

COMP251: proofs

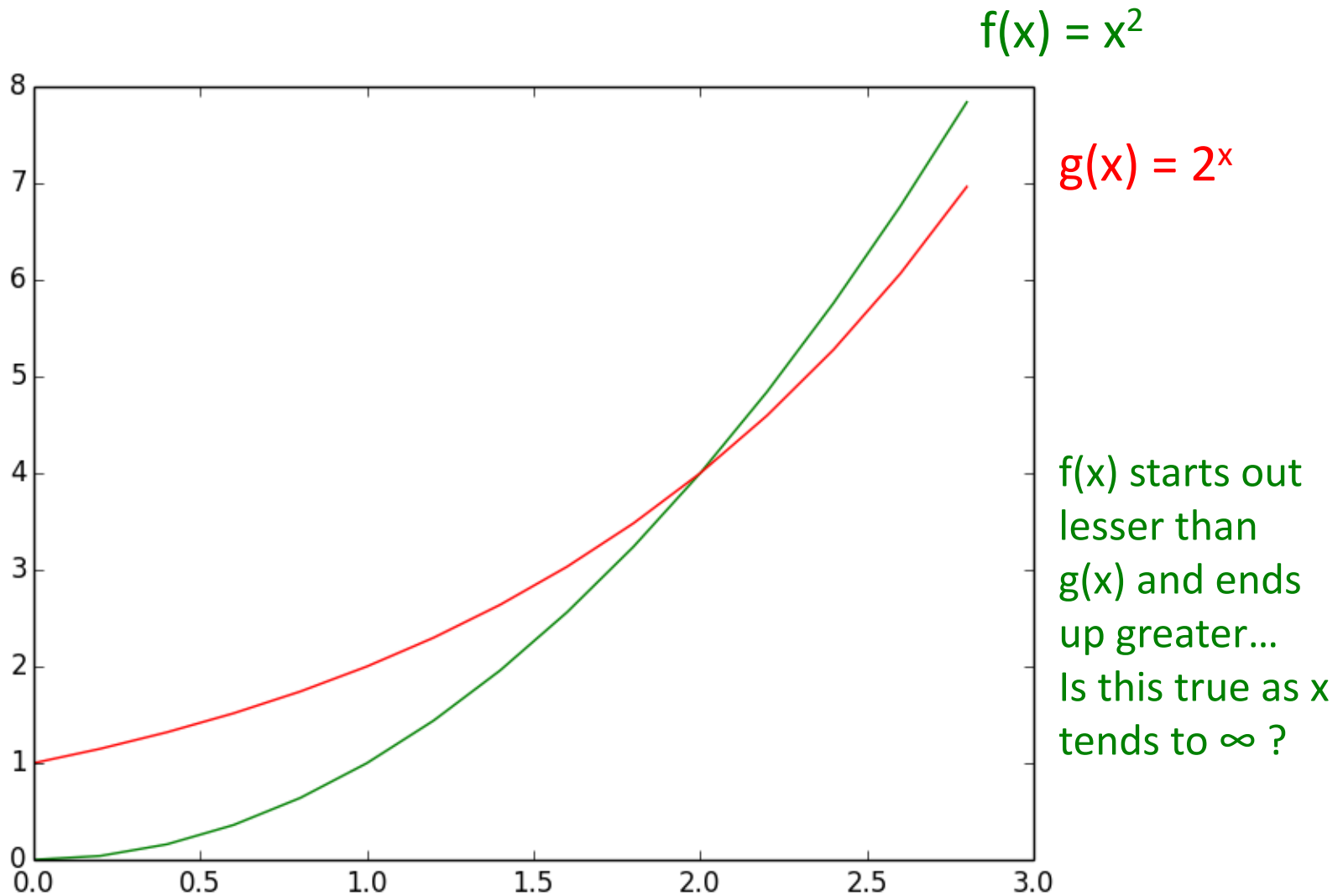
Jérôme Waldispühl & Giulia Alberini
School of Computer Science
McGill University

Based on slides from (Langer,2012), (CRLS, 2009) &
(Sora,2015)

