)
   ??
{{ Post: M[0, k] < x for any 0 ≤ k < j and x ≤ M[0, k] for any j ≤ k < n }}
```



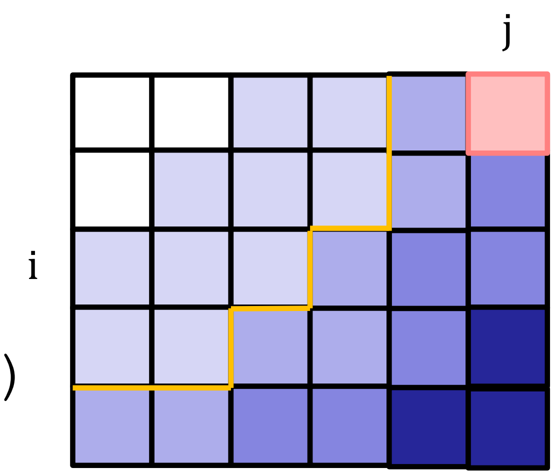
How do we exit so that the postcondition holds?

- **can no longer decrease j when** $j = 0$ **or** $M[0, j\text{-}1] < x$

Sorted Matrix Search

New Goal: find smallest j with $x \le M[0, \mathbf{k}]$ for any $j \le k < n$

```
int i = 0;
int j = n;
{{ Inv: x ≤ M[0, k] for any j ≤ k < n }}
while (j>0 && x <= M[0][j-1])
   ??
{{ Post: M[0, k] < x for any 0 ≤ k < j and x ≤ M[0, k] for any j ≤ k < n }}
```



Anything needed in the loop body?
(That is, other than `j = j - 1`?)

Sorted Matrix Search

New Goal: find smallest j **with** $x \le M[0, \mathbf{k}]$ for any $j \le k < n$

{{ **Inv**: $x \le M[0, k]$ for any $j \le k < n$ }}

```
while (j>0 && x <= M[0][j-1]) {
```

{{ $x \le M[0, k]$ for any $j \le k < n$ and $j > 0$ and $x \le M[0, j\text{-}1]$ }}

??

```
    j = j - 1;
```

{{ $x \le M[0, k]$ for any $j \le k < n$ }}

```
}
```

Sorted Matrix Search

New Goal: find smallest j **with** $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (j>0 && x <= M[0][j-1]) {
```

{{ $x \leq M[0, k]$ for any $j \leq k < n$ and $j > 0$ and $x \leq M[0, j\text{-}1]$ }}

??

{{ $x \leq M[0, k]$ for any $j - 1 \leq k < n$ }}

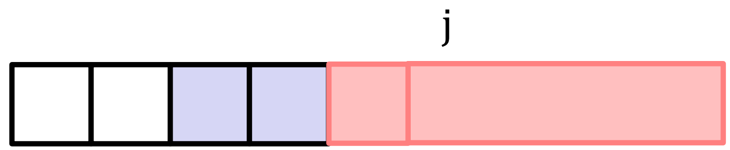
```
j = j - 1;
```

{{ $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
}
```

Sorted Matrix Search

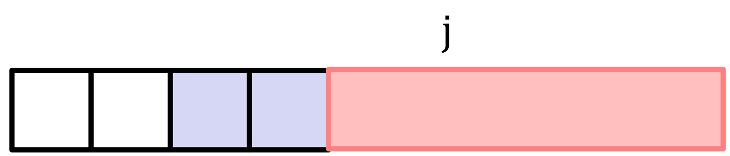
New Goal: find smallest j **with** $x \le M[0, \mathbf{k}]$ for any $j \le k < n$



{{ $x \le M[0, k]$ for any $j \le k < n$ and $j > 0$ and $x \le M[0, j\text{-}1]$ }}

??

{{ $x \le M[0, k]$ for any $j - 1 \le k < n$ }}



Nothing is missing!

Sorted Matrix Search

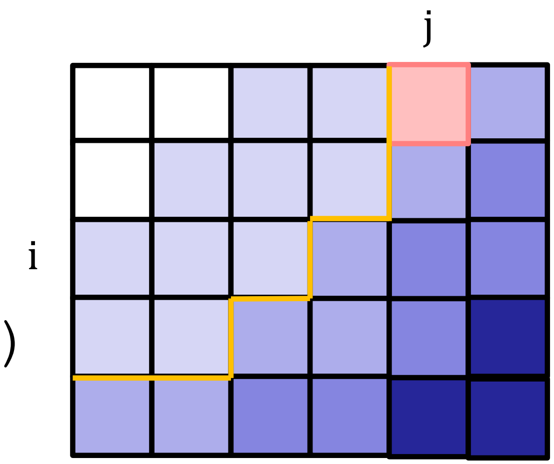
New Goal: find smallest j **with** $x \leq M[0, k]$ for any $j \leq k < n$

```
int i = 0;
int j = n;
```

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (j>0 && x <= M[0][j-1])
  j = j - 1;
```

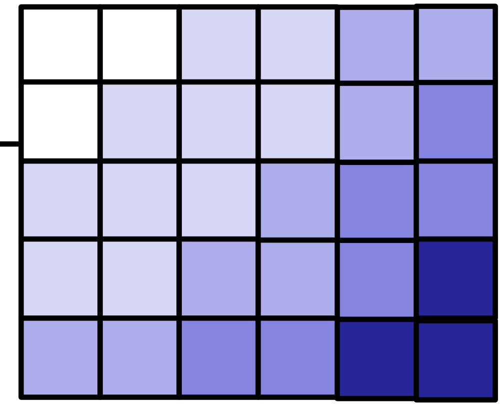
{{ **Post**: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ }}



