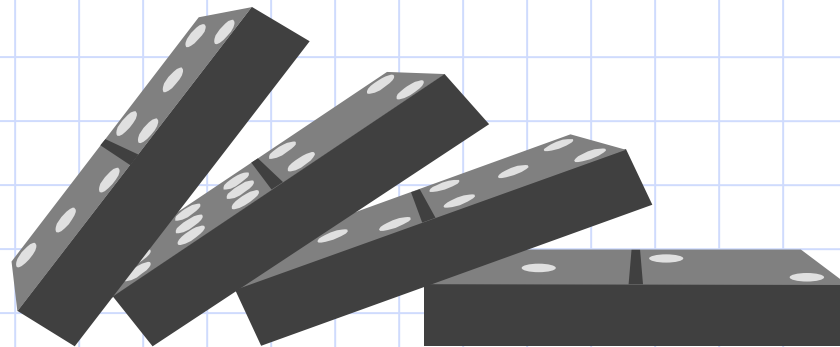
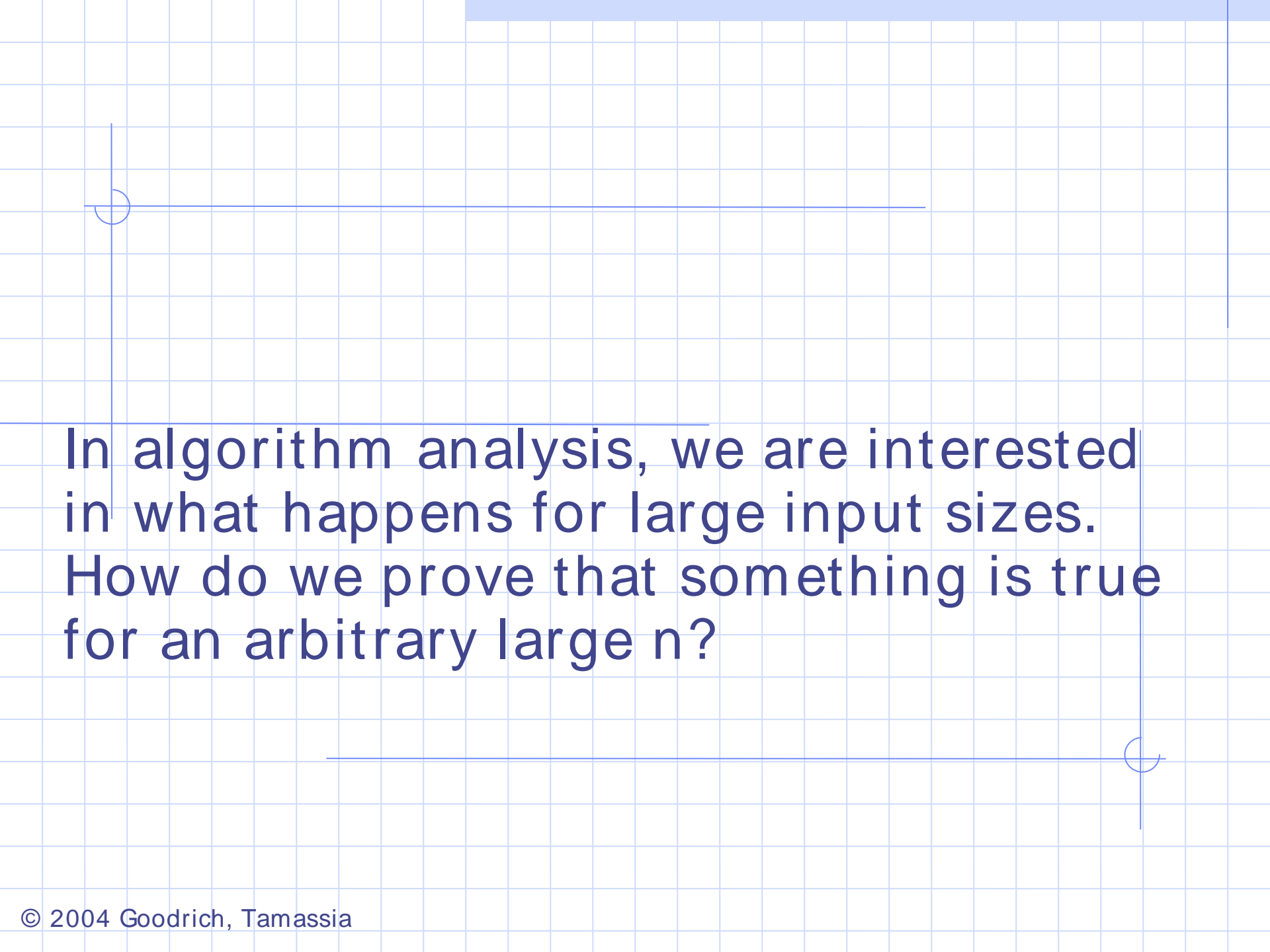


