COMP250: Induction proofs.

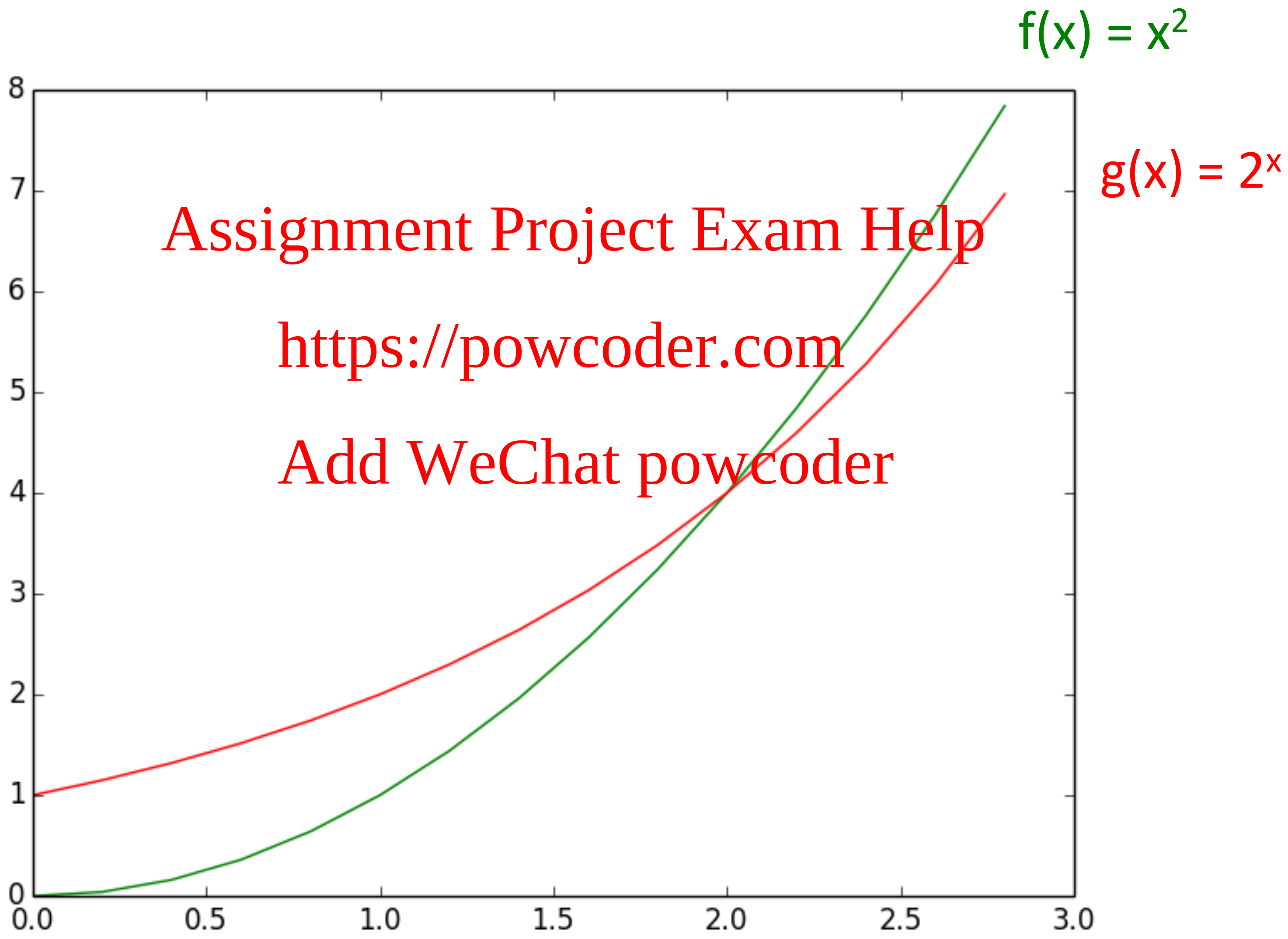
Jérôme Waldispühl
School of Computer Science
McGill University

