Measuring Performance

Rasmus Resen Amossen

http://rasmus.resen.org

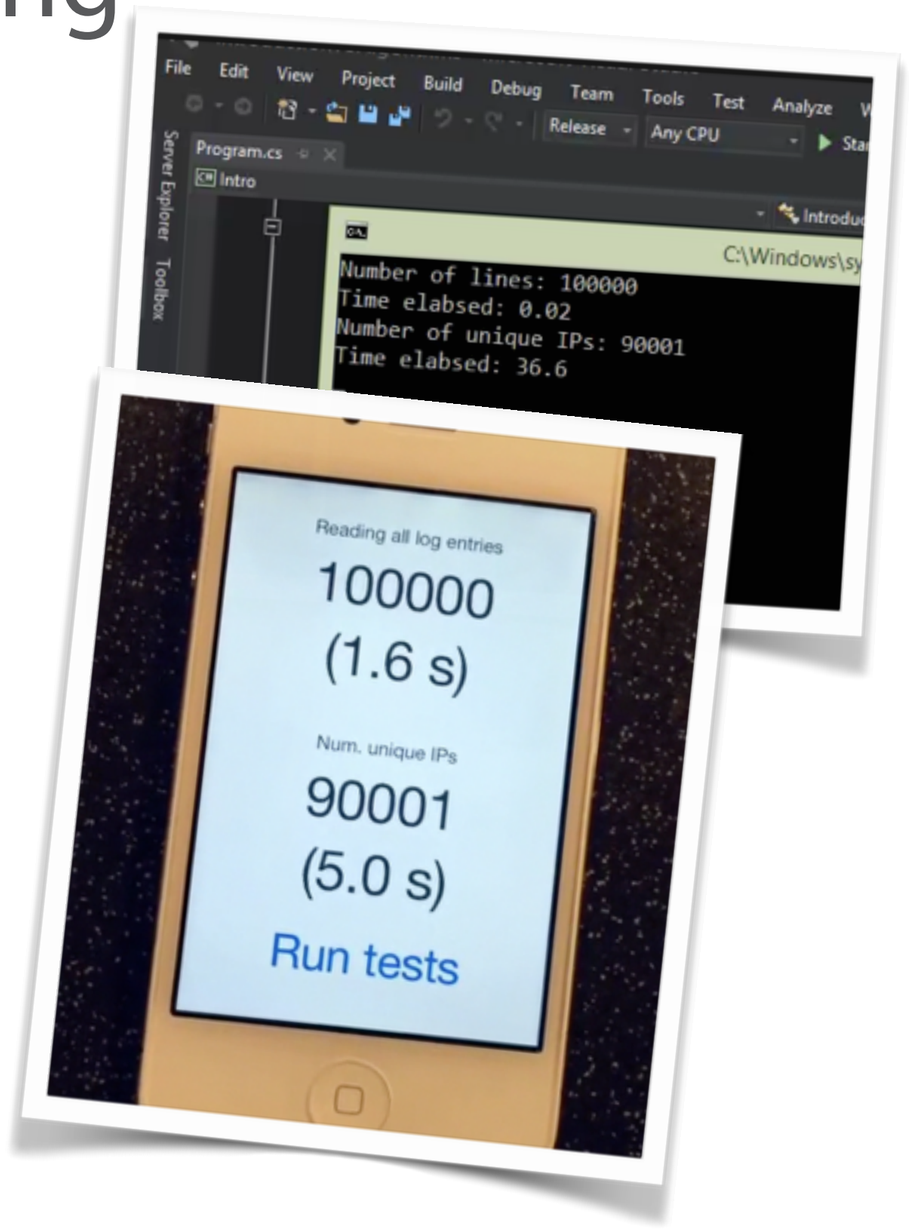
Ways of Measuring

Ways of Measuring

Timing with stopwatch

Ways of Measuring

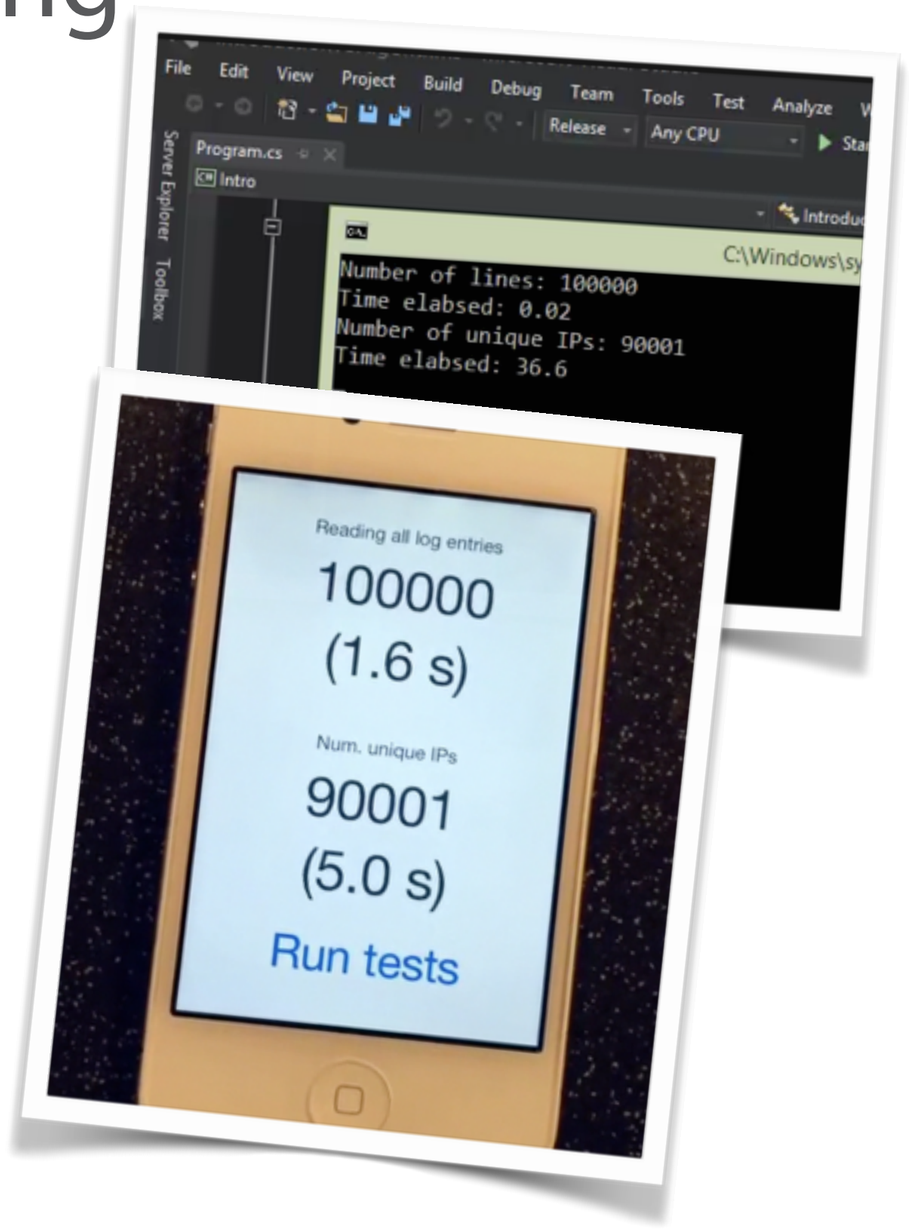
Timing with stopwatch



Ways of Measuring

Timing with stopwatch

Depends on…
hardware,
programming language,
environment,
etc.



Ways of Measuring

Timing with stopwatch

Counting instructions

Ways of Measuring

```
func readAllLogs() -> Int
{
    var reader = LogReader();
    var linesSeen = 0;
    for logLine in reader.GetLogLines() {
        logLine.getIP();
        linesSeen++;
    }
    return linesSeen;
}
```

Counting instructions

Ways of Measuring

```
func readAllLogs() -> Int
{
    var reader = LogReader();
    var linesSeen = 0;
    for logLine in reader.GetLogLines() {
        logLine.getIP();
        linesSeen++;
    }
    return linesSeen;
}
```

Counting instructions

100,000 lines:
2 + 100,000 · 4 + 1 instructions

Ways of Measuring

```
func readAllLogs() -> Int
{
    var reader = LogReader();
    var linesSeen = 0;
    for logLine in reader.GetLogLines() {
        logLine.getIP();
        linesSeen++;
    }
    return linesSeen;
}
```

Counting instructions

100,000 lines:
2 + 100,000 · 4 + 1 instructions

N lines:
2 + *N* · 4 + 1 instructions

Ways of Measuring

```
func readAllLogs() -> Int
{
    var reader = LogReader();
    var linesSeen = 0;
    for logLine in reader.GetLogLines() {
        logLine.getIP();
        linesSeen++;
    }
    return linesSeen;
}
```

Counting instructions

How does execution time grow, when *N* grows?