# Induction and loop invariants



Domino Principle: Line up any number of dominos in a row; knock the first one over and they will all fall.



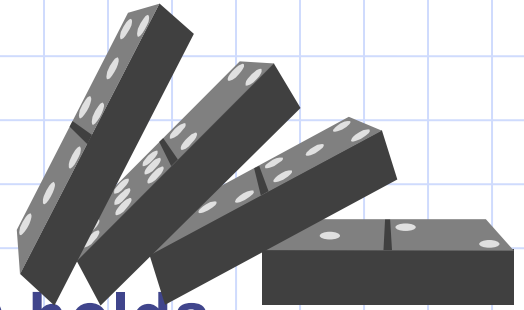
In algorithm analysis, we are interested in what happens for large input sizes. How do we prove that something is true for an arbitrary large  $n$ ?

# Induction: domino principle

In a row of dominos,

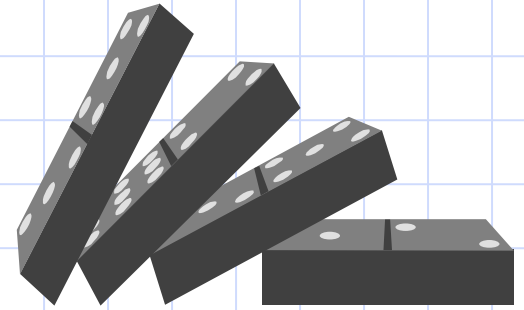
- ◆ If the first one falls
- ◆ If each domino falling knocks the next one
- ◆ Then all of them fall.

# Induction



- ◆ **Statement to prove: for all  $n$ ,  $P(n)$  holds.**
  - For example, for all  $n$ , sum of the first  $n$  elements is  $1 + 2 + \dots + n = n(n + 1) / 2$
- ◆ **Base Case:**
  - $P(0)$  holds (or  $P(1)$  holds).
  - For example,  $P(1)$ :  $1 = 1(1 + 1) / 2 = 1$
- ◆ **Induction hypothesis:**
  - $P(n)$  holds for some arbitrary  $n$ .
  - $1 + 2 + \dots + n = n(n + 1) / 2$
- ◆ **Induction step.**
  - From the fact that  $P(n)$  holds we can derive that  $P(n + 1)$  holds.

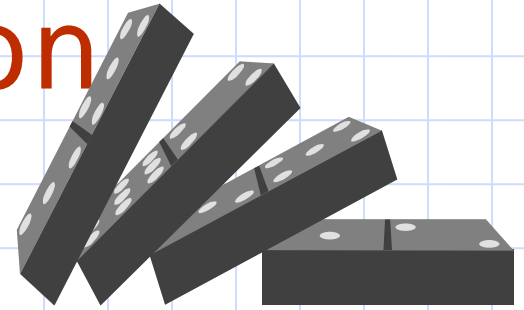
# Induction Step



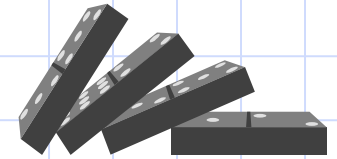
## ◆ Induction step.