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

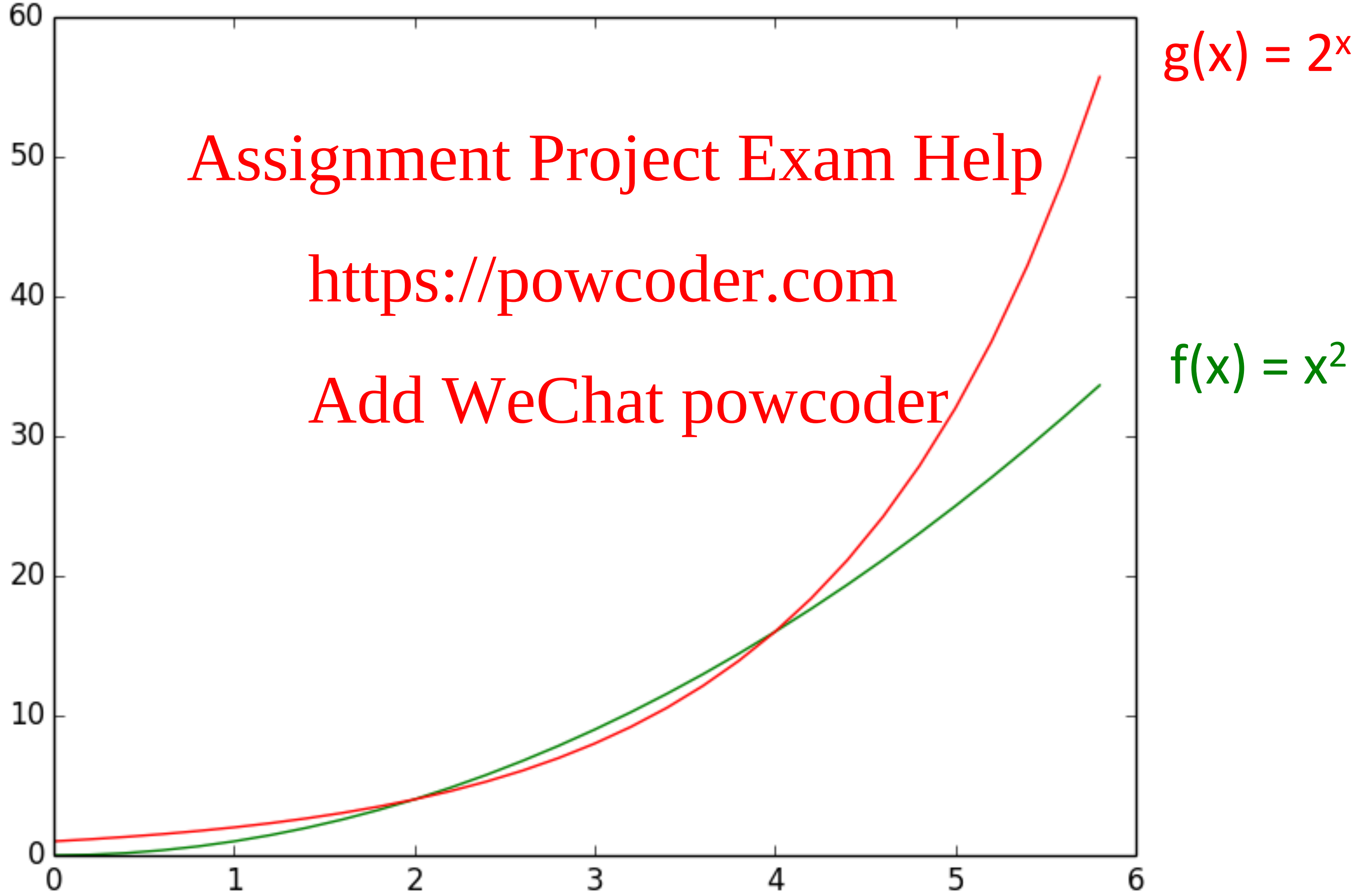
Outline

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

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



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



Motivation

How to prove these?

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

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

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

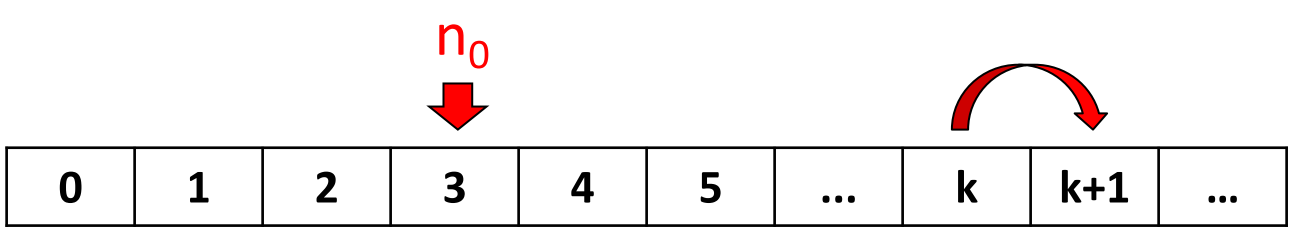
And in general, any statement of the form:
"for all n≥n_0 , P(n)" where *P(n) is some proposition.*