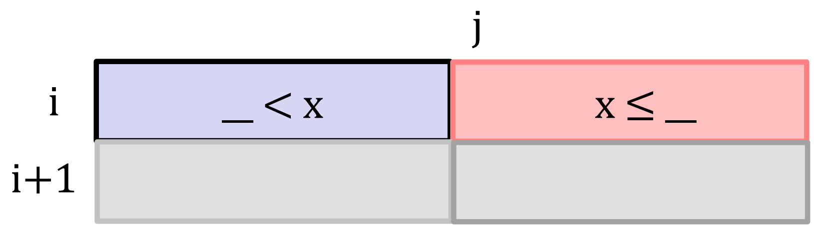
Can now check if $M[0, j] = x$

- if not, then it is not in the first row
- move on to the second row…

Sorted Matrix Search



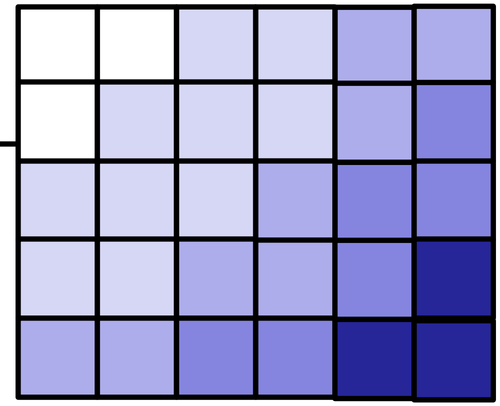
Moving from row i to row $i+1$



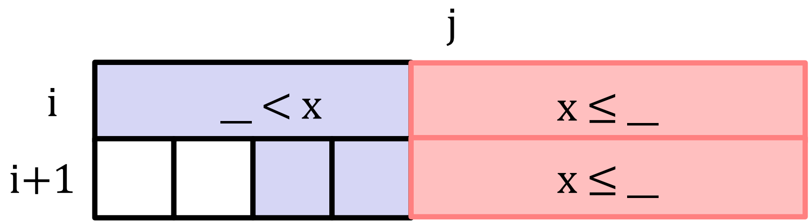
What does *vertical* sorting tell us about row $i+1$?

- right side is guaranteed to satisfy " $x \leq \_$ "
- left side **not** guaranteed to satisfy " $\_ < x$ "

Sorted Matrix Search

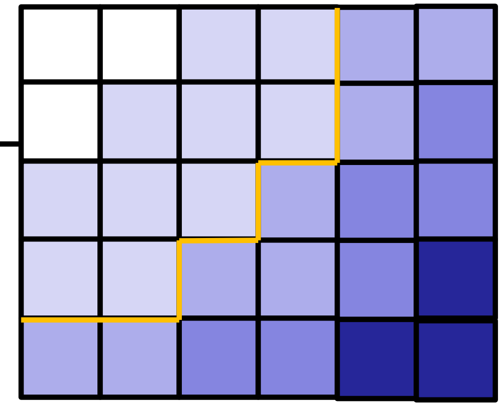


Moving from row i to row i+1

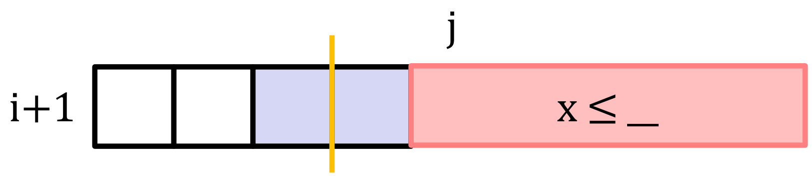


Next row looks like this

Sorted Matrix Search



Moving from row i to row i+1



How do we restore the invariant?

– find the index j with $M[i+1, j-1] < x \leq M[i+1, j]$

This is the same problem as before!

– move left until begining or $M[i+1, j-1] < x$ holds

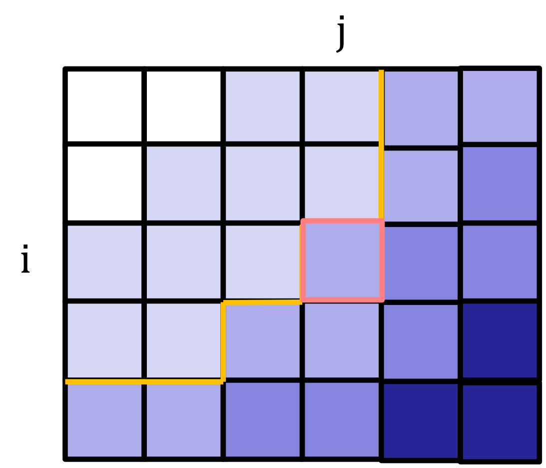
Sorted Matrix Search

```
int i = 0;
int j = n;
... move j to left...
if (M[i][j] == x) return true;
```

{{ **Inv**: (x is not in row k for any $0 \le k \le i$) and

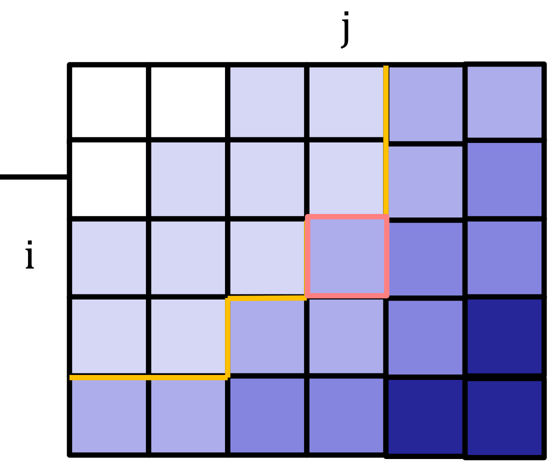
(M[i, k] < x for any $0 \le k < j$) and ($x \le$ M[i, k] for any $j \le k < n$) }}

```
while (i+1 != n) {
   ...
}
return false;
```