- From the fact that  $P(n)$  holds (and the base case) we can derive that  $P(n+1)$  holds.
- From  $1+2+\dots+n = n(n+1)/2$  derive  $1+2+\dots+n+n+1 = (n+1)(n+2)/2$
- Proof: by induction hypothesis,  
 $1+\dots+n+n+1 = n(n+1)/2 + (n+1)$ 
  - $= (n+1)(n+2)/2$

# Structure of induction proof



- ◆ **Statement to prove:**
  - For all  $n > k$ ,  $P(n)$  is true.
- ◆ **Base Case:**
  - $P(k)$  holds (usually  $P(0)$  or  $P(1)$ ).
- ◆ **Induction hypothesis:**
  - $P(n)$  holds for some arbitrary  $n$ .
- ◆ **Induction step.**
  - Assuming  $P(n)$  holds derive that  $P(n+1)$  holds.



# Another example

## ◆ Statement to prove:

- For all  $n > 4$ ,  $n^2 < 2^n$ .

## ◆ Base Case:

- $P(5)$ :  $5^2 = 25 < 2^5 = 32$

## ◆ Induction hypothesis:

- Assume  $n^2 < 2^n$  for  $n > 4$

## ◆ Induction step.

- Assuming  $n^2 < 2^n$  derive  $(n+1)^2 < 2^{(n+1)}$ .
- $(n+1)^2 = n^2 + 2n + 1 < 2n^2 < 2 \cdot 2^n = 2^{(n+1)}$

Because since  $n > 4$ ,  
 $n \cdot n > 4n = 2n + 2n > 2n + 1$

Induction hypothesis



### Statement to prove:

- For all  $n > k$ ,  $P(n)$  is true.



### Base Case:

- $P(k)$  holds.



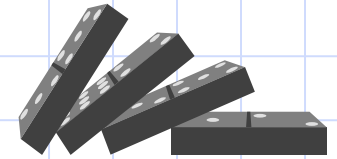
### Induction hypothesis:

- $P(n)$  holds for some  $n > k$ .



### Induction step.

- From  $P(n)$  derive  $P(n+1)$ .



# Strong induction

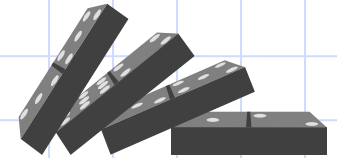
- ◆ **Statement to prove:**
  - Same: for all  $n > k$ ,  $P(n)$ .
- ◆ **Base Case:**
  - $P(k)$ , maybe also  $P(k+1)$ .
- ◆ **Induction hypothesis:**
  - Assume **for all  $m < n$   $P(m)$**  holds.
- ◆ **Induction step.**
  - Assuming for all  $m < n$   $P(m)$  holds, prove  $P(n)$ .

- ◆ **Statement to prove:**
  - For all  $n > k$ ,  $P(n)$  is true.
- ◆ **Base Case:**
  - $P(k)$  holds.
- ◆ **Induction hypothesis:**
  - $P(n)$  holds for some  $n > k$ .
- ◆ **Induction step.**
  - From  $P(n)$  derive  $P(n+1)$ .

Aren't we assuming the same thing as proving?

We are proving  $P(n)$  everywhere, assuming it only for the first several elements!

# Strong induction



## *Strong Induction*

- ◆ **Statement to prove:**
  - For all  $n > k$ ,  $P(n)$  is true.
- ◆ **Base Case:**
  - $P(k)$  holds.
- ◆ **Induction hypothesis:**
  - $P(m)$  holds for all  $m$ ,  $k < m < n$ ,
- ◆ **Induction step.**
  - From ind. hyp. derive  $P(n)$ .

