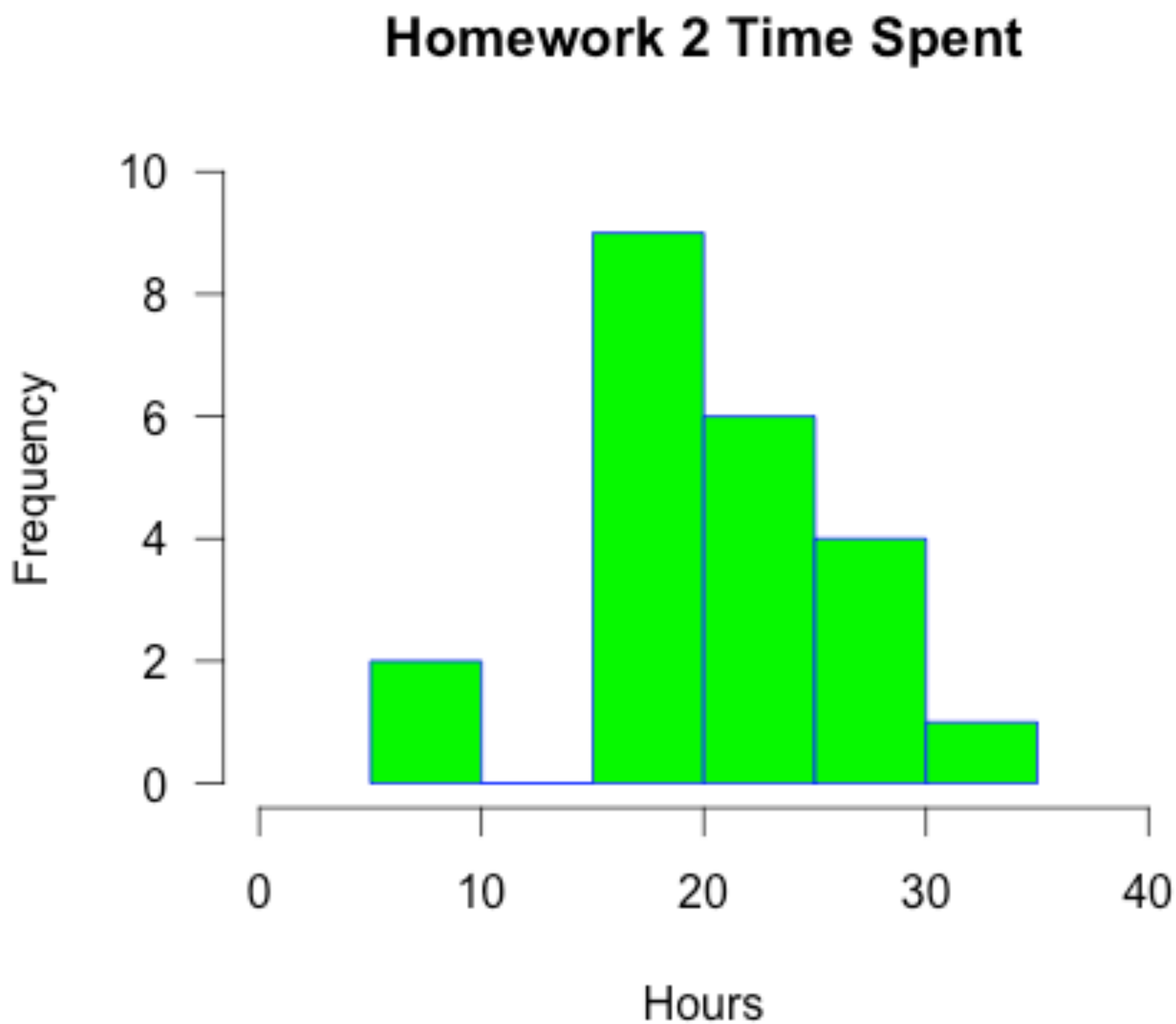
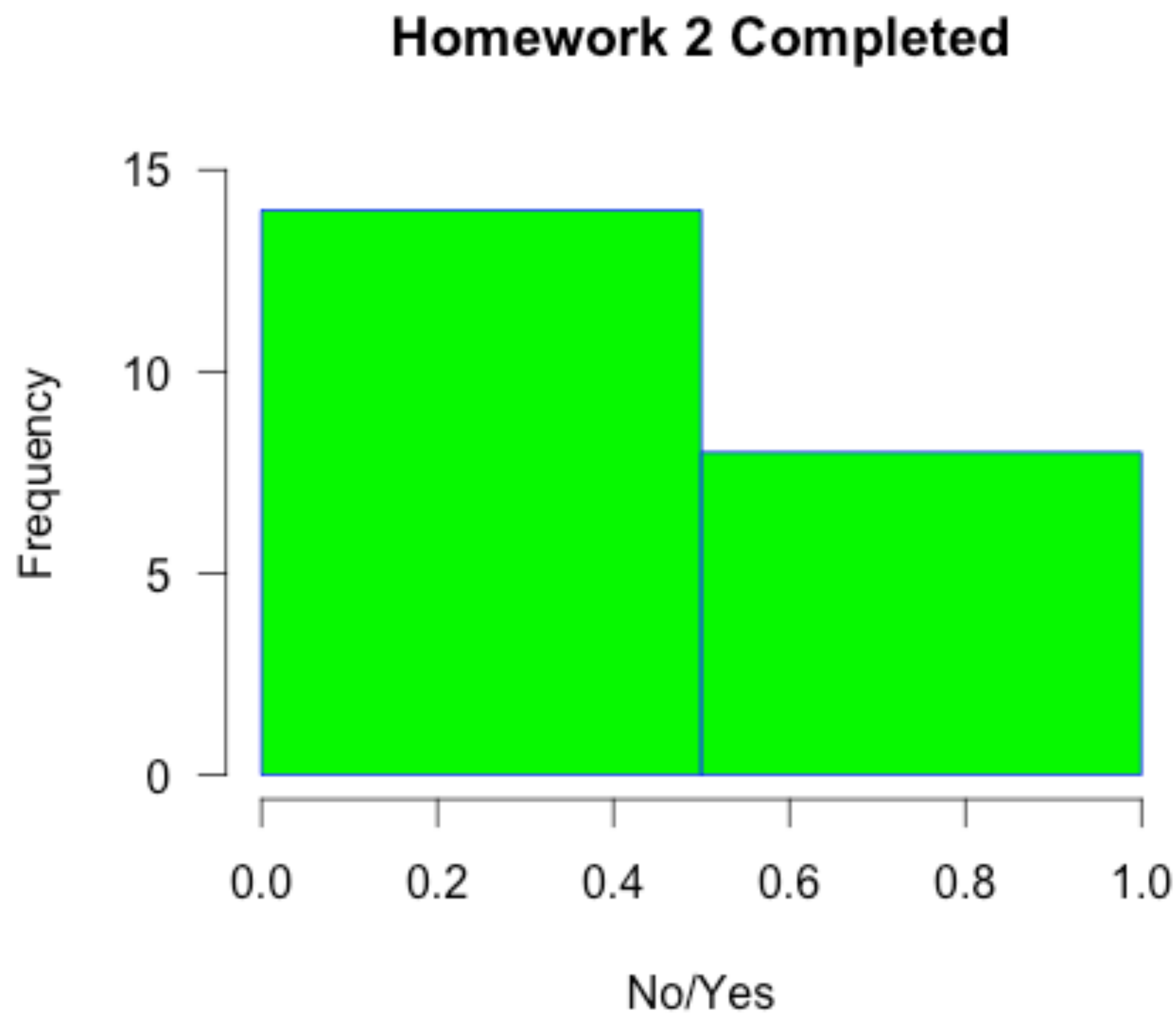
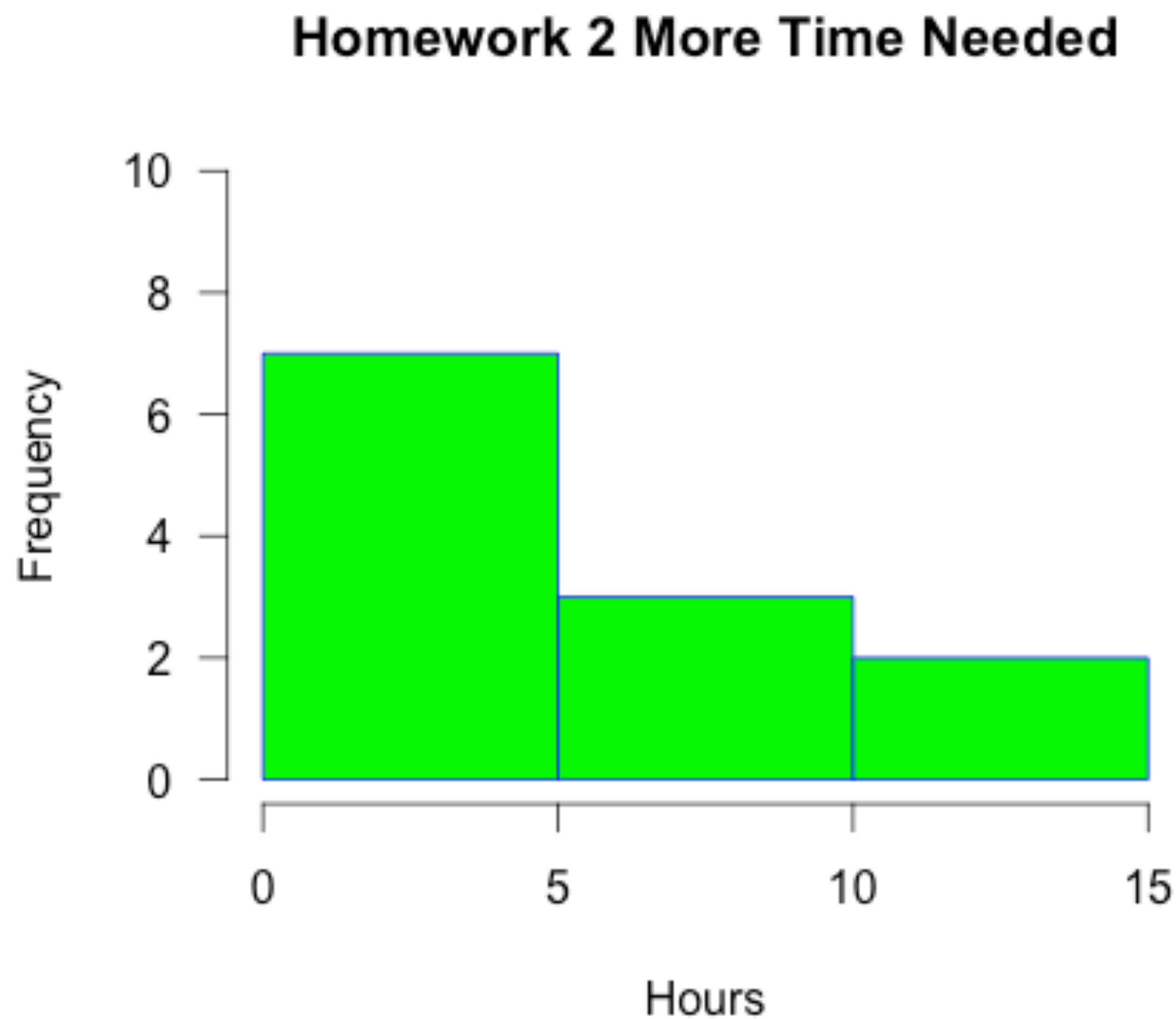
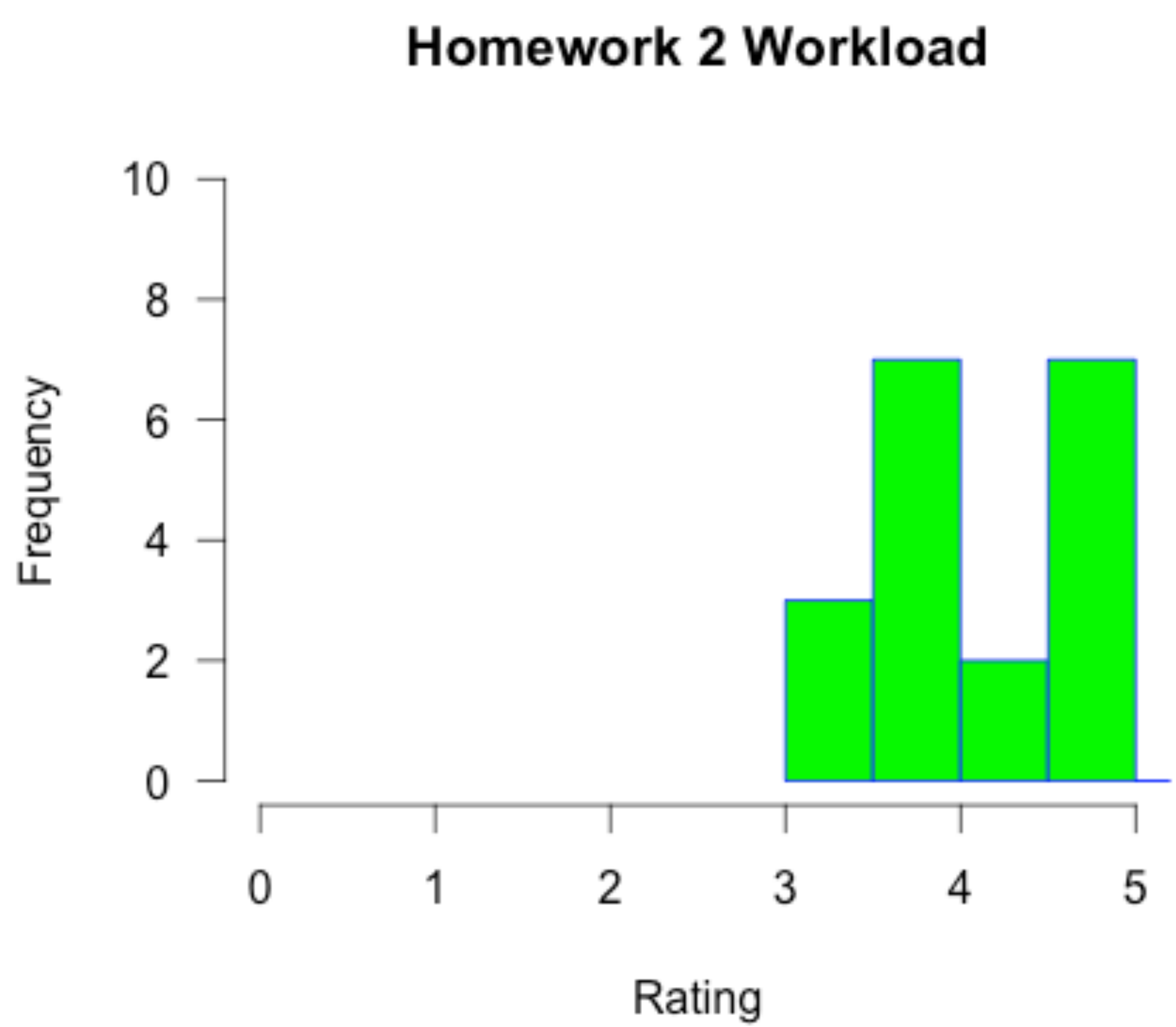
Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