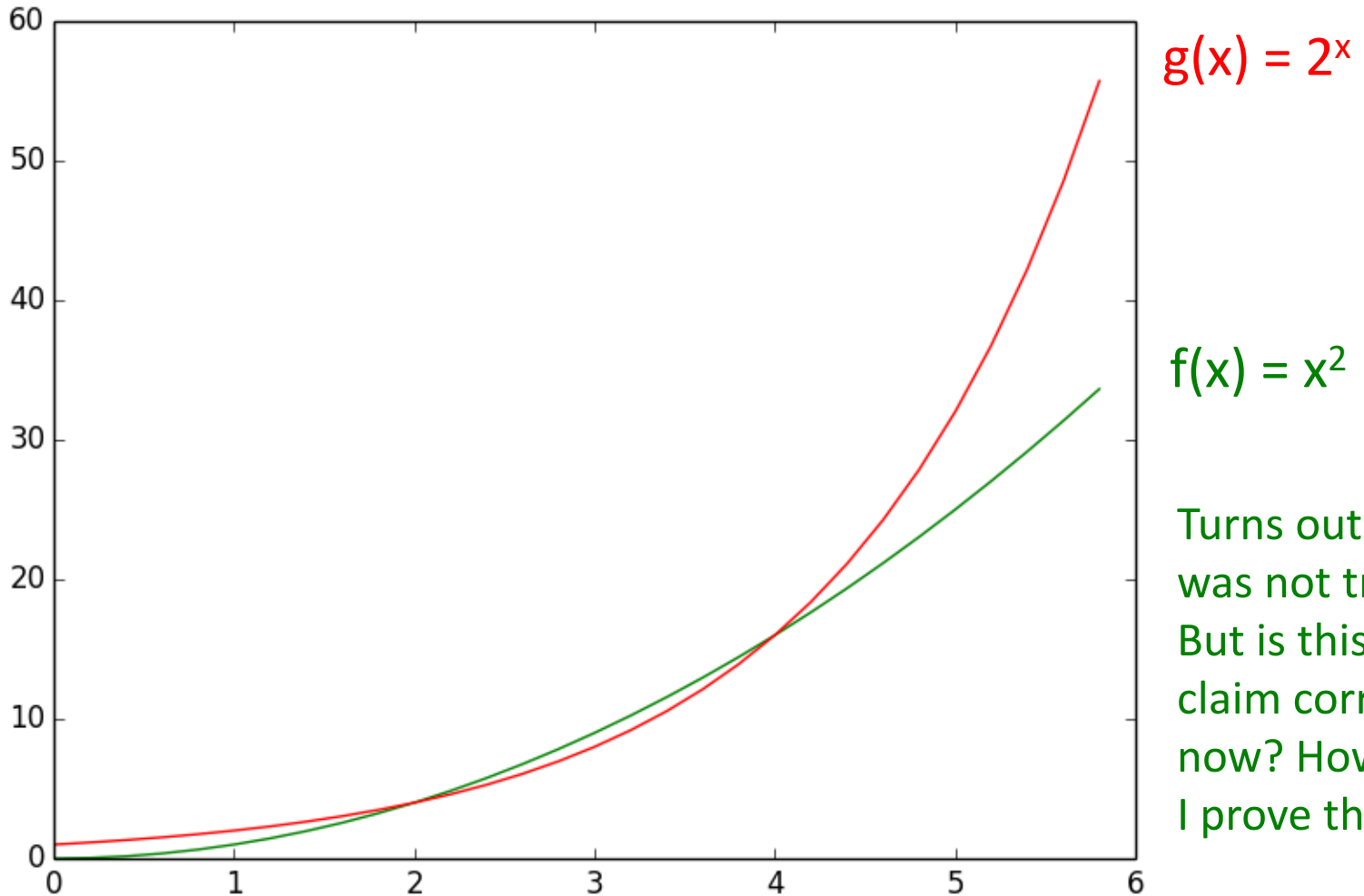
Outline

- **Induction proofs**
 - Introduction
 - Definition
 - Examples
- **Loop invariants**
 - Definition
 - Example (Insertion sort)
- **Recursive algorithms**
 - Analogy with induction proofs
 - Example (Merge sort)

for any $n \geq 2$, $n^2 \geq 2^n$?



for any $n \geq 5$, $n^2 \leq 2^n$?



$g(x) = 2^x$

$f(x) = x^2$

Turns out that was not true!
But is this new claim correct, now? How can I prove that?

Motivation

How to prove these?

$$\text{for any } n \geq 1, \quad 1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n + 1)}{2}$$

$$\text{for any } n \geq 1, \quad 1 + 3 + 5 + 7 + \cdots + (2 \cdot n - 1) = n^2$$

$$\text{for any } n \geq 5, \quad n^2 \leq 2^n$$

And in general, any statement of the form:

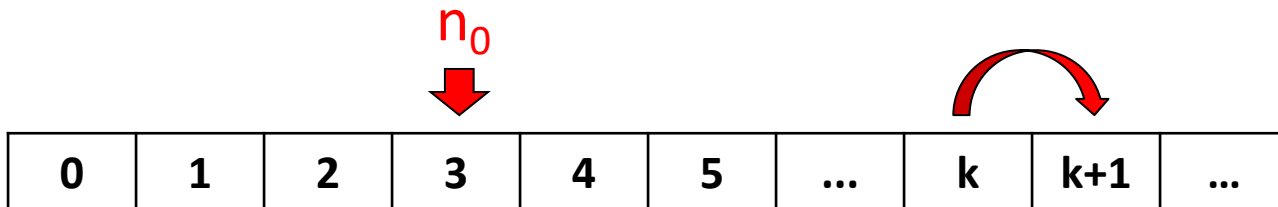
“for all $n \geq n_0$, $P(n)$ ” where $P(n)$ is some proposition.

Mathematical induction

Many statement of the form “*for all $n \geq n_0$, $P(n)$* ” can be proven with a logical argument called *mathematical induction*.

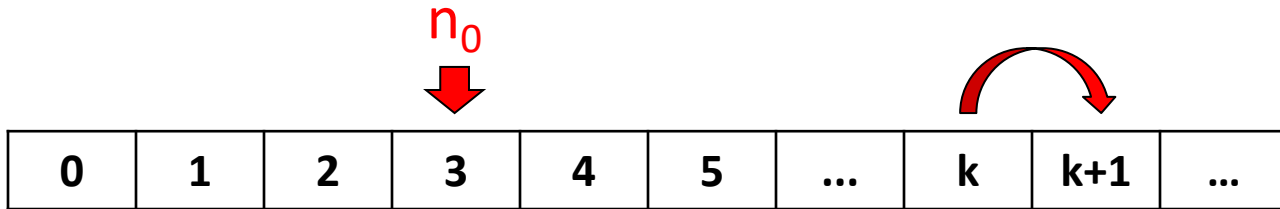
The proof has two components:

- **Base case:** $P(n_0)$
- **Induction step:** for any $n \geq n_0$, if $P(n)$ then $P(n+1)$

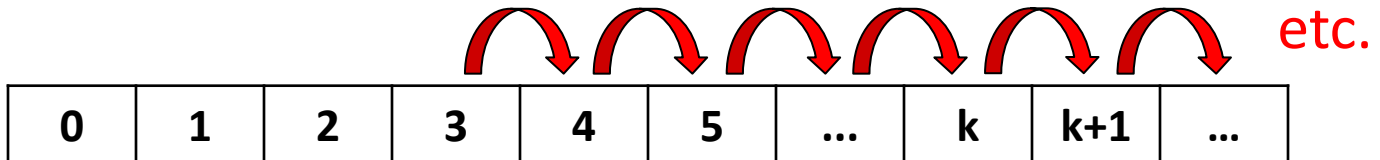


Is this sufficient to prove something is true between n_0 and very large values of n ?