Mathematical induction

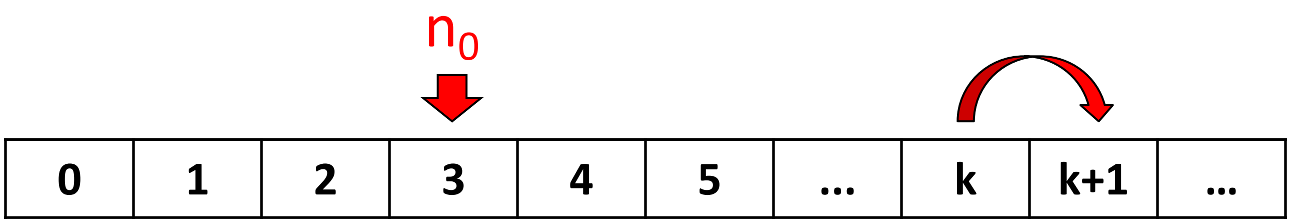
Many statement of the form *“for all $n \geq n_0$, $P(n)$”* can be proven with a logical argument call *mathematical induction.*

The proof has two components:

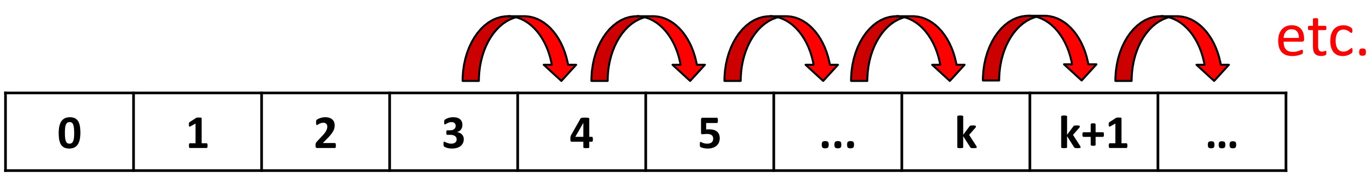
- **Base case**: $P(n_0)$
- **Induction step**: for any $n \geq n_0$, if $P(n)$ then $P(n+1)$



Principle



Implies



Example 1

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

Proof:

- Base case: $n = 1 \quad 1 = \dfrac{1\cdot 2}{2}$

- Induction step:

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

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

Example 1

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

$$\text{then} \quad 1+2+3+4+\cdots+k+(k+1)$$

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

Induction hypothesis

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

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

Summary

Base case: P(1)

Induction step: for any k≥1, if P(k) then P(k+1)

Thus for all n≥1, P(n) □

Example 2

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

Proof:

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

- Induction step:

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

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

Example 2

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

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

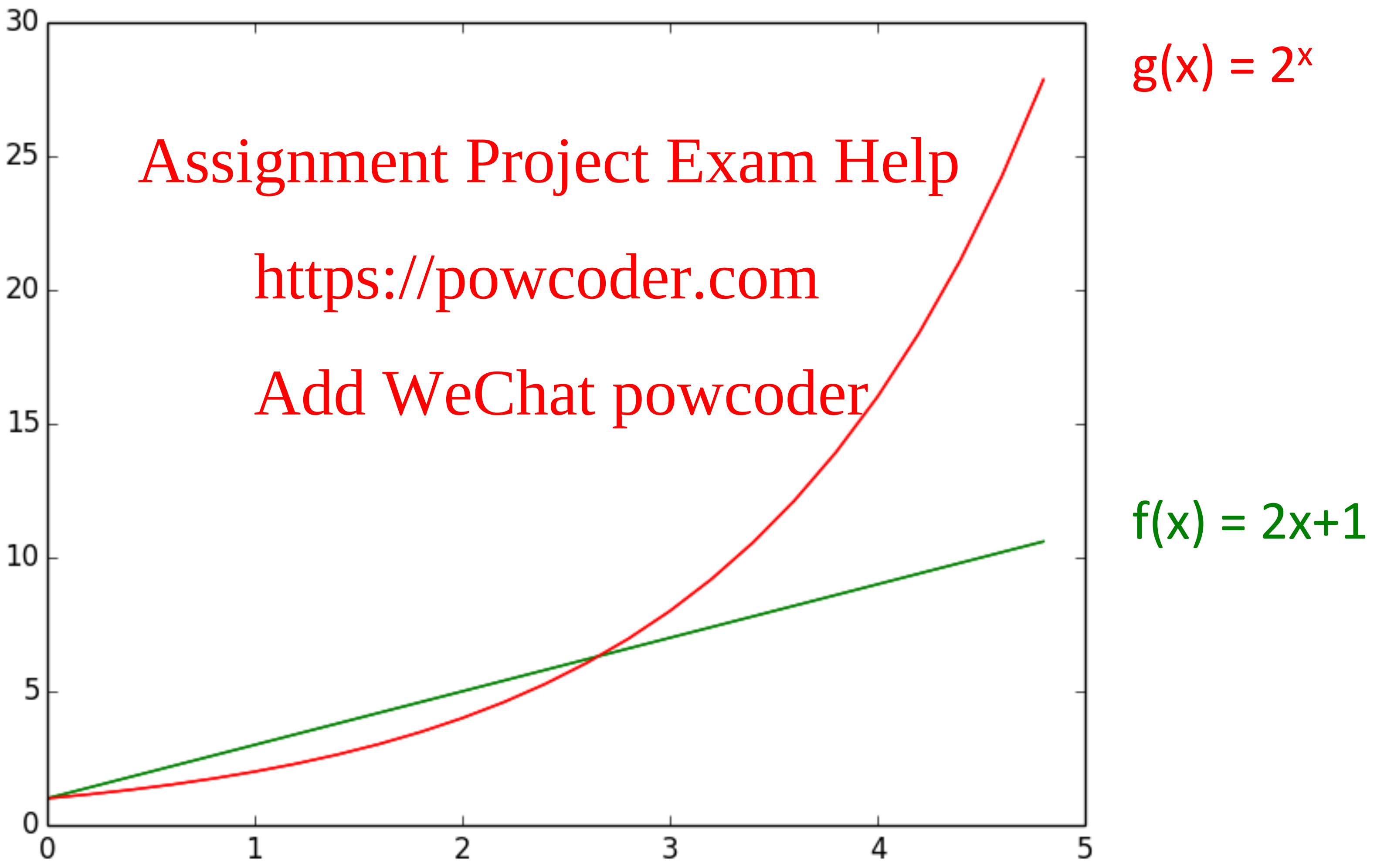
Induction hypothesis

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

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

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

Example 3



Example 3

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

Proof:

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

- Induction step:

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

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

Example 3

$$\textit{Assume} \quad 2 \cdot k + 1 < 2^k$$

$$\textit{then} \quad 2 \cdot (k+1) + 1$$

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

Induction hypothesis

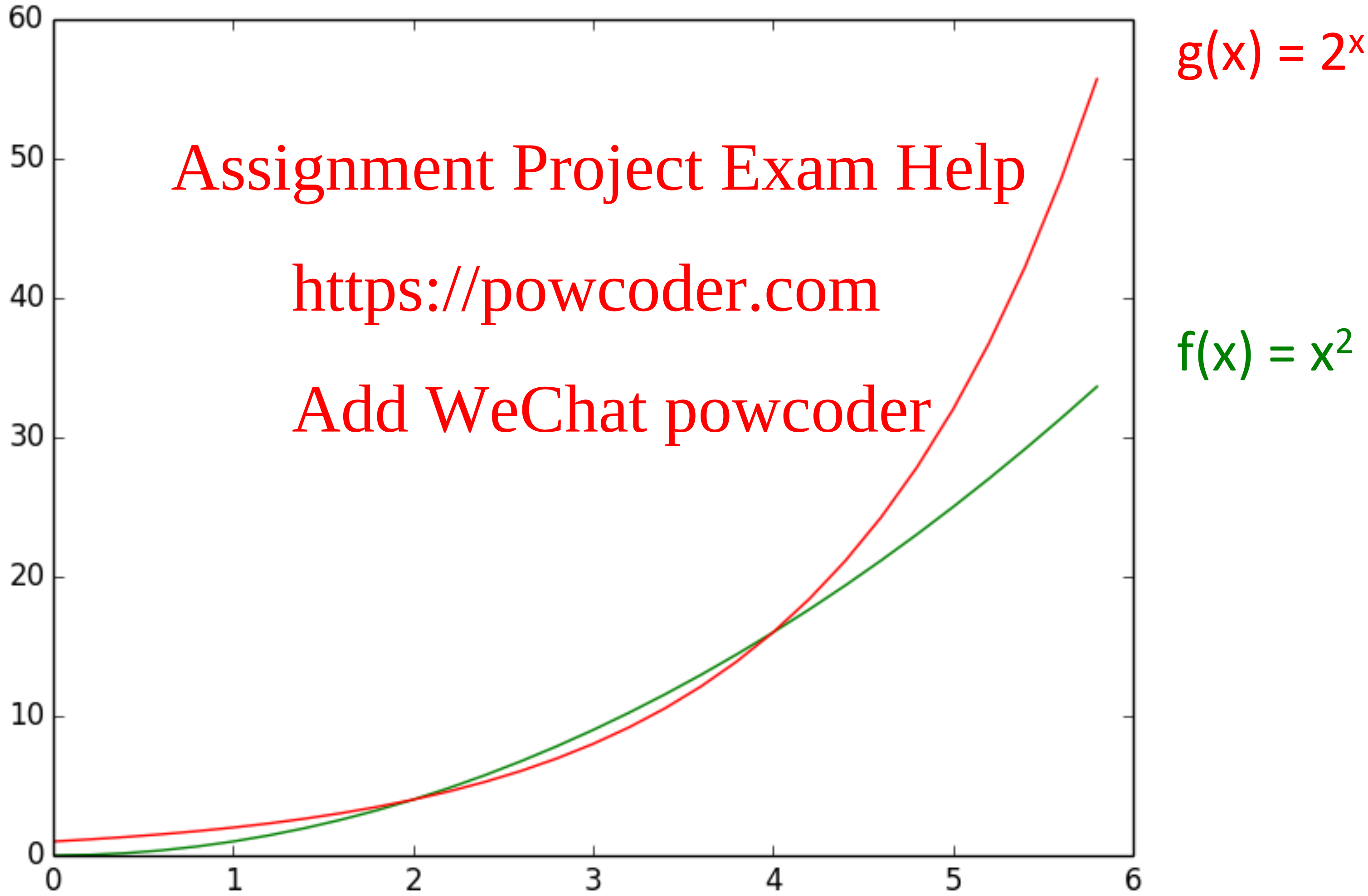
$$< 2^k + 2$$

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

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

Stronger than we need, but that works!

Example 4



Example 4

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

Proof:

- Base case: $n = 5 \qquad 25 \leq 32$

- Induction step:

$$for\ any\ k \geq 5, \quad if \quad k^2 \leq 2^n$$

$$then \quad (k+1)^2 \leq 2^{k+1}$$

Example 4

$$Assume \quad k^2 \le 2^k$$

$$then \quad (k+1)^2$$

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

Induction hypothesis

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

From previous example

$$\le 2^k + 2^k$$

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

Example 5

Fibonacci sequence:

$Fib_0 = 0$ base case

$Fib_1 = 1$ base case

$Fib_n = Fib_{n-1} + Fib_{n-2}$ for $n > 1$ recursive case

Claim: For all $n \geq 0$, $Fib_n < 2^n$

Base case: $Fib_0 = 0 < 2^0 = 1$, $Fib_1 = 1 < 2^1 = 2$

Q: Why should we check both Fib_0 and Fib_1?

Induction step: for any $i \leq k$, if $Fib_i < 2^i$ then $Fib_{k+1} < 2^{k+1}$

Example 5

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

Then $Fib_{k+1} = Fib_k + Fib_{k-1}$

$< \; 2^k + 2^{k-1}$ Induction hypothesis (x2)

$\leq \; 2^k + 2^k$ for $k \geq 1$

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

Proving the correctness of an algorithm

LOOP INVARIANTS

Algorithm specification

- An algorithm is described by:
  - Input data
  - Output data

  - ***Preconditions***: specifies restrictions on input data

  - ***Postconditions***: 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 precondition
- Correct output data: satisfies postcondition

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

How to prove the correctness of an algorithm?

- Recursive algorithm ⇒ Induction proofs

- Iterative algorithm (loops) ⇒ ???