Inv says we ruled out rows 0 .. i
and col j is line between _ < x and x ≤ _

Sorted Matrix Search




```
int i = 0;
int j = n;
... move j to left...
if (M[i][j] == x) return true;
```

{{ **Inv**: (x is not in row k for any $0 \leq k \leq i$) and

(M[i, k] < x for any $0 \leq k < j$) and ($x \leq$ M[i, k] for any $j \leq k < n$) }}

```
while (i+1 != n) {
  i = i + 1;
  ... move j to the left...
  if (M[i][j] == x) return true;
}
return false;
```

We can avoid writing this code twice (without writing a separate function)...

Don't try this at home!

Sorted Matrix Search

```
int i = 0;
int j = n;
while (i != n) {
  ... move j to left...
  if (M[i][j] == x) return true;
  {{ Inv: (x is not in row k for any 0 ≤ k ≤ i) and
        (M[i, k] < x for any 0 ≤ k < j) and (x ≤ M[i, k] for any j ≤ k < n) }}
  i = i + 1;
}
return false;
```

Loop condition was also changed

Inv is now checked in the middle of the loop!

Sorted Matrix Search

Final version is 9 lines of code.

Requires 6 lines of invariant assertions!

```
int i = 0;
int j = n;
while (i != n) {
```

{{ **Inv**: $x \le M[i, k]$ for any $j \le k < n$ }}

```
  while (j > 0 && x <= M[i][j-1])
    j = j - 1;
  if (M[i][j] == x)
    return true;
```

{{ **Inv**: (x is not in row k for any $0 \le k \le i$) and

($M[i, k] < x$ for any $0 \le k < j$) and ($x \le M[i, k]$ for any $j \le k < n$) }}

```
  i = i + 1;
}
return false;
```

Tail Recursion

Local Variable Mutation & Memory Use

- With only straight-line code & conditionals…
  - it seems like it saves memory
  - but it does not (compiler would fix anyway)

- With loops…
  - it really does save memory

    no improvement in **running time**
  - but low-memory loops cannot be used in all cases

    some problems really do require more memory

- When can low-memory loops be used and when not?

Sum of the Values in a List

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

Recursion can be directly translated into code

- **Loop to calculate sum of a list**

$\{\{\ L = L_0\ \}\}$

```
int s = 0;
```

$\{\{\ \textbf{Inv}\text{: } \mathrm{sum}(L_0) = s + \mathrm{sum}(L)\ \}\}$

```
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{\ s = \mathrm{sum}(L_0)\ \}\}$

Sum of the Values in a List

Loop

{{ $L = L_0$ }}

```
int s = 0;
```

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
```

{{ $s = \mathrm{sum}(L_0)$ }}

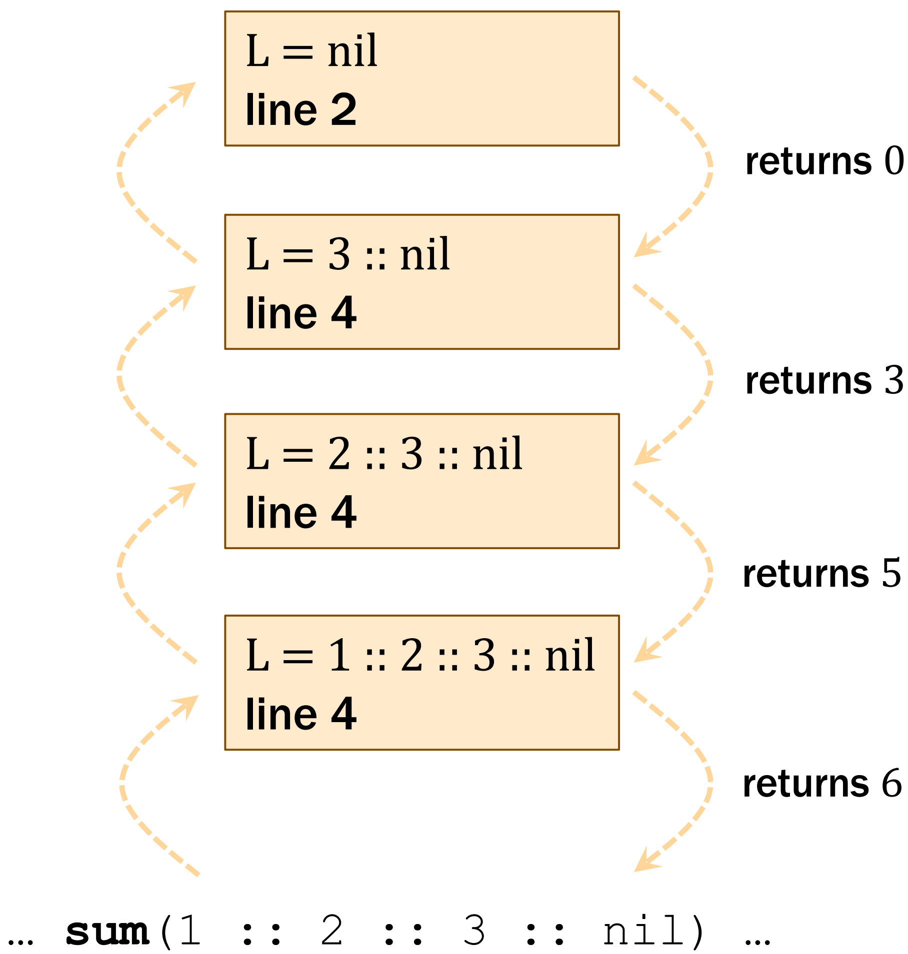
Recursion

```
int sum(List L) {
  if (L == null) {
    return 0;
  } else {
    return L.hd + sum(L.tl);
  }
}
```

Both run in $O(n)$ time where $n = \mathrm{len}(L)$

Loop uses $O(1)$ extra memory, but right does not…

Recursive Version of Sum




```
int sum(List L) {
1   if (L == null) {
2     return 0;
3   } else {
4     return L.hd + sum(L.tl);
5   }
}
```

List of length 3 takes 4 calls
List of length n takes $n+1$ calls.