Principle



Implies



Example 1

Claim: *for any* $n \geq 1$, $1 + 2 + 3 + 4 + \dots + n = \frac{n \cdot (n + 1)}{2}$

Proof:

• Base case: $n = 1$ $1 = \frac{1 \cdot 2}{2}$ ✓

• Induction step:

for any $k \geq 1$, *if* $1 + 2 + 3 + 4 + \dots + k = \frac{k \cdot (k + 1)}{2}$

then $1 + 2 + 3 + 4 + \dots + k + (k + 1) = \frac{(k + 1) \cdot (k + 2)}{2}$

Example 1

Claim: *for any* $n \geq 1$, $1 + 2 + 3 + 4 + \dots + n = \frac{n \cdot (n + 1)}{2}$

Proof:

- Base case: $n = 1$ $1 = \frac{1 \cdot 2}{2}$ ✓

- Induction step:

for any $k \geq 1$, *if* $1 + 2 + 3 + 4 + \dots + k = \frac{k \cdot (k + 1)}{2}$

then $1 + 2 + 3 + 4 + \dots + k + (k + 1) = \frac{(k + 1) \cdot (k + 2)}{2}$

Example 1


Assume $1 + 2 + 3 + 4 + \dots + k = \frac{k \cdot (k + 1)}{2}$

then $1 + 2 + 3 + 4 + \dots + k + (k + 1)$

$$= \frac{k \cdot (k + 1)}{2} + (k + 1)$$

$$= \frac{k \cdot (k + 1) + 2 \cdot (k + 1)}{2}$$

$$= \frac{(k + 2) \cdot (k + 1)}{2}$$

 Induction hypothesis

We substitute 1...k in the left-side expression to resolve

Summary

Base case: $P(1)$ ✓

Induction step: for any $k \geq 1$, if $P(k)$ then $P(k+1)$ ✓

Thus for all $n \geq 1$, $P(n)$ □

The proposition is true!

Example 2

Claim: *for any* $n \geq 1$, $1 + 3 + 5 + 7 + \cdots + (2 \cdot n - 1) = n^2$

Proof:

- Base case: $n = 1$ $1 = 1^2$



- Induction step:

for any $k \geq 1$, *if* $1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) = k^2$

then $1 + 3 + 5 + 7 + \cdots + (2 \cdot (k + 1) - 1) = (k + 1)^2$

Example 2

Assume $1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) = k^2$

then


$$1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) + (2 \cdot (k + 1) - 1)$$