100,000 lines:
$2 + 100{,}000 \cdot 4 + 1$ instructions

N lines:
$2 + N \cdot 4 + 1$ instructions

Ways of Measuring

Timing with stopwatch

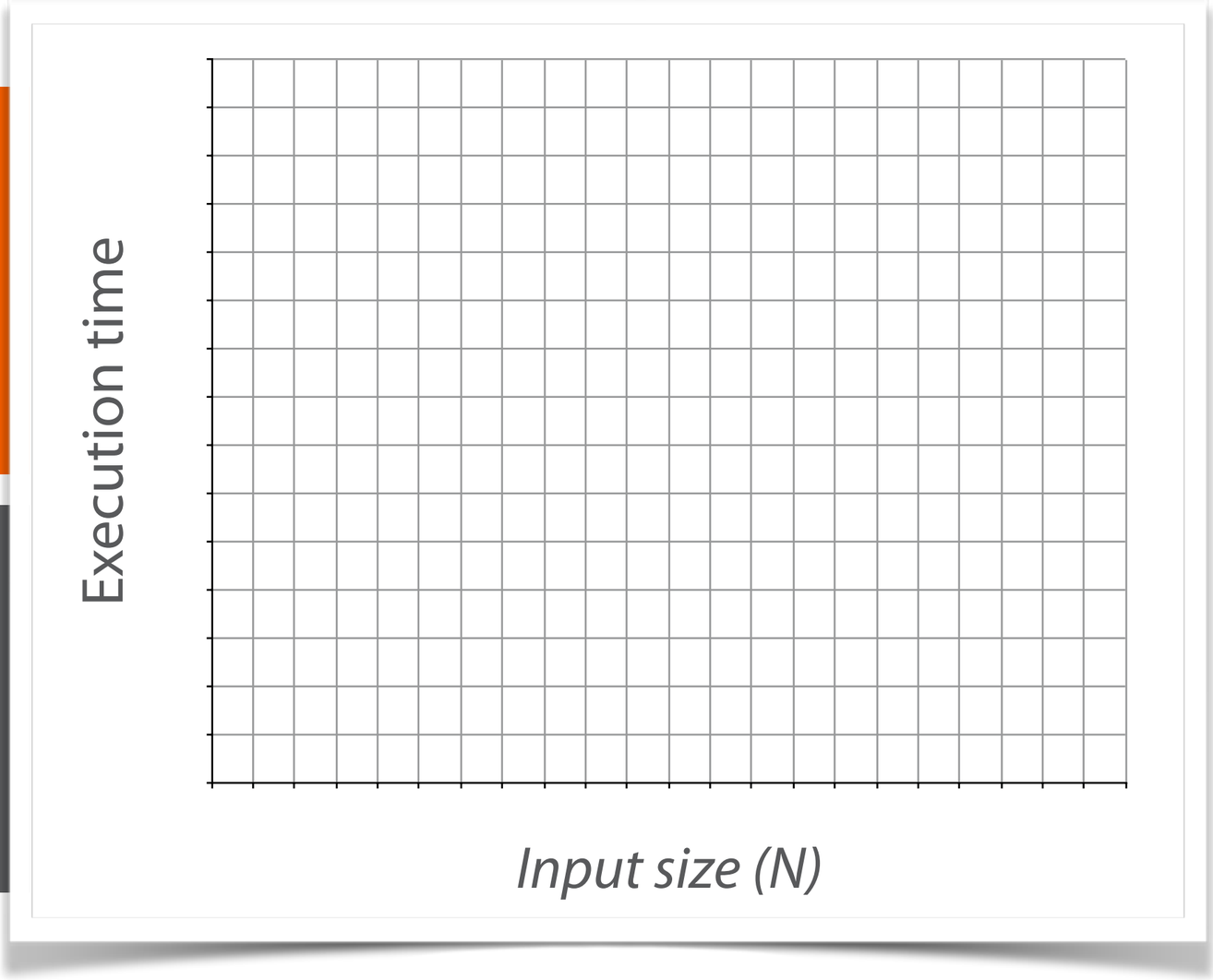
Counting instructions

Looking at the curve

Ways of Measuring

Timing with stopwatch

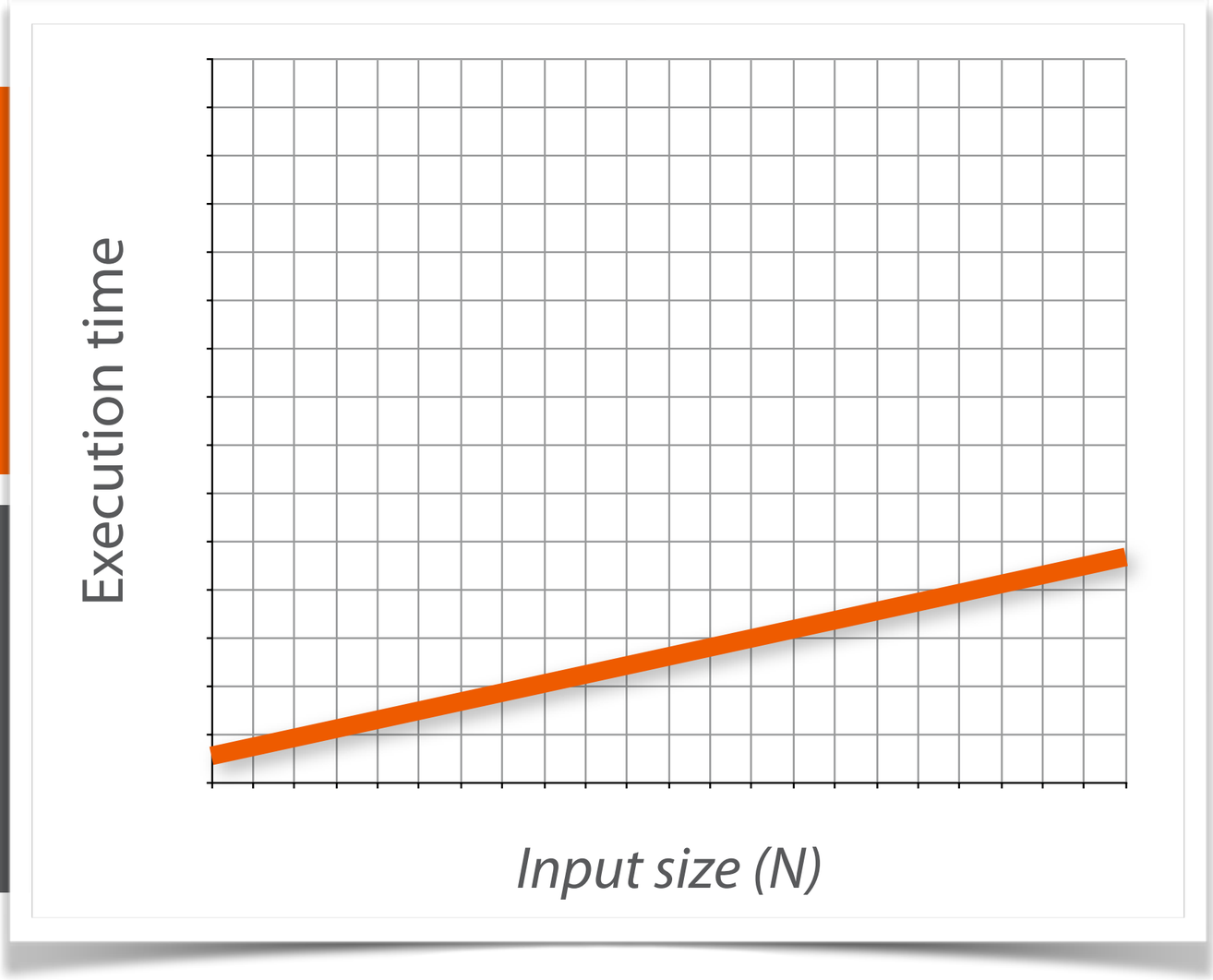
Looking at the curve



Ways of Measuring

Timing with stopwatch

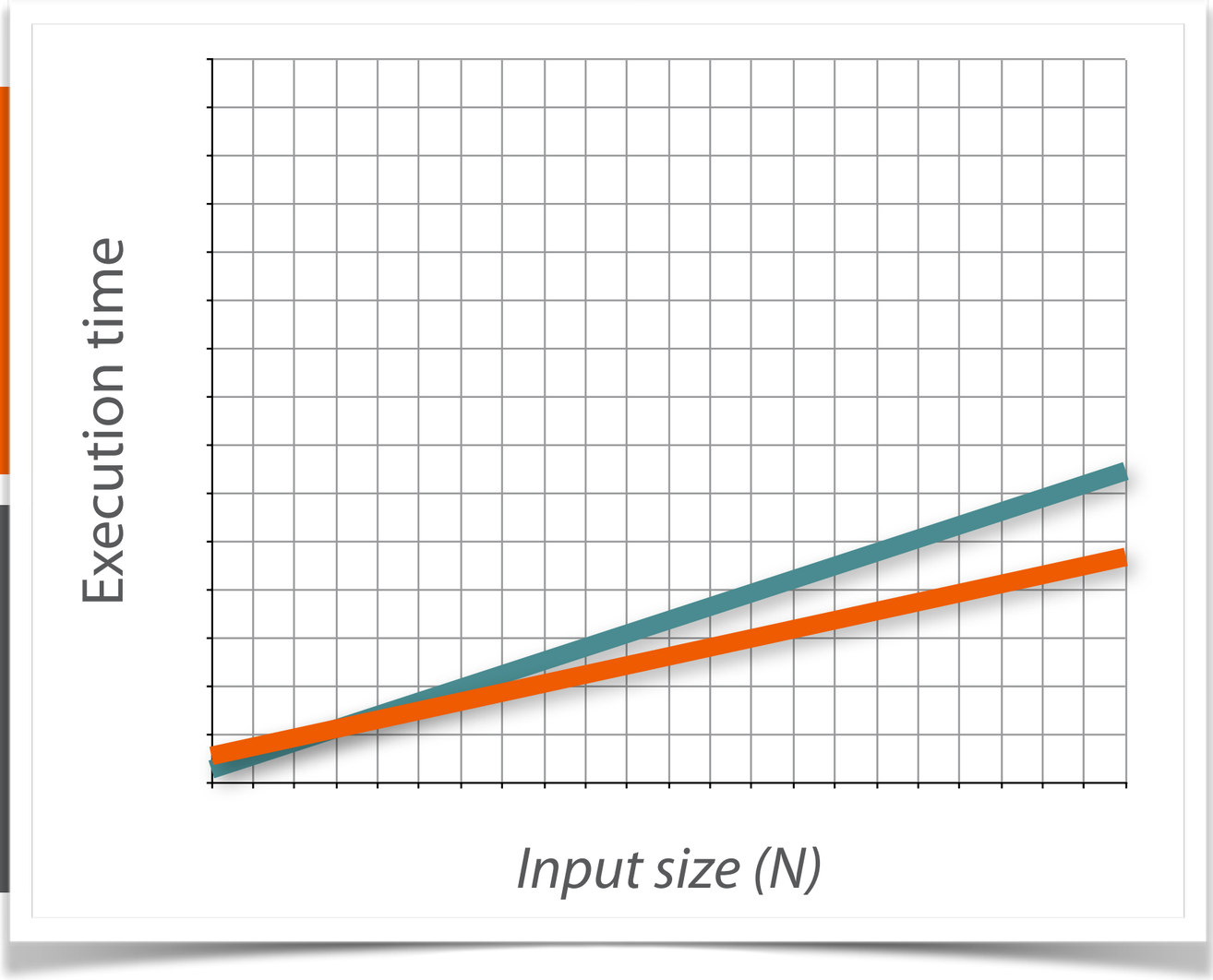
Looking at the curve



Ways of Measuring

Timing with stopwatch

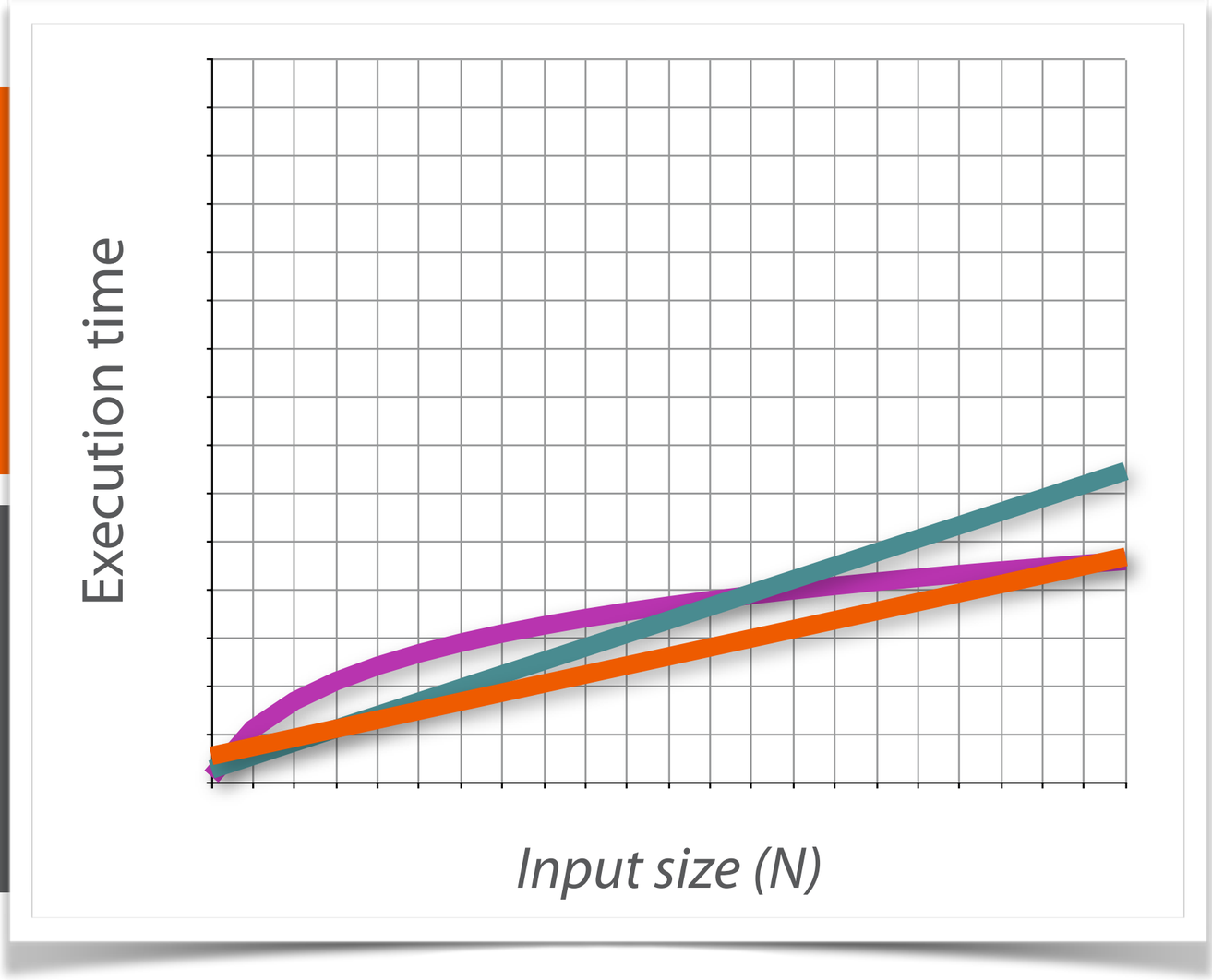
Looking at the curve



Ways of Measuring

Timing with stopwatch

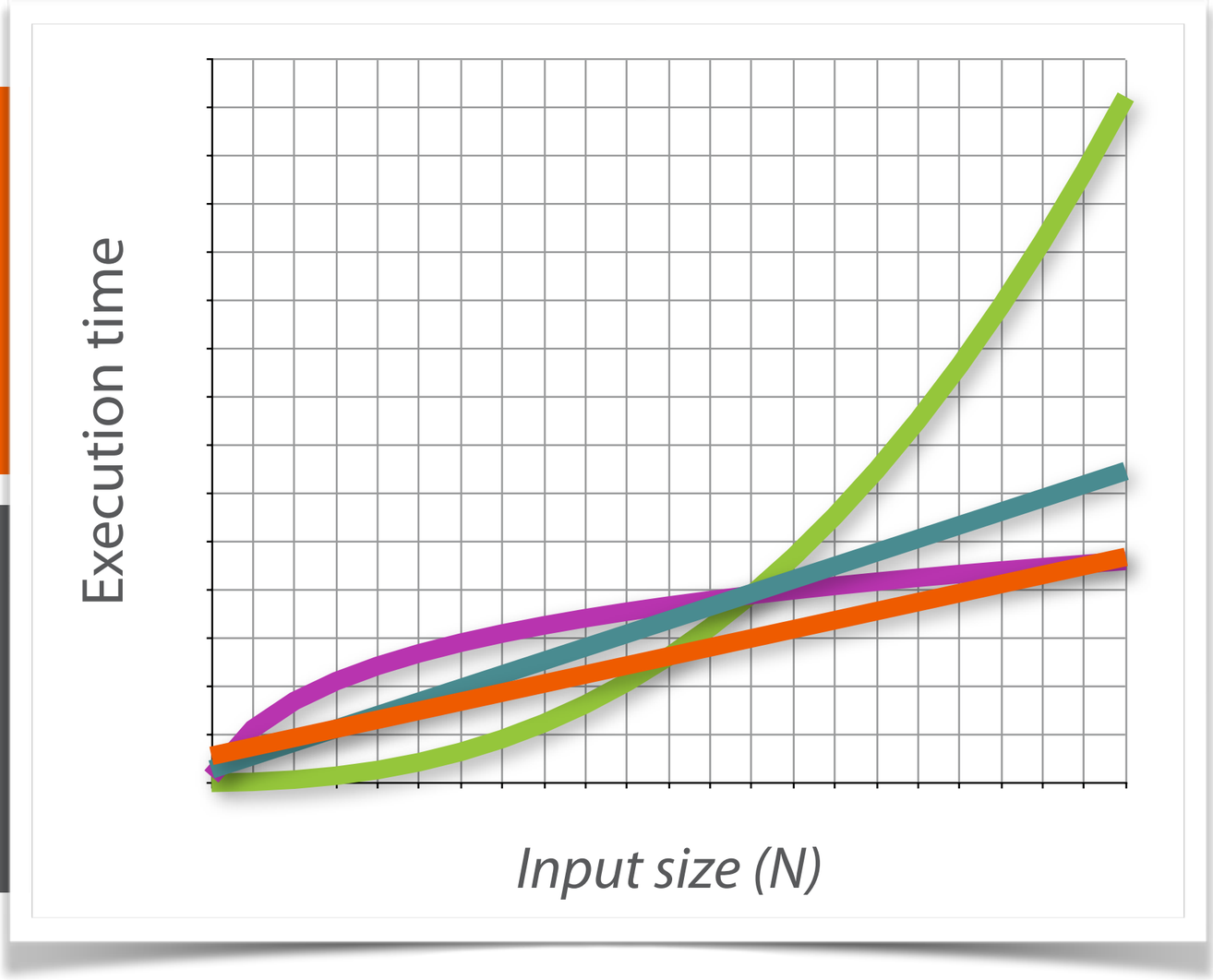
Looking at the curve



Ways of Measuring

Timing with stopwatch

Looking at the curve



Ways of Measuring

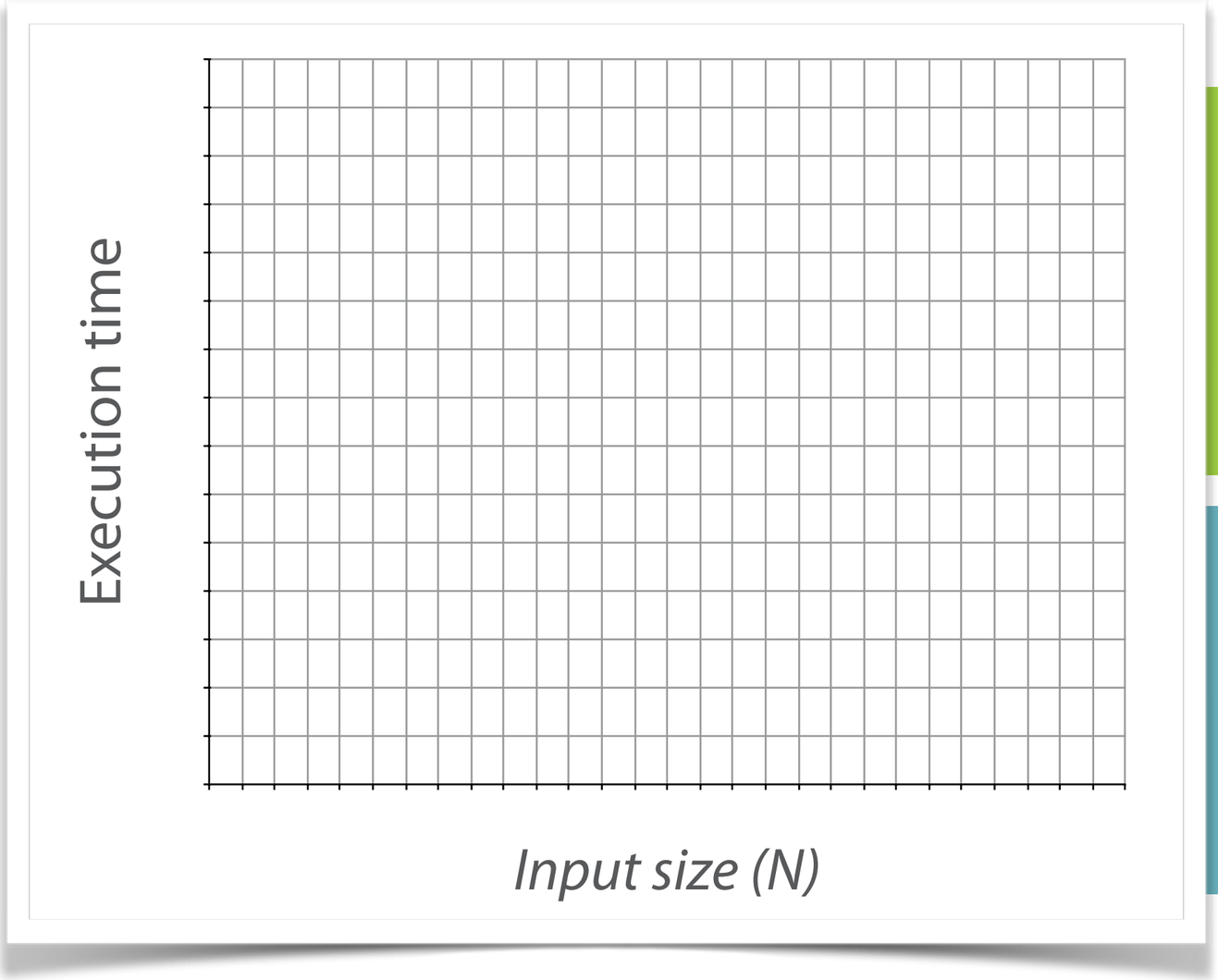
Timing with stopwatch

Counting instructions

Looking at the curve

Best case
Worst case
Average

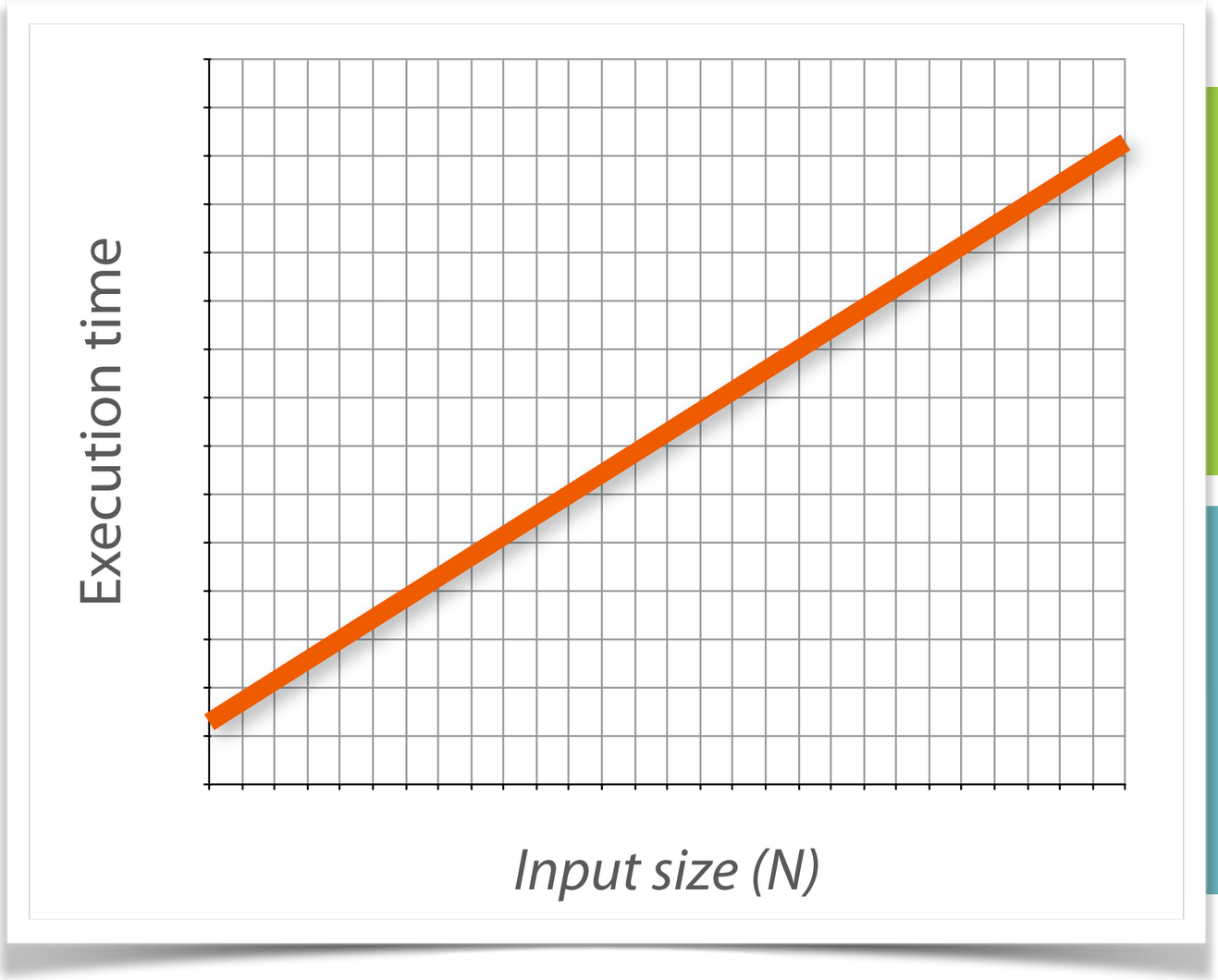
Ways of Measuring



Counting instructions

Best case
Worst case
Average

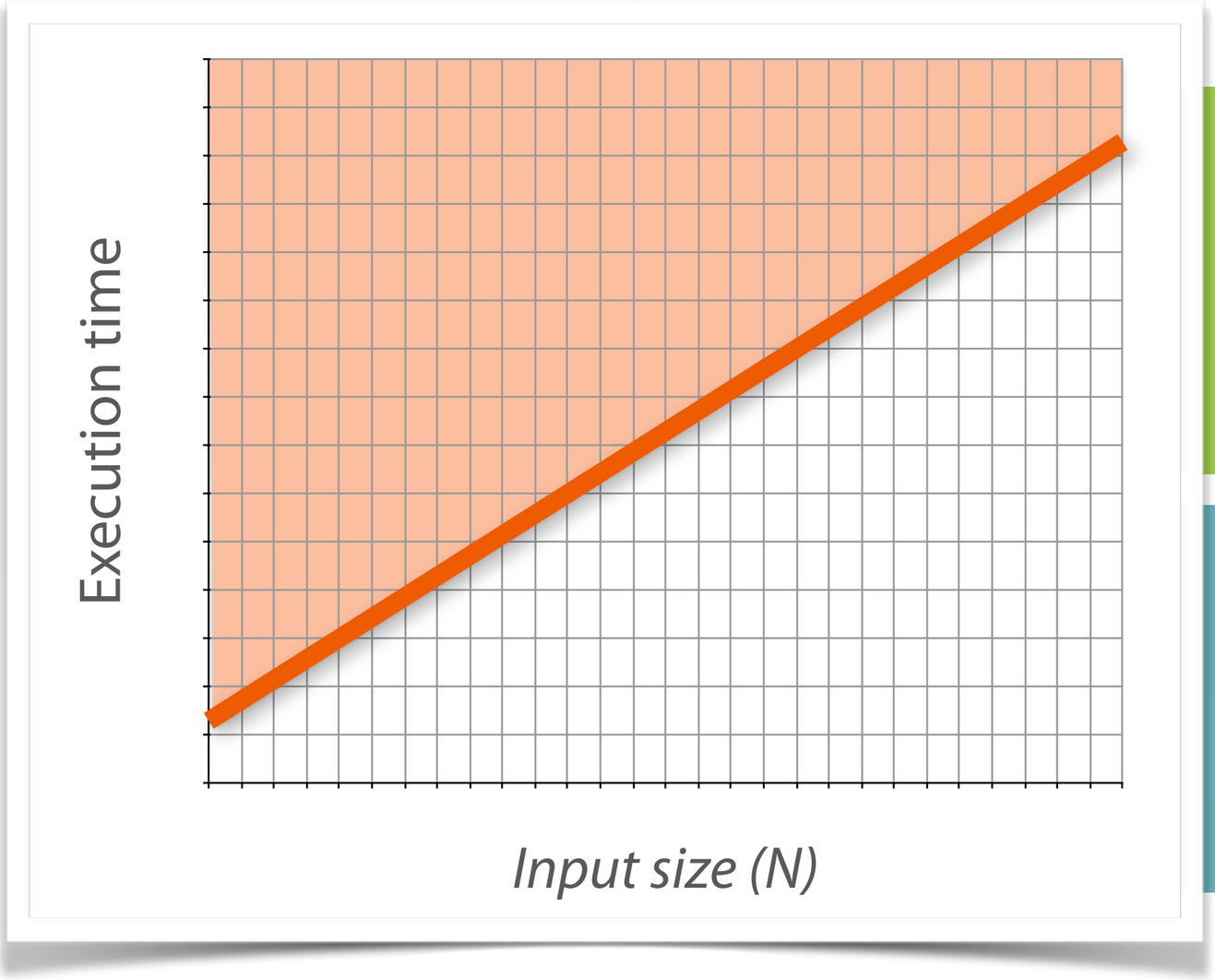
Ways of Measuring



Counting instructions

Best case
Worst case
Average

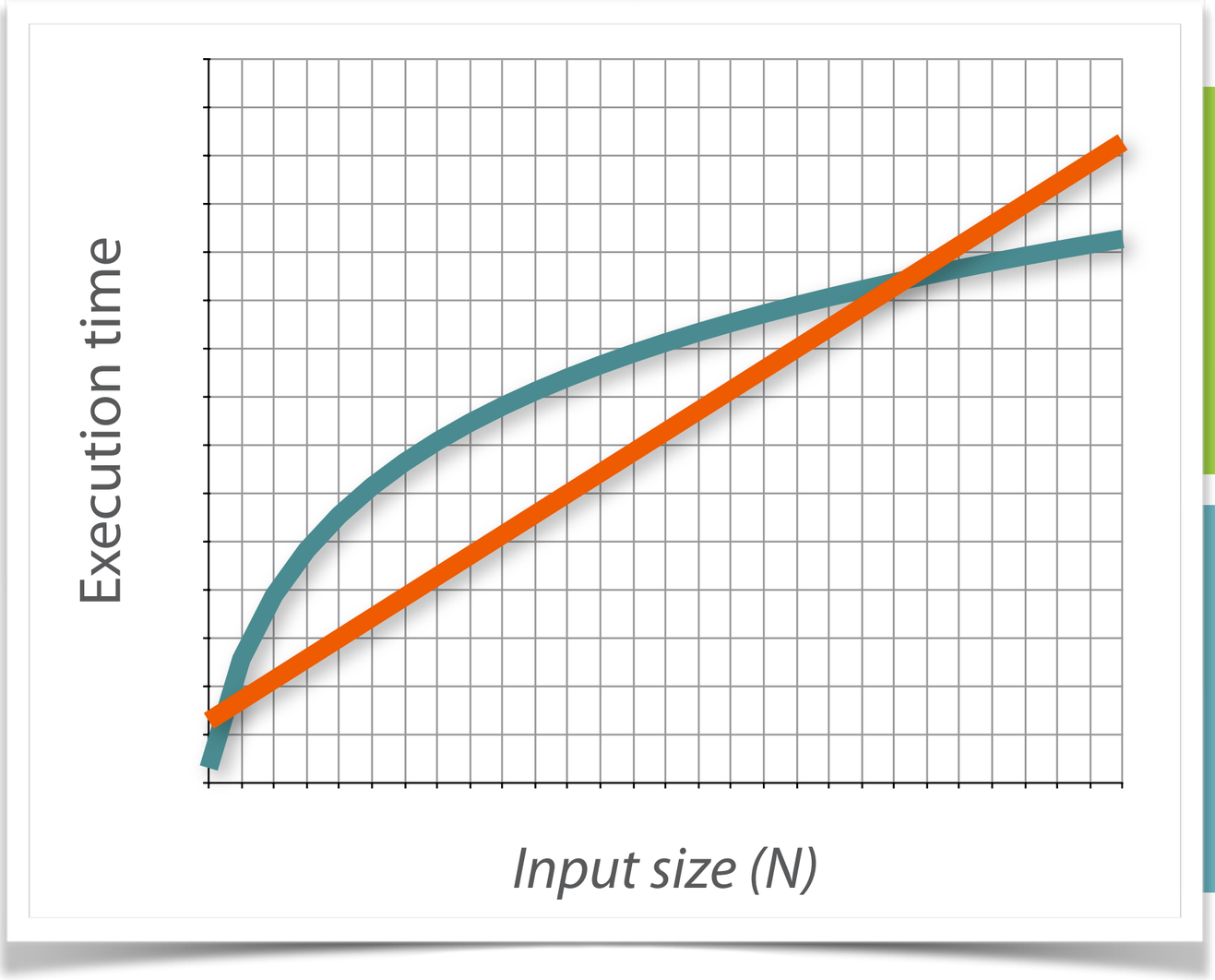