### ◆ **Statement to prove:**

- For all  $n$ ,  $F(n) < 2^n$
- $F(n)$  is  $n^{\text{th}}$  Fibonacci number

### ◆ **Base Cases:**

- $F(0) = 0 < 1$ ,  $F(1) = 1 < 2$

### ◆ **Induction hypothesis:**

- Assume  $F(m) < 2^m$  for all  $m < n$

### ◆ **Induction step.**

- Assuming induction hypothesis derive  $F(n) < 2^n$ .
- $F(n) = F(n-1) + F(n-2)$ . By induction hypothesis,  $F(n-1) < 2^{n-1}$  and  $F(n-2) < 2^{n-2}$ .
- So  $F(n) < 2^{n-1} + 2^{n-2} < 2 * 2^{n-1} = 2^n$

# Algorithm correctness

✂ **Precondition:** something that is true before a part of an algorithm (e.g, a loop).

- CurrentMax contains A[0].

✂ **Postcondition:** something that is true after it finished running.

- CurrentMax contains the maximum element of A.

✂ **Loop invariant:** .

- CurrentMax is the maximum of elements of A seen so far.

Algorithm *arrayMax(A, n)*

**Input** array A of n integers

**Output** maximum element of

A

*currentMax* ← A[0]

**for** i ← 1 **to** n - 1 **do**

**if** A[i] > *currentMax* **then**

*currentMax* ← A[i]

**return** *currentMax*

# Induction and correctness

- ✂ **Precondition:** like base case.
  - CurrentMax contains A[0].
- ✂ **Loop invariant:** if true at step  $i$ , still true after one more loop.
  - If currentMax contains the maximum of the A[0.. $i$ ], and the if then is executed, currentMax will contain the maximum of A[0.. $i+1$ ].
- ✂ **Postcondition:** follows from loop invariant.
  - CurrentMax is the maximum element of A.

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element
  of A

  currentMax ← A[0]
  for  $i \leftarrow 1$  to  $n - 1$  do
    if  $A[i] > \textit{currentMax}$ 
      then
        currentMax ←
        A[i]
  return currentMax
```

# Induction and correctness

✂ **Precondition:** like base case.

- $\text{CurrentMax} = A[0]$ .

✂ **Loop invariant:** if true at step  $i$ , still true after one more loop.

- Suppose  $\text{CurrentMax}$  is the max of  $A[0..i-1]$ . Two cases:
  - a)  $A[i] > \text{CurrentMax}$ . Then  $\text{CurrentMax}$  is the max of  $A[0..i]$  since it is in the array ( $A[i]$ ) and it is the largest ( $>$  than previous).
  - b)  $A[i]$  is not  $>$  than  $\text{CurrentMax}$ . Then  $\text{CurrentMax}$  is the max of  $A[0..i]$  since it was the max of  $A[0..i-1]$  and  $A[i]$  is not bigger.

◆ By induction, true for all iterations including termination.

```
Algorithm arrayMax( $A, n$ )  
  Input array  $A$  of  $n$  integers  
  Output maximum element  
  of  $A$   
  
   $\text{currentMax} \leftarrow A[0]$   
  for  $i \leftarrow 1$  to  $n - 1$  do  
    if  $A[i] > \text{currentMax}$   
  then  
     $\text{currentMax} \leftarrow$   
   $A[i]$   
  return  $\text{currentMax}$ 
```

# Algorithm analysis

- ◆ If an algorithm is a sequence of (constantly many) parts, its running time is the maximal over running times of parts.
  - E.g., if the algorithm sorts inputs in time  $O(n \log n)$  then looks at each one once in  $O(n)$ , total time is  $O(\max(n \log n, n)) = O(n \log n)$ .
- ✂ If an algorithm has  $k$  nested loops, with  $n$  as a bound, then the running time is  $n^k$ .
  - First prefix average algorithm ran in time  $n^2$ .
- ✂ If an algorithm makes  $n$  recursive calls each time calling itself twice, it is exponential. Calling itself once gives linear factor.
  - First and second Fibonacci algorithms.