$$= k^2 + 2 \cdot (k + 1) - 1$$

$$= k^2 + 2 \cdot k + 1$$

$$= (k + 1)^2$$

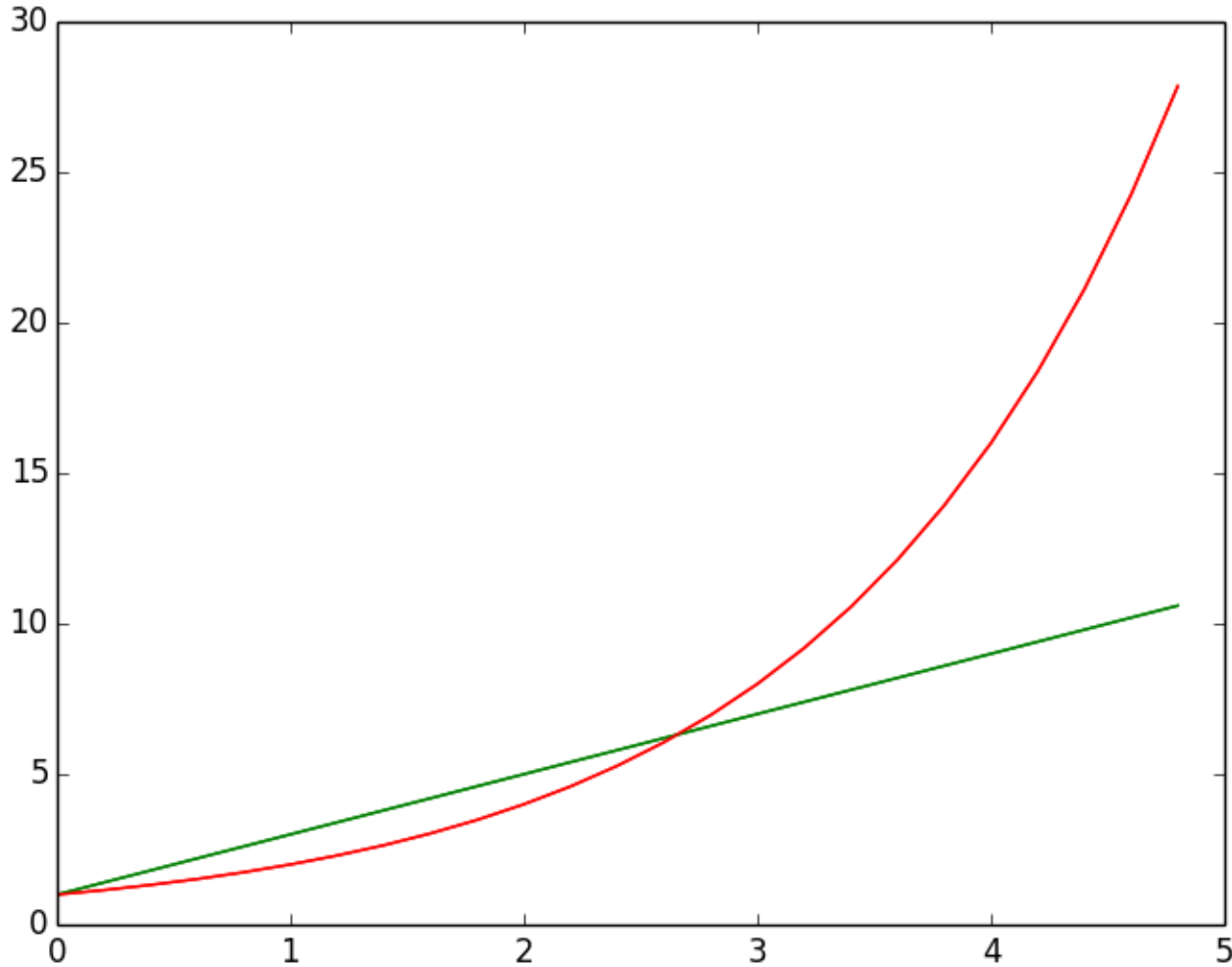
Induction
hypothesis



What if instead we
have inequalities?

Example 3

for any $n \geq 3$, $2 \cdot n + 1 < 2^n$



$$g(x) = 2^x$$

$$f(x) = 2x + 1$$

Example 3

Claim: $\text{for any } n \geq 3, \quad 2 \cdot n + 1 < 2^n$

Proof:

- Base case: $n = 3 \quad 2 \cdot 3 + 1 = 7 < 2^3 = 8$



- Induction step:

$\text{for any } k \geq 3, \quad \text{if } 2 \cdot k + 1 < 2^k$

$\text{then } 2 \cdot (k + 1) + 1 < 2^{k+1}$

Example 3

Assume $2 \cdot k + 1 < 2^k$


then $2 \cdot (k + 1) + 1$

$$= 2 \cdot k + 2 + 1$$

$$< 2^k + 2$$

$$\leq 2^k + 2^k \quad \text{for } k \geq 1$$

$$= 2^{k+1}$$

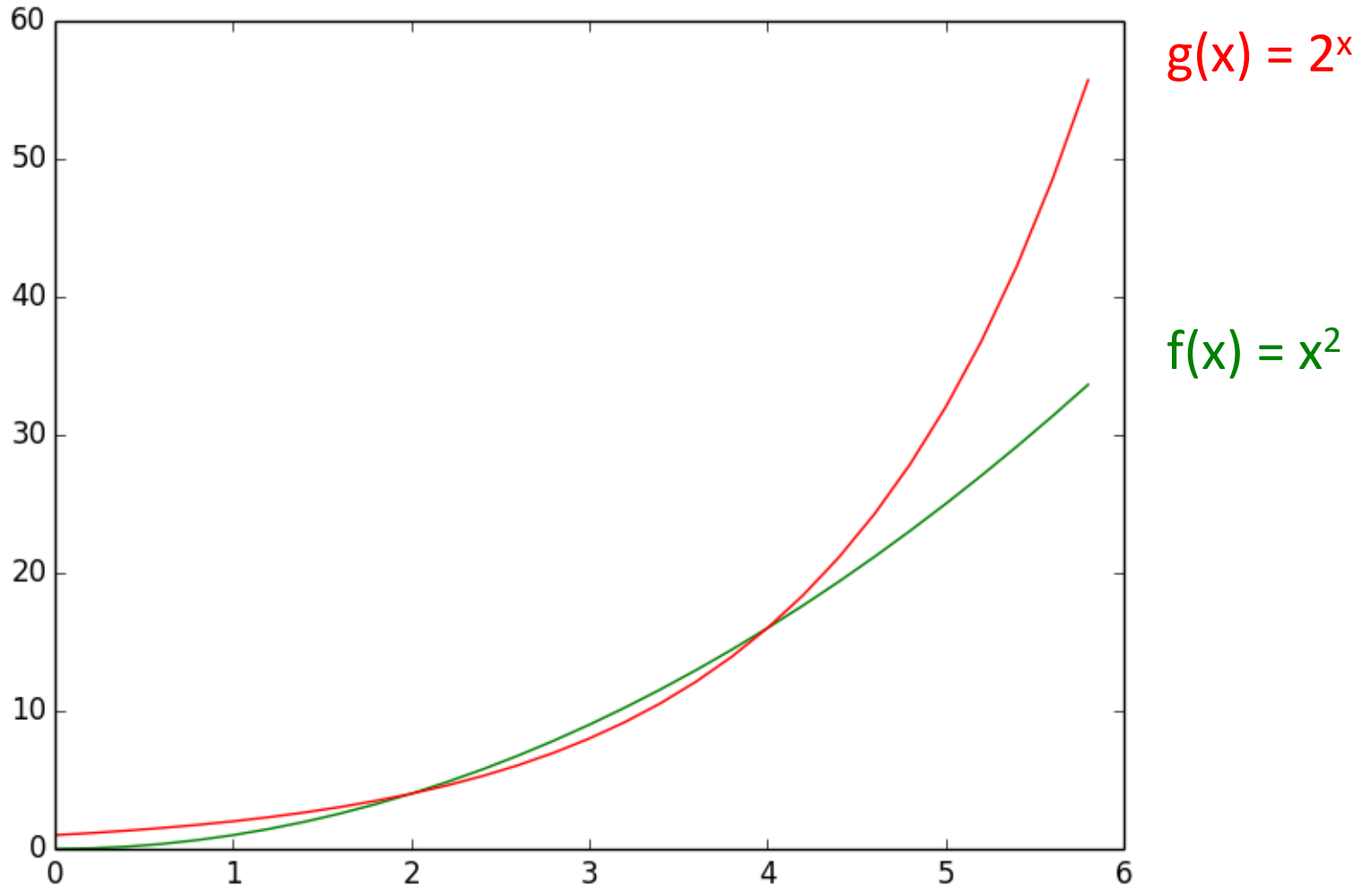
 Induction hypothesis

Here we can use the fact that $2 \leq$ a bigger, more convenient value

Stronger than we need, but that works!