Ways of Measuring



Counting instructions

Best case
Worst case
Average

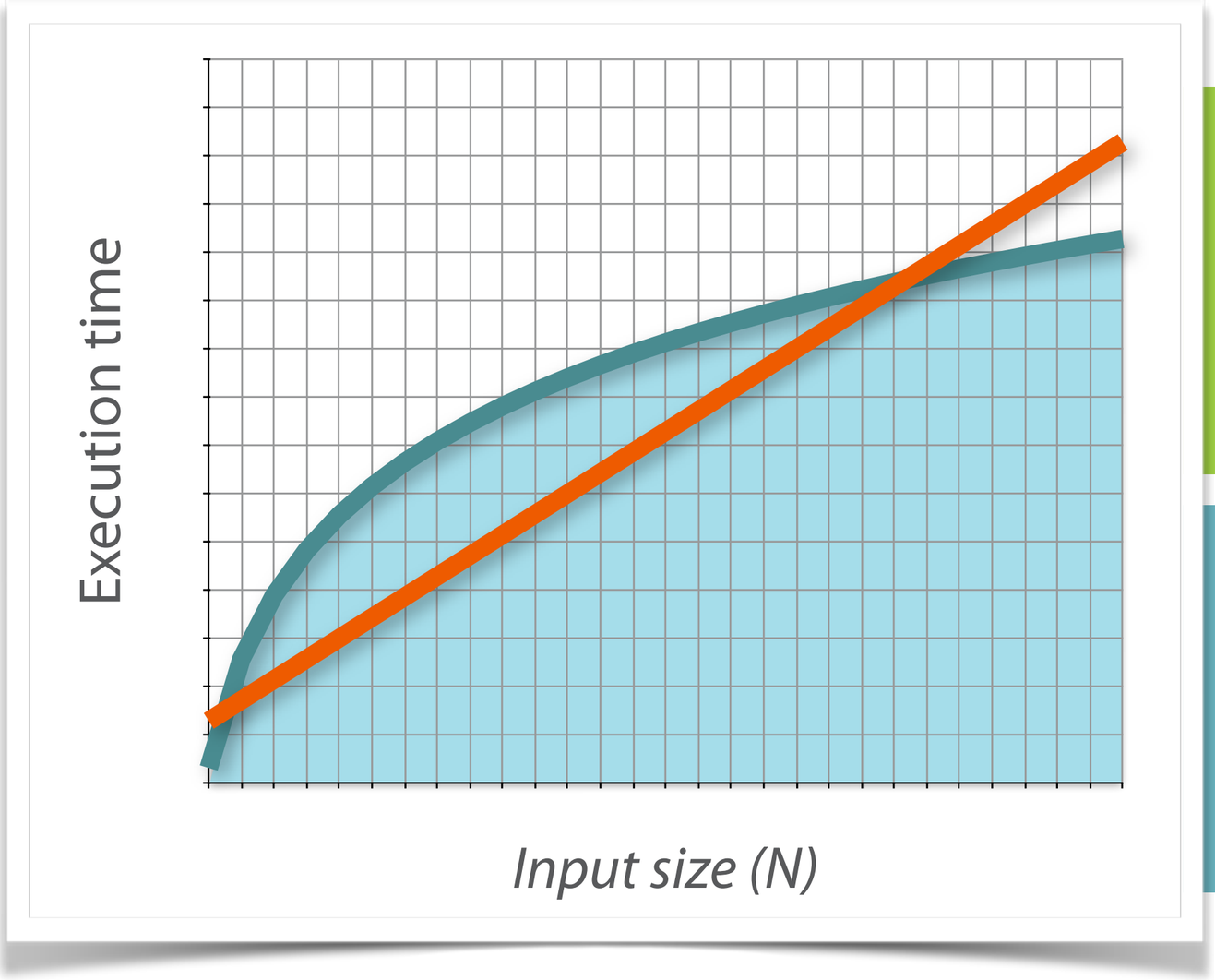
Ways of Measuring



Counting instructions

Best case
Worst case
Average

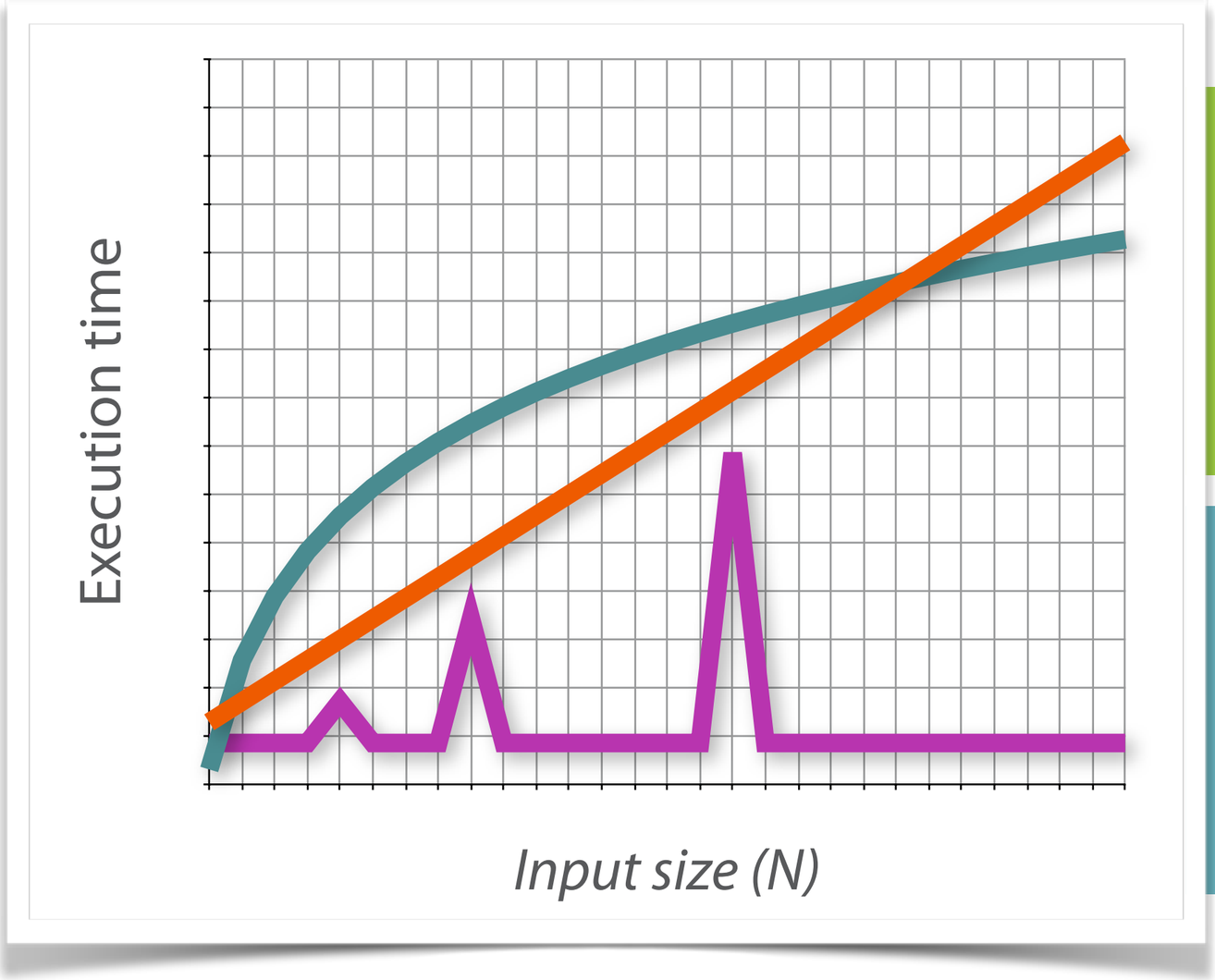
Ways of Measuring



Counting instructions

Best case
Worst case
Average

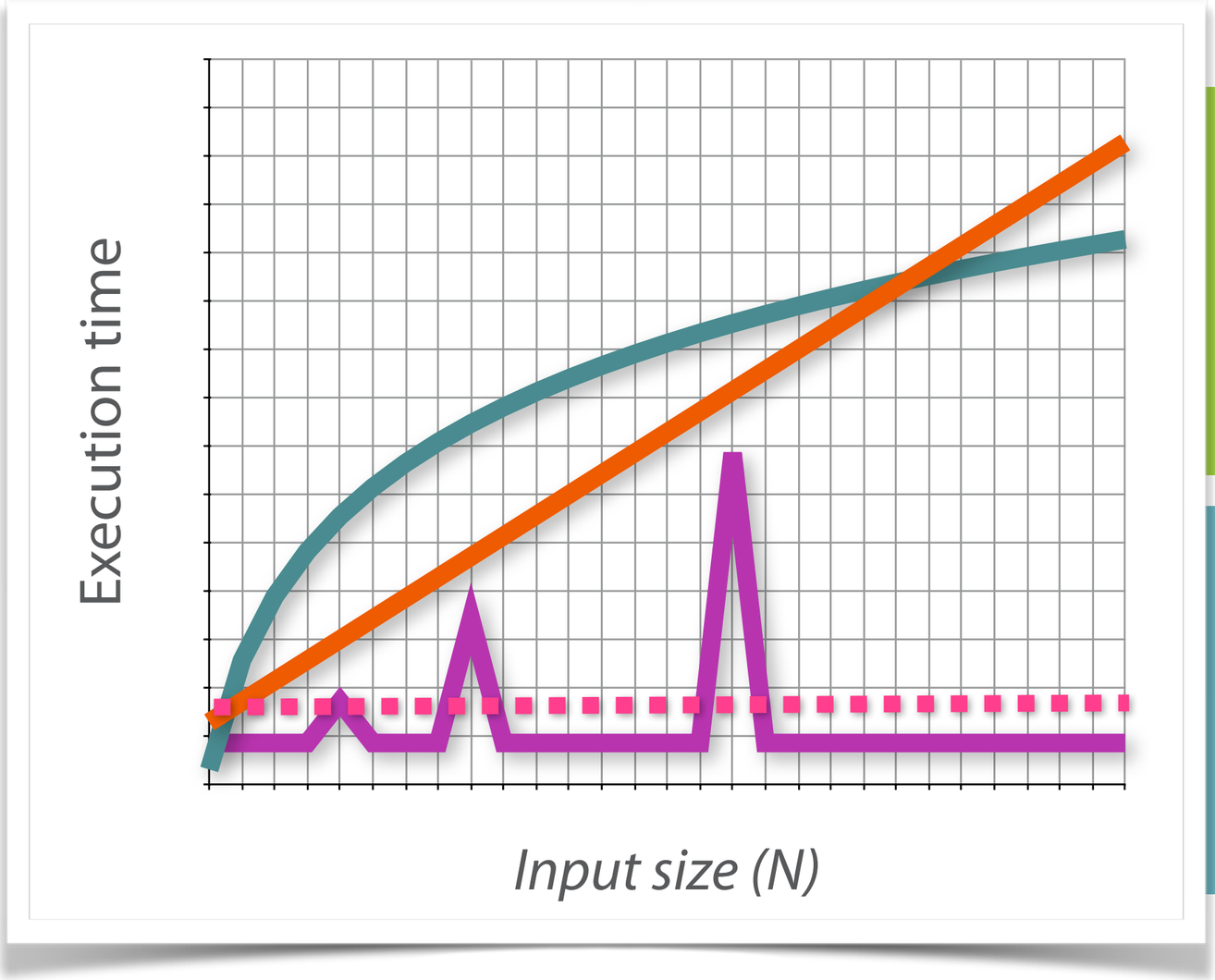
Ways of Measuring



Counting instructions

Best case
Worst case
Average

Ways of Measuring



Counting instructions

Best case
Worst case
Average

Ways of Measuring

Timing with stopwatch

Counting instructions

Looking at the curve

Best case
Worst case
Average

Asymptotic Performance

Asymptotic Performance

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

Asymptotic Performance

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

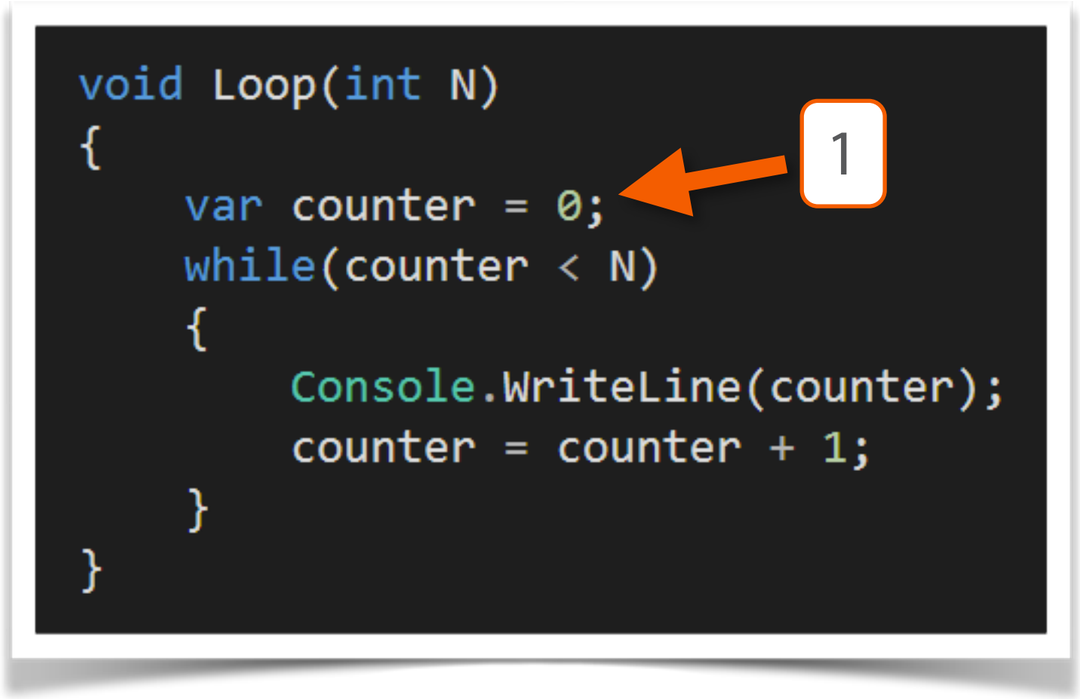
How complex?

Asymptotic Performance

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

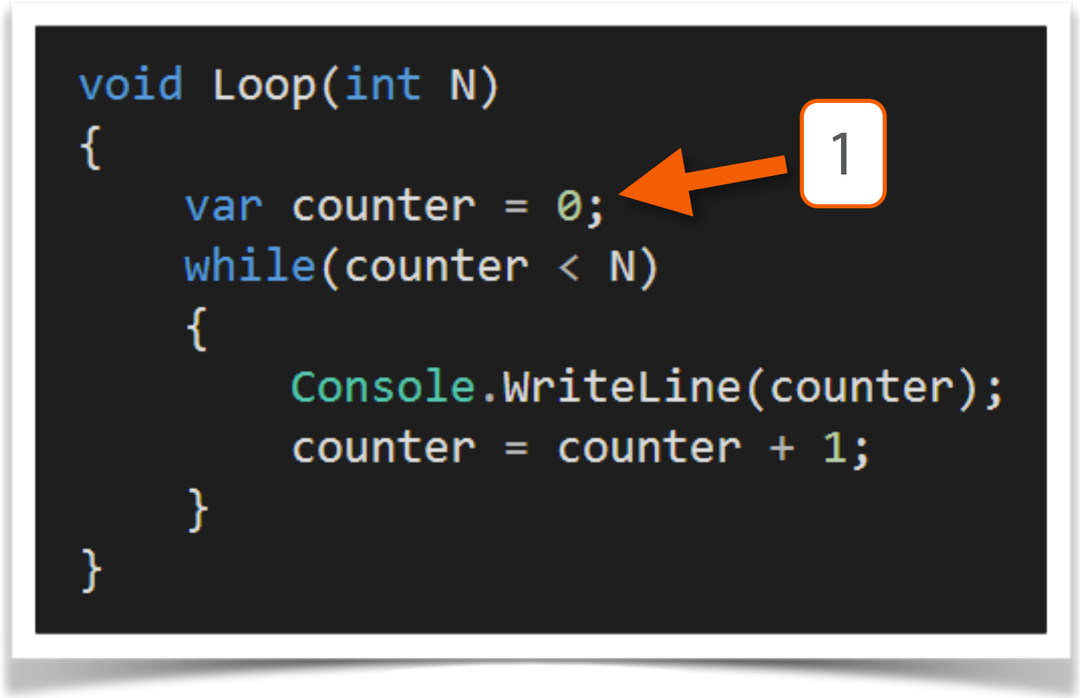
Complexity (*N*):

Asymptotic Performance



Complexity (*N*):

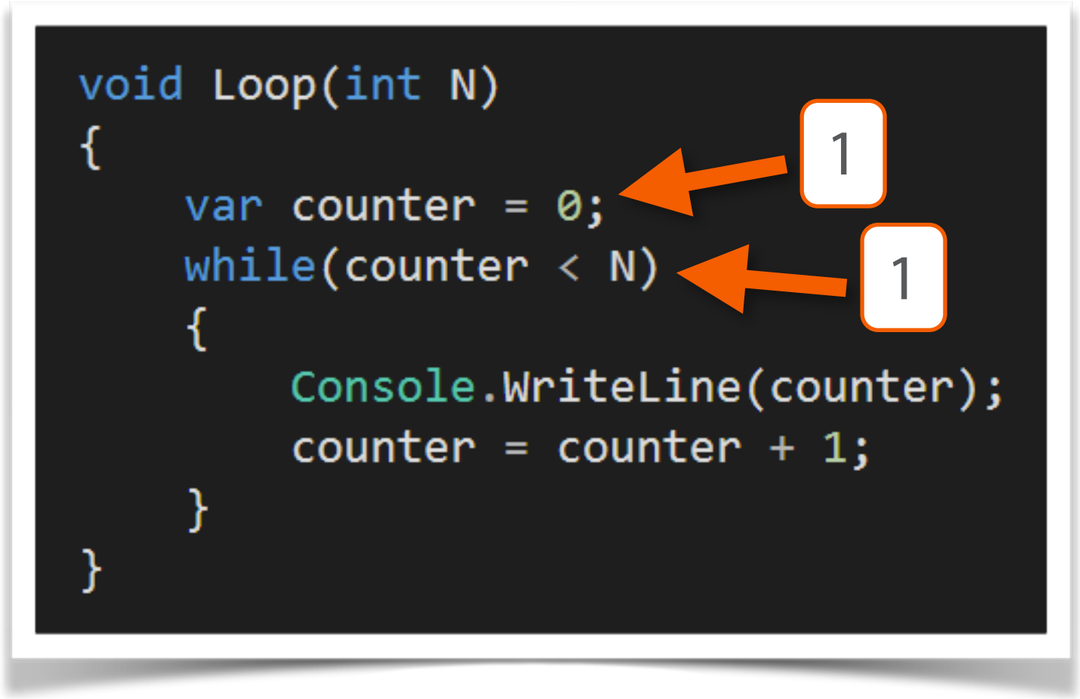
Asymptotic Performance



Complexity (*N*):