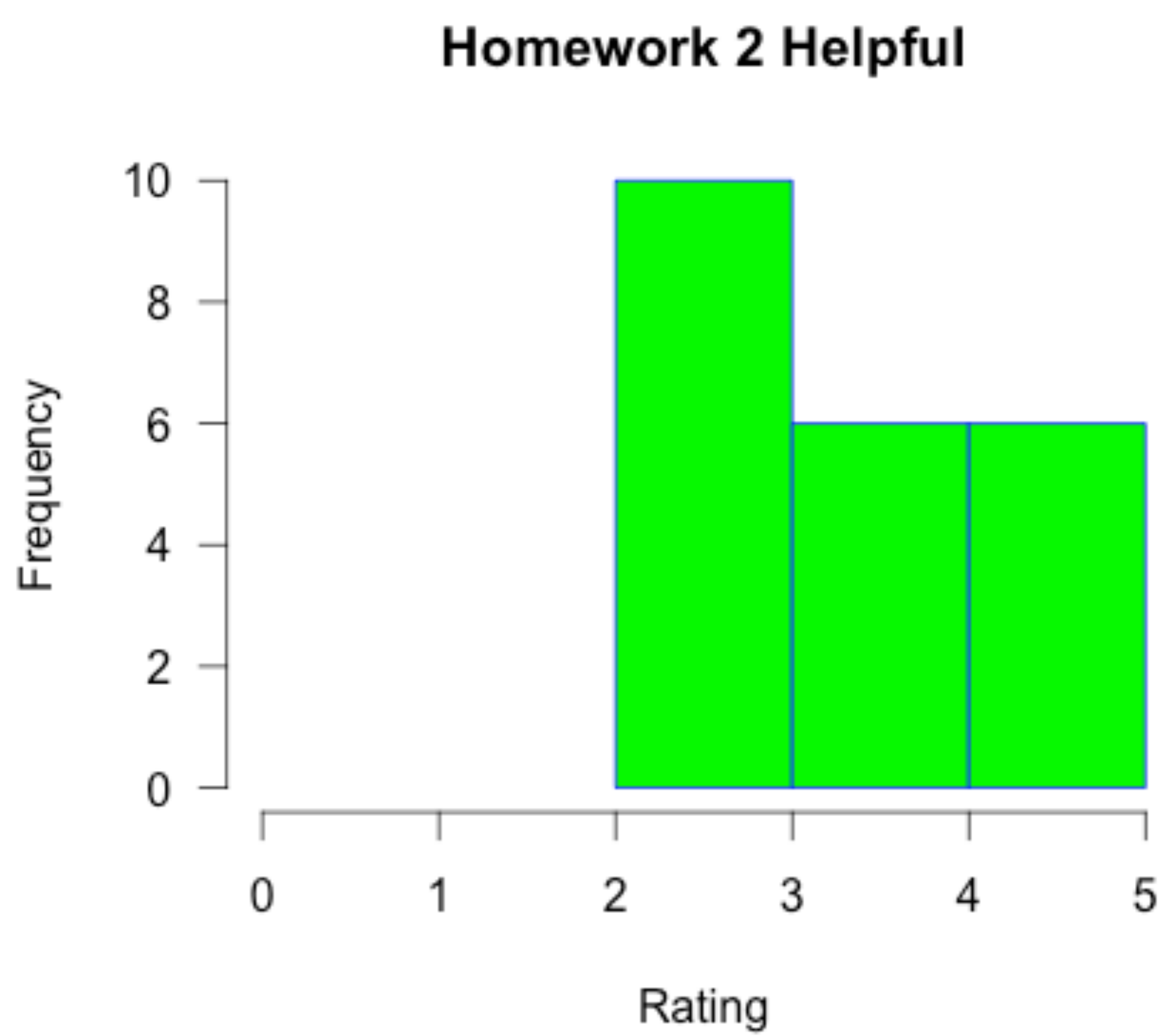

Homework 2 Time Spent
Frequency
0
2
4
6
8
10
0
10
20
30
40
Hours


Homework 2 Completed
Frequency
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
No/Yes

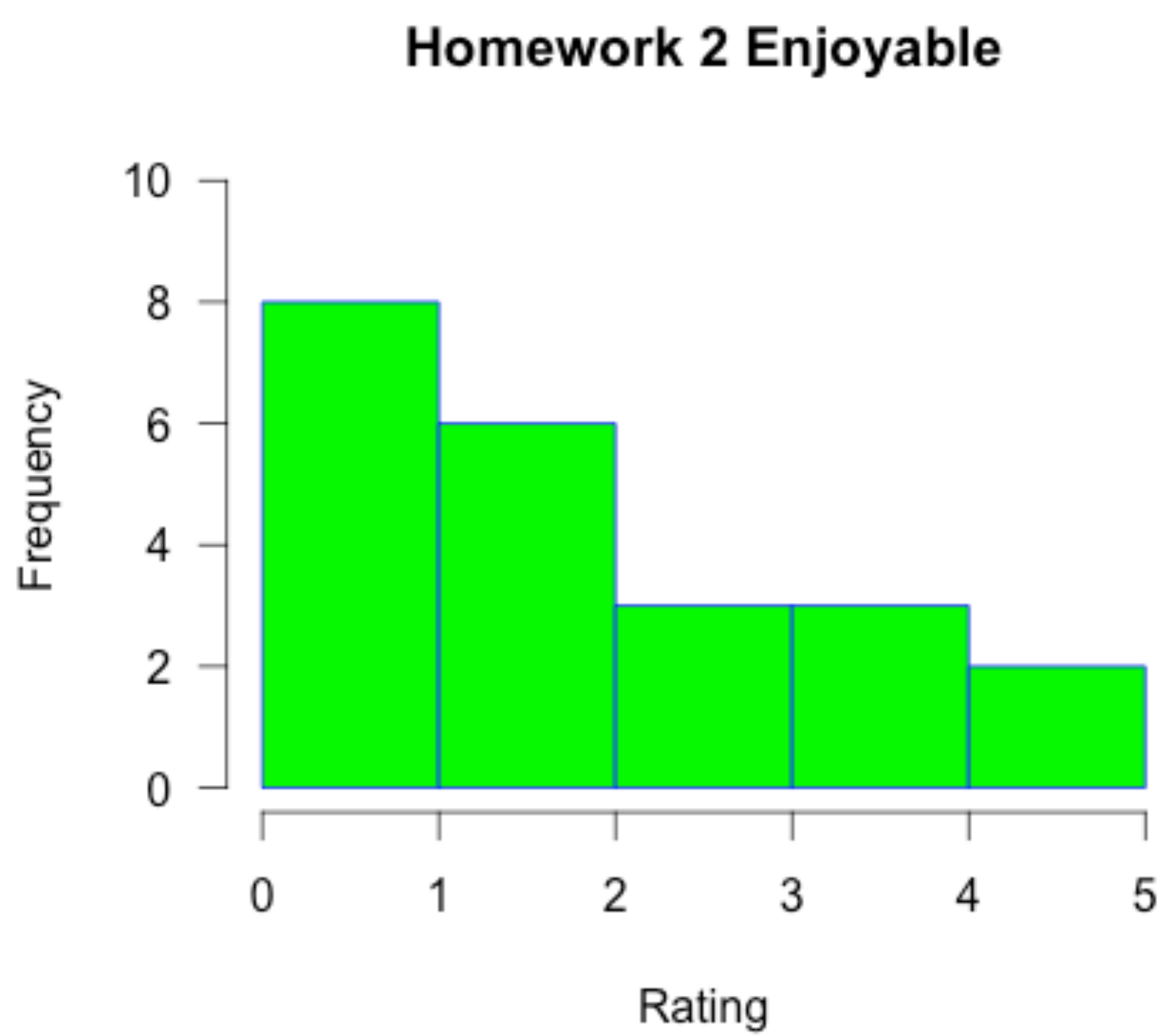

Homework 2 More Time Needed
Frequency
Hours
0
2
4
6
8
10
0
5
10
15



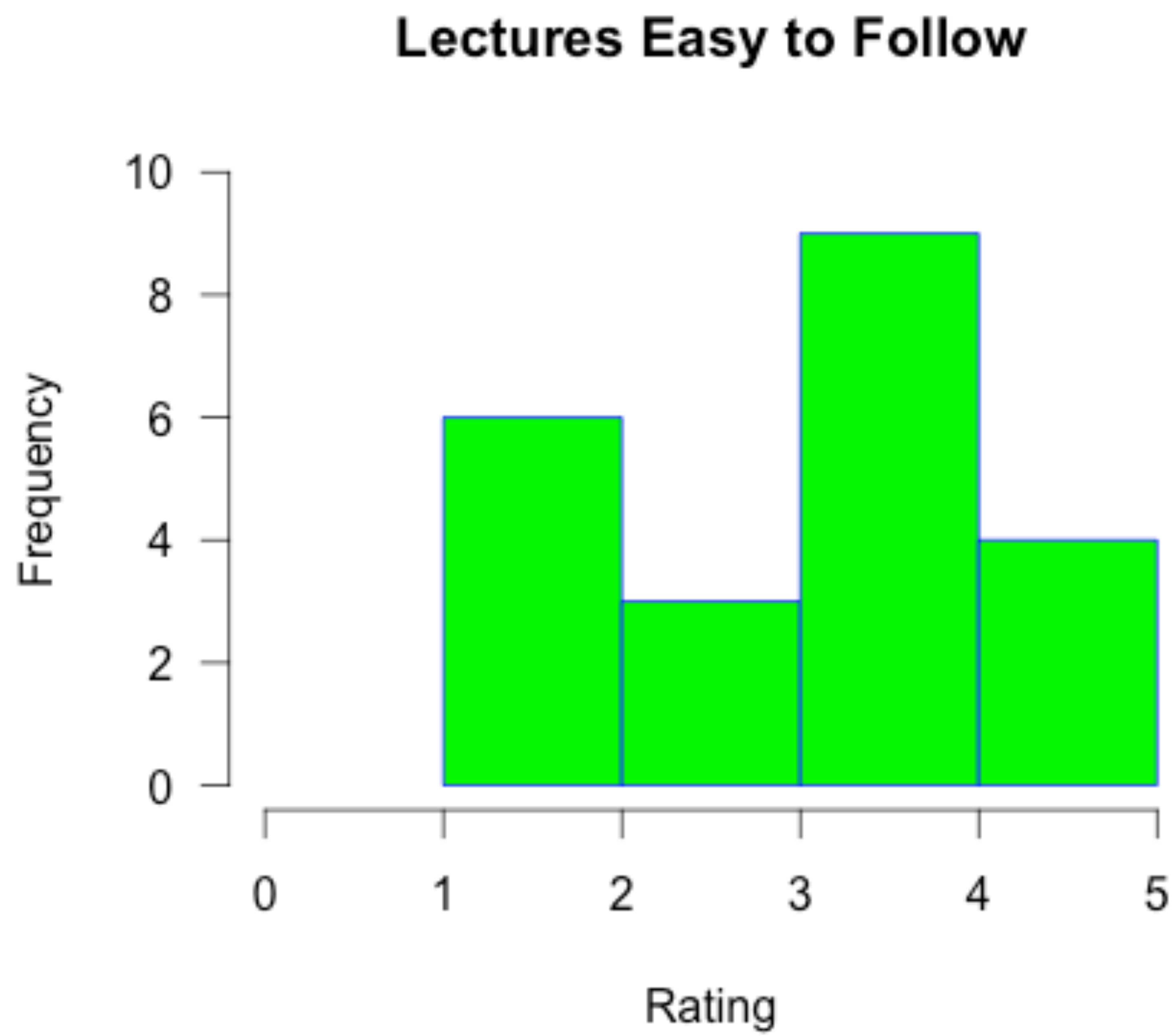
Homework 2 Workload
Frequency
Rating
0
2
4
6
8
10
0
1
2
3
4
5



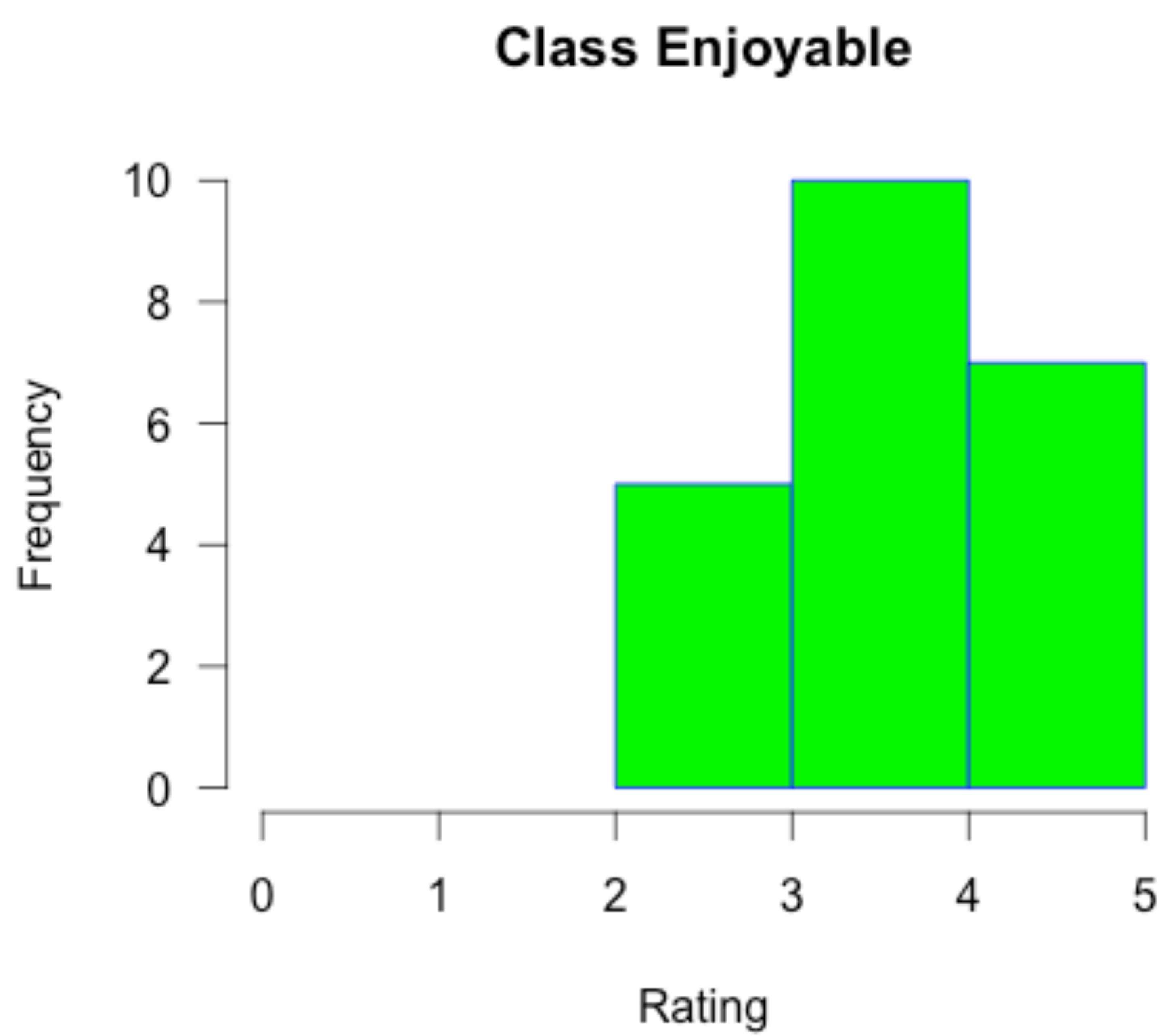
Homework 2 Helpful
Frequency
Rating
0
2
4
6
8
10
0
1
2
3
4
5