Loop invariant

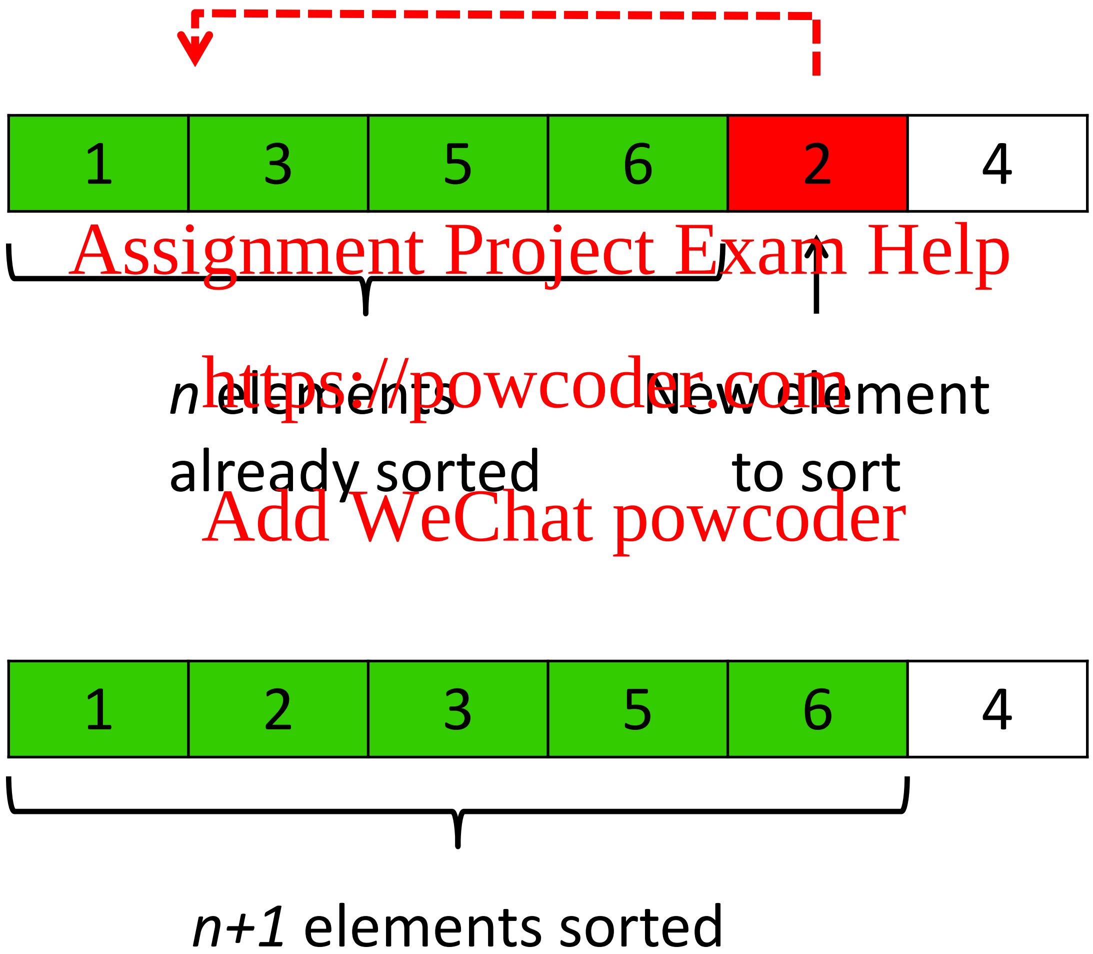
A **loop invariant** is a loop property that hold before and after each iteration of a loop.