Call uses $O(n)$ memory,
where $n = \text{len}(L)$

How much does this matter?

- In principle, this extra memory usually not a problem
  - $\mathrm{O(n)}$ time is usually the more important constraint

- In practice, sometimes we are memory constrained
  - in the browser, $\mathrm{sum(L)}$ exceeds stack size at $\mathrm{len(L)} = 10{,}000$

- Loops $\gg$ Recursion?

- Nope!
  1. Loops do not <u>always</u> use less memory.
  2. Recursion can solve <u>more problems</u> than low-memory loops
  3. Extra memory use pays for some other benefits.

Another Sum of the Values in a List

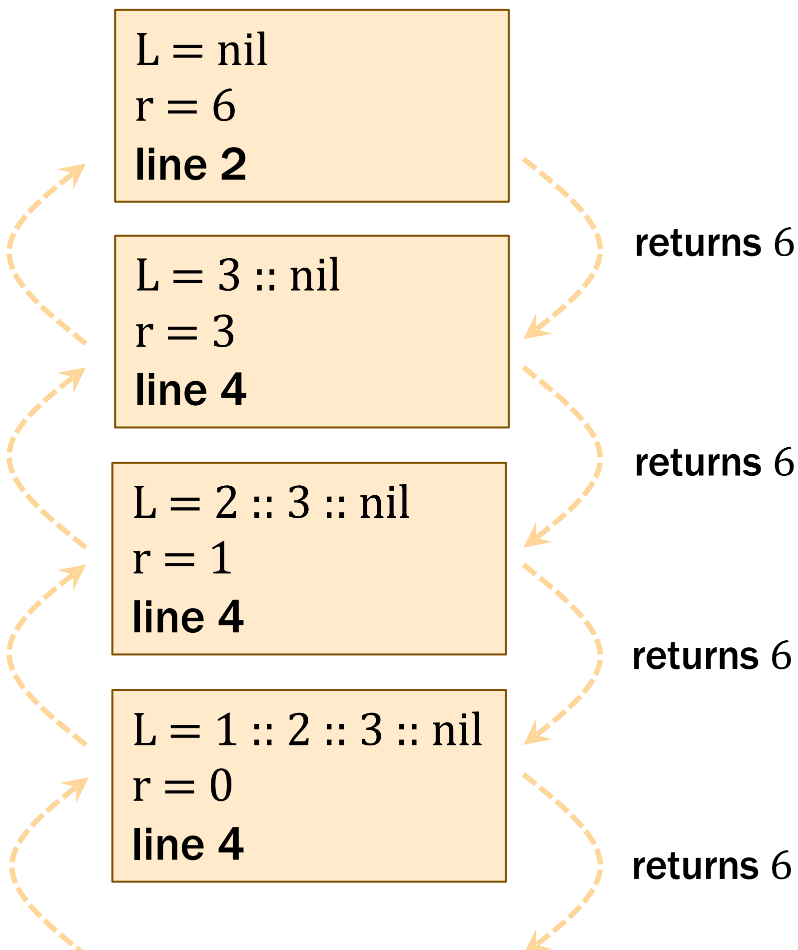
- **Saw another summation function in Topic 2**

$$\begin{aligned} \text{sum-acc}(\text{nil}, r) &:= r \\ \text{sum-acc}(x :: L, r) &:= \text{sum-acc}(L, x + r) \end{aligned}$$

- **Translates to the following code**

```
int sum_acc(List L, int r) {
  if (L == null) {
    return r;
  } else {
    return sum_acc(L.tl, L.hd + r);
  }
}
```

Recursive Version of Sum




```
int sum_acc(List L, int r) {
1   if (L == null) {
2     return r;
3   } else {
4     return sum_acc(L.tl, L.hd + r);
5   }
}
```

This is a "tail call" and "tail recursion".

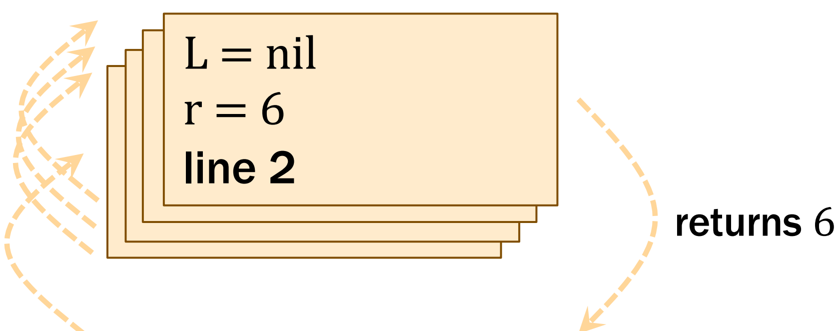
Same return value means no need to remember where we were.