Homework 2 Enjoyable
Frequency
Rating
0
2
4
6
8
10
0
1
2
3
4
5


Lectures Easy to Follow
Frequency
10
8
6
4
2
0
0
1
2
3
4
5
Rating


Course Pace
Frequency
0
2
4
6
8
10
0
1
2
3
4
5
Rating


Class Enjoyable
Frequency
10
8
6
4
2
0
0
1
2
3
4
5
Rating

General Functional Programming vs Scala

- The vast majority of topics we have discussed are relevant to any functional programming language:
  - The Substitution and Environment Models
  - The Design Recipe and Templates
  - Abstract and Recursive Datatypes
  - Arrow Types, First-Class Functions
  - Continuations

General Functional Programming vs Scala

- The vast majority of topics we have discussed are relevant to any functional programming language:
  - Parametric Polymorphism
  - Covariance, Contravariance
  - Monads
  - Lexical vs. Dynamic Scoping
  - Call-by-Value vs. Call-by-Name

More on Traits

Thin vs Rich Interfaces

- Traits provide a way to resolve the tension between "thin" and "rich" interfaces:
    - Thin interface: Include only essential methods in an interface
        - Good for implementors
    - Rich interface: Include a rich set of methods in an interface
        - Good for clients

Thin vs Rich Interfaces

- With traits, we can define an interface to include only a small number of essential methods, but then include traits to build rich functionality based on the essential methods
  - Implementors win
  - Clients win

Thin vs Rich Interfaces

- Consider our implementations of Interval, Rational, Measurement
  - We want to include all comparison operators on them:

    <   <=   >=   >

    - With traits, we could define just one operator < and mix in a trait to define the rest in terms of <

Thin vs Rich Interfaces

```
case class Measurement(magnitude: BigDecimal,
                       unit: PhysicalUnit)
extends Ordered[Measurement]

  def compare(that: Measurement) =
    val (u,m1,m2) = this.unit commonUnits that.unit
    (m1 * magnitude) - (m2 * that.magnitude)
  }
  …
}
```

Traits as Stackable Modifiers

```
abstract class IntMap {
  def insert(s: String, n: Int): IntMap
  def retrieve(s: String): Int
}
```

Traits as Stackable Modifiers

```
case class IntListMap(elements: List[(String,Int)] = Nil)
extends IntMap {

  def insert(s: String, n: Int): IntMap =
    IntListMap((s -> n) :: elements)

  def retrieve(s: String) = {
    def retrieve(xs: List[(String, Int)]): Int = {
      xs match {
        case Nil => throw new IllegalArgumentException(s)
        case (t, n) :: ys if (s == t) => n
        case y :: ys => retrieve(ys)
      }
    }
    retrieve(elements)
  }
}
```

Traits as Stackable Modifiers

```
trait Incrementing extends IntMap {
  abstract override def insert(s: String, n: Int) =
    super.insert(s, n + 1)
}
```

This super call depends on how the trait is mixed into a particular class

Traits as Stackable Modifiers

```
trait Filtering extends IntMap {
  abstract override def insert(s: String, n: Int) = {
    if (n >= 0) super.insert(s, n)
    else this
  }
}
```

As does this one

Traits as Stackable Modifiers

```
> val m = new IntListMap() with Incrementing with Filtering
m: IntListMap with Incrementing with Filtering = IntListMap(List())
```

The order in which the traits are listed is important.
The trait furthest to the right is called first

Traits as Stackable Modifiers

```
> m.insert("a", -1)
res0: IntMap = IntListMap(List())
```

Traits as Stackable Modifiers

```
> res0.retrieve("a")
java.lang.IllegalArgumentException: a
```

Traits as Stackable Modifiers

```
> m.insert("a", 1)
res2: IntMap = IntListMap(List((a,2)))
```

Traits as Stackable Modifiers

```
> res2.retrieve("a")
res3: Int = 2
```

Traits as Stackable Modifiers

```
> val m = new IntListMap() with Filtering with Incrementing
m: IntListMap with Filtering with Incrementing = IntListMap(List())
```

Now we have reversed the order

Traits as Stackable Modifiers

```
> m.insert("a", -1)
res0: IntMap = IntListMap(List((a,0)))
```

Now the integer is incremented before filtering, and so it passes the filter

Traits as Stackable Modifiers

```
> res0.retrieve("a")
res5: Int = 0
```

Traits vs Multiple Inheritance

Traits vs Multiple Inheritance

- The key property of traits that distinguishes them from multiple inheritance is *linearization*
- With traditional multiple inheritance, which implementation of insert would be called:

```
class MyMap() extends IntListMap() with Filtering with Incrementing

new MyMap().insert("b",2)
```

Traits vs Multiple Inheritance

- With traits, the effect of a super call is determined by the linearization of traits, which enables:
  - Multiple trait implementation of the same method to be called
  - Multiple ways to compose the traits depending on circumstances

Trait Linearization

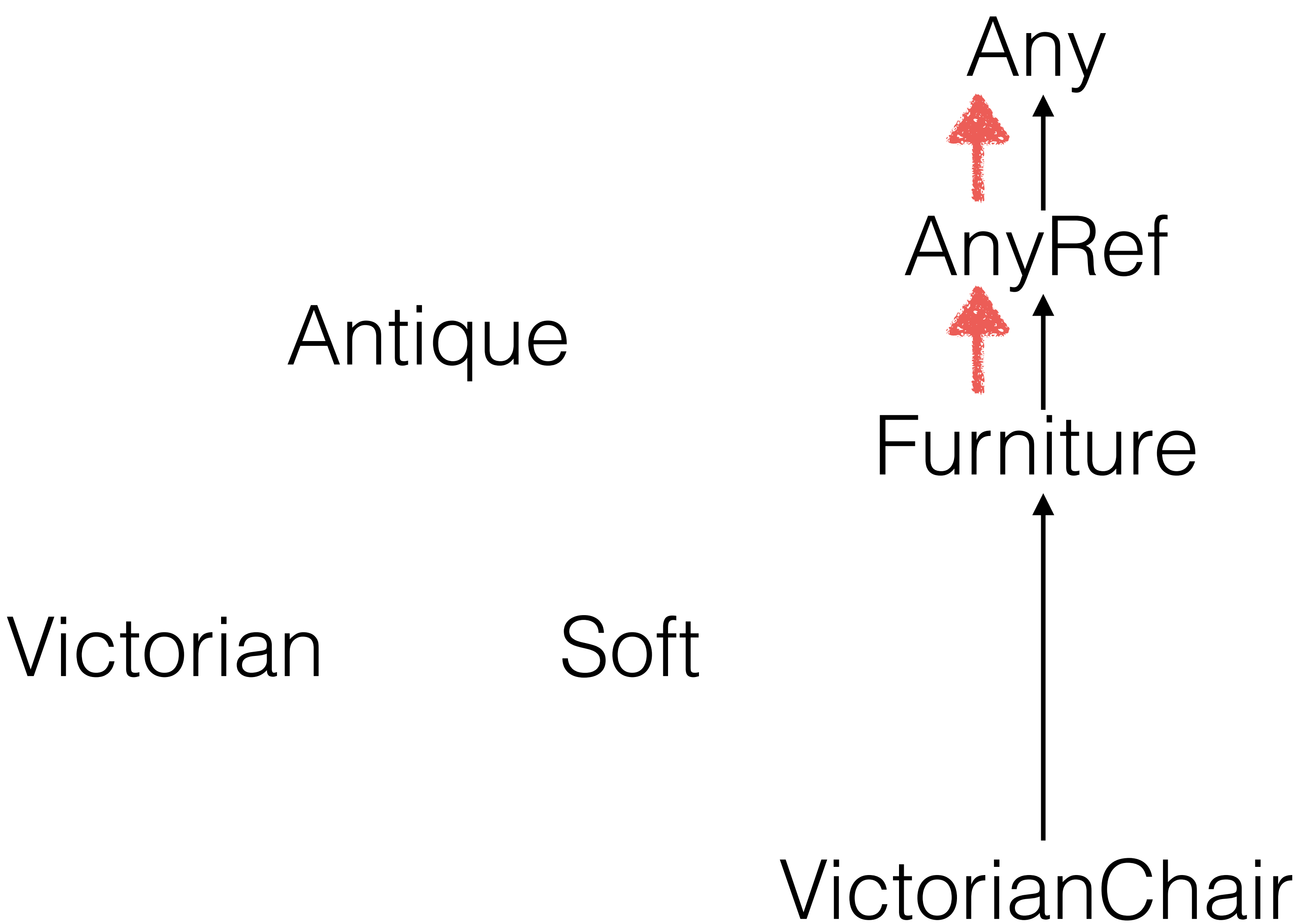
```
class C() extends D() with T1… with TN {
   …
}
```

- To linearize class C
    - Linearize class D
    - Extend with the linearization of T1, leaving out classes already linearized
    - Continue until extending with the linearization of TN, leaving out classes already linearized
    - Finally, extend with the body of class C

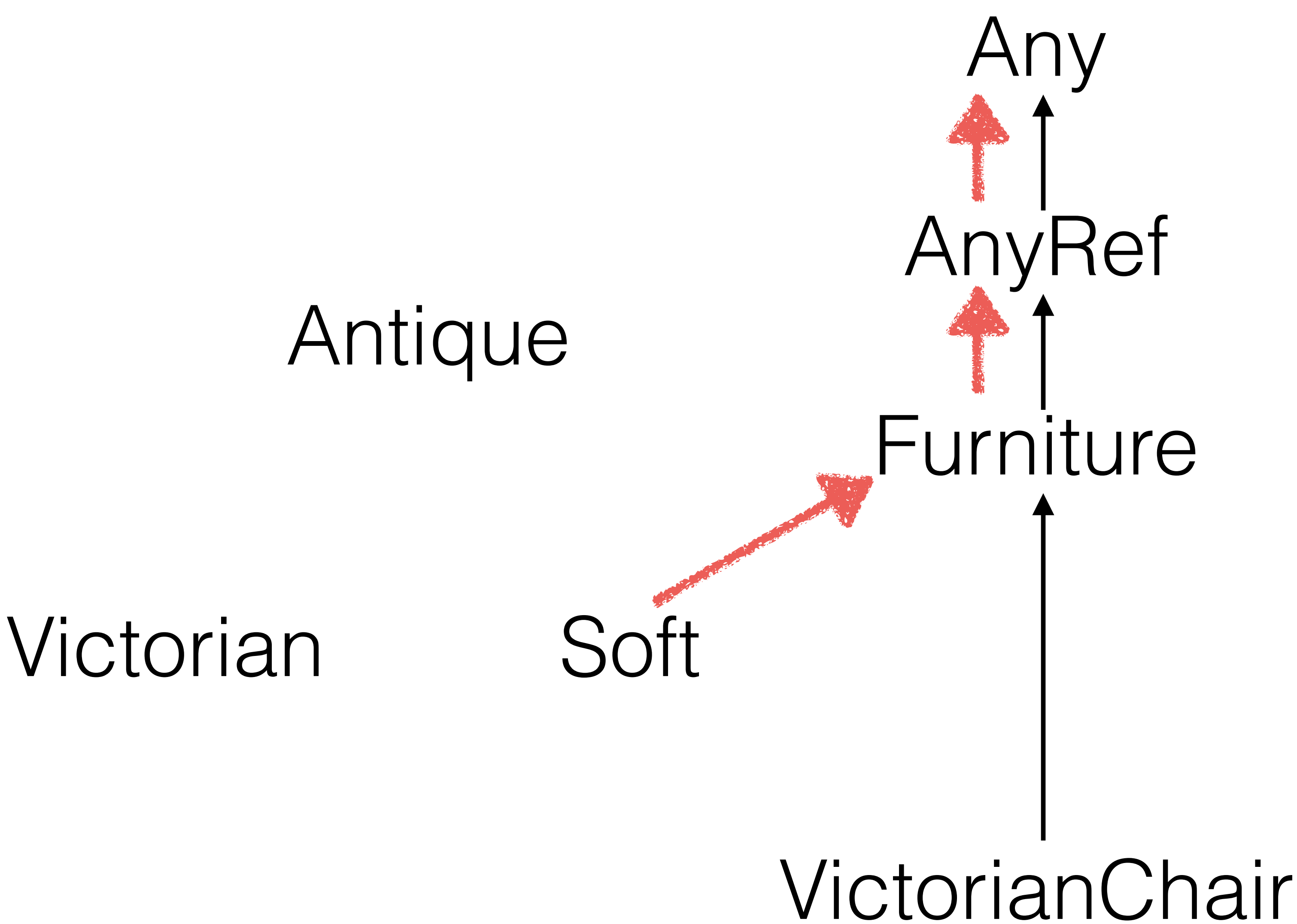
Trait Linearization

```
class Furniture
trait Soft extends Furniture
trait Antique extends Furniture
trait Victorian extends Antique
class VictorianChair extends Furniture with Soft with Victorian
```