Example 4

for any $n \geq 5$, $n^2 \leq 2^n$



Example 4

Claim: *for any $n \geq 5$, $n^2 \leq 2^n$*

Proof:

• Base case: $n = 5$ $25 \leq 32$ ✓

• Induction step:

*for any $k \geq 5$, if $k^2 \leq 2^k$
then $(k+1)^2 \leq 2^{k+1}$*

Example 4

Assume $k^2 \leq 2^k$

then $(k+1)^2$

$$= k^2 + 2 \cdot k + 1$$

$$\leq 2^k + 2 \cdot k + 1$$

$$\leq 2^k + 2^k$$

$$= 2^{k+1}$$



Induction hypothesis

From previous example

Remember example 3!

Example 5

This also works with recursively defined functions!

Fibonacci sequence:

$\text{Fib}_0 = 0$ base case

$\text{Fib}_1 = 1$ base case

$\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$ for $n > 1$ recursive case

Claim: For all $n \geq 0$, $\text{Fib}_n < 2^n$

Base case: $\text{Fib}_0 = 0 < 2^0 = 1$, $\text{Fib}_1 = 1 < 2^1 = 2$

Q: Why should we check both Fib_0 and Fib_1 ?

Induction step: for any $i \leq k$, if $\text{Fib}_i < 2^i$ then $\text{Fib}_{k+1} < 2^{k+1}$

Example 5

Assume that *for all $i \leq k$, $Fib_i < 2^i$* (Note variation of induction hypothesis)

$$\begin{aligned} \text{Then } Fib_{k+1} &= Fib_k + Fib_{k-1} \\ &< 2^k + 2^{k-1} \\ &\leq 2^k + 2^k \\ &= 2^{k+1} \end{aligned}$$



Induction hypothesis (x2)

for $k \geq 1$

Strong induction assumes the hypothesis is true for all ranks until k , not just for k .

Remember example 3 where we substituted a greater, more convenient value.

Proving the correctness of an algorithm

LOOP INVARIANTS

Algorithm specification

- An algorithm is described by:
 - Input data
 - Output data
 - ***Pre-conditions***: specifies restrictions on input data
 - ***Post-conditions***: specifies what is the result
- Example: Binary Search
 - Input data: `a:array of integer; x:integer;`
 - Output data: `index:integer;`
 - Precondition: `a` is sorted in ascending order
 - Postcondition: `index` of `x` if `x` is in `a`, and `-1` otherwise.

Correctness of an algorithm

An algorithm is correct if:

- for any correct input data:
 - it stops and
 - it produces correct output.
- Correct input data: satisfies pre-condition
- Correct output data: satisfies post-condition

Problem: Proving the correctness of an algorithm may be complicated when it is repetitive or contains loop instructions.

How to prove the correctness of an algorithm?

- Recursive algorithm \Rightarrow Induction proofs
- Iterative algorithm (loops) \Rightarrow ???

Loop invariant

A **loop invariant** is a loop property that hold before and after each iteration of a loop.

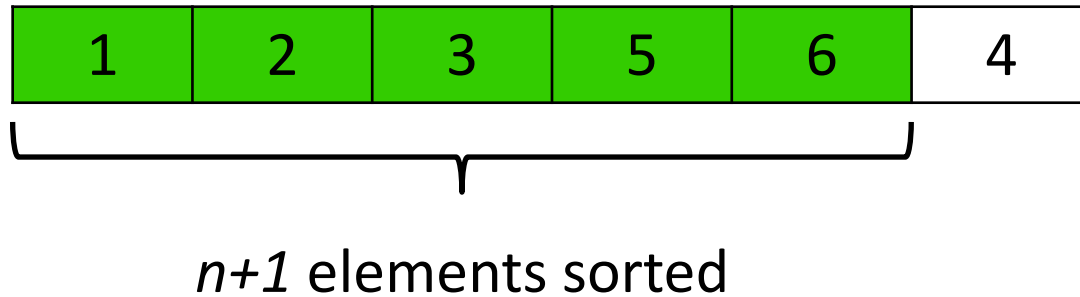
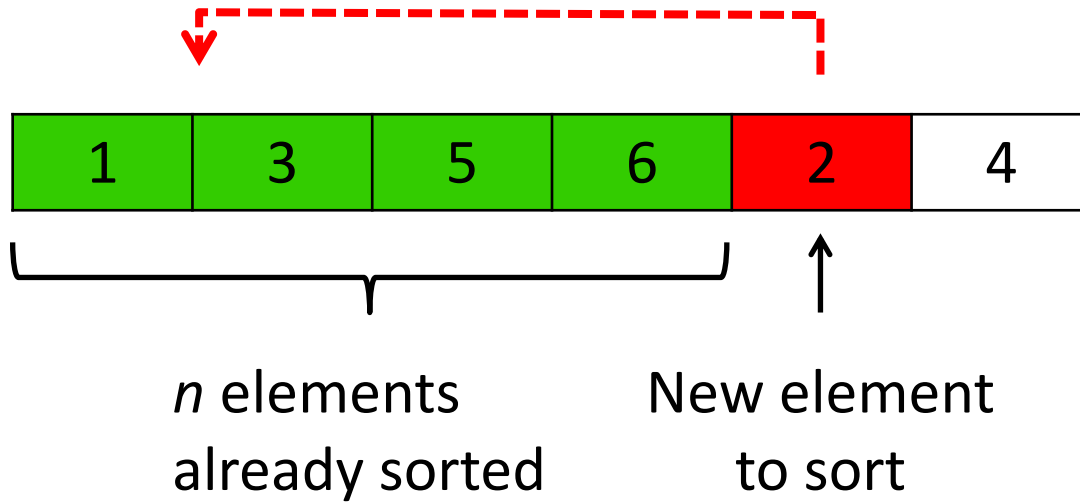
Preferably, a property that will help to show that the algorithm compute what it is expected to do.