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
```

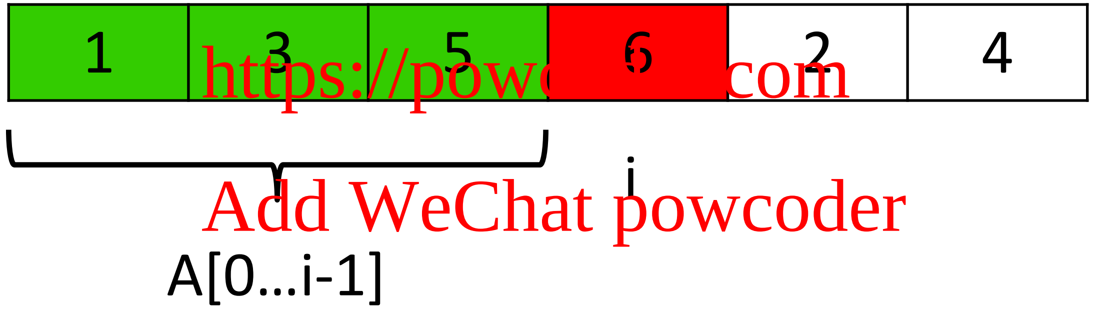

(Seen in previous lecture)

Insertion sort



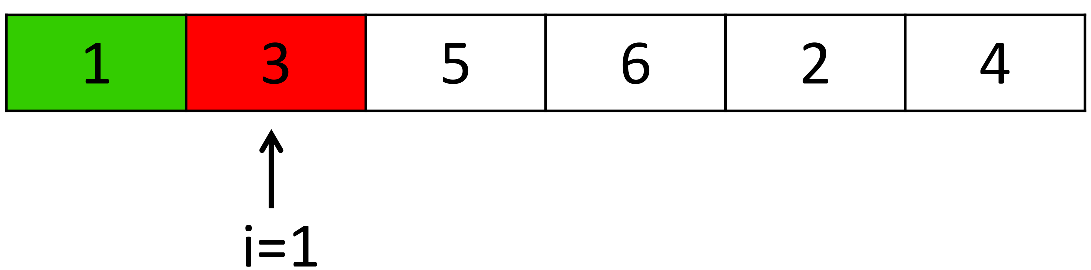
Loop invariant

The array A[0…i-1] is fully sorted.

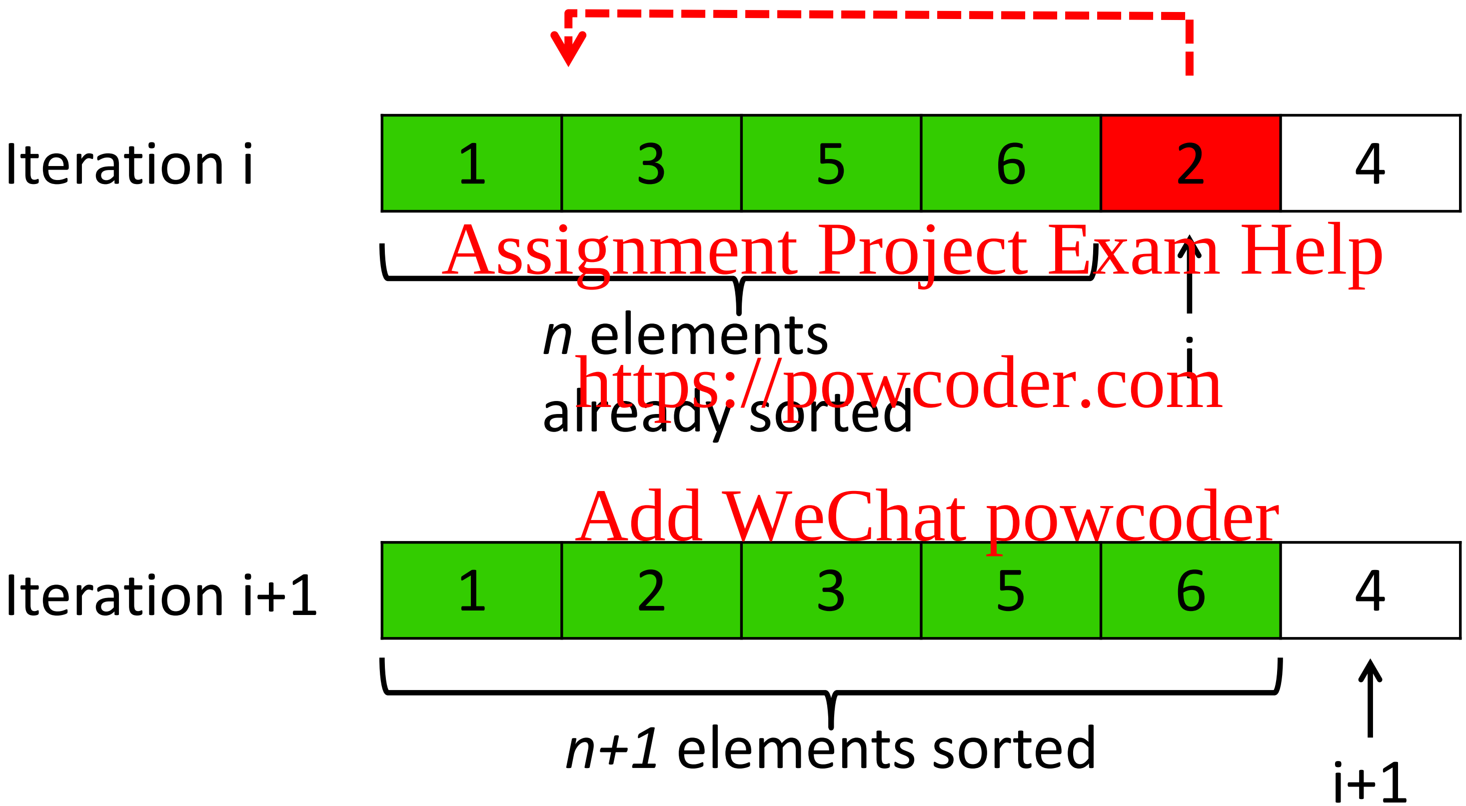


Initialization

Just before the first iteration (i = 1), the sub-array A[0 … 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 precise, we would need to state and prove a loop invariant for the ``inner'' **while** loop.

Termination

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

Therefore, $i = length(A)$.

Plugging *length(A)* in for *i*-1 in the loop invariant, the subarray *A*[0 … *length(A)-1*] consists of the elements originally in *A*[0 … *length(A)*-1] but in sorted order.

A[0 … *length(A)*-1] contains *length(A) elements (i.e. all initial elements!) and no element is duplicated/deleted.*

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

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.

Analogy to induction proofs

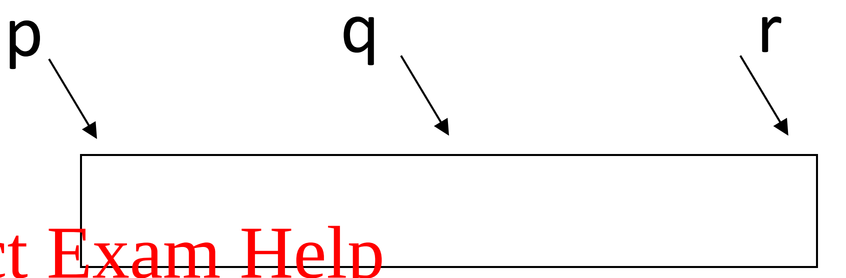
Using loop invariants is like mathematical induction.

- 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≤r)

Postcondition:

The elements between indexes p and r are sorted

Merge Sort (Merge method)