1

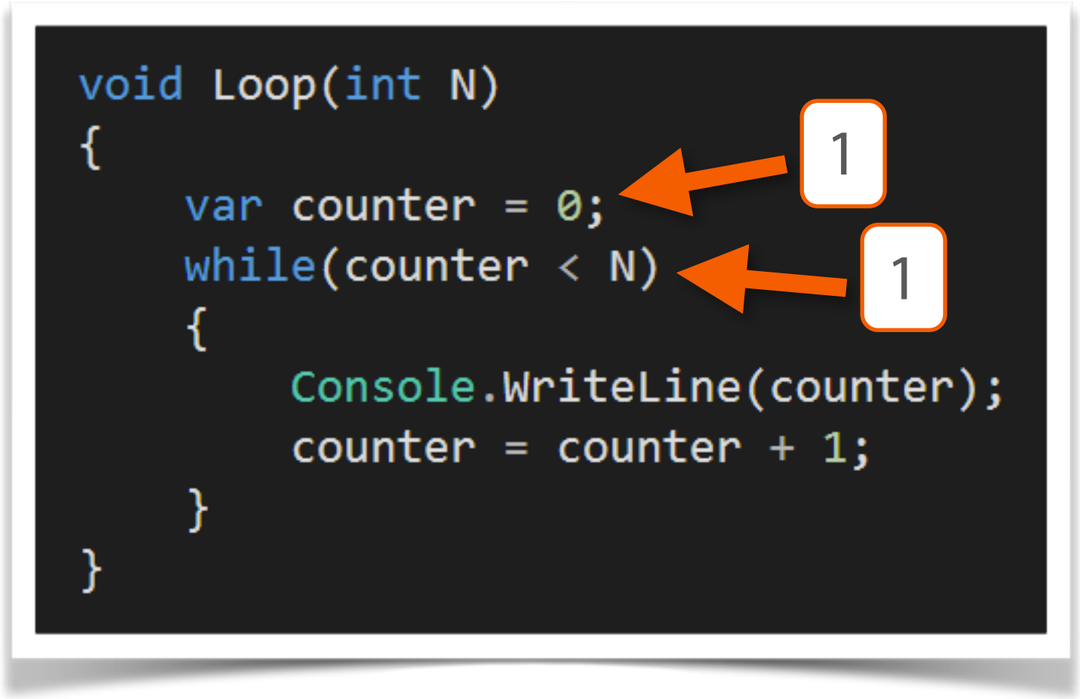
Asymptotic Performance



Complexity (*N*):

1

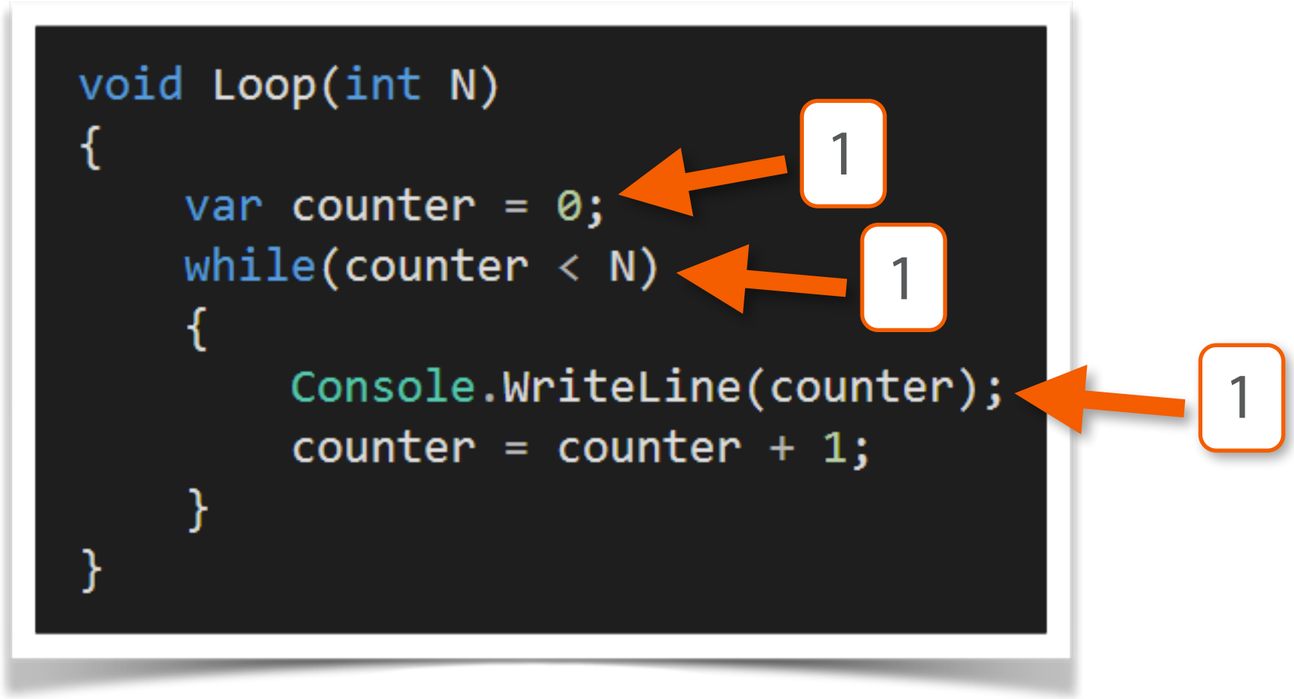
Asymptotic Performance



Complexity (*N*):

1 + 1

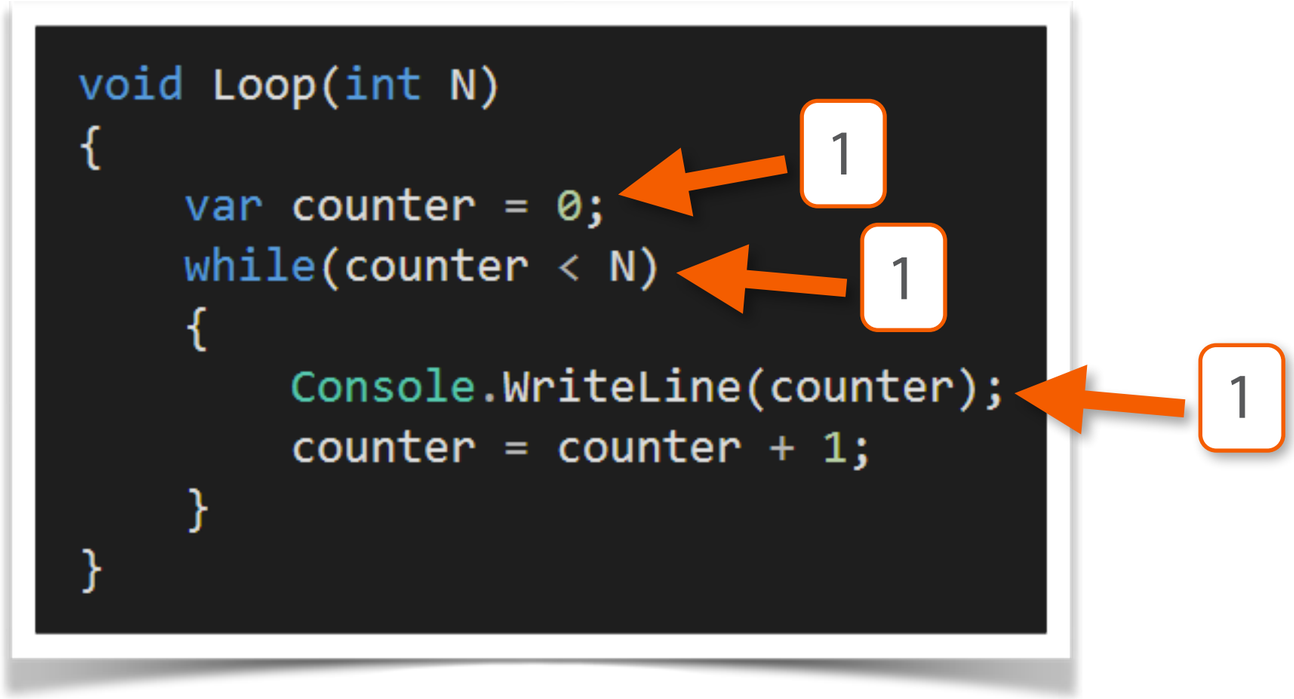
Asymptotic Performance



Complexity (*N*):

1 + 1

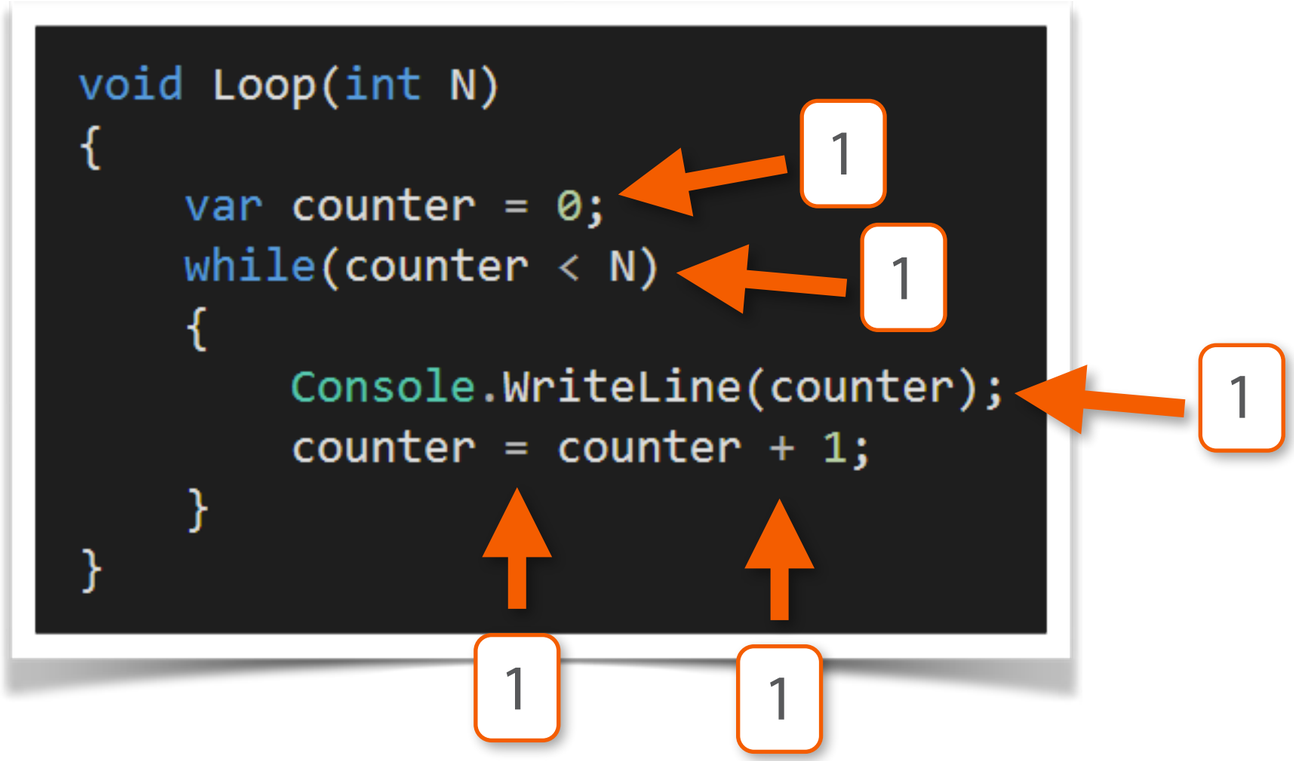
Asymptotic Performance



Complexity (*N*):

1 + 1 + 1

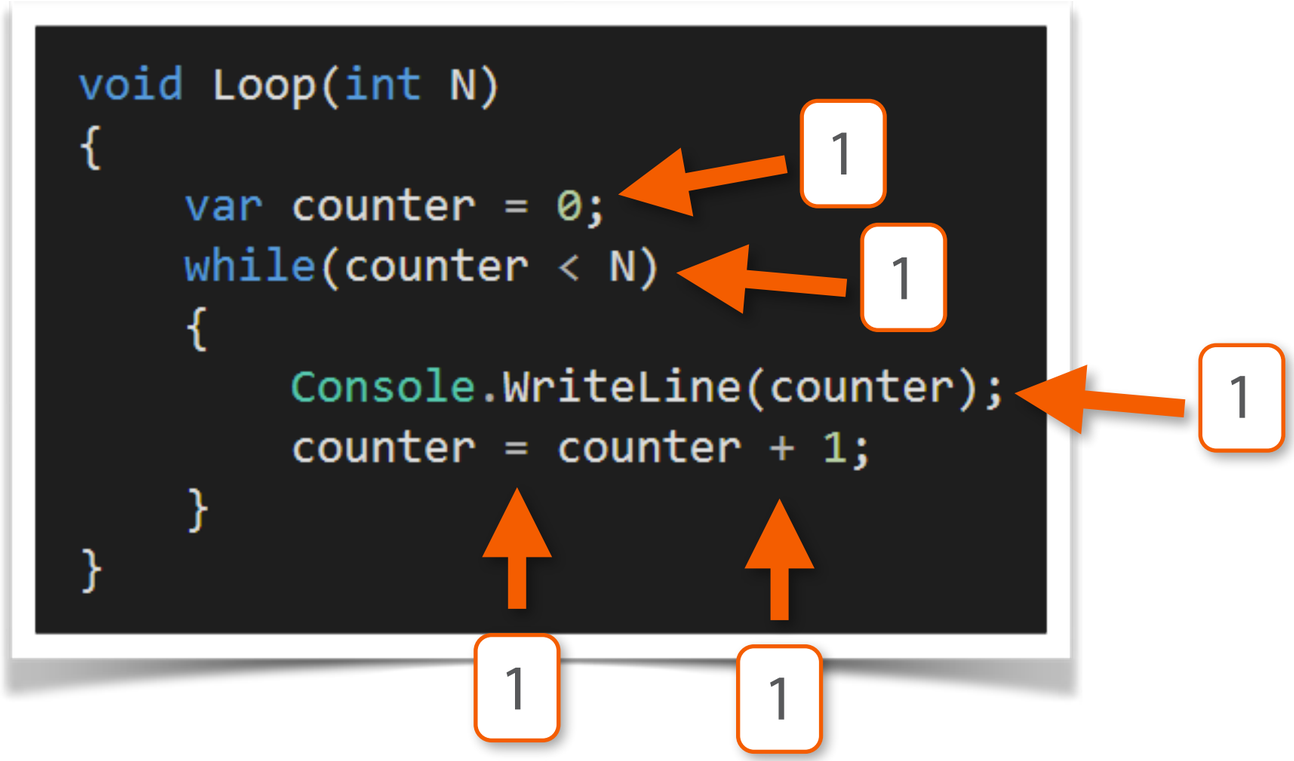
Asymptotic Performance



Complexity (*N*):

1 + 1 + 1

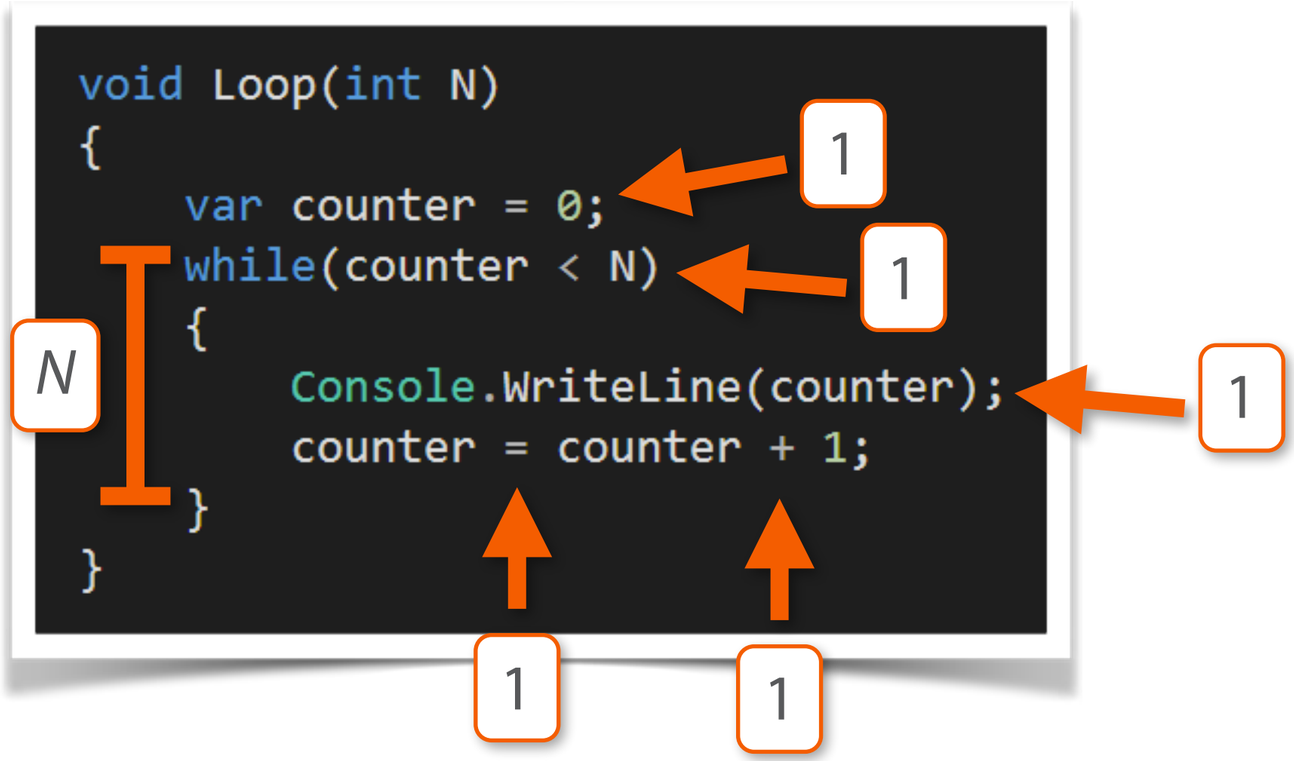
Asymptotic Performance



Complexity (*N*):

1 + 1 + 1 + 2

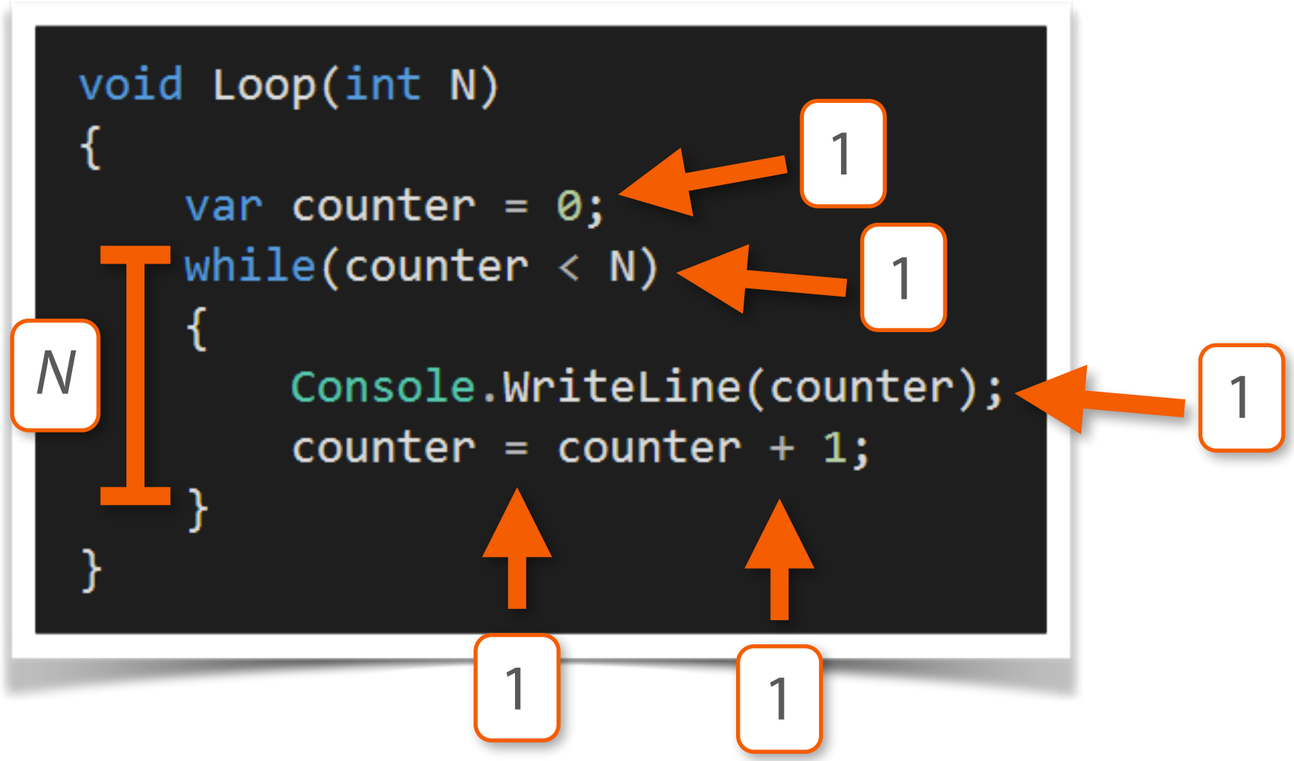
Asymptotic Performance



Complexity (*N*):