Insertion sort

```
for i ← 1 to length(A) - 1
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  end while
end for
```

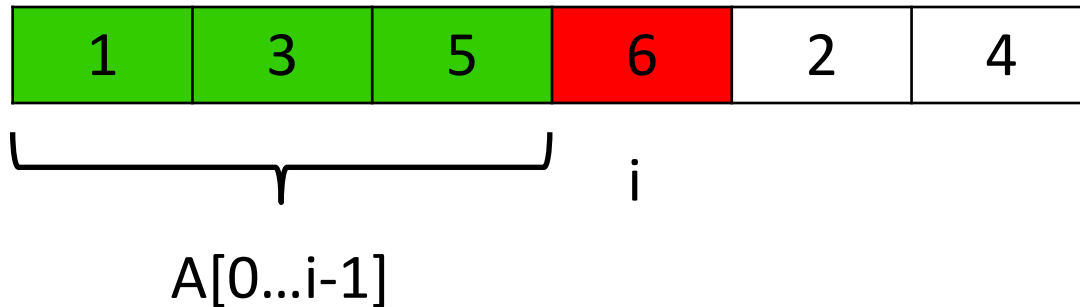
(Seen in previous lecture)

Insertion sort



Loop invariant

The array $A[0\dots i-1]$ is fully sorted.



In other words:

At the time of insert i into the pre-sorted list, every element to the left of i is already sorted

Overview of the proof

①

```
for i ← 1 to length(A) - 1
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  end while
```

②

③

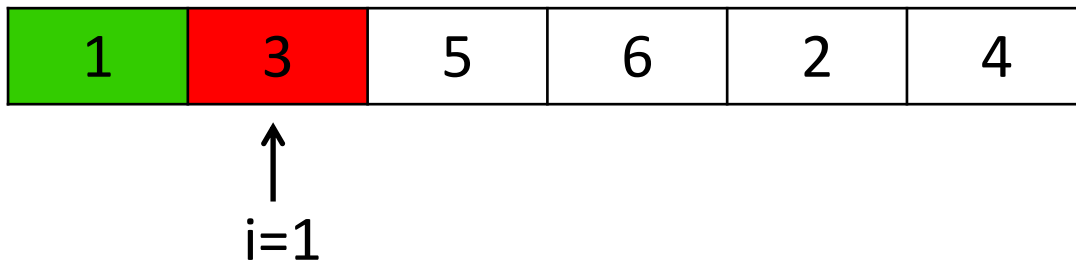
```
end for
```

Once you established the loop invariant property, you want to check that:

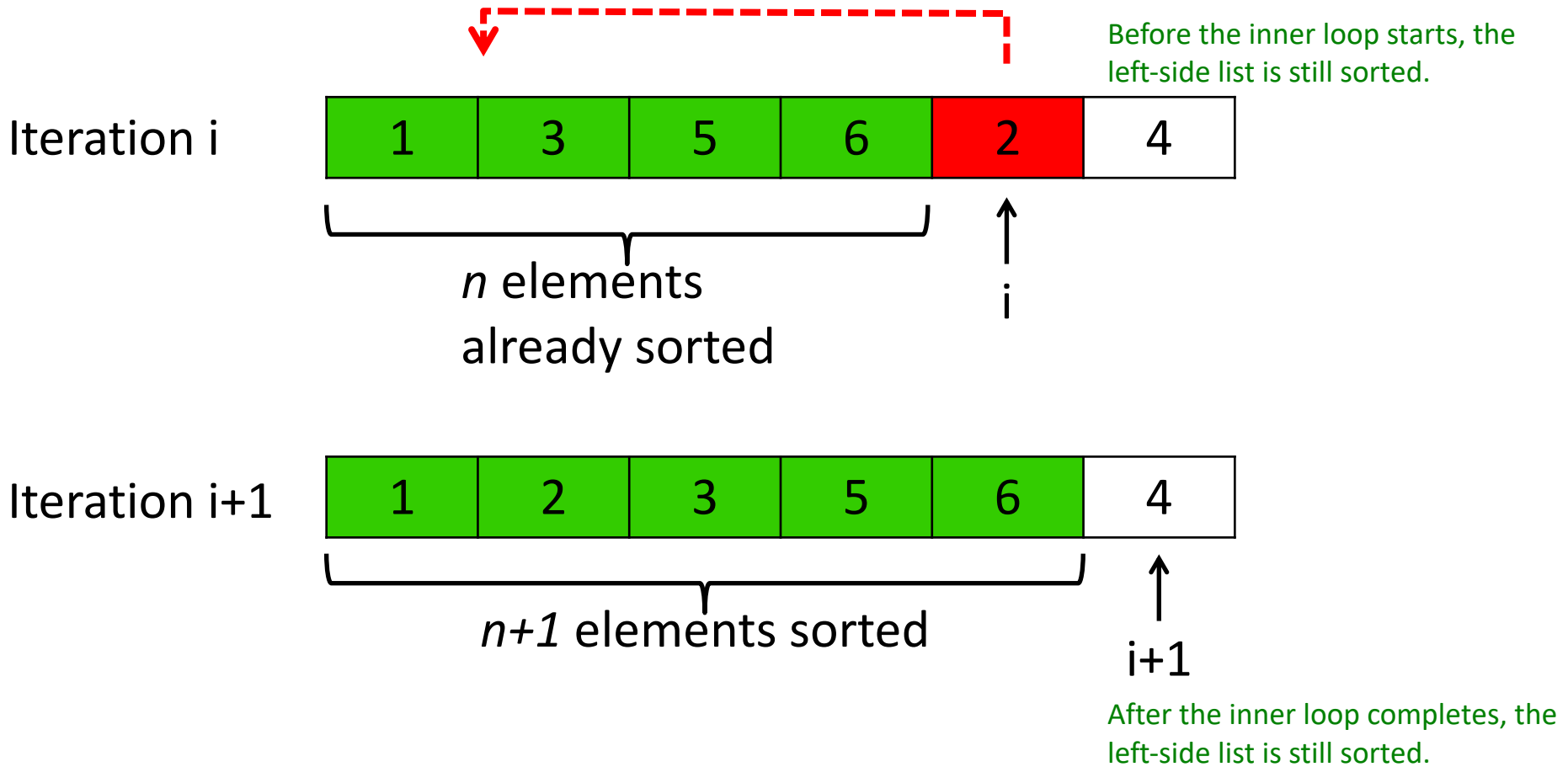
1. The property holds true at the beginning
2. The instruction block within the loop restore the property when we add one more element
3. The loop terminates and all elements are sorted.