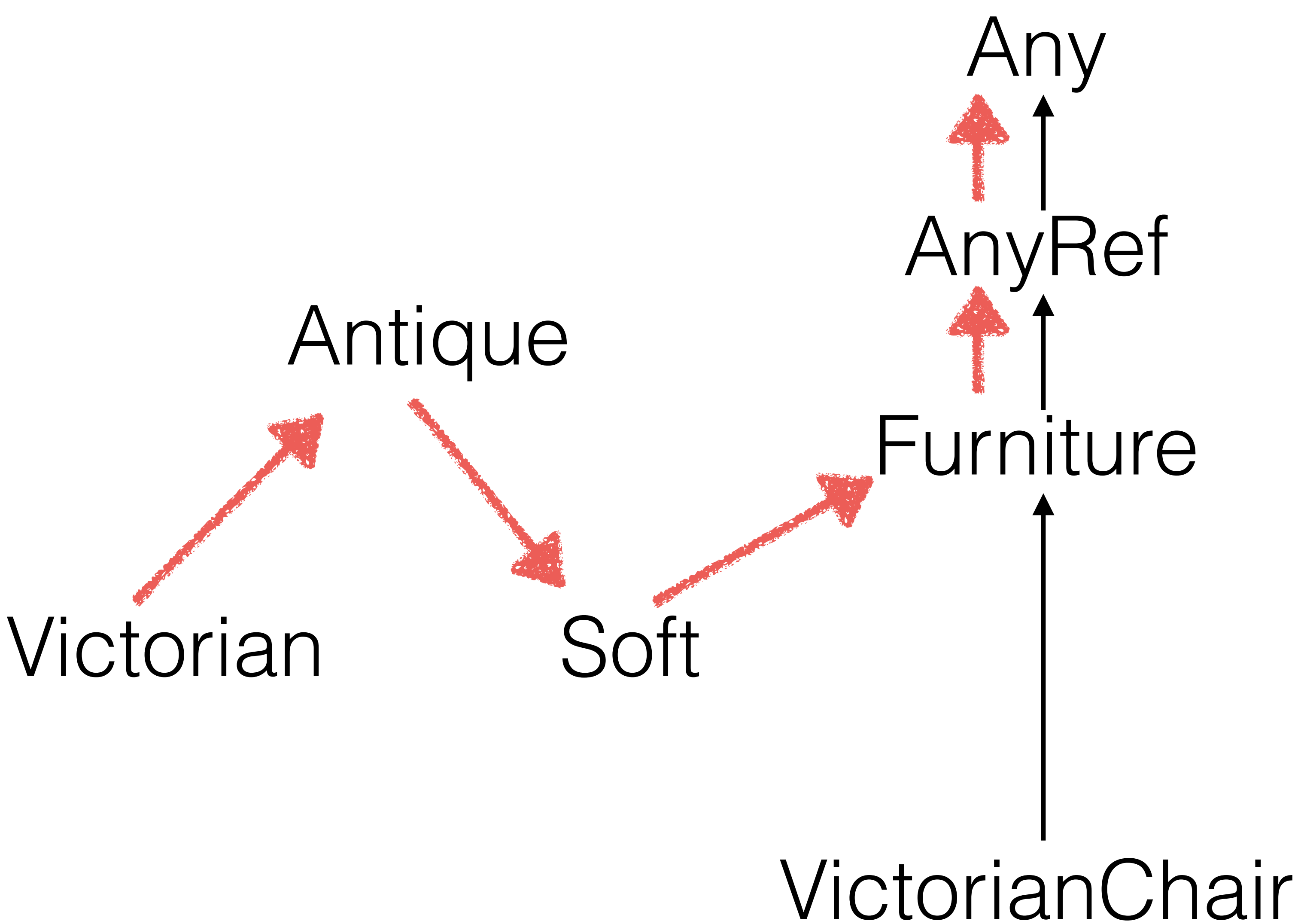
Trait Linearization



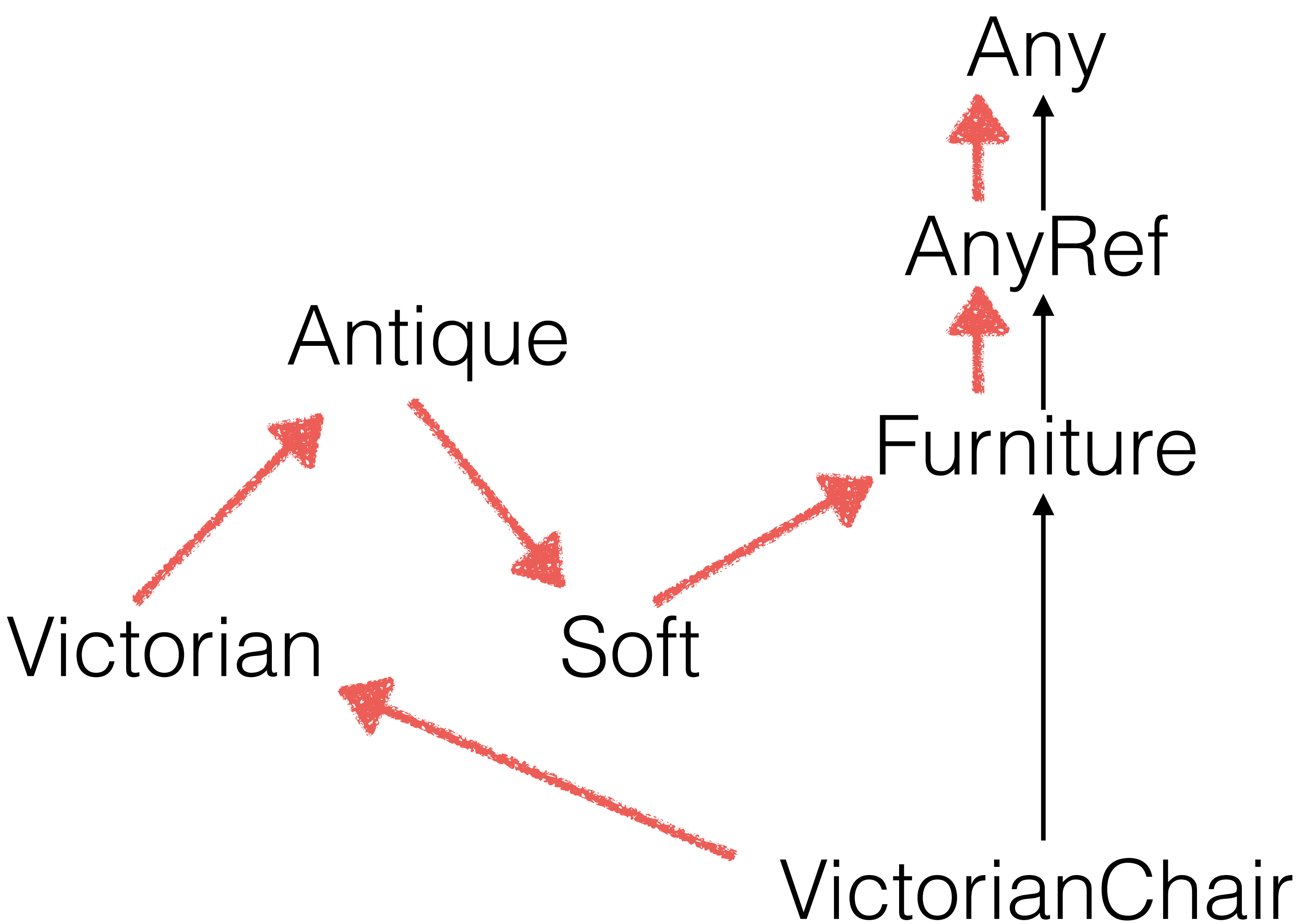
Trait Linearization



Trait Linearization



Trait Linearization



Guidelines on Using Traits

- Use concrete classes when the behavior is not reused
- Use traits to capture behavior that is reused in multiple, unrelated classes
- If clients will inherit the behavior, try to make it an abstract class

Generative Recursion

Generative vs Structural Recursion

- The functions we have studied to this point have (mostly) followed a common pattern:
  - Break into cases
  - Decompose data into components
  - Process components (usually recursively)
- Functions that follow this pattern are referred to as *structurally recursive functions*

Generative vs Structural Recursion

- Some problems are not amenable to solution by recursive descent
  - Instead, a deeper insight or “eureka” is required
  - Often a result from mathematics or computer science must be applied to discover important structure
  - Consider Euclid’s Algorithm for GCD
- The discovery of these insights and construction of solutions using them is the study of *algorithms*

Generative vs Structural Recursion

- Typically the design of an algorithm distinguishes two kinds of problems:
  - Base cases (or trivially solvable cases)
  - Problems that can be reduced to other problems of the same form
- The design of algorithms using this approach is referred to as *generative recursion*

Square Roots

- We would like to define a function `sqrt` that takes a non-negative value of type `Double` and returns the square root of that value

- There is no obvious way to apply structural recursion to this problem

Newton's Method

- We can use derivatives to find successively better approximations to the zeroes of a real-valued function:

$$f(x) = 0$$

Newton's Method

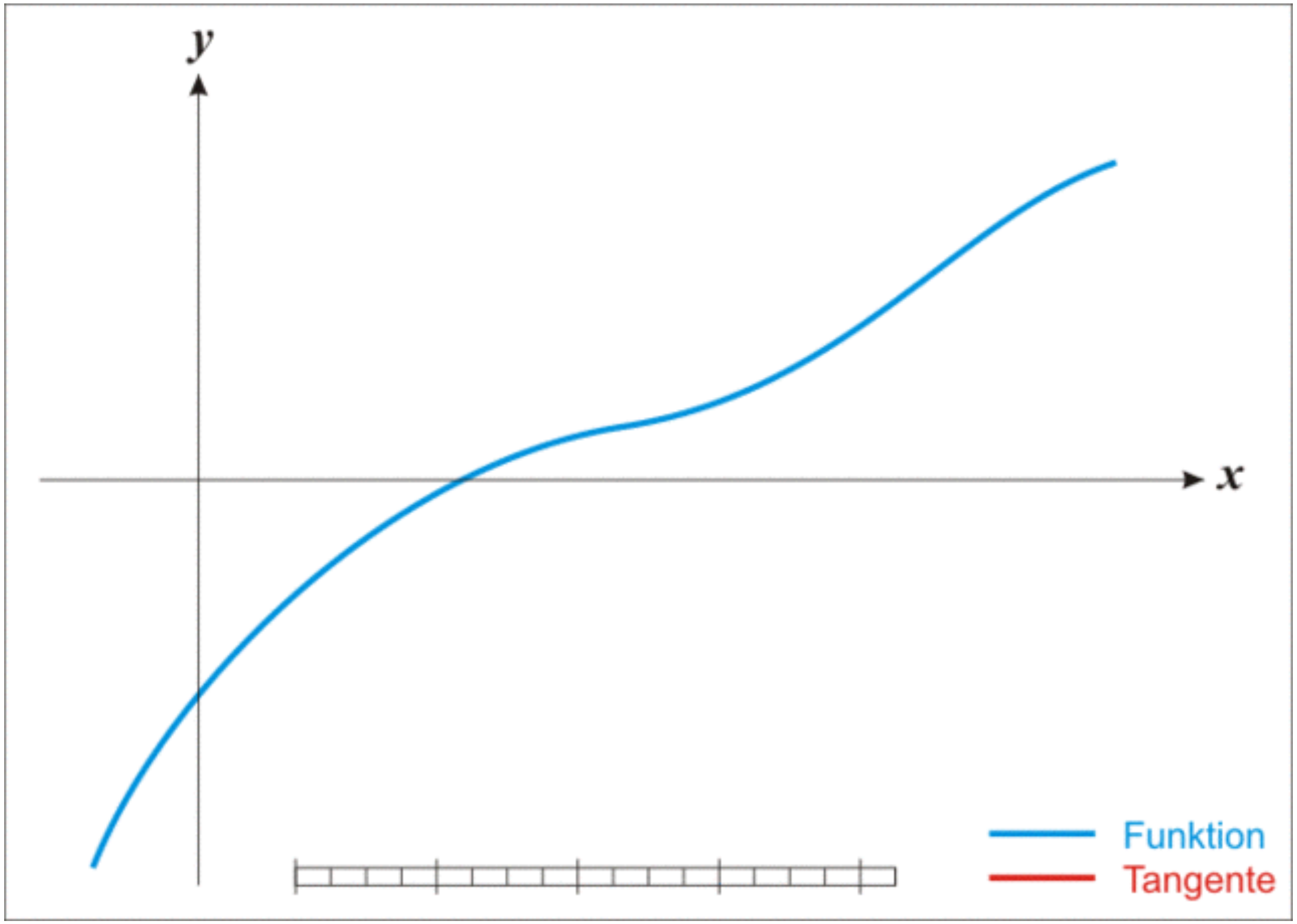
- We start with some guess for a value of `x`