No need to keep stack old frames!
Tail call optimization reuses them...

```
… sum_acc(1 :: 2 :: 3 :: nil, 0) …
```

Recursive Version of Sum

```
int sum_acc(List L, int r) {
1   if (L == null) {
2     return r;
3   } else {
4     return sum_acc(L.tl, L.hd + r);
5   }
}
```



… **sum_acc**(1 :: 2 :: 3 :: nil, 0) …

Tail call optimization reuses stack frames so only O(1) memory

What does this look like?   A loop!

`sum_acc` calculates the *same values* in the *same order* as the loop

Loops vs Tail Recursion

- Tail-call optimization turns tail recursion into a <u>loop</u>

- Functional languages implement tail-call optimization
  - standard feature of such languages
  - you don't write loops; you write tail recursive functions

- Chrome added tail-call optimization… then dropped it!
  - loops / tail-call optimization have downsides (more later…)
  - it no longer does this automatically

    you must manually convert to a loop if you require O(1) memory

Loops vs Tail Recursion

Ordinary Loops $\leq$ Tail Recursion (with tail-call optimization)

- Tail recursion can solve all problems loop can
  - any loop can be translated to tail recursion
  - both use O(1) memory with tail-call optimization

- Translation is simple and important to understand

- Tells us that Ordinary Loops $\ll$ Recursion
  - correspond to the *special* case of tail recursion

Loop to Tail Recursion

```
T myLoop(List R) {
  T s = f(R);
  while (R != null) {
    s = g(s, R.hd);
    R = R.tl;
  }
  return h(s);
};
```

{{ **Inv:** my-acc(R_0, s_0) = my-acc(R, s) }}

- **Tail-recursive function that does same calculation:**

my-acc(nil, s)	:= h(s)	**after loop**
my-acc(x :: L, s)	:= my-acc(L, g(s, x))	**loop body**
my-func(L)	:= my-acc(L, f(L))	**before loop**

Example 1: Loop to Tail Recursion

```
int sumLoop(List R) {
  int s = 0;
  while (R != null) {
    s = s + R.hd;
    R = R.tl;
  }
  return s;
};
```

- **Tail-recursive function that does same calculation:**

$$\begin{aligned} \text{sum-acc}(\text{nil}, s) &:= h(s) \\ \text{sum-acc}(x :: L, s) &:= \text{my-acc}(L, g(s, x)) \end{aligned}$$

$h(s) \to s$

$g(s, x) \to s + x$

$$\text{sum-func}(L) := \text{my-acc}(L, f(L))$$

$f(L) \to 0$

Example 1: Loop to Tail Recursion

```
int sumLoop(List R) {
  int s = 0;
  while (R != null) {
    s = s + R.hd;
    R = R.tl;
  }
  return s;
};
```

{{ Inv: $\text{sum-acc}(R_0, s_0) = \text{sum-acc}(R, s)$ }}

- **Tail-recursive function that does same calculation:**

$$\begin{aligned}\text{sum-acc}(\text{nil}, s) &:= s \\ \text{sum-acc}(x :: L, s) &:= \text{sum-acc}(L, s + x)\end{aligned}$$

$$\text{sum-func}(L) := \text{sum-acc}(L, 0)$$

Example 2: Max Value in a List

```
int maxLoop(List R) {
  if (R == null) throw …
  int s = R.hd;
  R = R.tl;
  while (R != null) {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

maxLoop(1 :: 3 :: 4 :: 2 :: nil)

Iteration	R	s

Example 2: Max Value in a List

```
int maxLoop(List R) {
  if (R == null) throw …
  int s = R.hd;
  R = R.tl;
  while (R != null) {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

maxLoop(1 :: 3 :: 4 :: 2 :: nil)

Iteration	R	s
0	3 :: 4 :: 2 :: nil	1
1	4 :: 2 :: nil	3
2	2 :: nil	4
3	nil	4

Example 2: Loop to Tail Recursion

```
int maxLoop(List R) {
  if (R == null) throw …
  int s = R.hd;
  R = R.tl;
  while (R != null) {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

$$\begin{aligned} \text{max-acc}(\text{nil}, s) &:= h(s) \\ \text{max-acc}(x :: R, s) &:= \text{max-acc}(R, g(s, x)) \end{aligned}$$

$$h(s) \rightarrow s$$

$$g(s, x) \rightarrow \begin{cases} x & \textbf{if } x > s \\ s & \textbf{if } x \leq s \end{cases}$$

$$\text{max}(R) := \text{max-acc}(R, f(R))$$

$$f(R) \rightarrow R.\text{hd} \;\; \textbf{if } L \neq \text{nil}$$

Example 2: Loop to Tail Recursion

```
int maxLoop(List R) {
  if (R == null) throw …
  int s = R.hd;
  R = R.tl;
  while (R != null) {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

{{ **Inv:** $\text{max-acc}(R_0, s_0) = \text{max-acc}(R, s)$ }}

max-acc(nil, s)	:= s	
max-acc(x :: R, s)	:= max-acc(R, x)	if $x > s$
max-acc(x :: R, s)	:= max-acc(R, s)	if $x \leq s$
max(x :: R)	:= max-acc(R, x)	