Initialization

Just before the first iteration ($i = 1$), the sub-array $A[0 \dots i-1]$ is the single element $A[0]$, which is the element originally in $A[0]$, and it is trivially sorted.



Maintenance



Note: To be self-content, we would also need to state and prove a loop invariant for the “inner” **while** loop.

Termination

The outer **for** loop ends when $i \geq \text{length}(A)$ and increment by 1 at each iteration starting from 1.

Therefore, at exit $i = \text{length}(A)$.

By the loop invariant property, the sub-array $A[0 \dots \text{length}(A)-1]$ consists of the elements originally in $A[0 \dots \text{length}(A)-1]$ but in sorted order.

$A[0 \dots \text{length}(A)-1]$ contains $\text{length}(A)$ elements (i.e., ***all initial elements!***) and no element is duplicated/deleted.

In other words, **the entire array is sorted.**

We need to show that the algorithm terminates and respects the postcondition. Here, the loop invariant helps us convince ourselves that the output is correct.

Proof using loop invariants

We must show:

- 1. Initialization:** It is true prior to the first iteration of the loop.
- 2. Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- 3. Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

We want a property that, if we show it is respected throughout the execution, will help us show the output is correct. The difficulty is to find the right property!

Proving the correctness of a recursive algorithm

LOOP INVARIANTS AND INDUCTION PROOFS

Analogy to induction proofs

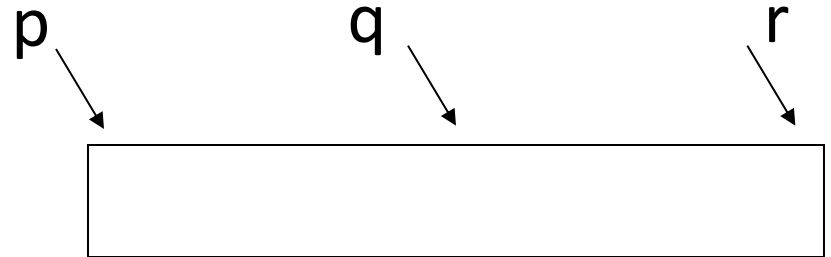
Using loop invariants is like mathematical induction.

- You prove a **base case** and an **inductive step**.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The **termination** part differs from classical mathematical induction. Here, we stop the “induction” when the loop terminates instead of using it infinitely.

We can show the three parts in any order.

Merge Sort

```
MERGE-SORT (A, p, r)
  if p < r then
    q = (p+r) / 2
    MERGE-SORT (A, p, q)
    MERGE-SORT (A, q+1, r)
    MERGE (A, p, q, r)
```



Precondition:

Array A has at least 1 element between indexes p and r ($p \leq r$)

Postcondition:

The elements between indexes p and r are sorted

There are two proofs to make:

1. The recursive function MERGE-SORT (by induction)
2. The function MERGE (using loop invariants)

Merge method

- MERGE-SORT calls a function $\text{MERGE}(A, p, q, r)$ to merge the sorted subarrays of A into a single sorted one
- The proof of MERGE can be done separately, using loop invariants

MERGE (A, p, q, r)

Precondition: A is an array and p , q , and r are indices into the array such that $p \leq q < r$. The subarrays $A[p..q]$ and $A[q+1..r]$ are sorted