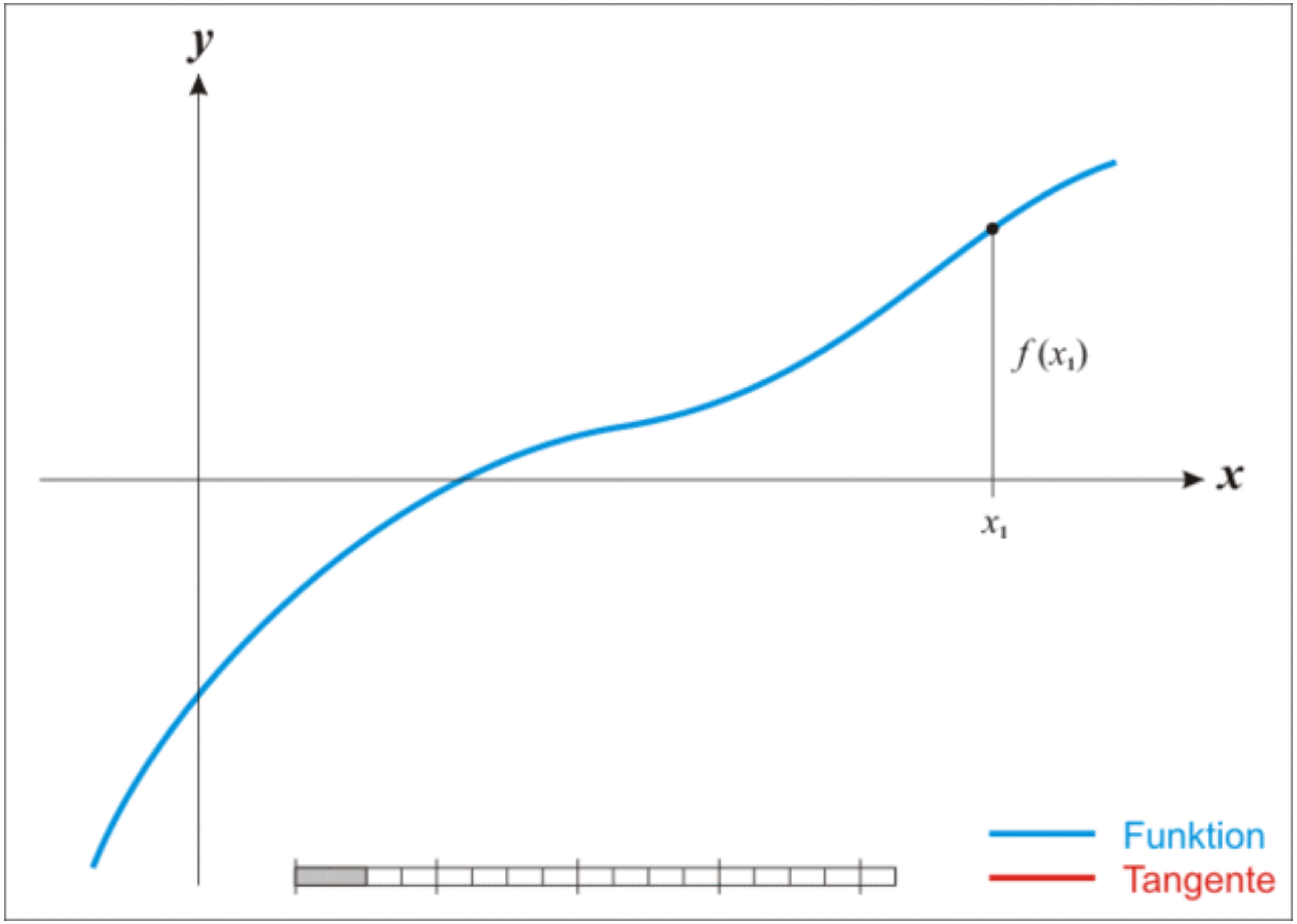
$$x_0 = \texttt{guess}$$

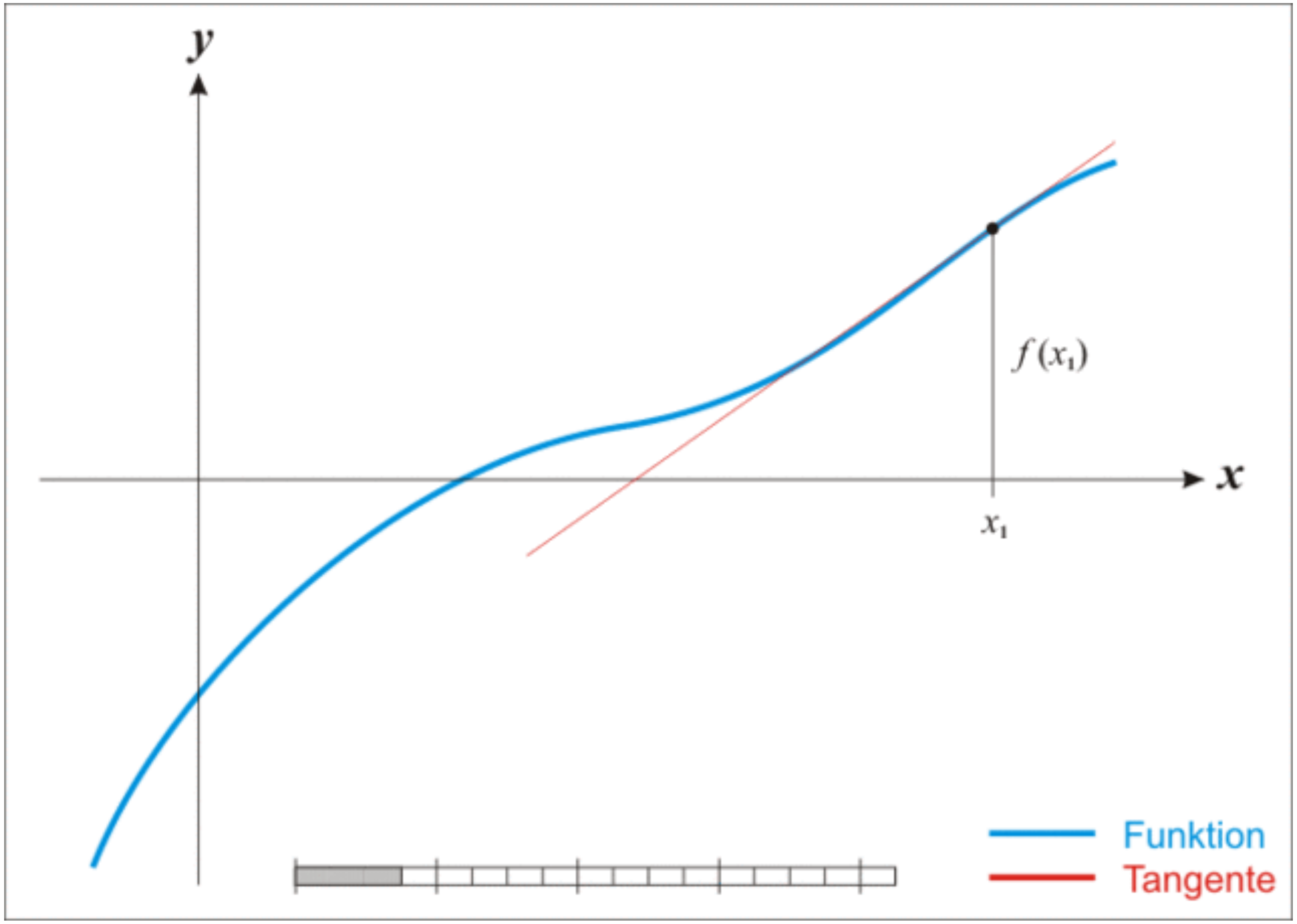
Newton's Method

- Then we construct a better approximation with the following formula:

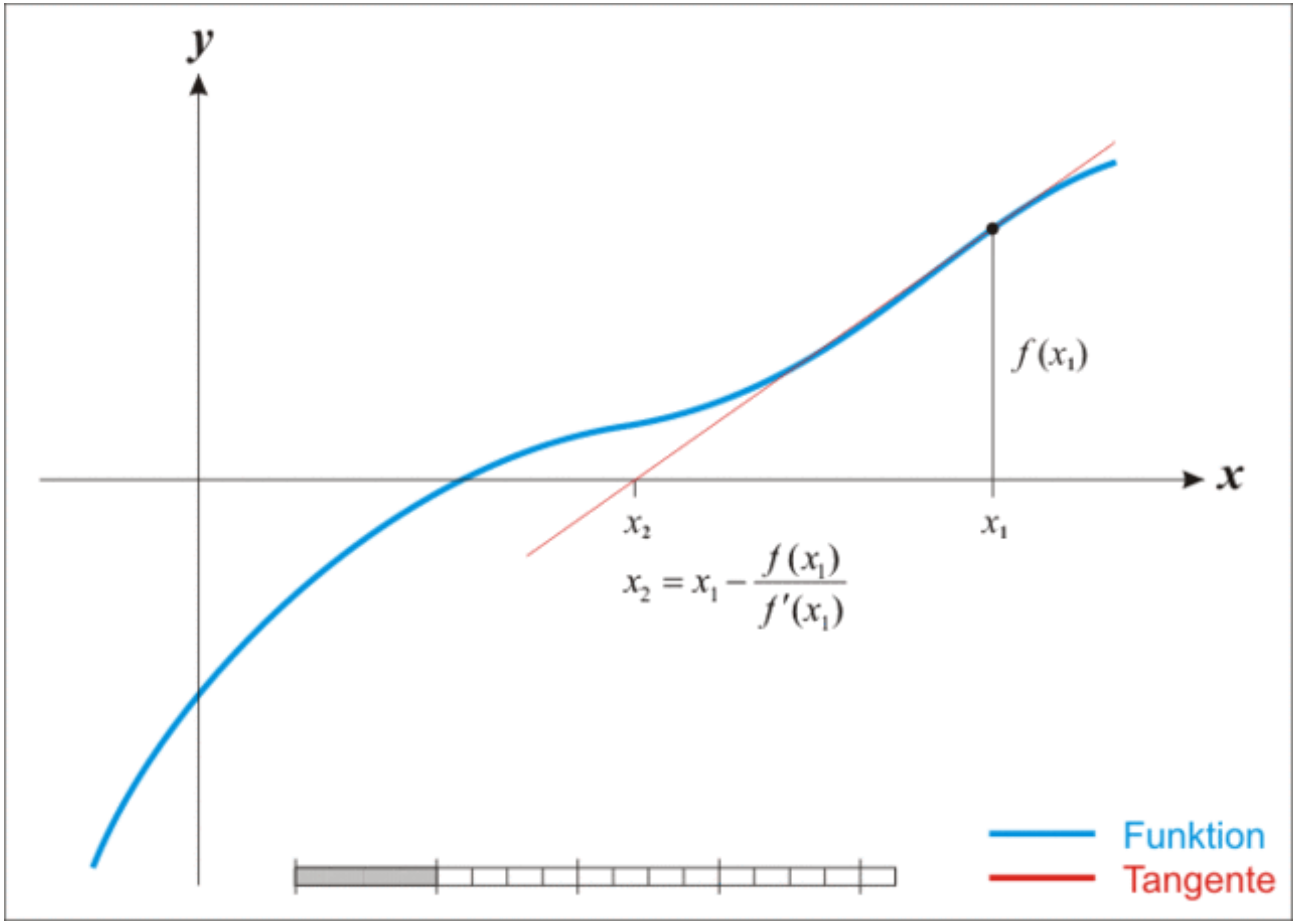
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$