Example 2: Loop to Tail Recursion

```
int maxLoop(List R) {
  if (R == null) throw …
  let s = R.hd;
  R = R.tl;
  while (R != null) {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

$\boxed{\max(1 :: 3 :: 4 :: 2 :: \text{nil})}$

$\max(1 :: 3 :: 4 :: 2 :: \text{nil})$
$= \text{max-acc}(3 :: 4 :: 2 :: \text{nil}, 1)$ **def of ...**
$= \text{max-acc}(4 :: 2 :: \text{nil}, 3)$ (**since** $3 > 1$)
$= \text{max-acc}(2 :: \text{nil}, 4)$ (**since** $4 > 3$)
$= \text{max-acc}(\text{nil}, 4)$ (**since** $2 \le 4$)
$= 4$

$\text{max-acc}(\text{nil}, s) := s$
$\text{max-acc}(x :: R, s) := \text{max-acc}(R, x)$ if $x > s$
$\text{max-acc}(x :: R, s) := \text{max-acc}(R, s)$ if $x \le s$

$\max(x :: R) := \text{max-acc}(R, x)$

Loops vs Tail Recursion

- Tail recursion gives nicer notation for loop operation

$\text{maxLoop}(1 :: 3 :: 4 :: 2 :: \text{nil})$

Iteration	R	s
0	$3 :: 4 :: 2 :: \text{nil}$	1
1	$4 :: 2 :: \text{nil}$	3
2	$2 :: \text{nil}$	4
3	nil	4

$\text{max}(1 :: 3 :: 4 :: 2 :: \text{nil})$

$$
\begin{aligned}
&\text{max}(1 :: 3 :: 4 :: 2 :: \text{nil}) \\
&\quad = \text{max-acc}(3 :: 4 :: 2 :: \text{nil}, 1) && \textbf{def of ...} \\
&\quad = \text{max-acc}(4 :: 2 :: \text{nil}, 3) && (\textbf{since } 3 > 1) \\
&\quad = \text{max-acc}(2 :: \text{nil}, 4) && (\textbf{since } 4 > 3) \\
&\quad = \text{max-acc}(\text{nil}, 4) && (\textbf{since } 2 \le 4) \\
&\quad = 4
\end{aligned}
$$

- Loops are hard to describe with math
  - math never mutates anything, so loops are not a good fit
  - tail recursive notation shows loop operation in calculation block

More Loops vs Tail Recursion

- Ordinary loops use less memory than (non-tail) recursion

- This is a tradeoff
  - save memory at the loss of information…

Example 2: Max Value in a List

```
int maxLoop(List R) {
1 if (R == null) throw …
2 int s = R.hd;
3 R = R.tl;
4 while (R != null) {
5   if (R.hd > s)
6     s = R.hd;
7   R = R.tl;
8 }
9 return s;
};
```

Suppose we are at line 5
with $\mathrm{R} = 4 :: 2 :: \mathrm{nil}$ **and** $\mathrm{s} = 3$

Could have started out with...

$\mathrm{R} = 1 :: 3 :: 4 :: 2 :: \mathrm{nil}$

$\mathrm{R} = 3 :: 4 :: 2 :: \mathrm{nil}$

$\mathrm{R} = 0 :: 1 :: 3 :: 3 :: 1 :: 1 :: 1 :: 0 :: 4 :: 2 :: \mathrm{nil}$

...

Could have been one of infinitely many lists!

Example 2: Max Value in a List

```
int maxLoop(List R) {
1 if (R == null) throw …
2 let s = R.hd;
3 R = R.tl;
4 while (R != null) {
5   if (R.hd > s)
6     s = R.hd;
7   R = R.tl;
8 }
9 return s;
};
```

Suppose we are at line 4
with $R = 4 :: 2 :: \mathrm{nil}$ and $s = 3$

Could have been one of infinitely many lists!

Is there a situation where knowing
<u>how</u> we got to a line is important?

It matters when **debugging**!

Loop saves memory at the cost of harder debugging.

This is why (I think) Chrome removed the optimization.

Key Takeaways

- Any loop can be translated to tail recursion
  - they describe the same *calculation*
    - tail recursive version *is a* loop (with tail call optimization)
  - tail recursive notation is also useful for analyzing the loop
- Ordinary loops are strictly *less powerful* than recursion
  - not all recursive functions can be written as tail recursion
  - many problems cannot be solved in O(1) memory
    - e.g., tree traversals *require* extra space
    - many (most?) list operations require extra space
- Ordinary loops save memory but are harder to debug
  - information thrown away tells you how you got there

Faster Len

$$\begin{aligned} \mathrm{len}(\mathrm{nil}) \quad &:= 0 \\ \mathrm{len}(x :: L) \quad &:= 1 + \mathrm{len}(L) \end{aligned}$$

$$\begin{aligned} \text{len-acc}(\mathrm{nil}, r) \quad &:= r \\ \text{len-acc}(x :: L, r) \quad &:= \text{len-acc}(L, r + 1) \end{aligned}$$

- **Both versions are recursive and $O(n)$ time**
  - **second version is tail recursive**
- **Can see that** $\mathrm{len}(S) = \text{len-acc}(S, 0)$

Tail Recursion to a Loop

$$\begin{aligned} \text{len-acc}(\text{nil}, r) &:= r \\ \text{len-acc}(x :: L, r) &:= \text{len-acc}(L, r + 1) \end{aligned}$$

- **Could implement** len-acc **with a loop as:**

```
int len_acc(List S, int r) {
  {{ Inv: len-acc(S_0, r_0) = len-acc(S, r) }}
  while (S != null) {
    r = r + 1;
    S = S.tl;
  }
  return r;
};
```

recursive cases (tail calls)

base cases

Tail Recursion to a Loop

$$\begin{aligned} &\text{sum-acc}(\text{nil}, r) &&:= r \\ &\text{sum-acc}(x :: L, r) &&:= \text{sum-acc}(L, x + r) \end{aligned}$$

- **Could implement** sum-acc **with a loop as:**