1 + 1 + 1 + 2

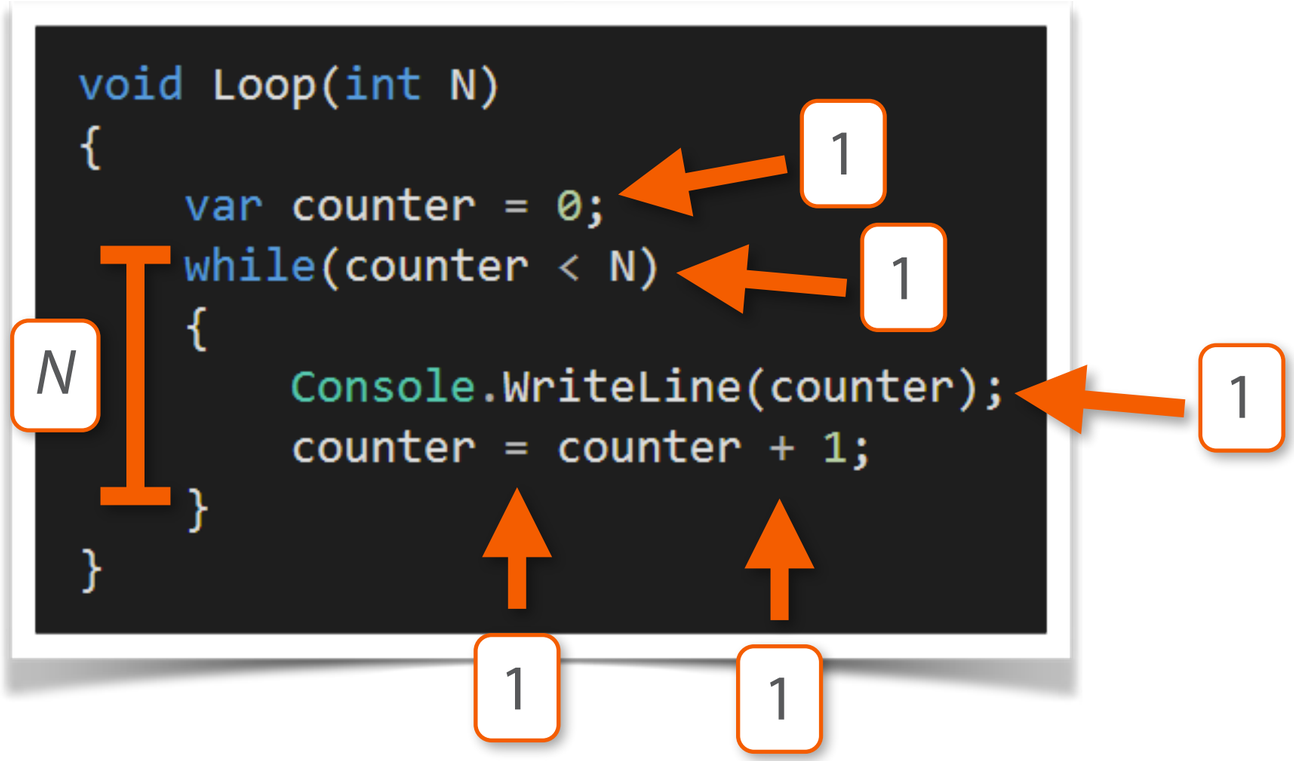
Asymptotic Performance



Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

Asymptotic Performance

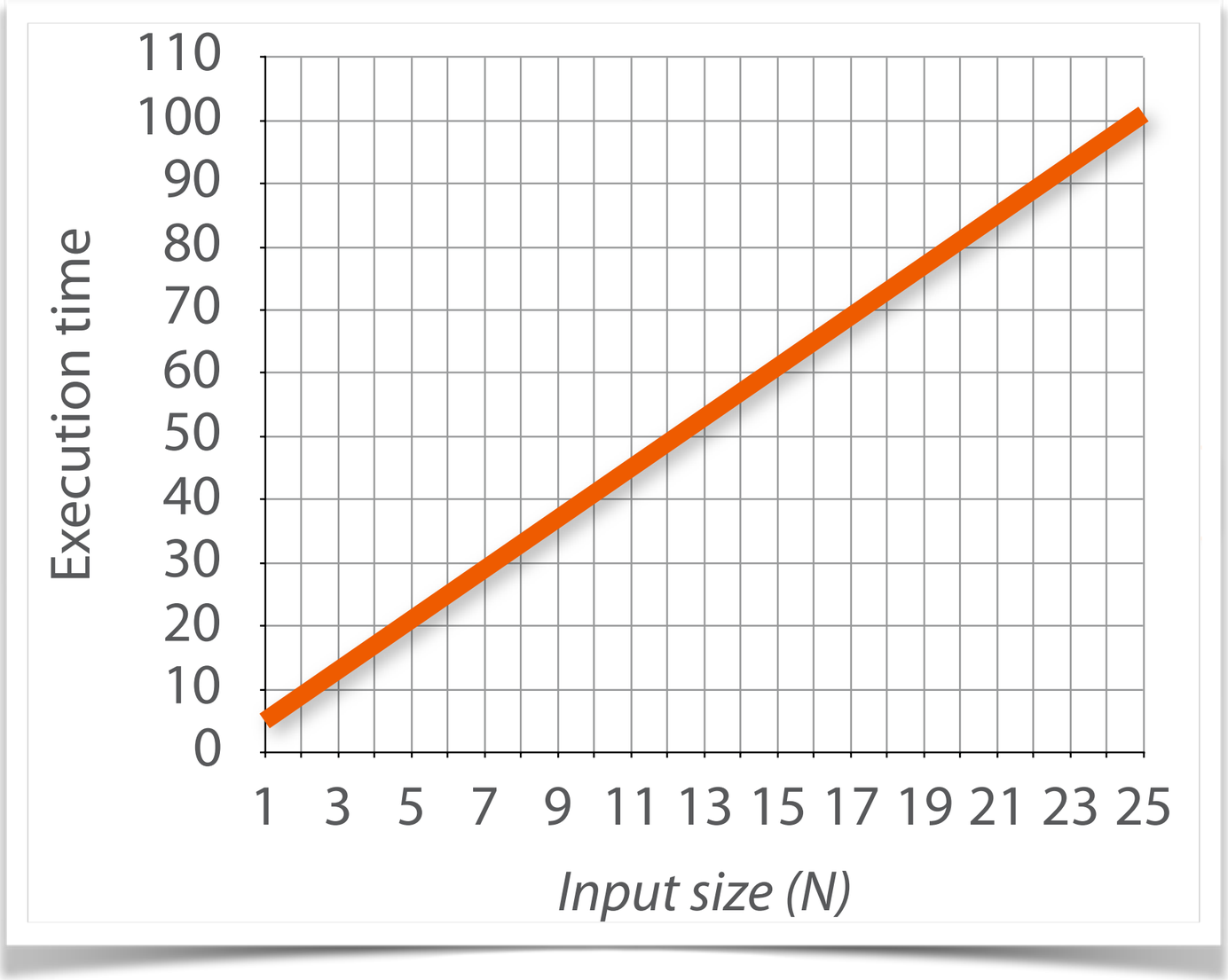


Complexity (*N*):

$$1 + (1 + 1 + 2)\,N$$

$$= 1 + 4\,N$$

Asymptotic Performance

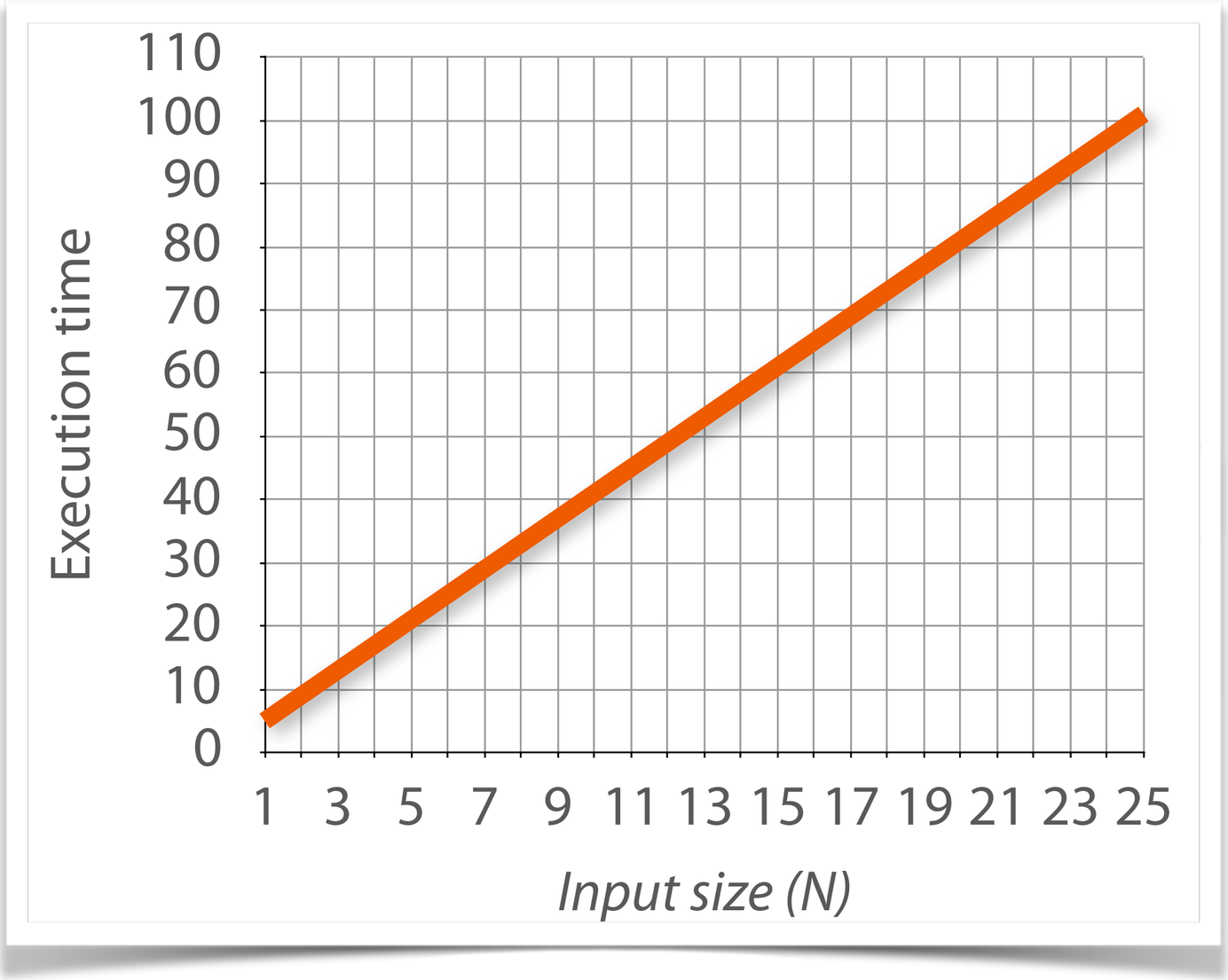


Complexity (*N*):

$$1 + (1 + 1 + 2)\,N$$

$$= 1 + 4\,N$$

Asymptotic Performance

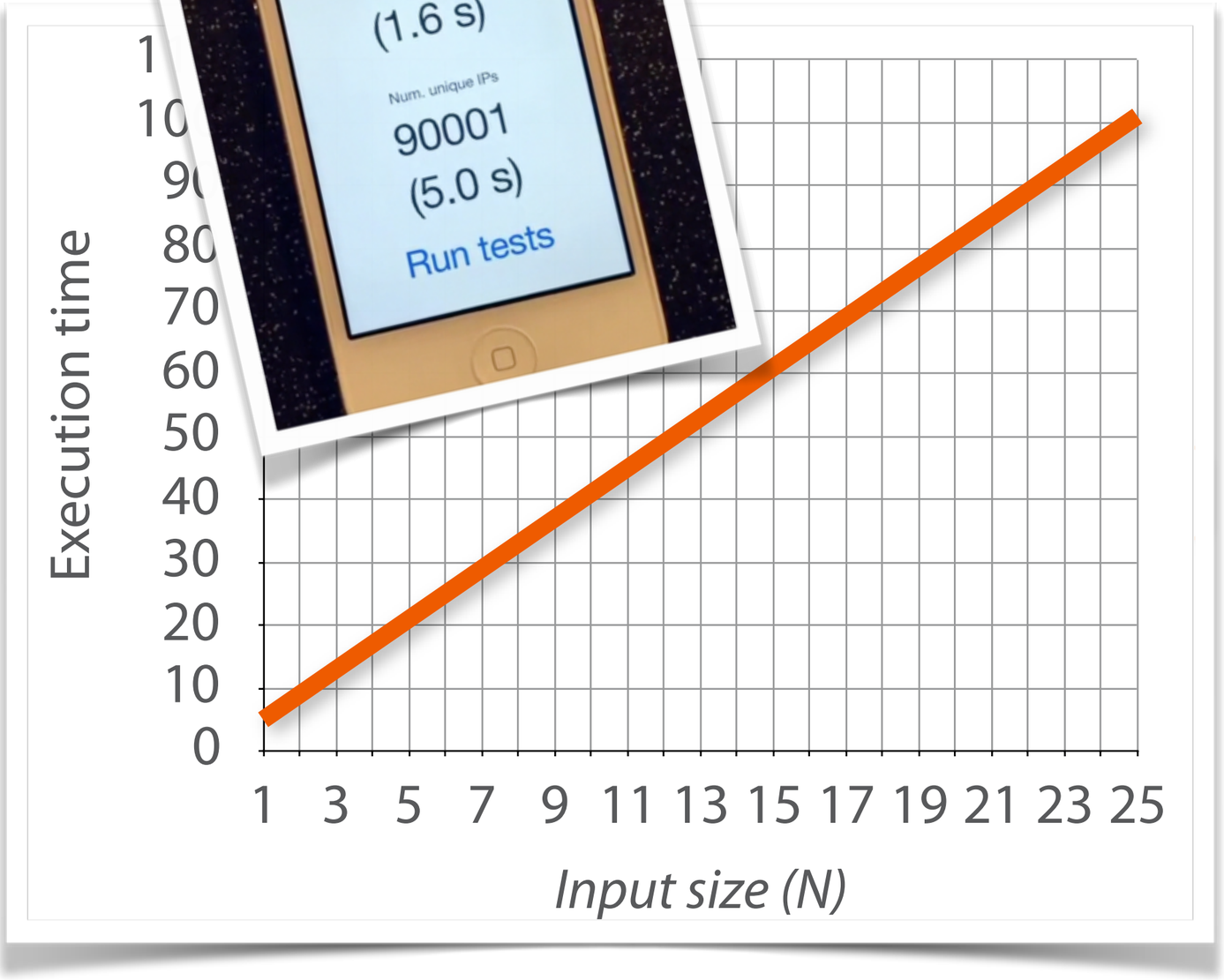


Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$

nptotic Performance

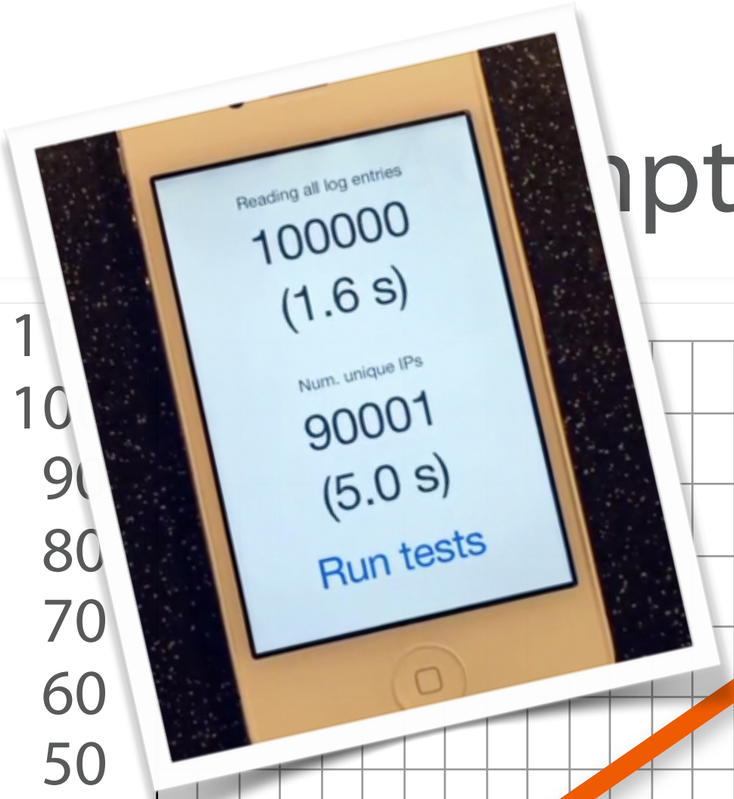


Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$

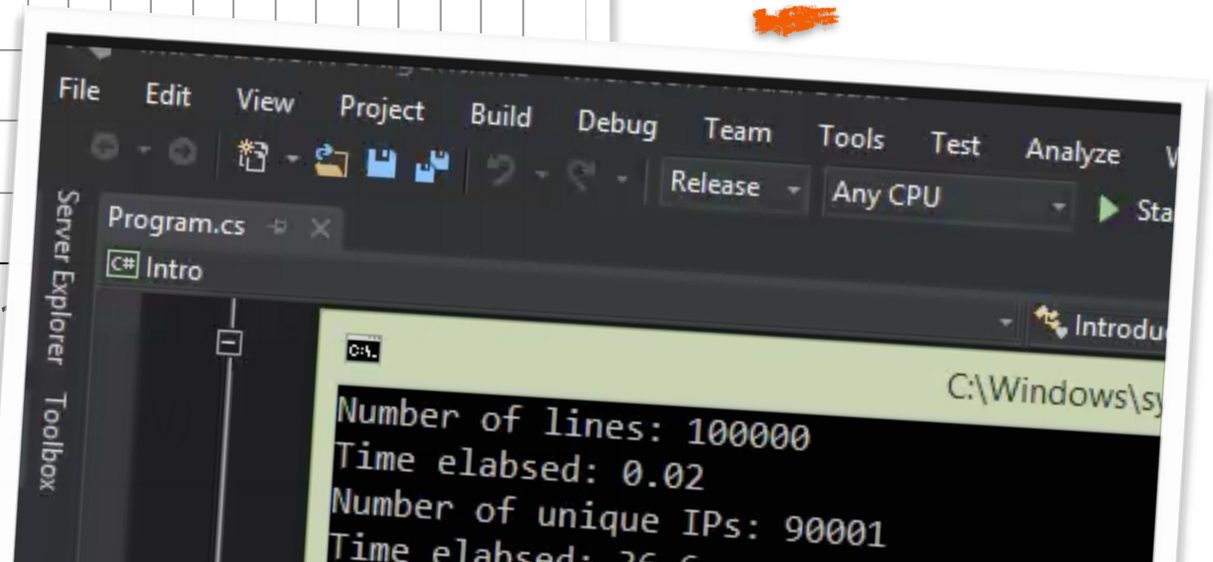
nptotic Performance



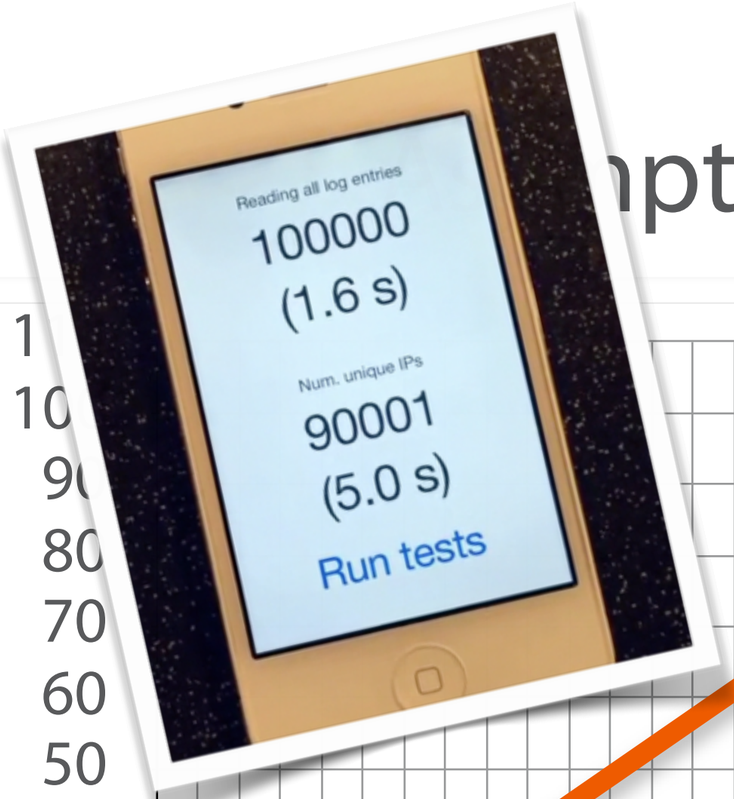
Complexity (*N*):