y
x
Funktion
Tangente

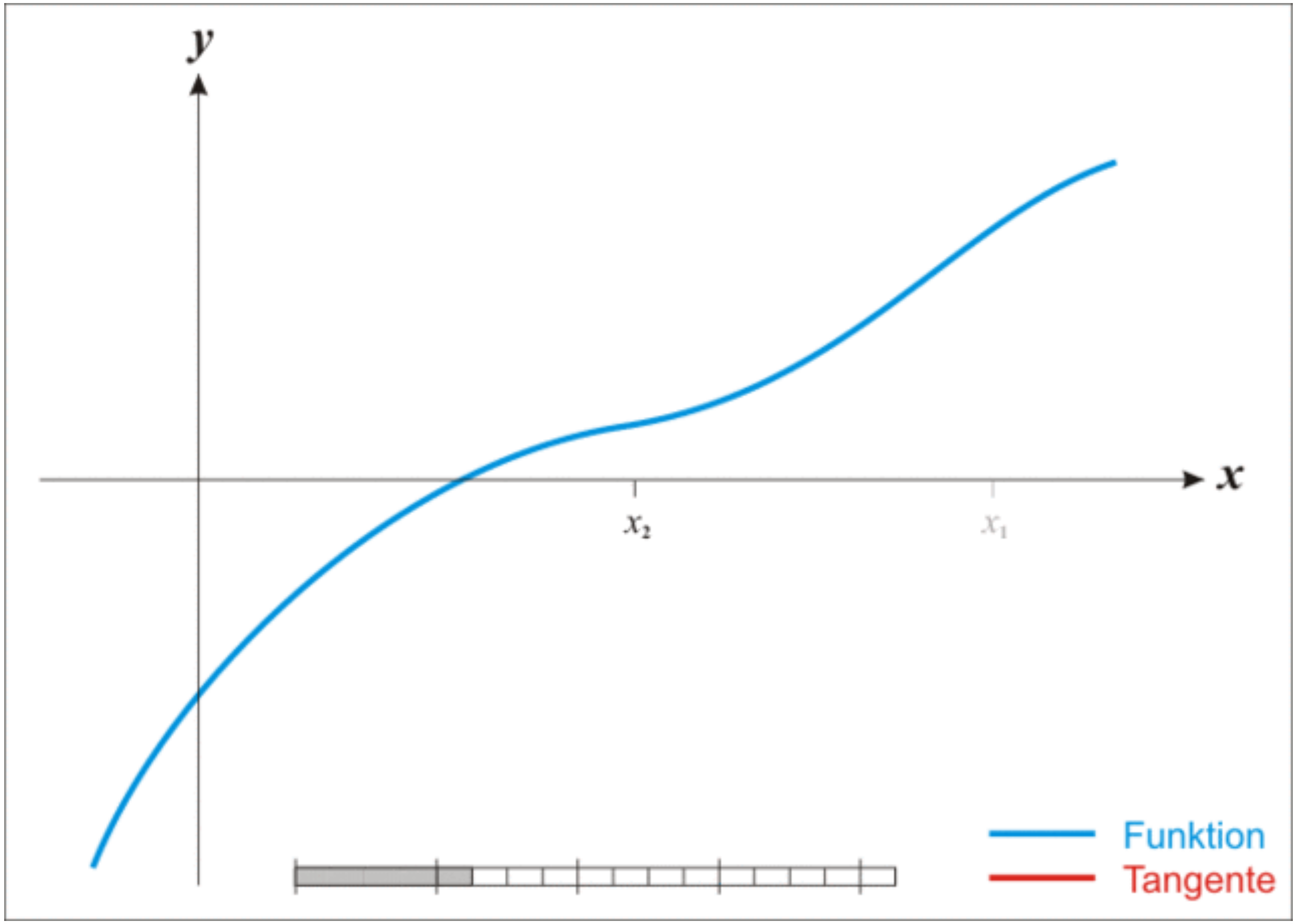

y
x
f (x1)
x1
Funktion
Tangente

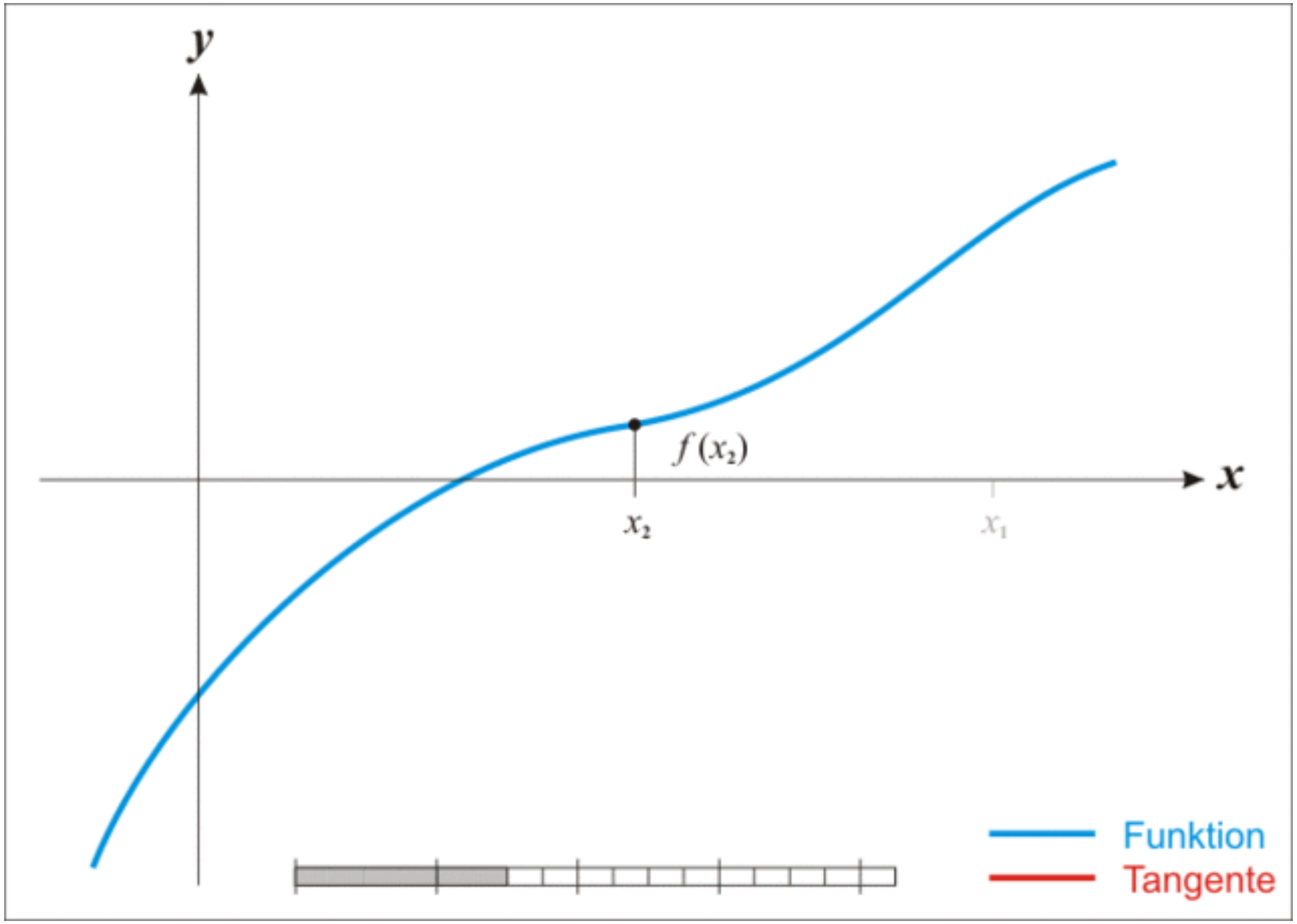


y
x
f (x1)
x1
Funktion
Tangente

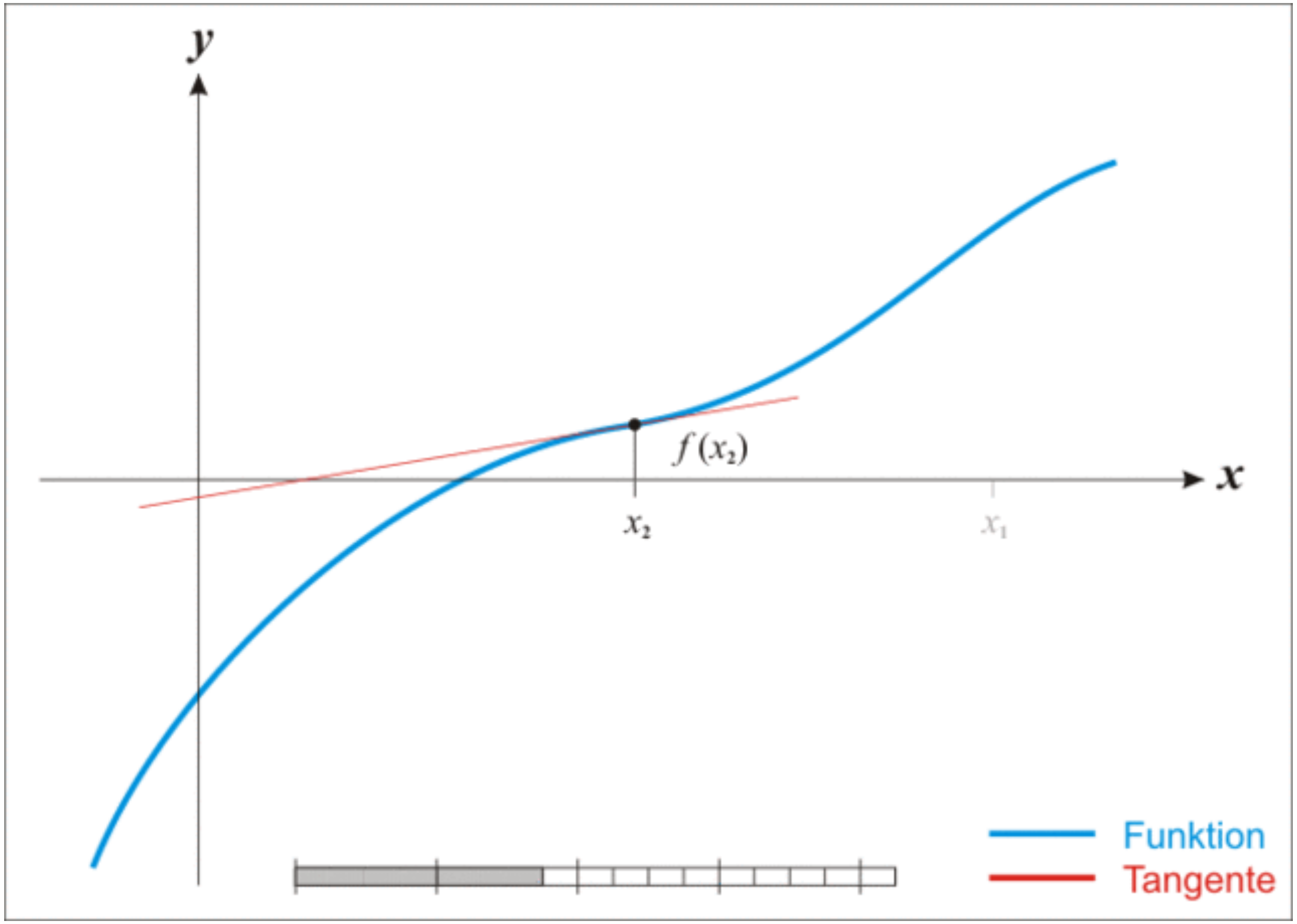


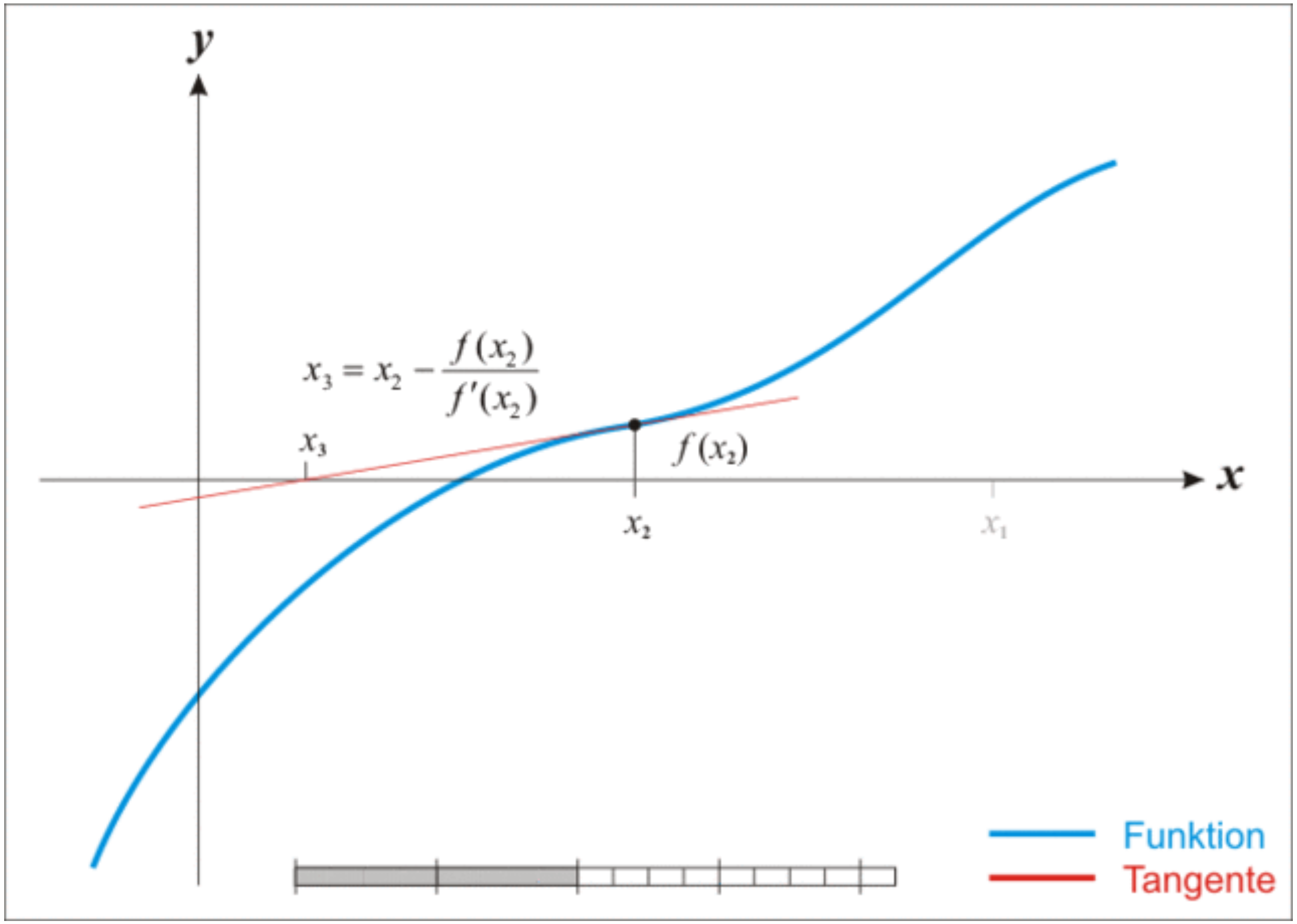
y
x
$f(x_1)$
x_2
x_1
$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$
Funktion
Tangente