```
int sum_acc(List S, int r) {
  {{ Inv: sum-acc(S_0, r_0) = sum-acc(S, r) }}
  while (S != null) {
    r = r + S.hd;
    S = S.tl;
  }
  return r;
};
```

recursive cases (tail calls)

base cases

Tail Recursion to a Loop

$$\left.\begin{array}{ll} f(\ldots\ p_1\ \ldots,\ r) & :=\ \ldots \\ \ldots & \\ f(\ldots\ p_n\ \ldots,\ r) & :=\ \ldots \end{array}\right\}\ \text{base cases}$$

$$\left.\begin{array}{ll} f(\ldots\ q_1\ \ldots,\ r) & :=\ f(\ldots) \\ \ldots & \\ f(\ldots\ q_n\ \ldots,\ r) & :=\ f(\ldots) \end{array}\right\}\ \text{recursive cases (tail calls } \textit{only}\text{)}$$

- **Tail-recursive function becomes a loop:**

```
// Inv: f(args₀) = f(args)
while (args /* match some q pattern */) {
  args = /* right-side of appropriate q pattern */;
}
return /* right-side of appropriate p pattern */;
```

Last Element

last(nil)	:= undefined
last(x :: nil)	:= x
last(x :: y :: L)	:= last(y :: L)

- **Returns the last element of the list**
  - **only defined if the list is non-empty**

    otherwise, there is no last element

- **This is already tail recursive**
  - **so we can translate it into a loop…**

Last Element

last(nil) := undefined
last(x :: nil) := x
last(x :: y :: L) := last(y :: L)] tail recursive case

- **Translate to a loop:**

```
// @param S a non-empty list
int last(List S) {
  // Inv: last(S0) = last(S)
  while (args /* match some recursive pattern */) {
    args = /* right-side of recursive pattern */;
  }
  return /* right-side of base case pattern */;
};
```

Last Element

last(nil)	:= undefined	base cases
last(x :: nil)	:= x	
last(x :: y :: L)	:= last(y :: L)	tail recursive case

- **Translate to a loop:**

```
// @param S a non-empty list
int last(List S) {
  // Inv: last(S₀) = last(S)
  while (S != null && S.tl != null) {
    S = S.tl;
  }
  return /* right-side of base case pattern */;
};
```

Last Element

$$\begin{aligned} \text{last}(\text{nil}) &:= \text{undefined} \\ \text{last}(x :: \text{nil}) &:= x \end{aligned} \Bigg] \text{ base cases}$$

$$\text{last}(x :: y :: L) := \text{last}(y :: L) \Big] \text{ tail recursive case}$$

- **Translate to a loop:**

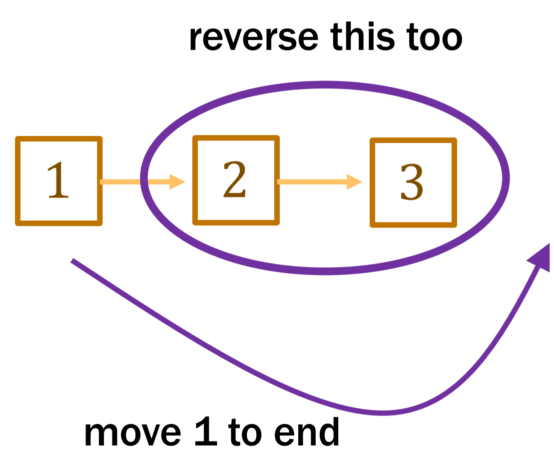
```
// @param S a non-empty list
int last(List S) {
  // Inv: last(S0) = last(S)
  while (S != null && S.tl != null) {
    S = S.tl;
  }
  if (S == null)
    throw new Error("no last element!");
  return S.hd;
};
```

Reversing a List

- **Mathematical definition of** $\mathrm{rev}(S)$

$$
\begin{aligned}
\mathrm{rev}(\mathrm{nil}) &:= \mathrm{nil} \\
\mathrm{rev}(x :: L) &:= \mathrm{rev}(L) \mathbin{+\!\!+} [x]
\end{aligned}
$$

  - **note that** rev **uses** concat $(\mathbin{+\!\!+})$ **as a helper function**



Reversing a List (Slowly)

$$\begin{aligned} \mathrm{rev}(\mathrm{nil}) &:= \mathrm{nil} \\ \mathrm{rev}(x :: L) &:= \mathrm{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where n is length of L
  - n calls to concat takes $\Theta(n^2)$ time

- Can we do this faster?
  - yes, but we need a helper function

Reversing a List Quickly

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$
\begin{aligned}
\text{rev-acc}(\text{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}
$$

$$\text{rev-acc}\left(\boxed{1} \to \boxed{2} \to \boxed{3} \to \text{nil}, \ \text{nil}\right)$$

Reversing a List Quickly

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$
\begin{aligned}
\text{rev-acc}(\text{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}
$$

$$
\begin{aligned}
&\text{rev-acc}\left(\boxed{1} \to \boxed{2} \to \boxed{3} \to \text{nil},\ \text{nil}\right) \\
&= \text{rev-acc}\left(\boxed{2} \to \boxed{3} \to \text{nil},\ \boxed{1} \to \text{nil}\right)
\end{aligned}
$$

Reversing a List Quickly

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$
\begin{aligned}
\text{rev-acc}(\text{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}
$$

$$
\begin{aligned}
&\text{rev-acc}\left(\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \text{nil},\ \text{nil}\right) \\
&= \text{rev-acc}\left(\boxed{2} \rightarrow \boxed{3} \rightarrow \text{nil},\ \boxed{1} \rightarrow \text{nil}\right) \\
&= \text{rev-acc}\left(\boxed{3} \rightarrow \text{nil},\ \boxed{2} \rightarrow \boxed{1} \rightarrow \text{nil}\right)
\end{aligned}
$$

Reversing a List Quickly

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\begin{aligned}
\text{rev-acc}(\text{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}$$

$$\begin{aligned}
&\text{rev-acc}\left(\boxed{1} \to \boxed{2} \to \boxed{3} \to \text{nil}\ ,\ \text{nil}\right) \\
&= \text{rev-acc}\left(\boxed{2} \to \boxed{3} \to \text{nil}\ ,\ \boxed{1} \to \text{nil}\right) \\
&= \text{rev-acc}\left(\boxed{3} \to \text{nil}\ ,\ \boxed{2} \to \boxed{1} \to \text{nil}\right) \\
&= \text{rev-acc}\left(\text{nil}\ ,\ \boxed{3} \to \boxed{2} \to \boxed{1} \to \text{nil}\right)
\end{aligned}$$

Reversing a List Quickly

$$\begin{aligned}
\mathrm{rev}(\mathrm{nil}) &:= \mathrm{nil} \\
\mathrm{rev}(x :: L) &:= \mathrm{rev}(L) \mathbin{+\!\!+} [x]
\end{aligned}$$

$$\begin{aligned}
\text{rev-acc}(\mathrm{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}$$

- **Second, also works, with** $\mathrm{rev}(L) = \text{rev-acc}(L, \mathrm{nil})$

Implementing rev-acc

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **Tail-recursive function becomes a loop:**