$$1 + (1 + 1 + 2)\,N$$

$$= 1 + 4\,N$$



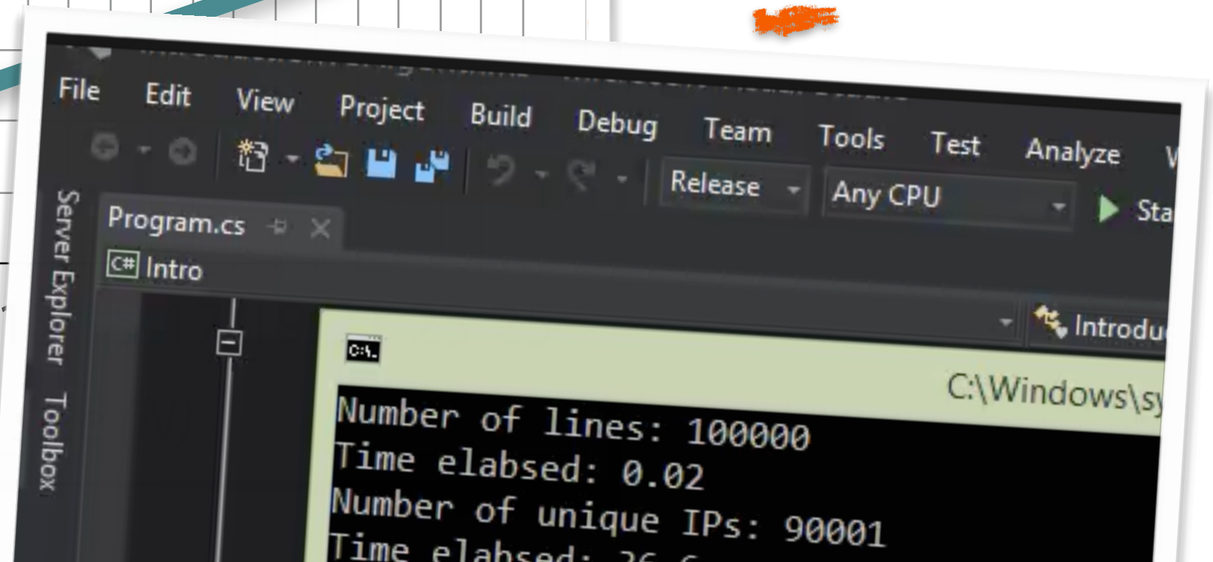
nptotic Performance



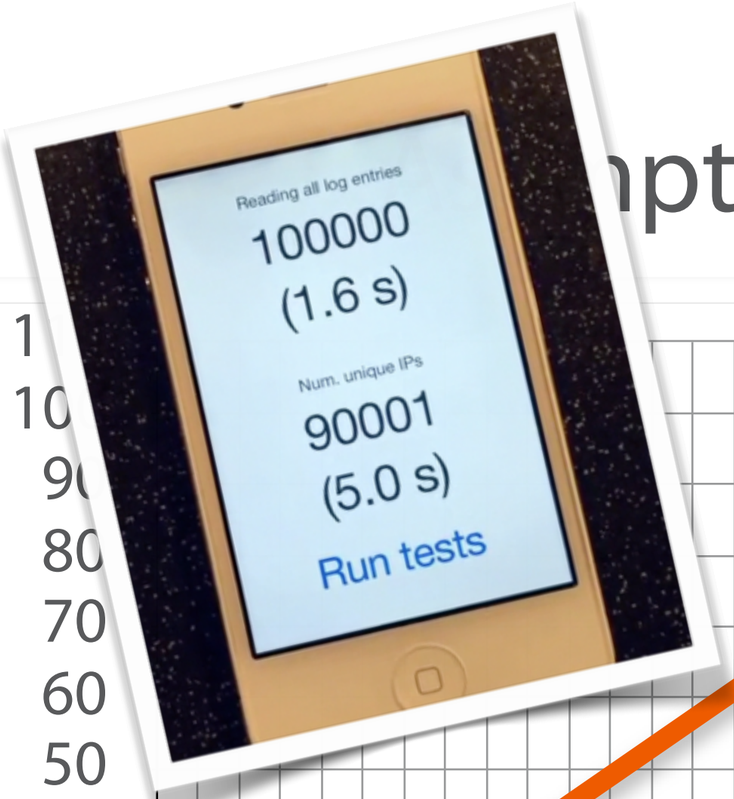
Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$



nptotic Performance

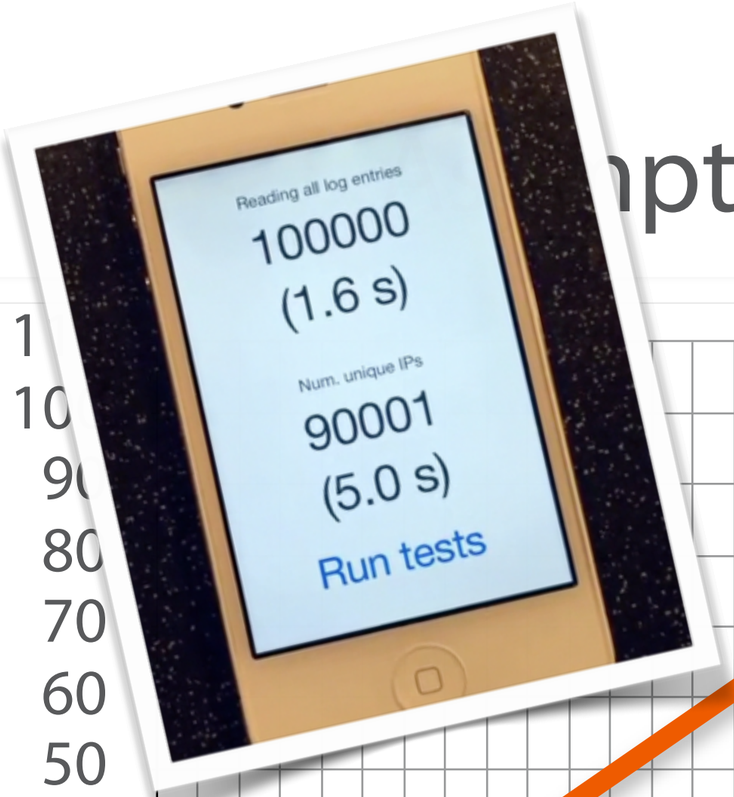


Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$

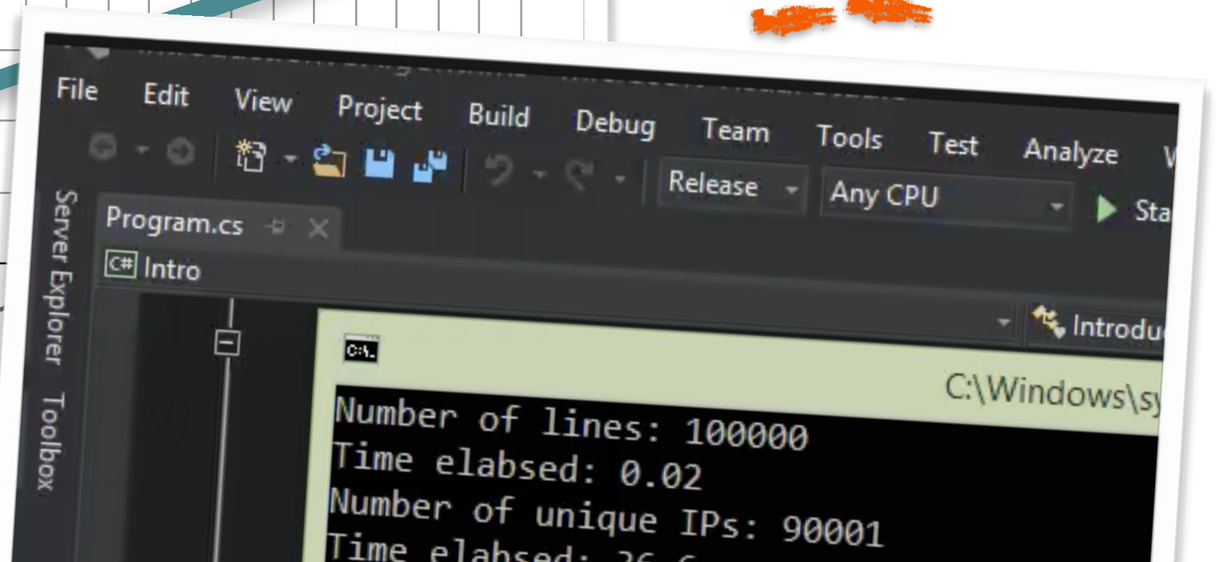
nptotic Performance



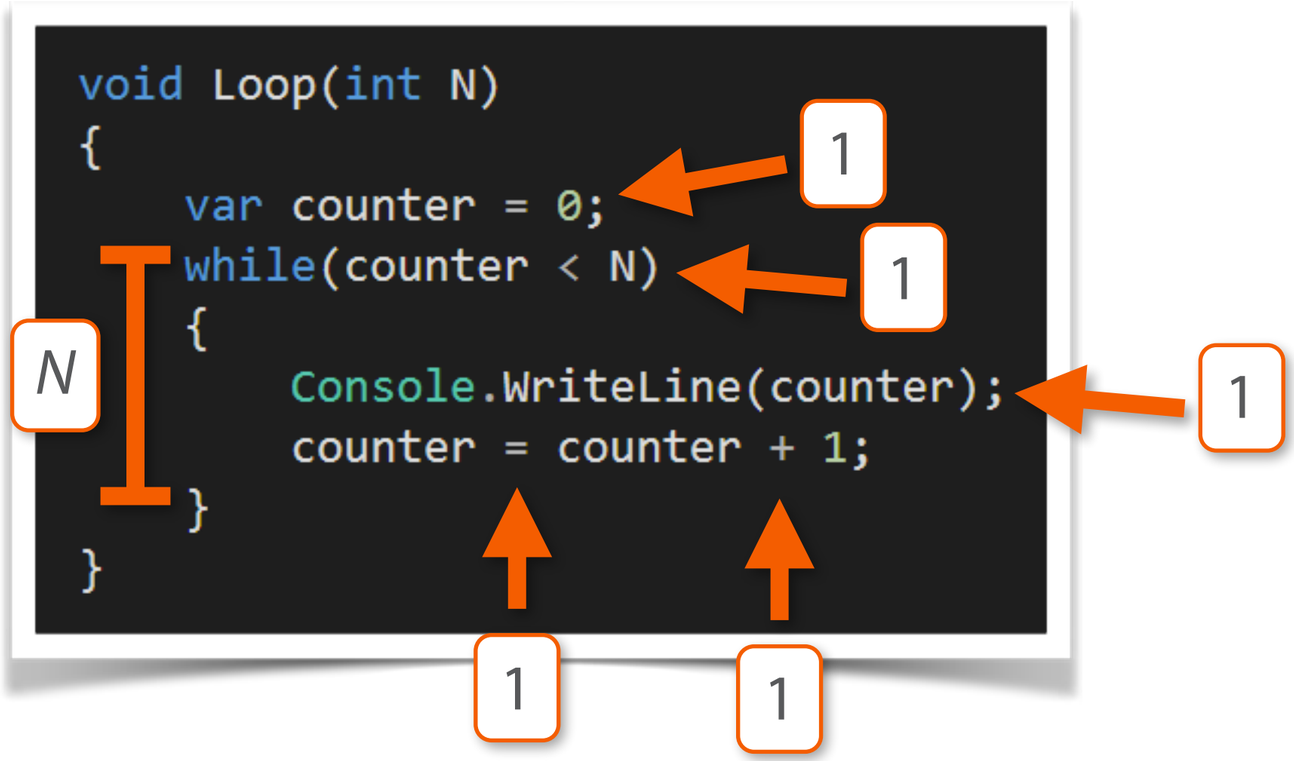
Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$



Asymptotic Performance

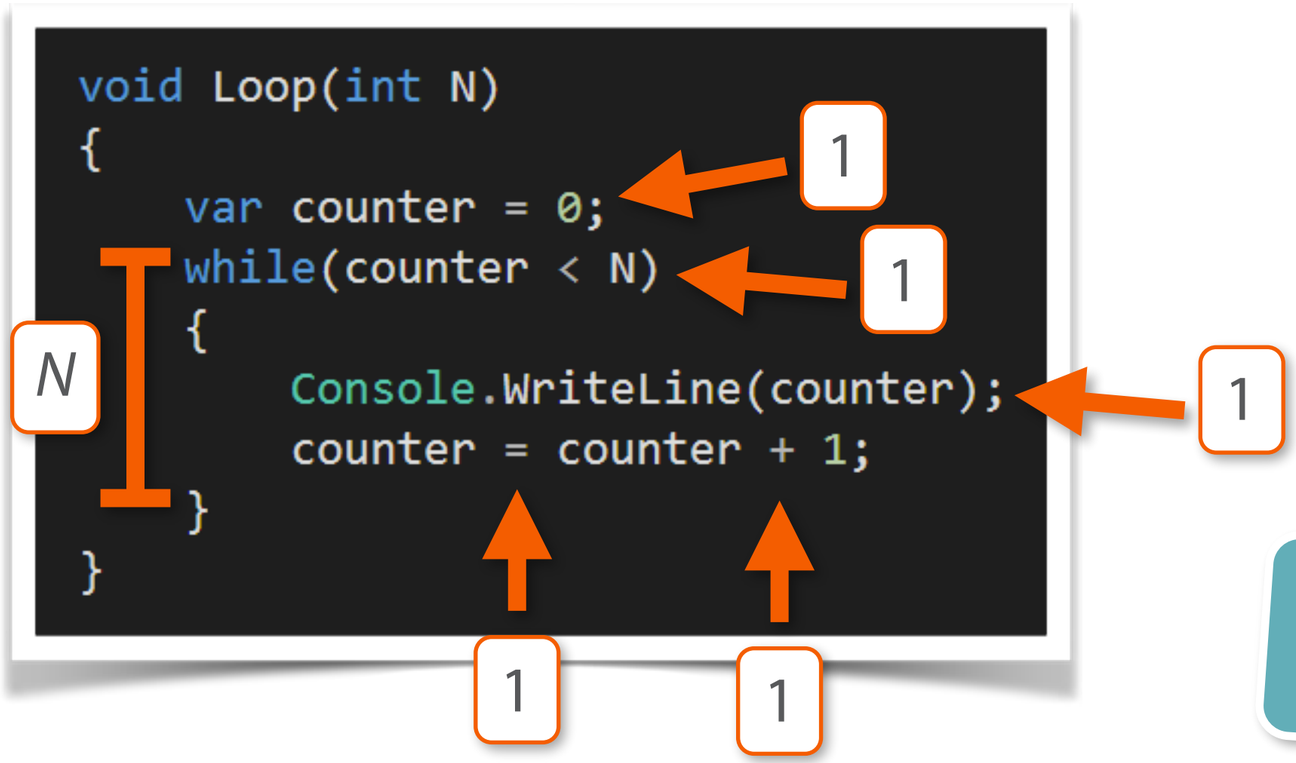


Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$

Asymptotic Performance



Complexity (*N*):

$1 + (1 + 1 + 2)\,N$

$= 1 + 4\,N$

$N \cdot$ (some number)

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

N

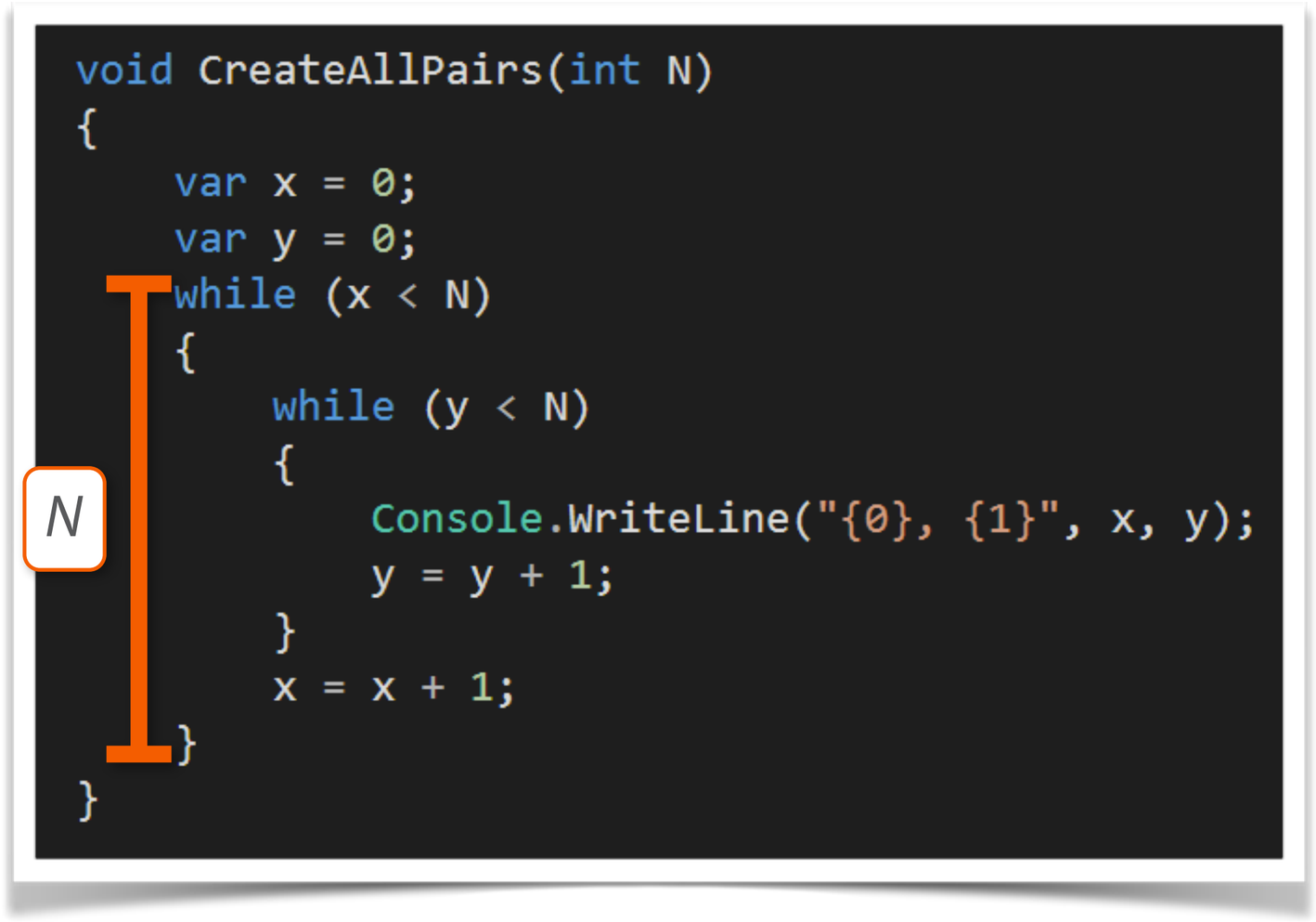
Complexity (*N*):