- MERGE-SORT calls a function 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 <= q < r. The subarrays A[p.. q] and A[q +1.. r] are sorted

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

Procedure Merge

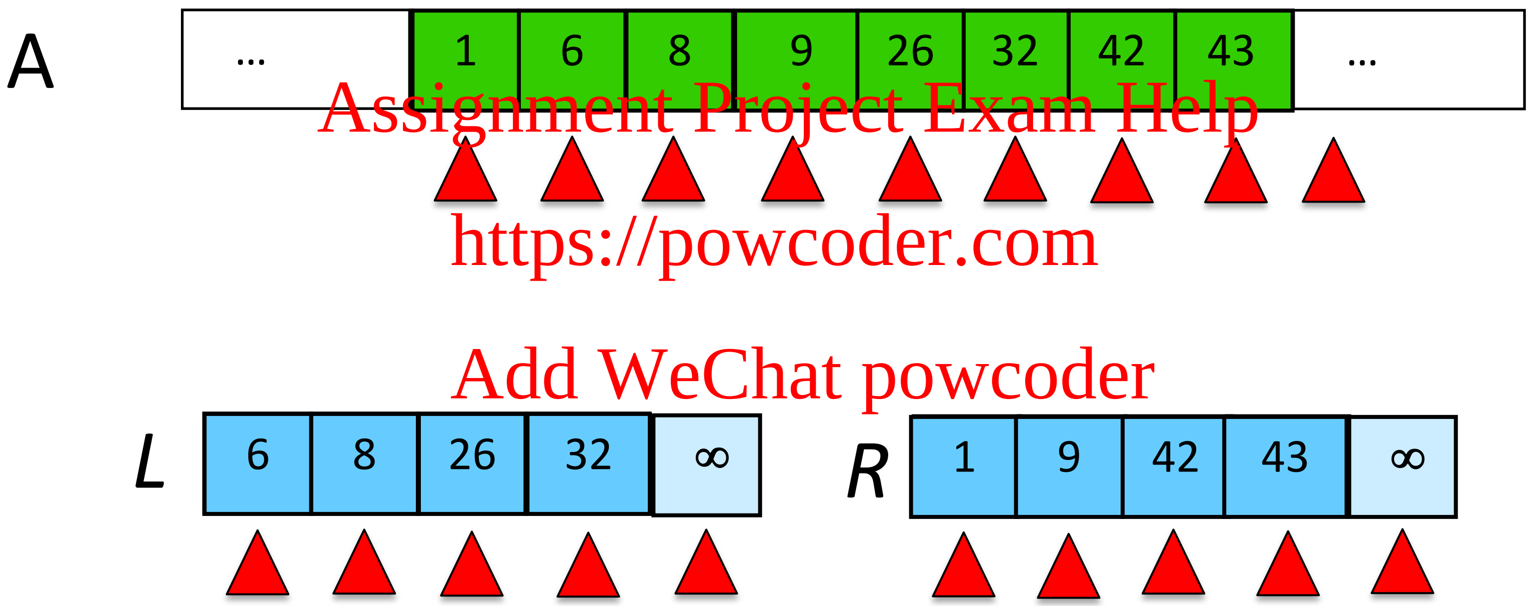
Merge(A, p, q, r)

```
n1 ← q − p + 1
n2 ← r − q
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← 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 satisfy 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 ⇒ All elements are sorted.

Correctness proof for **Merge Sort**

- Recursive property: Elements in A[p,r] are be sorted.
- **Base Case**: p = r
  - A contains a single element (which is trivially "sorted")
- **Inductive Hypothesis:**
  - Assume that MergeSort correctly sorts n=1, 2, ..., **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, ... , **k** elements
- **Inductive Step:**

  - Show that MergeSort correctly sorts n = **k + 1** elements.

- **Proof:**
  - First recursive call $n_1=q-p+1=(k+1)/2 \leq k$

    => subarray A[p .. q] is sorted
  - Second recursive call $n_2=r-q=(k+1)/2 \leq k$

    => subarray A[q+1 .. r] is sorted
  - A, p q, r fulfill now the precondition of Merge
  - The post-condition of Merge guarantees that the array A[p .. r] is sorted => post-condition of MergeSort satisfied.

Termination Step

We have to find a quantity that 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.