```
// Inv: rev-acc(S0, R0) = rev-acc(S, R)
while (S != null) {
  R = cons(S.hd, R);
  S = S.tl;
}
return R;
```

- **Now, use this to calculate** $\text{rev}(S) = \text{rev-acc}(S, \text{nil})$

Loop Version of rev-acc

rev-acc(nil, R) := R
rev-acc(x :: L, R) := rev-acc(L, x :: R)

- **Calculate** rev(S) **with loop:**

```
List rev(List S) {
  let R = nil;
  // Inv: rev-acc(S₀, R₀) = rev-acc(S, R)
  while (S != null) {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
}
```

Can see this calculates rev-acc(S_0, nil)

How do we know this is rev(S)**?**

Recall: Sum of the Values in a List

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) \quad &:= 0 \\ \text{sum}(x :: L) \quad &:= x + \text{sum}(L) \end{aligned}$$

- **Loop to calculate sum of a list**

$\{\{\ L = L_0\ \}\}$

```
int s = 0;
```

$\{\{$ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)\ \}\}$

```
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{\ s = \text{sum}(L_0)\ \}\}$

Why is this invariant not just $\text{sum-acc}(L_0, s_0) = \text{sum-acc}(L, s)$**?**

Recall: Length of a List with Tail Recursion

$$\begin{aligned} \text{len-acc}(\text{nil}, r) &:= r \\ \text{len-acc}(x :: L, r) &:= \text{len-acc}(L, r + 1) \end{aligned}$$

- **Could implement** len **with a loop as:**

```
int len(List S) {
  int r = 0;
  {{ Inv: len(S_0) = r + len(S) }}
  while (S != null) {
    r = r + 1;
    S = S.tl;
  }
  return r;
};
```

Why is this the invariant not just $\text{len-acc}(L_0, r_0) = \text{len-acc}(L, r)$**?**

Faster Len

$$\text{len}(\text{nil}) \quad := 0$$
$$\text{len}(x :: L) \quad := 1 + \text{len}(L)$$

$$\text{len-acc}(\text{nil}, r) \quad := r$$
$$\text{len-acc}(x :: L, r) \quad := \text{len-acc}(L, r + 1)$$

- **Both versions are recursive and $O(n)$ time**
  - **<u>second</u> version is tail recursive**
- **Can show that $\text{len-acc}(S, r) = \text{len}(S) + r$**
  - **tells us that $\text{len-acc}(S, 0) = \text{len}(S)$**

Relating Tail Recursion to Ordinary Recursion

- Need to understand the relationship between these functions

- With previous examples, the relationships are:

$$\text{len-acc}(S, r) = r + \text{len}(S)$$
$$\text{sum-acc}(S, r) = r + \text{sum}(S)$$

  - need to explain the role of the "accumulator variable" also
  - substituting this gives the invariant from the code

- How do we prove this?

Reversing a List Quickly

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!\!+} [x] \end{aligned}$$

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **The general relationship between the two is this:**

$$\text{rev-acc}(S, R) = \text{rev}(S) \mathbin{+\!\!\!+} R$$

- **Can now substitute this into the invariant…**

Loop Version of rev-acc

$$\text{rev-acc}(\text{nil}, R) \quad := \; R$$
$$\text{rev-acc}(x :: L, R) \quad := \; \text{rev-acc}(L, x :: R)$$

- **Calculate** $\text{rev}(S)$ **with loop:**

```
List rev(List S) {
  List R = null;
  // Inv: rev(S₀) ++ R₀ = rev(S) ++ R
  while (S != null) {
    R = cons(S.hd, R);
    R = R.tl;
  }
  return R;
}
```

We know $R_0 = []$

And $\text{rev}(S) \mathbin{+\!\!+} [] = \text{rev}(S)$

Loop Version of rev-acc

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **Calculate** rev(S) **with loop:**

```
List rev(List S) {
  List R = null;
  // Inv: rev(S₀) = rev(S) ++ R
  while (S != null) {
    R = cons(S.hd, R);
    R = R.tl;
  }
  return R;
}
```

Options for proving **correctness**:

- Prove relationship btw two recursive functions. Then, implement tail recursion with template.
- Prove loop correct with Floyd logic.