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

N

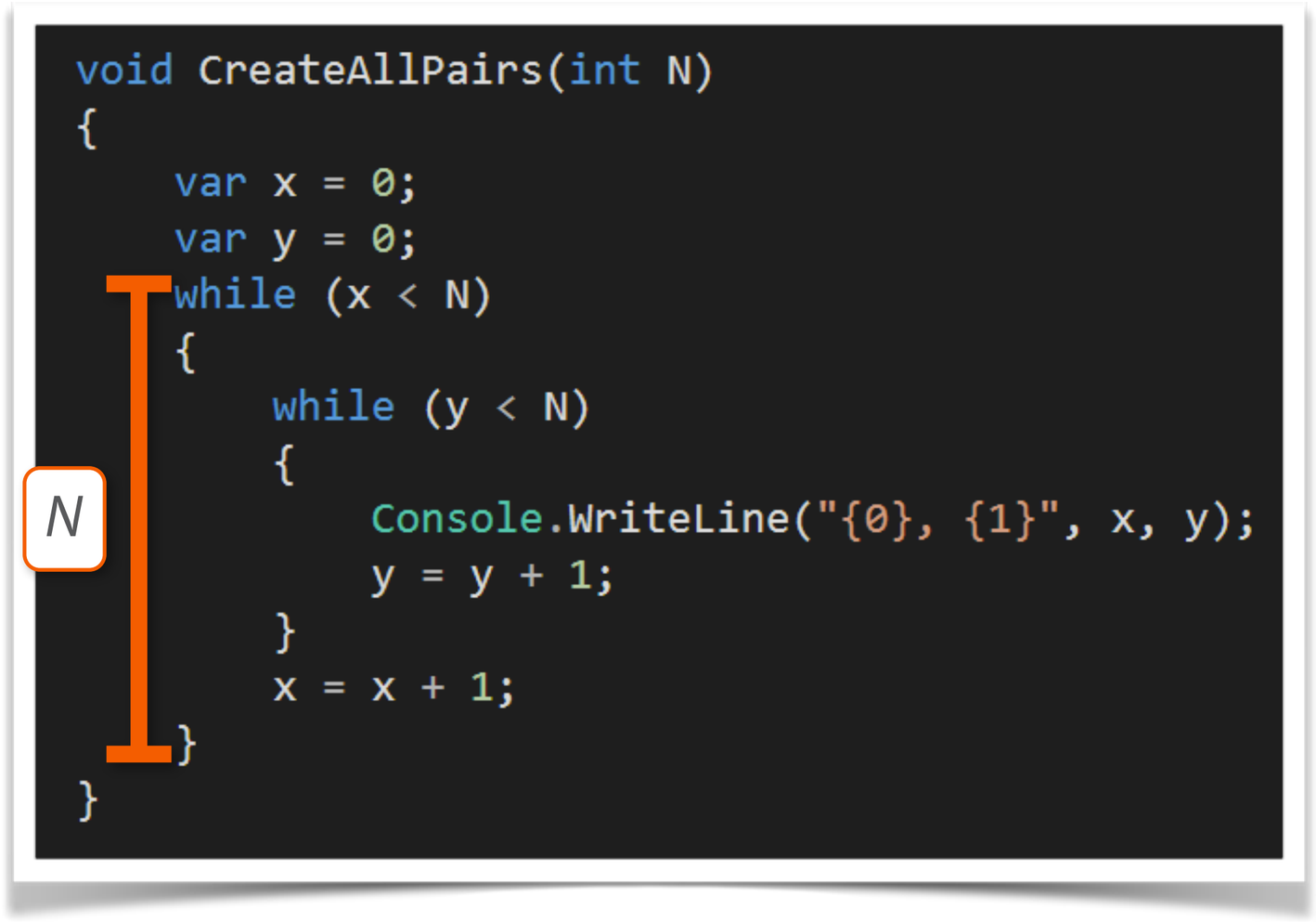
Complexity (*N*):

$$N\,(\,1+\,1\,+\,2\,)$$



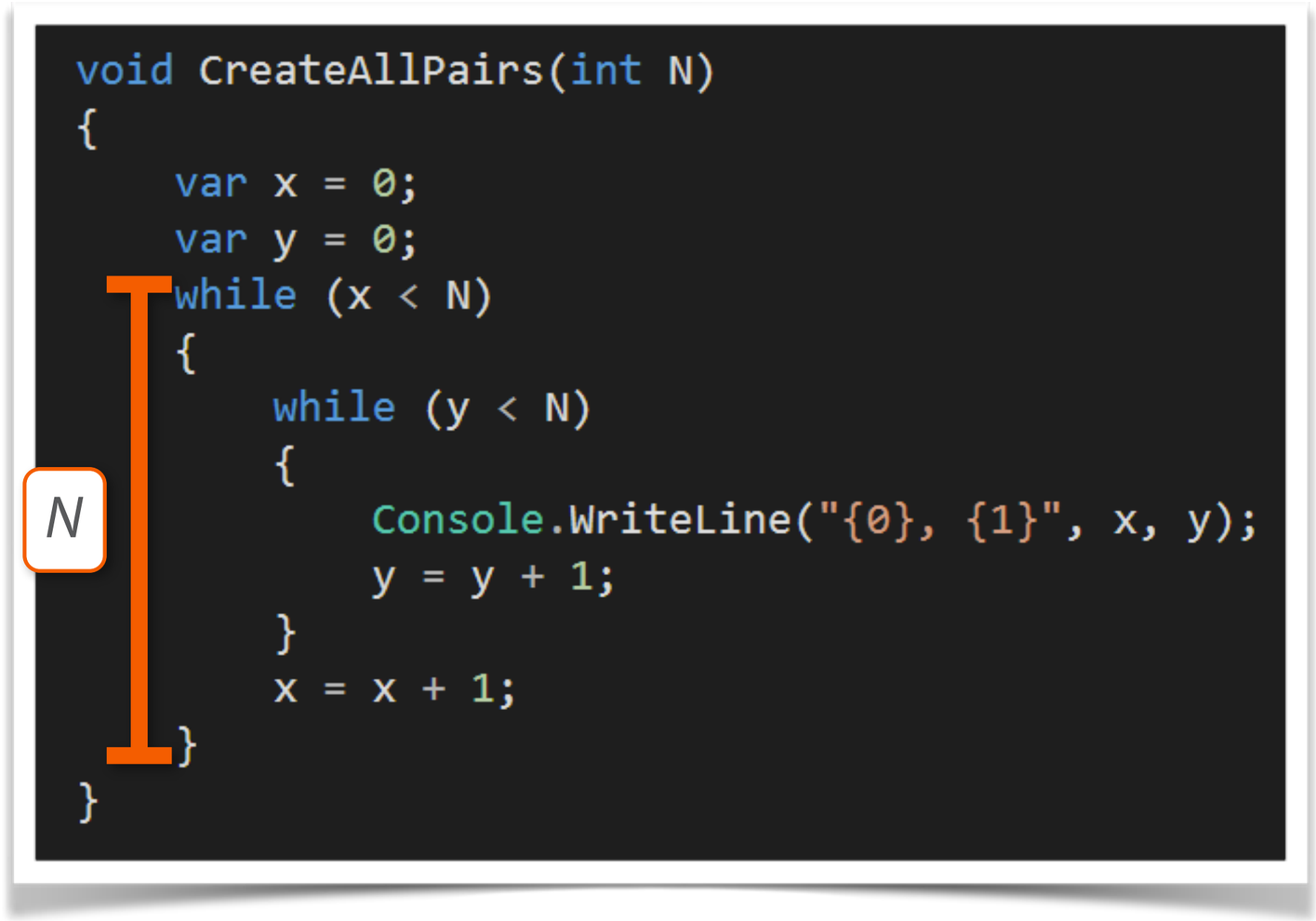
Complexity (*N*):

$$N\,(\,1+1+2\,)$$



Complexity (*N*):

$$N\,(\,1 + N\,(\,1+1+2\,)\,+2)$$



Complexity (*N*):

$$N\,(\,1 + N\,(\,1 + 1 + 2\,) + 2)$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

$$= 2 + N(3 + 4N)$$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

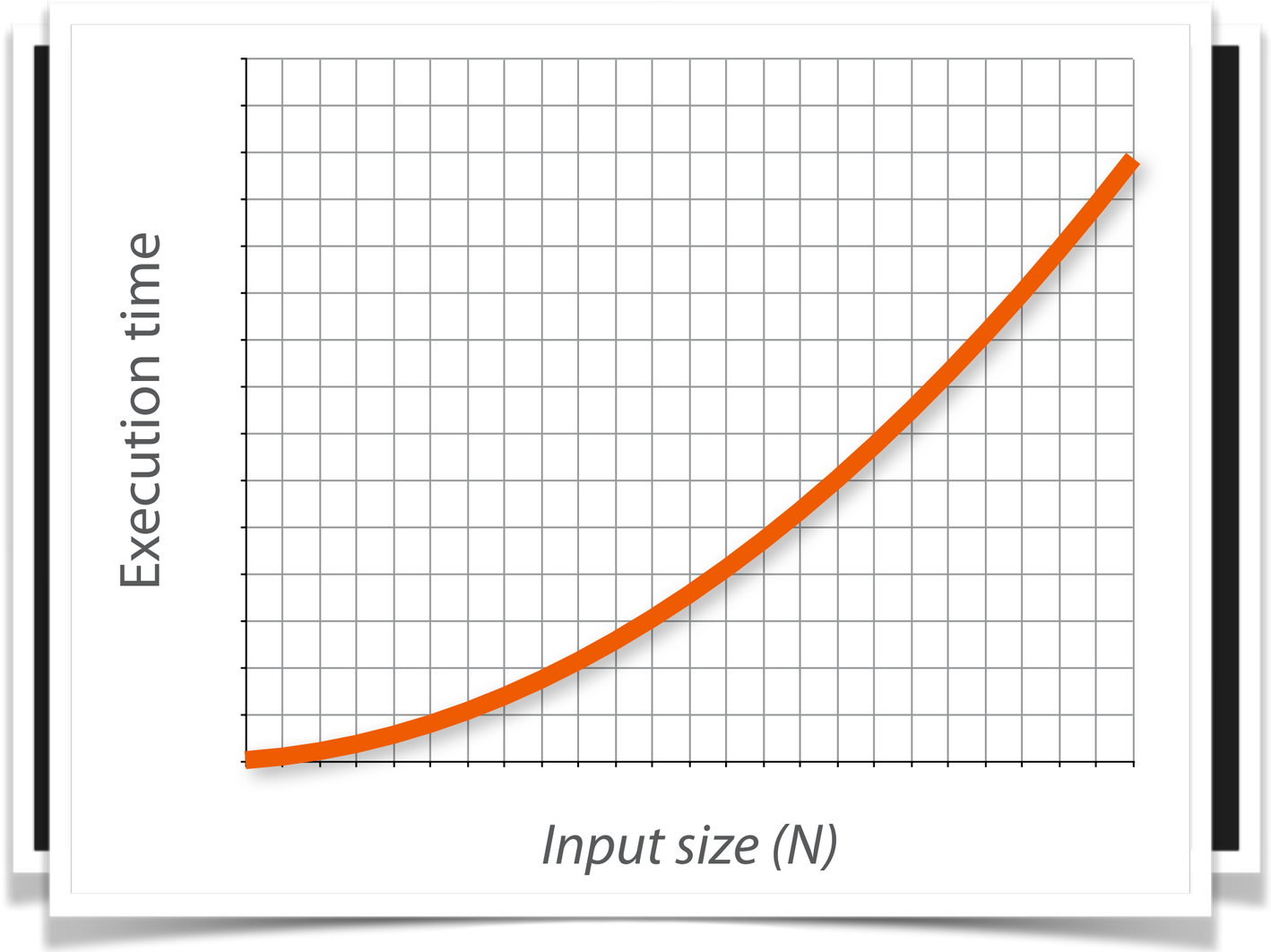
Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

$$= 2 + N(3 + 4N)$$

$$= 2 + 3N + 4N^2$$



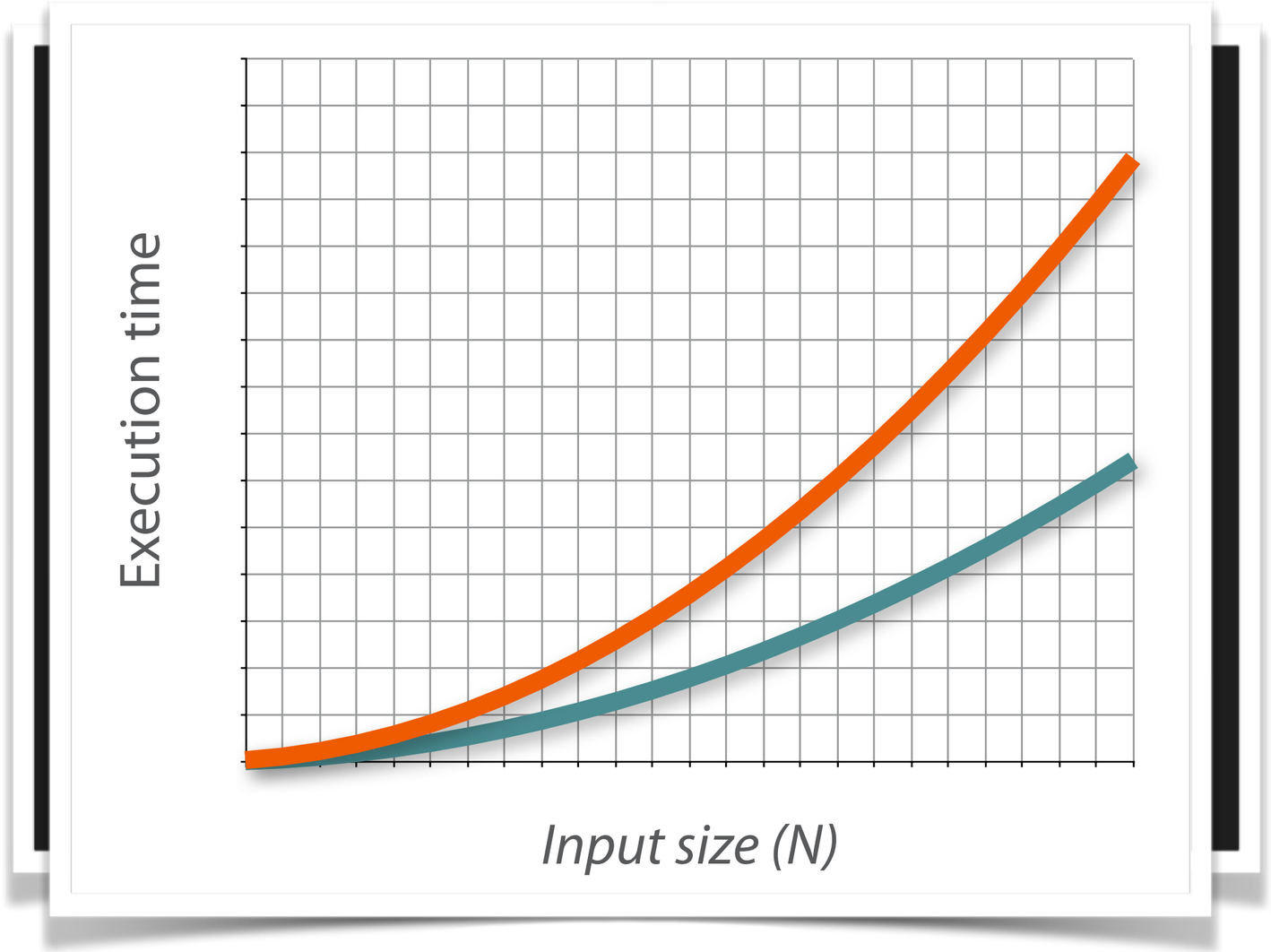
Complexity (*N*):

$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

$$= 2 + N(3 + 4N)$$

$$= 2 + 3N + 4N^2$$



Complexity (*N*):

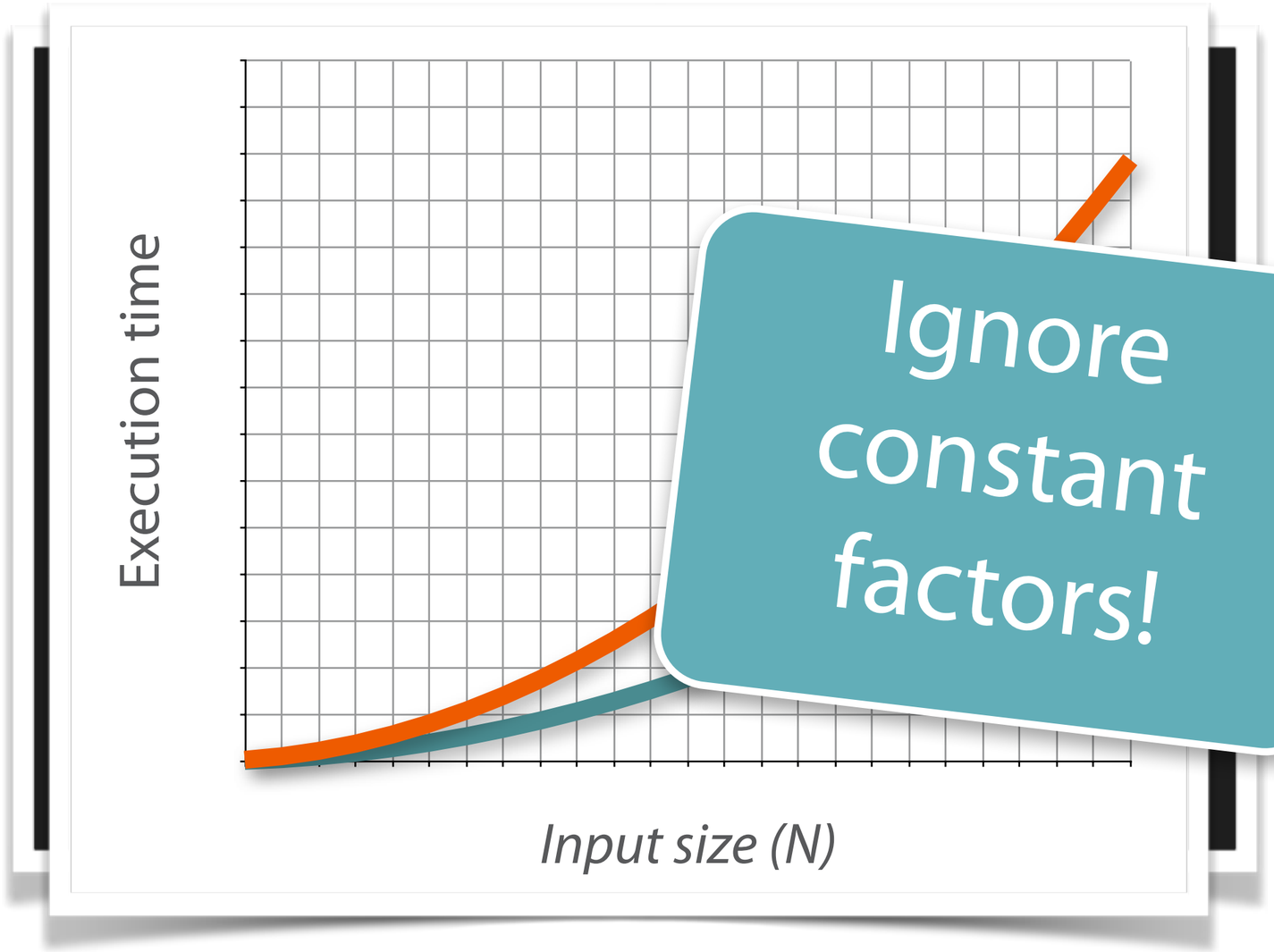
$$1 + 1 + N(1 + N(1 + 1 + 2) + 2)$$

$$= 2 + N(3 + N(4))$$

$$= 2 + N(3 + 4N)$$

$$= 2 + 3N + 4N^2$$

$$(2 + 3N + 4N^2) \cdot 0.5$$



Complexity (*N*):

$$+ 1 + \; N (1 + N (1 + 1 + 2) + 2)$$

$$= 2 \quad + N (3 + N (\quad 4 \quad) \quad)$$

$$= 2 + N (3 + 4N)$$

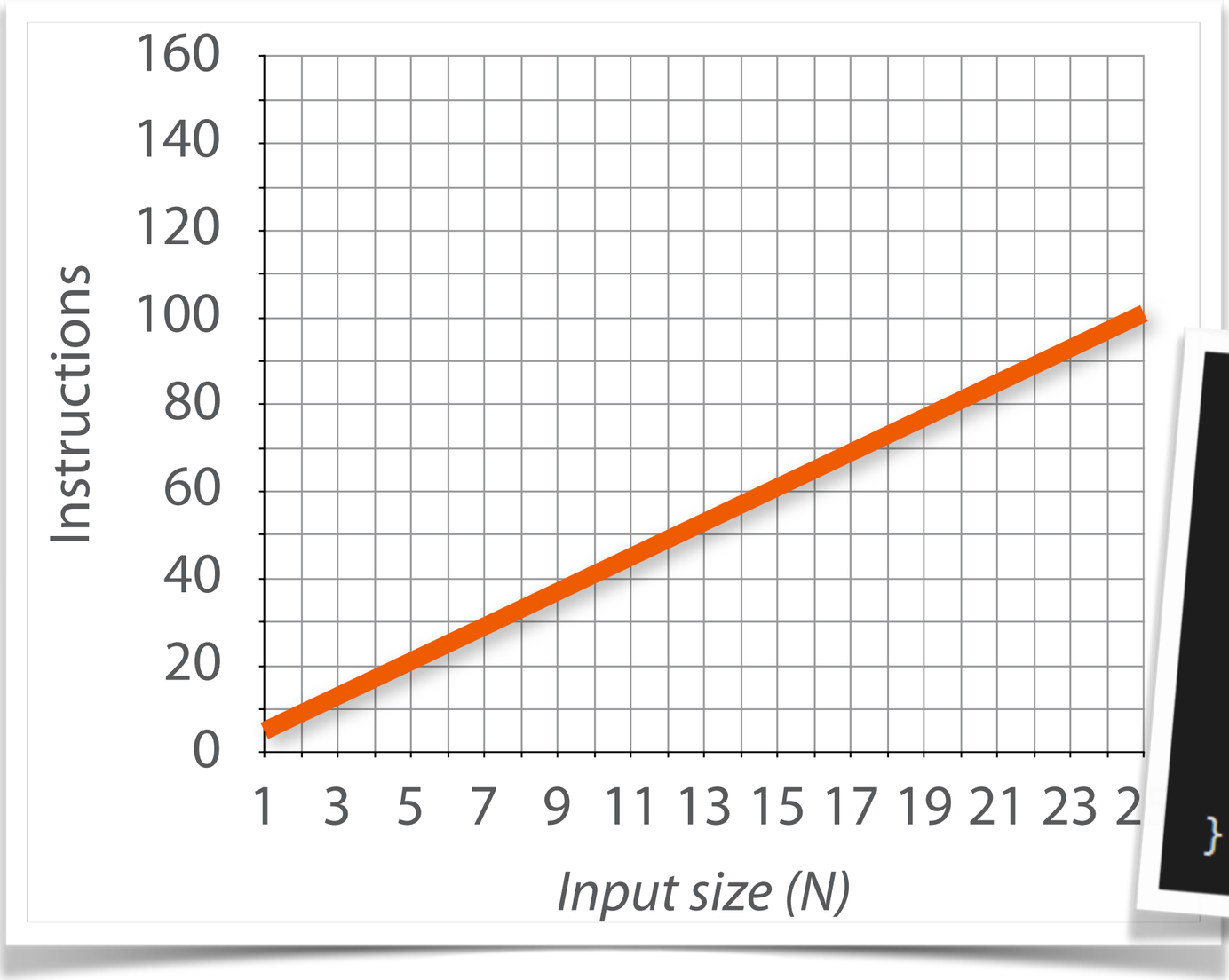
$$= 2 + 3N + 4N^2$$

$$(2 + 3N + 4N^2) \cdot 0.5$$

Big Theta (Θ)