y
x
x_2
x_1
Funktion
Tangente

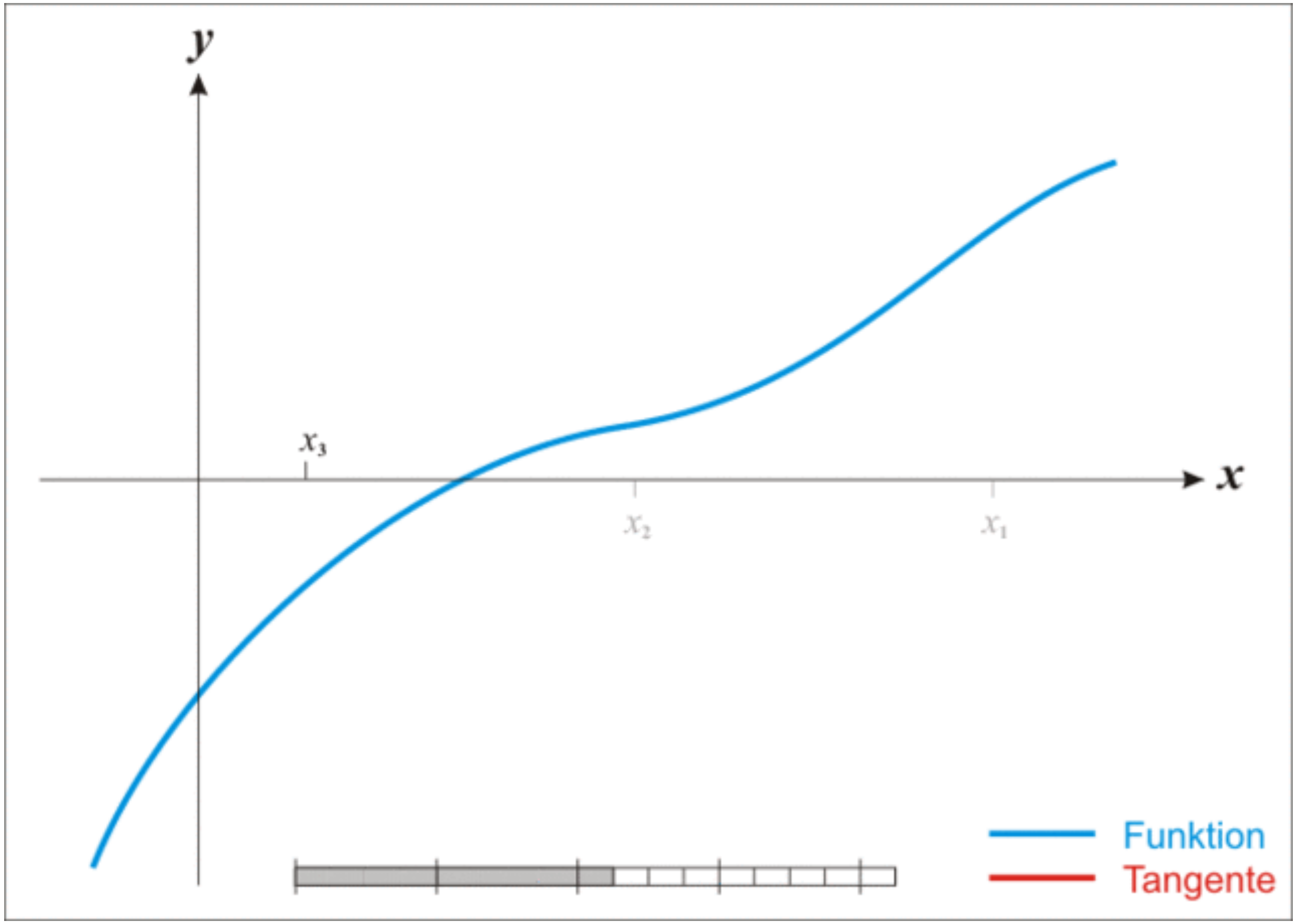


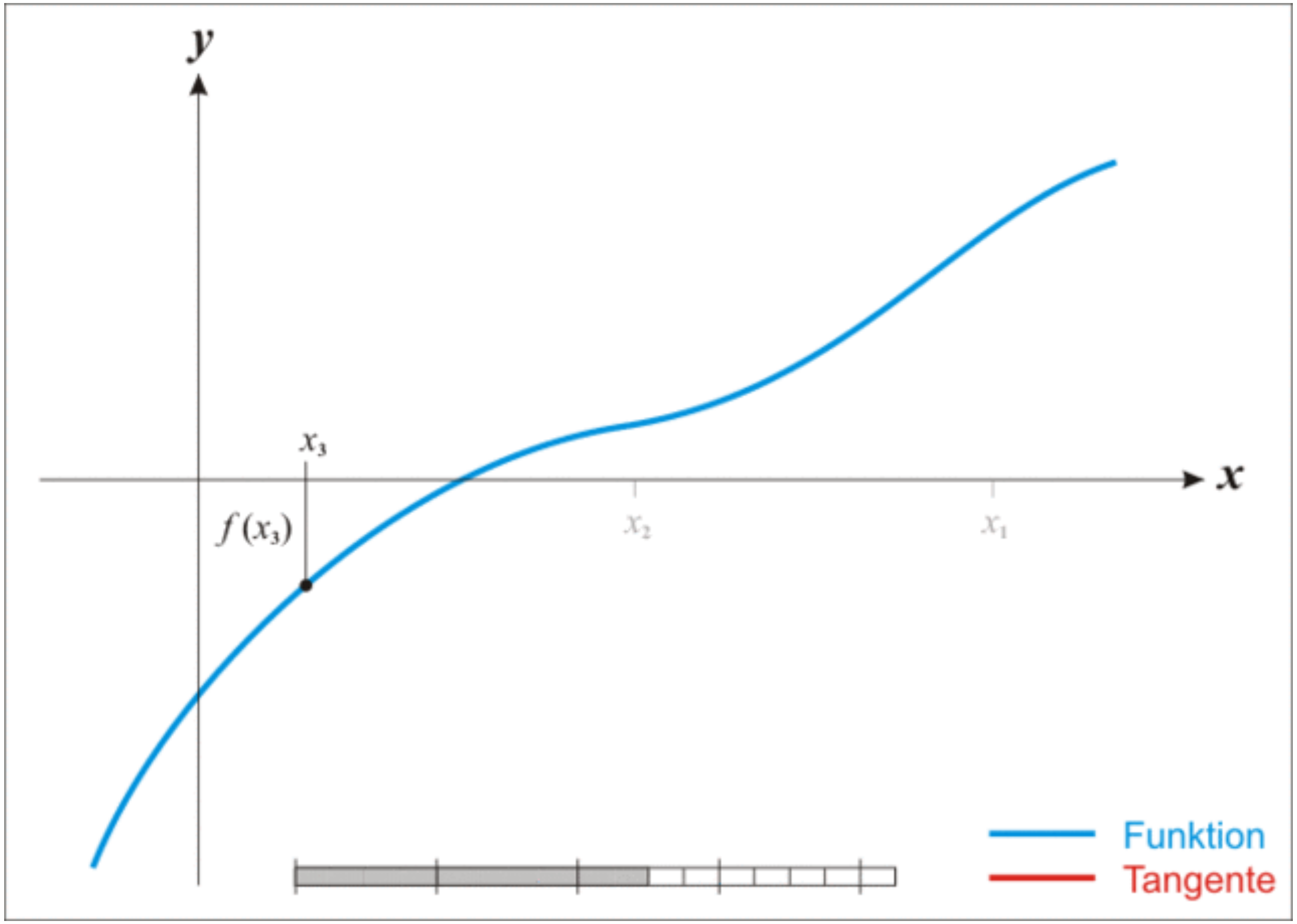
y
x
f (x2)
x2
x1
Funktion
Tangente


y
x
f(x2)
x2
x1
Funktion
Tangente

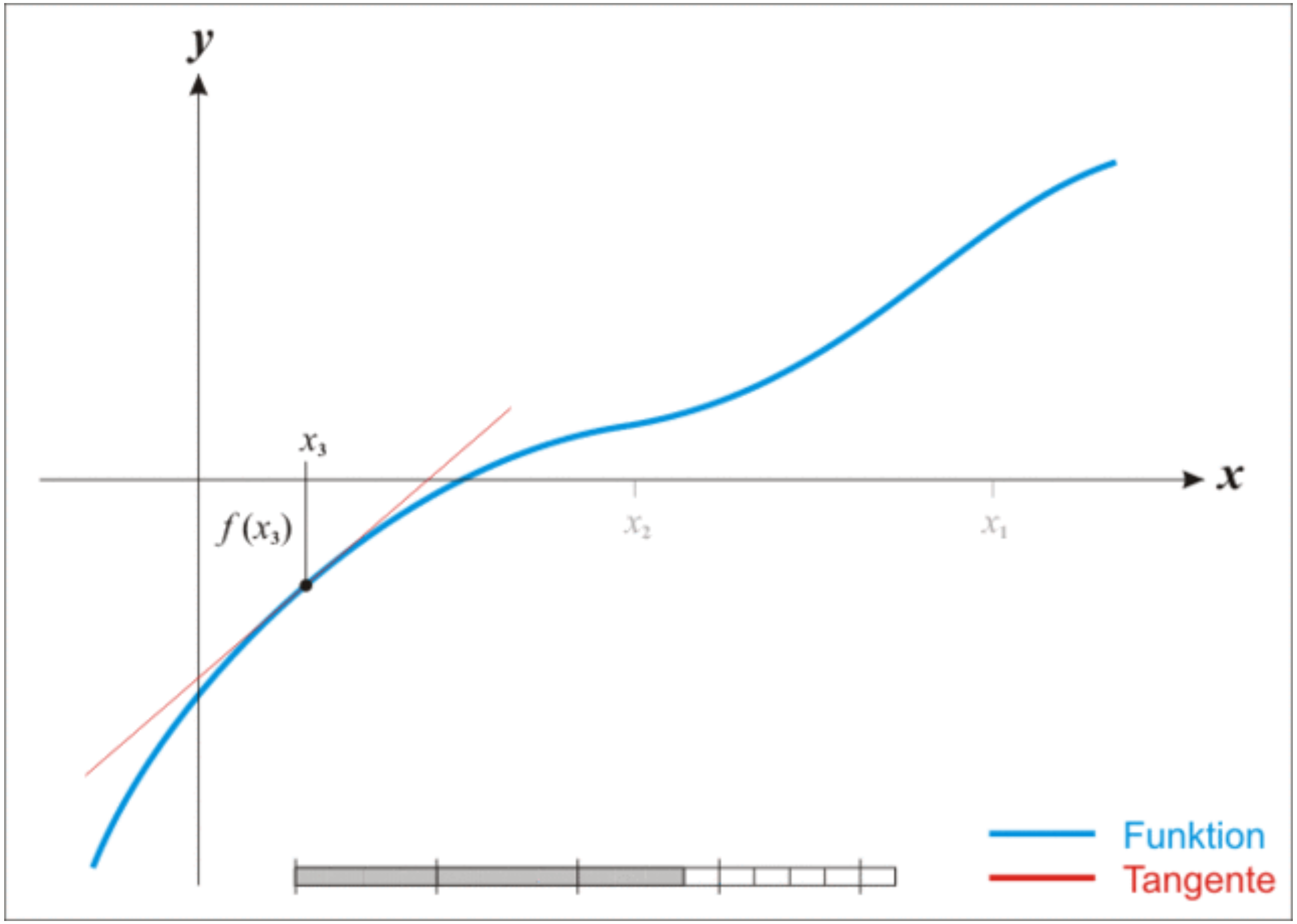


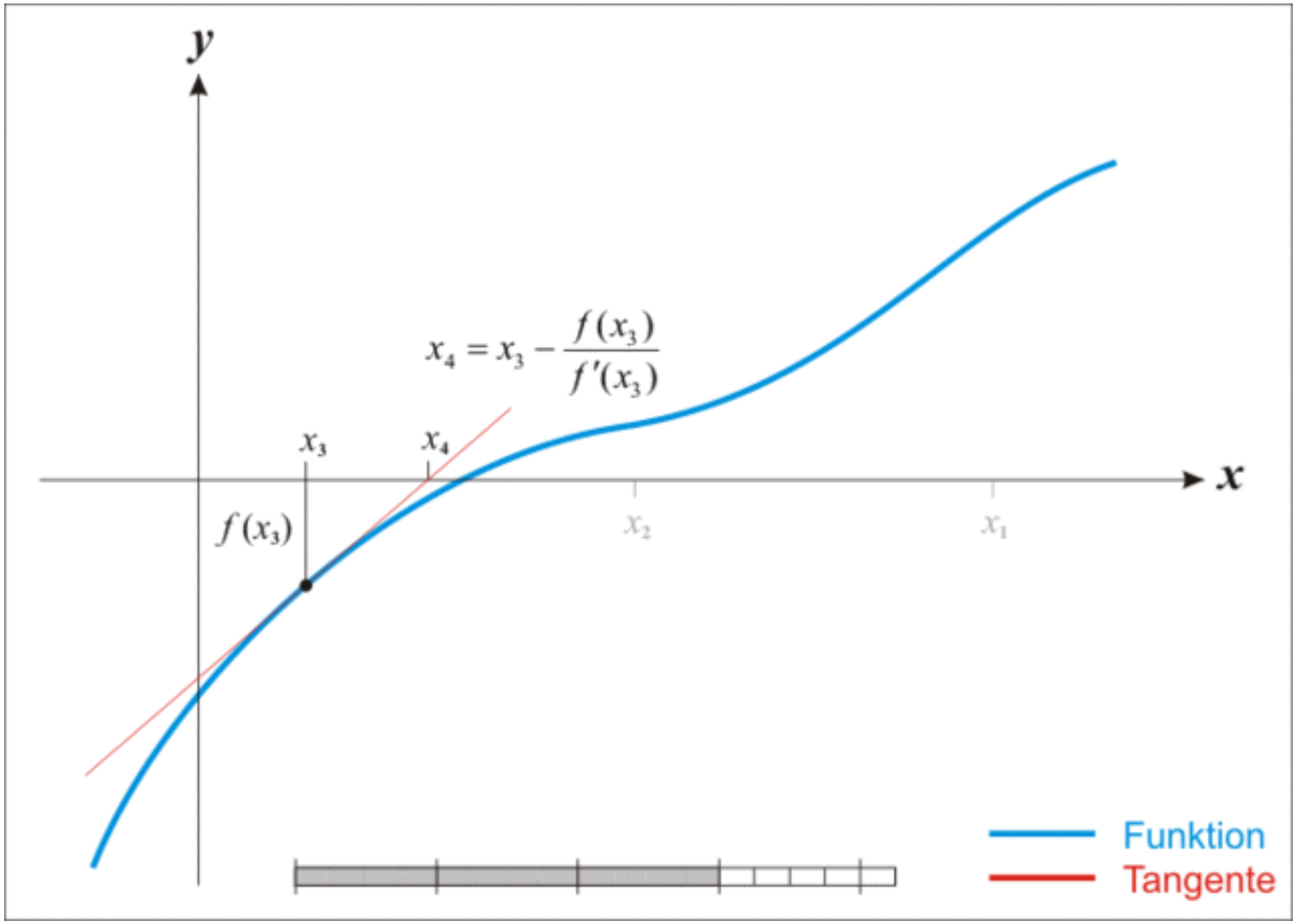
y
x
$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
x_3
$f(x_2)$
x_2
x_1
Funktion
Tangente

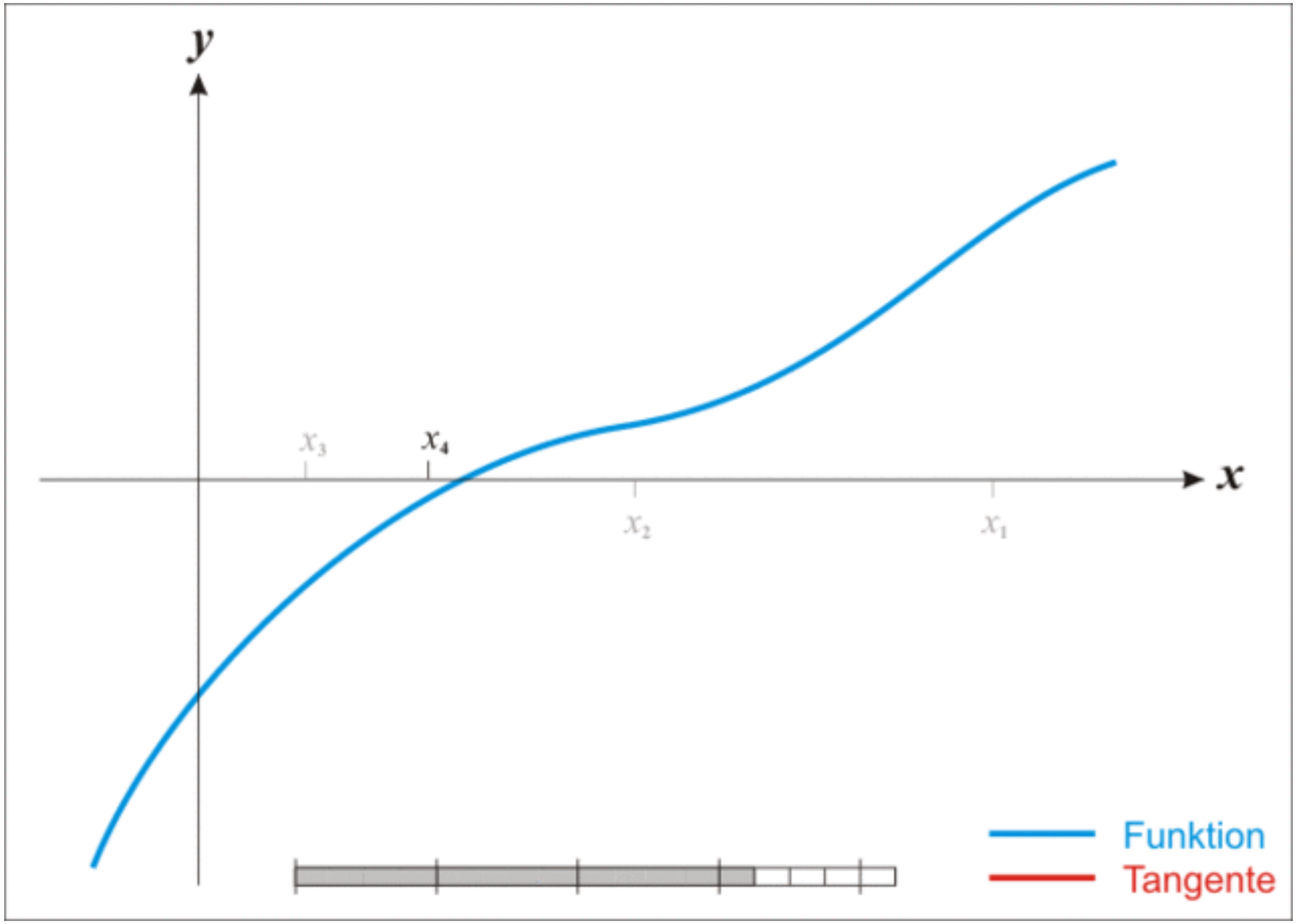

y
x
x_3
x_2
x_1
Funktion
Tangente



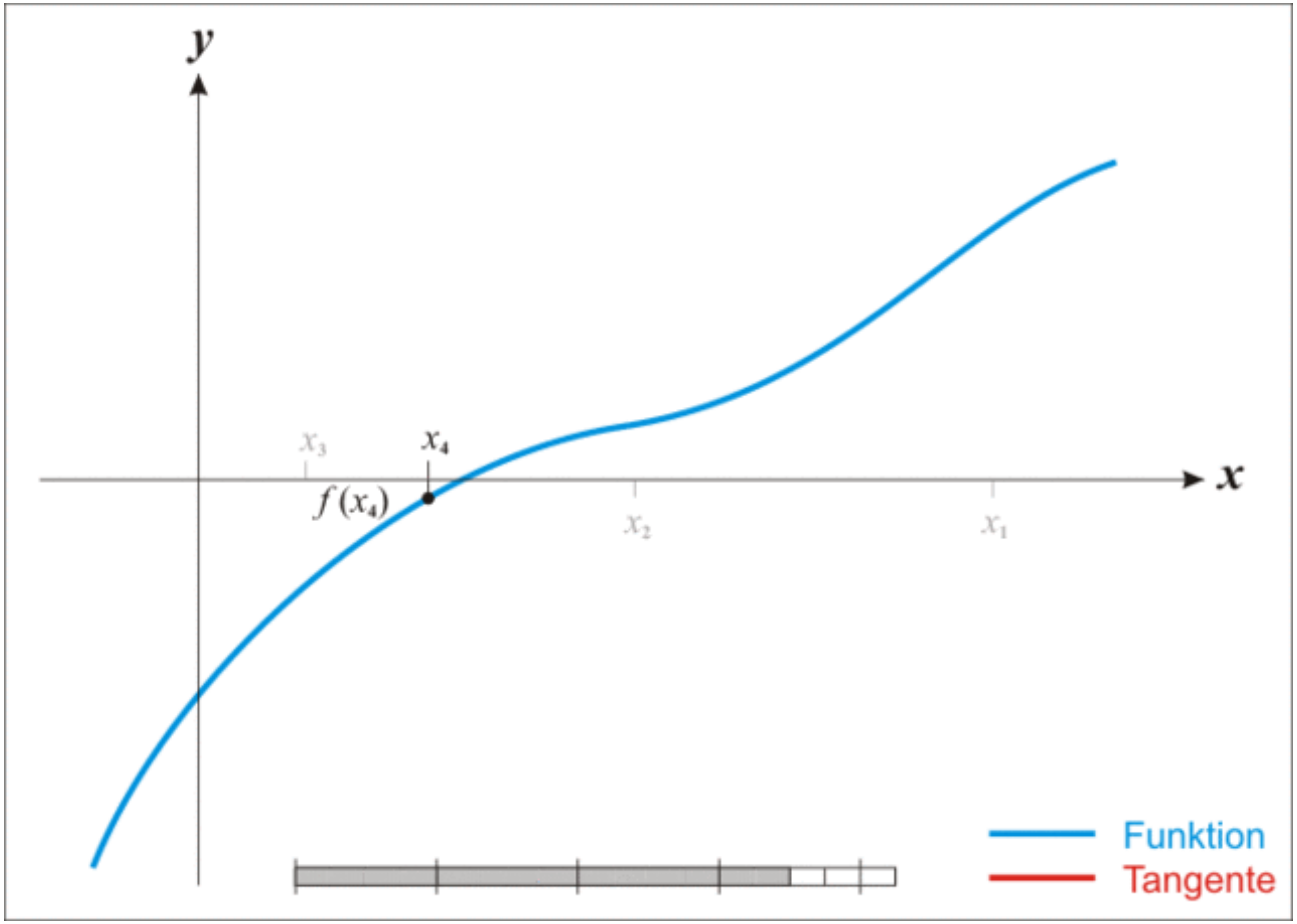
y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente


y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente

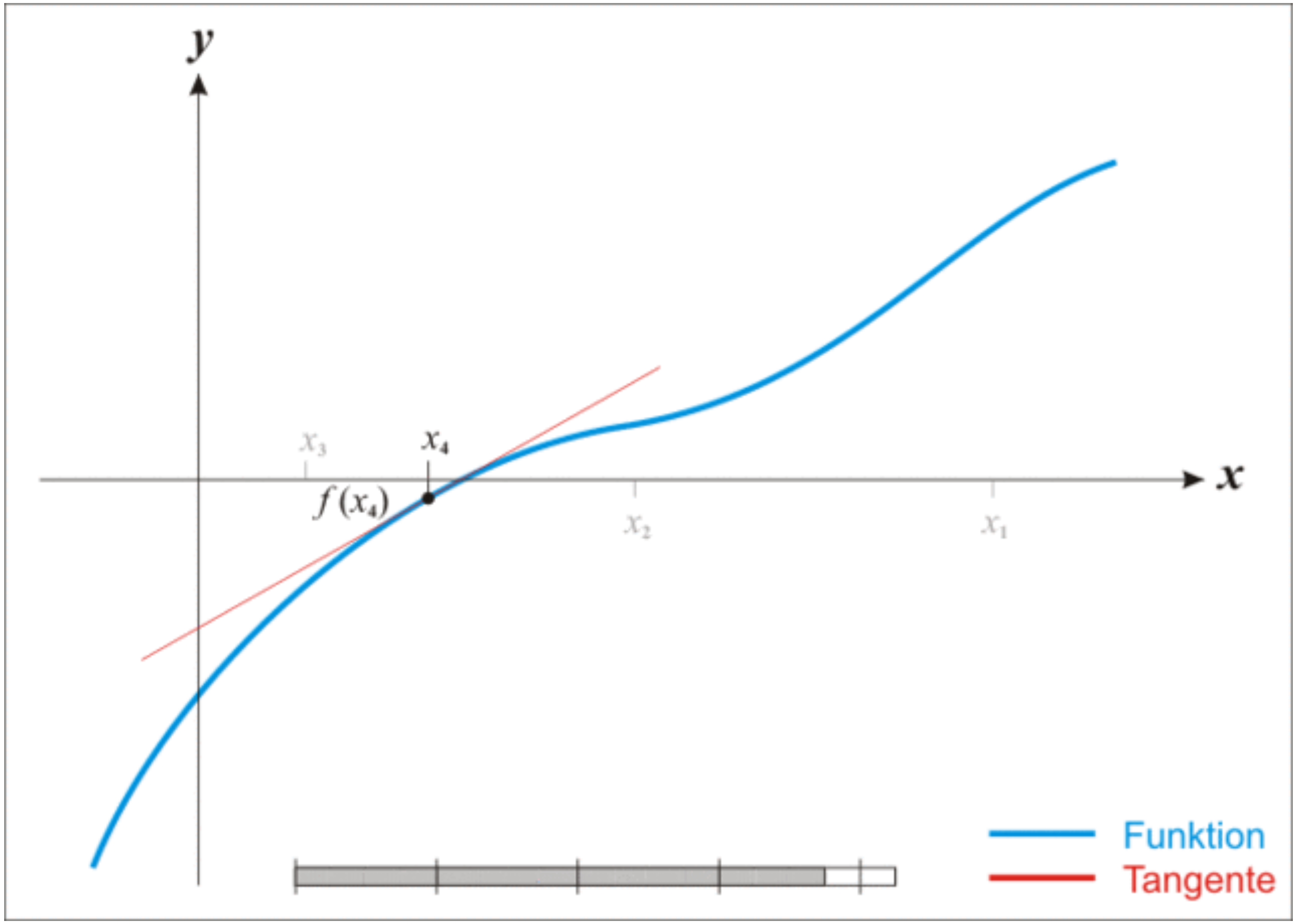

y
x
$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)}$
x_3
x_4
$f(x_3)$
x_2
x_1
Funktion
Tangente

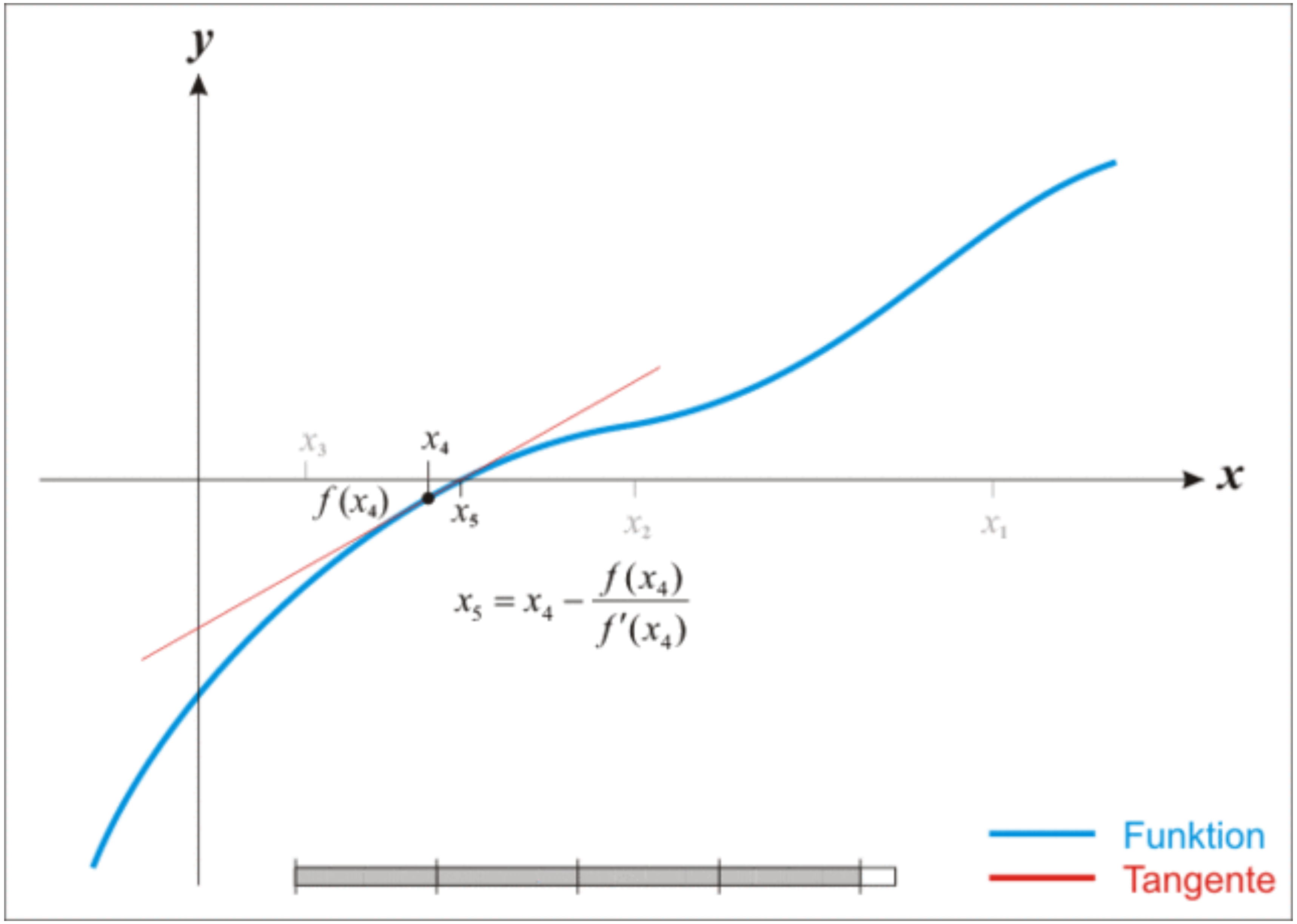


y
x
x_3
x_4
x_2
x_1
Funktion
Tangente

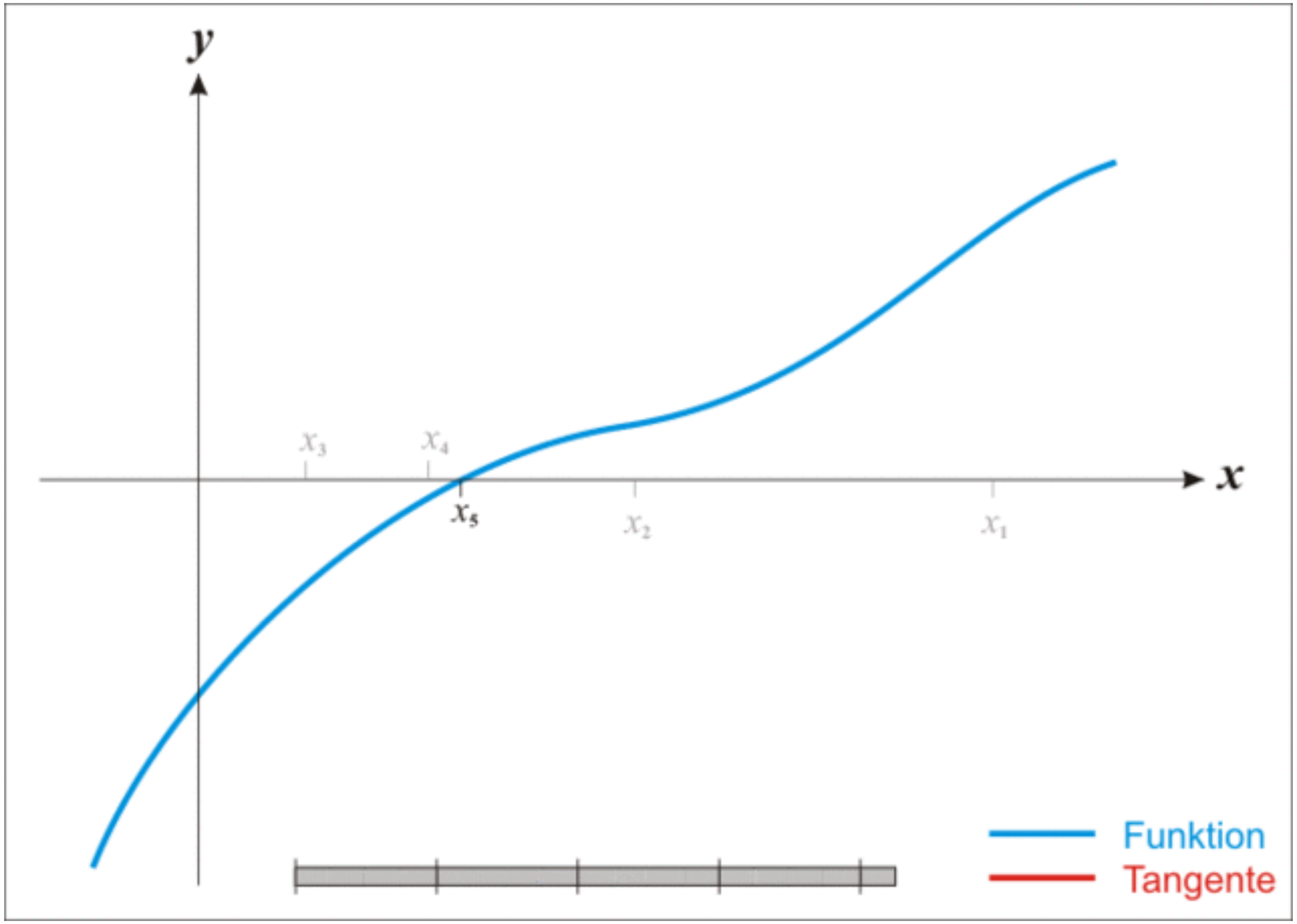


y
x
x3
x4
f (x4)
x2
x1
Funktion
Tangente


y
x
x3
x4
f (x4)
x2
x1
Funktion
Tangente



y
x
x_3
x_4
$f(x_4)$
x_5
x_2
x_1
$x_5 = x_4 - \frac{f(x_4)}{f'(x_4)}$
Funktion
Tangente


y
x
x_3
x_4
x_5
x_2
x_1
Funktion
Tangente

Applying Newton's Method to Finding Square Roots

- We can view the process of finding the square root of a number y as finding a solution to the equation:

$$x^2 = y$$

Applying Newton's Method to Finding Square Roots

- We can view the process of finding the square root of a number y as finding a solution to the equation:

$$x^2 - y = 0$$

Applying Newton's Method to Finding Square Roots

- Equivalently, we want to find a zero to the function:

$$f(x) = x^2 - y$$

Newton's Method

- Plugging in our function `f`:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton's Method

- Plugging in our function `f`:

$$x_{n+1} = x_n - \frac{x_n^2 - y}{2x_n}$$

Newton's Method

```
def abs(x: Double) = if (x < 0) -x else x
def square(x: Double) = x * x
```

Newton’s Method

- To encode Newton’s Method as an application of generative recursion:
  - We need to choose an initial guess
  - We need to encode computation of the next guess from our current guess
  - We need to determine our base case

Newton's Method

- For square roots:
  - Our initial guess can be the parameter
  - Our base case is that our current guess falls within some tolerance of the true square root

Newton's Method

```
def next(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else next(guess - ((square(guess) - x) /
                     (2 * guess)))
```

Newton's Method

```scala
val epsilon = 0.000000000000001

def isGoodEnough(guess: Double) =
  abs(square(guess) - x) <= epsilon
```

Newton’s Method

```
def sqrt(x: Double) = {
  val epsilon = 0.000000000000001

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) <= epsilon

  def next(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else next(guess - ((square(guess) - x) /
                      (2 * guess)))

  next(x)
}
```

Generalizing to an Arbitrary Function

```
def newtonsMethod(f: Double => Double) = {
  val epsilon = 0.000000000000001
  val delta = 0.000000001

  def isGoodEnough(guess: Double) = abs(f(guess)) <= epsilon

  def fPrime(x: Double) = (f(x + delta) - f(x)) / delta

  def next(guess: Double): Double = {
    if (isGoodEnough(guess)) guess
    else next(guess - f(guess) / fPrime(guess))
  }
  next(2)
}
```

Generalizing to an Arbitrary Function

```
> newtonsMethod((x: Double) => x*x - 2)
res1: Double = 1.414213562373095

> newtonsMethod((x: Double) => x*x*x - 1000)
res0: Double = 10.0
```

Not All Applications of Newton's Method Terminate

- Consider:

$$f(x) = x^2 - x$$

$$f'(x) = 2x - 1$$

- An initial guess of 0.5 leads us to find the root of a tangent with slope zero (which has no root!)

Not All Applications of Newton's Method Terminate

```
newtonsMethod((x: Double) => x*x - x) ↦ ⊥
```

Design Recipe for Generative Recursion

- Data analysis and design
- Contract, purpose, header: Should now include some description of how the function works
- Examples: Include examples that illustrate how the function proceeds (not just input/output)

Design Recipe for Generative Recursion

- Template:
  - What is trivially solvable?
  - We new sub-problems do we generate?
  - How do we combine solutions to the sub-problems?
- Tests
- A termination argument

A Termination Argument

- With structural recursion, the computation follows the structure of the data
- Because immutable data has no cycles, the computation is certain to terminate
- With generative recursion, the sub-problems might be as large as the original problem
- Thus, we should include an explicit argument that the algorithm terminates