Postcondition: The subarray $A[p..r]$ is sorted

Procedure Merge

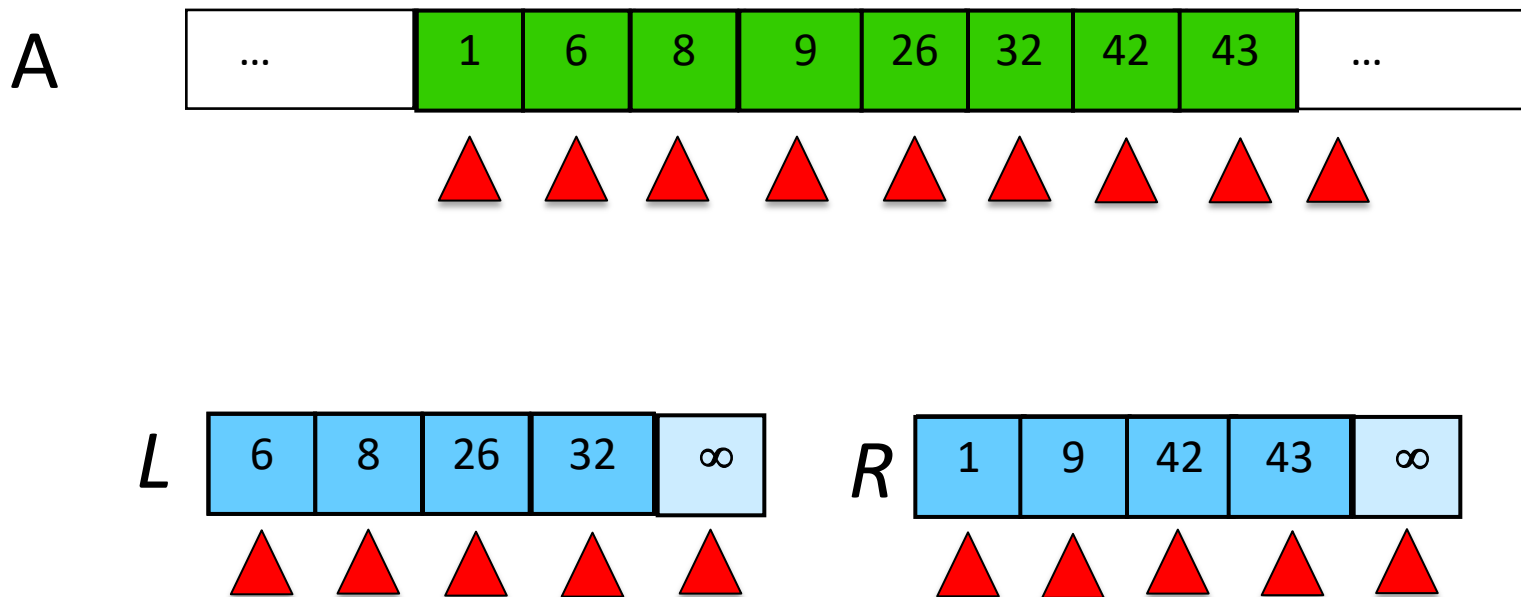
Merge(A, p, q, r)

```
1  $n_1 \leftarrow q - p + 1$ 
2  $n_2 \leftarrow r - q$ 
3   for  $i \leftarrow 1$  to  $n_1$ 
4     do  $L[i] \leftarrow A[p + i - 1]$ 
5   for  $j \leftarrow 1$  to  $n_2$ 
6     do  $R[j] \leftarrow A[q + j]$ 
7    $L[n_1 + 1] \leftarrow \infty$ 
8    $R[n_2 + 1] \leftarrow \infty$ 
9    $i \leftarrow 1$ 
10   $j \leftarrow 1$ 
11  for  $k \leftarrow p$  to  $r$ 
12    do if  $L[i] \leq R[j]$ 
13      then  $A[k] \leftarrow L[i]$ 
14           $i \leftarrow i + 1$ 
15      else  $A[k] \leftarrow R[j]$ 
16           $j \leftarrow j + 1$ 
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Merge/combine – Example



Idea: The lists L and R are **already sorted**.

Correctness proof for Merge

- **Loop Invariant:** The array $A[p,k]$ stored the $(k-p+1)$ smallest elements of L and R sorted in increasing order.
- **Initialization:** $k = p$
 - A contains a single element (which is trivially “sorted”)
 - $A[p]$ is the smallest element of L and R
- **Maintenance:**
 - Assume that Merge satisfies the loop invariant property until k .
 - $(k-p+1)$ smallest elements of L and R are already sorted in A.
 - Next value to be inserted is the smallest one remaining in L and R.
 - This value is larger than those previously inserted in A.
 - Loop invariant property satisfied for $k+1$.
- **Termination Step:**
 - Merge terminates when $k=r$, thus when $r-p+1$ elements have been inserted in A \Rightarrow All elements are sorted.

The loop invariant we chose tells us something important about the algorithm and its postcondition!

Correctness proof for Merge Sort

- **Recursive property:** For $0 \leq k = r - p + 1$, $A[p,r]$ is sorted.
- **Base Case:** $p = r$
 - A contains a single element (which is trivially “sorted”)
- **Inductive Hypothesis:**
 - Assume that MergeSort correctly sorts $n=1, 2, \dots, k$ elements
- **Inductive Step:**
 - Show that MergeSort correctly sorts $n = k + 1$ elements.
- **Termination Step:**
 - MergeSort terminate and all elements are sorted.

Note: Merge Sort is a recursive algorithm. We already proved that the Merge procedure is correct. Here, we complete the proof of correctness of the main method using induction.

Inductive step

- **Inductive Hypothesis:**
 - Assume MergeSort correctly sorts $n=1, \dots, k$ elements
- **Inductive Step:**
 - Show that MergeSort correctly sorts $n = k + 1$ elements.
- **Proof:**
 - First recursive call $n_1=q-p+1=(k+1)/2 \leq k$ Our inductive hypothesis is that we can correctly store arrays smaller than k
 - \Rightarrow subarray $A[p \dots q]$ is sorted
 - Second recursive call $n_2=r-q=(k+1)/2 \leq k$
 - \Rightarrow subarray $A[q+1 \dots r]$ is sorted
 - A, p, q, r fulfill now the precondition of Merge
 - The post-condition of Merge guarantees that the array $A[p \dots r]$ is sorted \Rightarrow post-condition of MergeSort satisfied.

Termination Step

We have to show the size of the problem decreases with every recursive call: the length of the subarray of A to be sorted MergeSort.

At each recursive call of MergeSort, the length of the subarray is strictly decreasing.

When MergeSort is called on an array of size ≤ 1 (i.e. the base case), the algorithm terminates without making additional recursive calls.

Calling MergeSort(A,0,n) returns a fully sorted array.