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

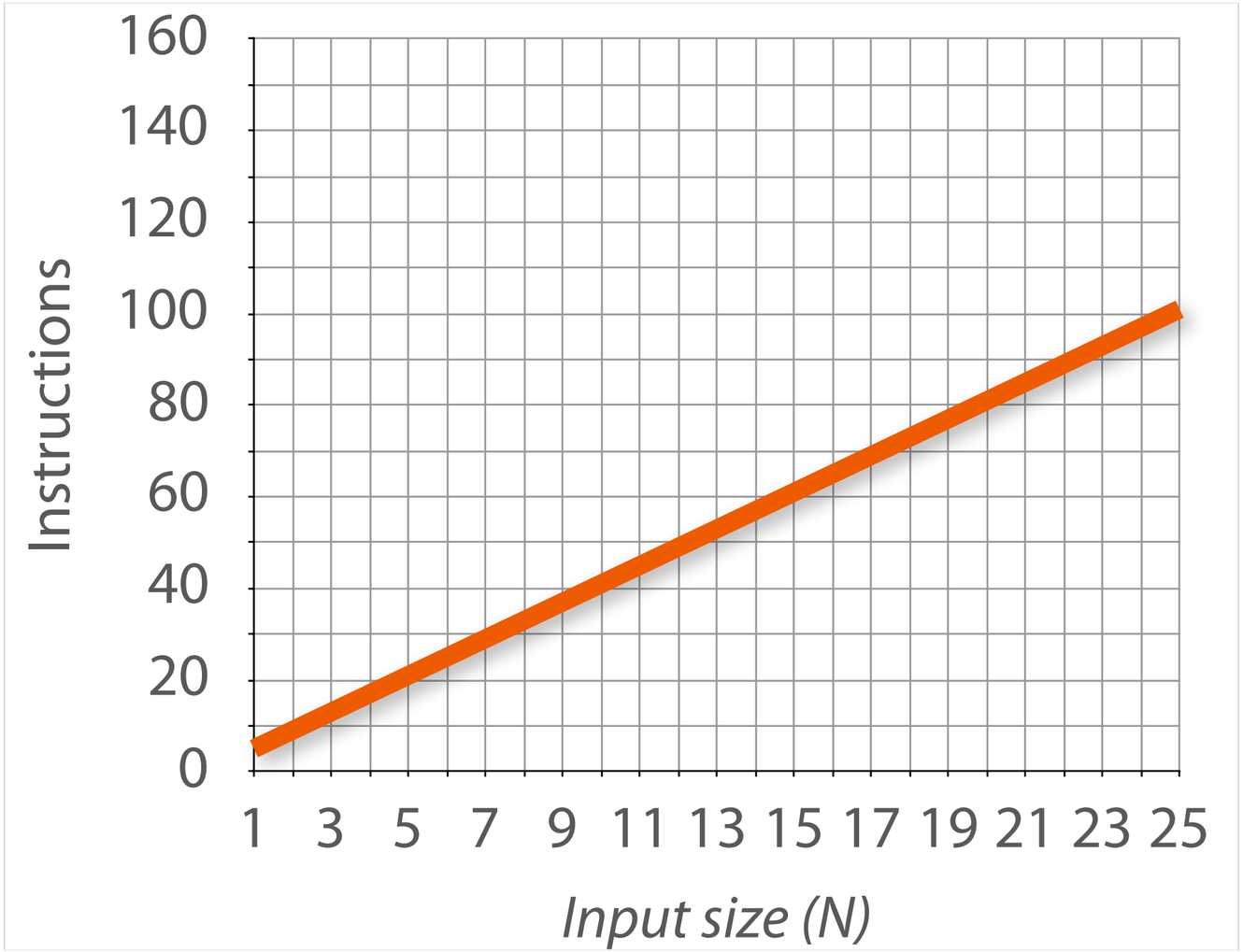
Big Theta (Θ)



$f(N) = 4N + 1$

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

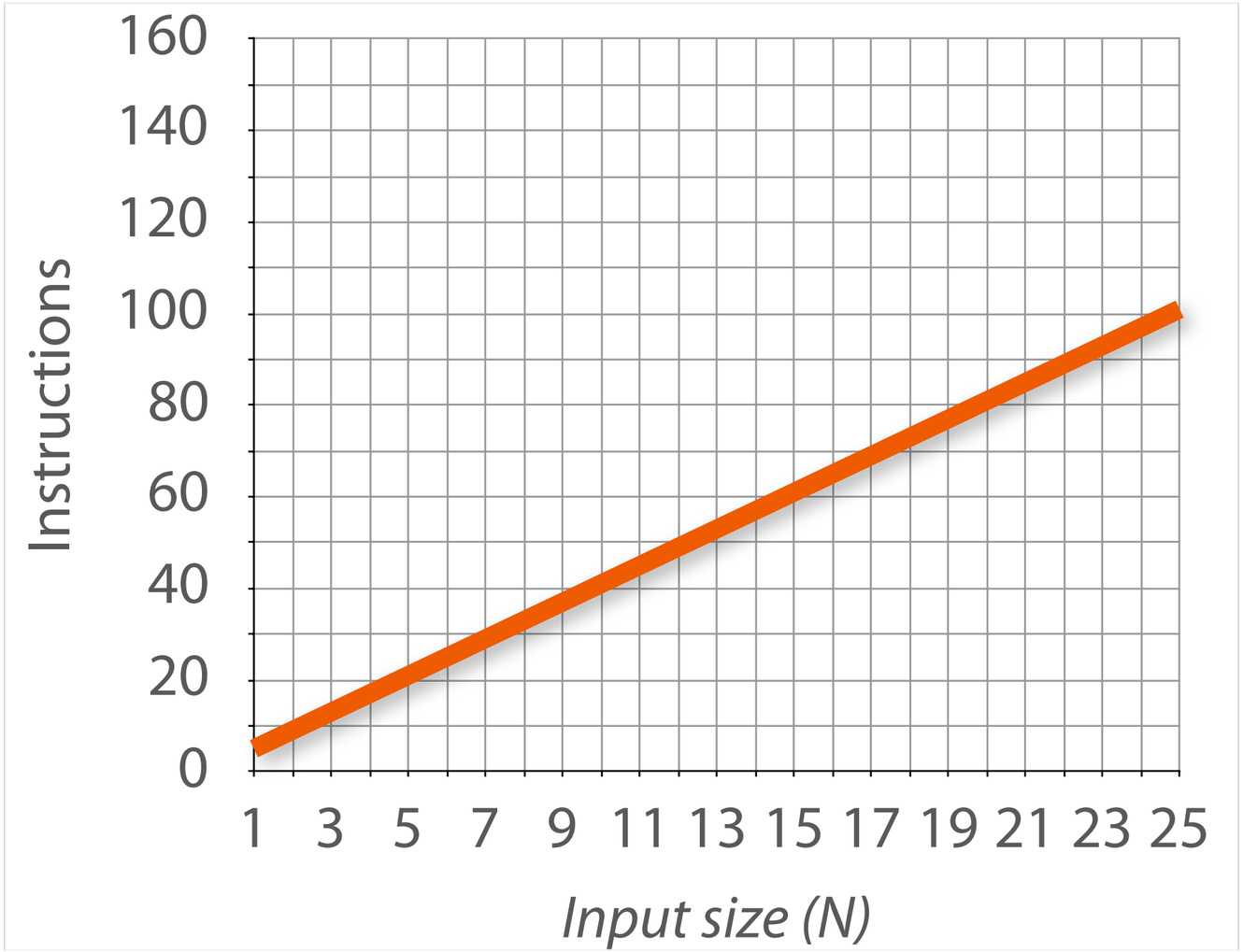
Big Theta (Θ)



$f(N) = 4N + 1$

$g(N) = N$

Big Theta (Θ)

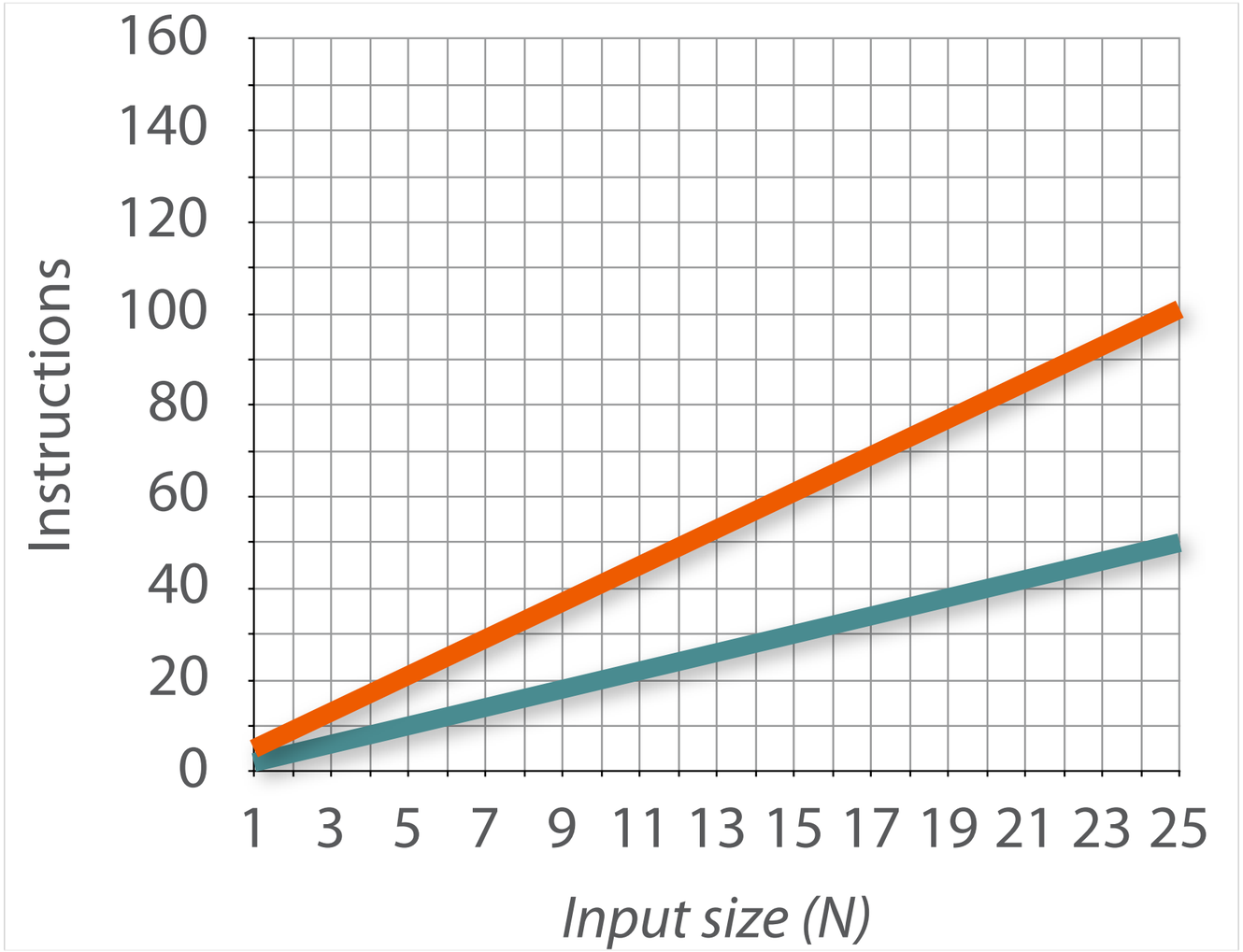


$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)

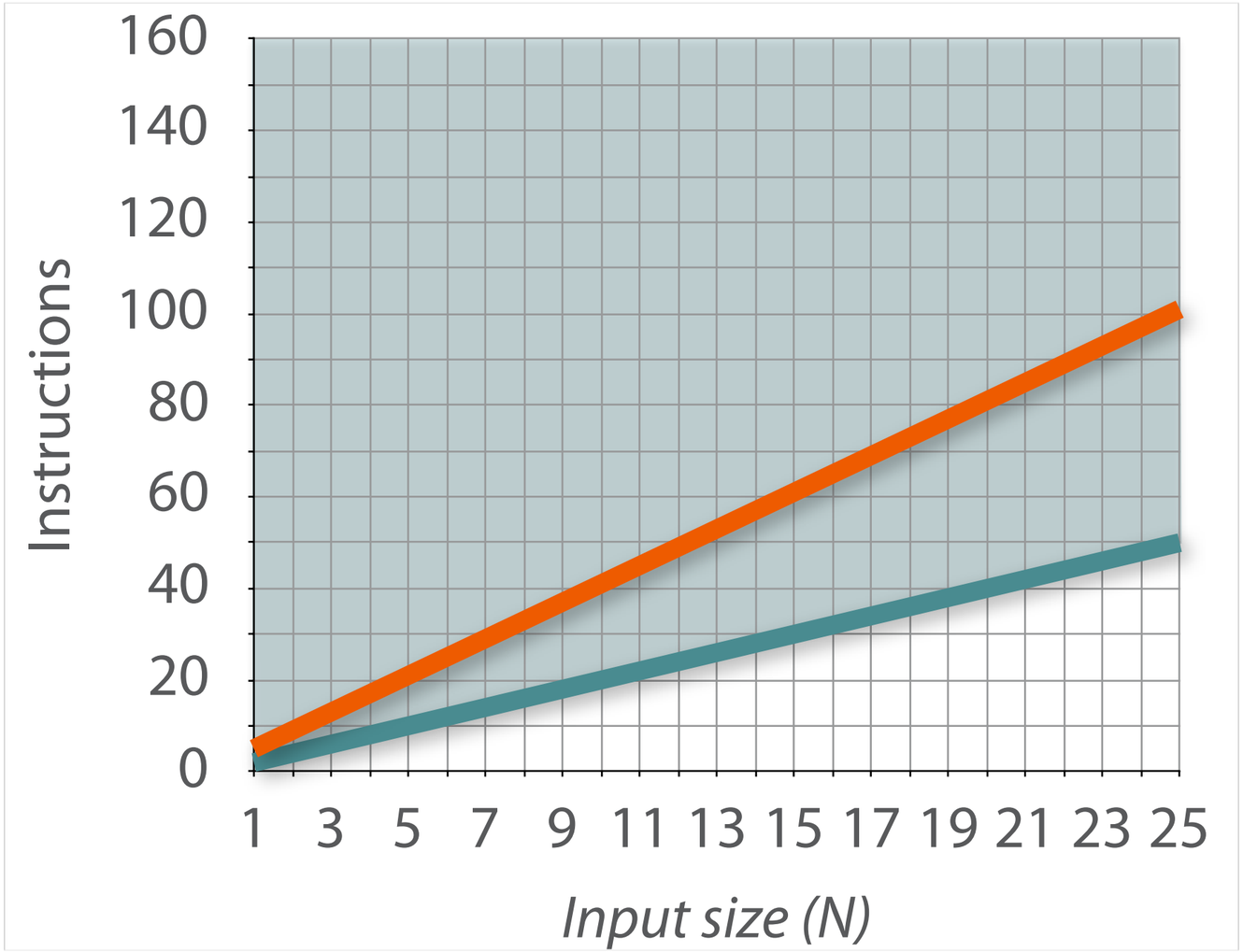


$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)

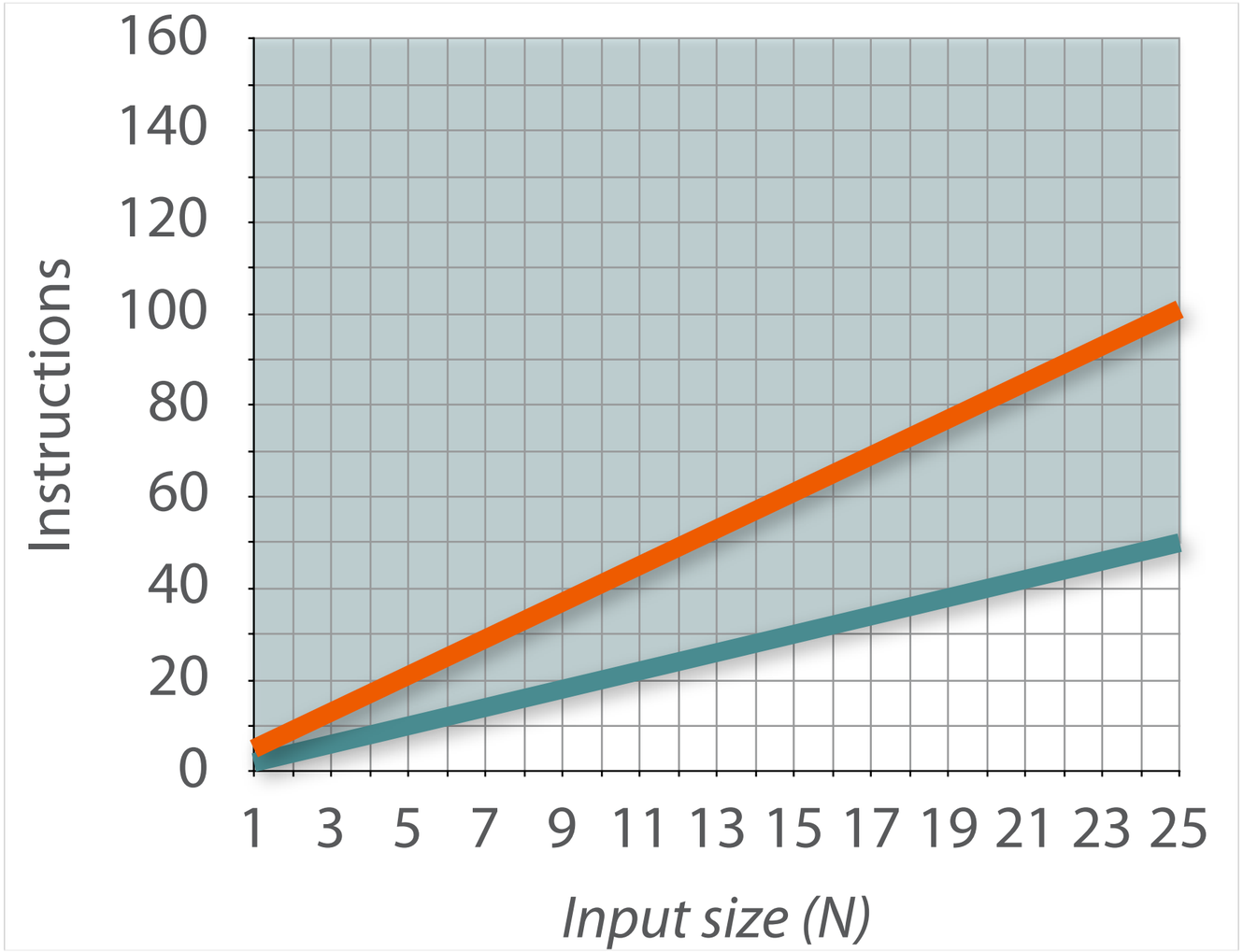


$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)



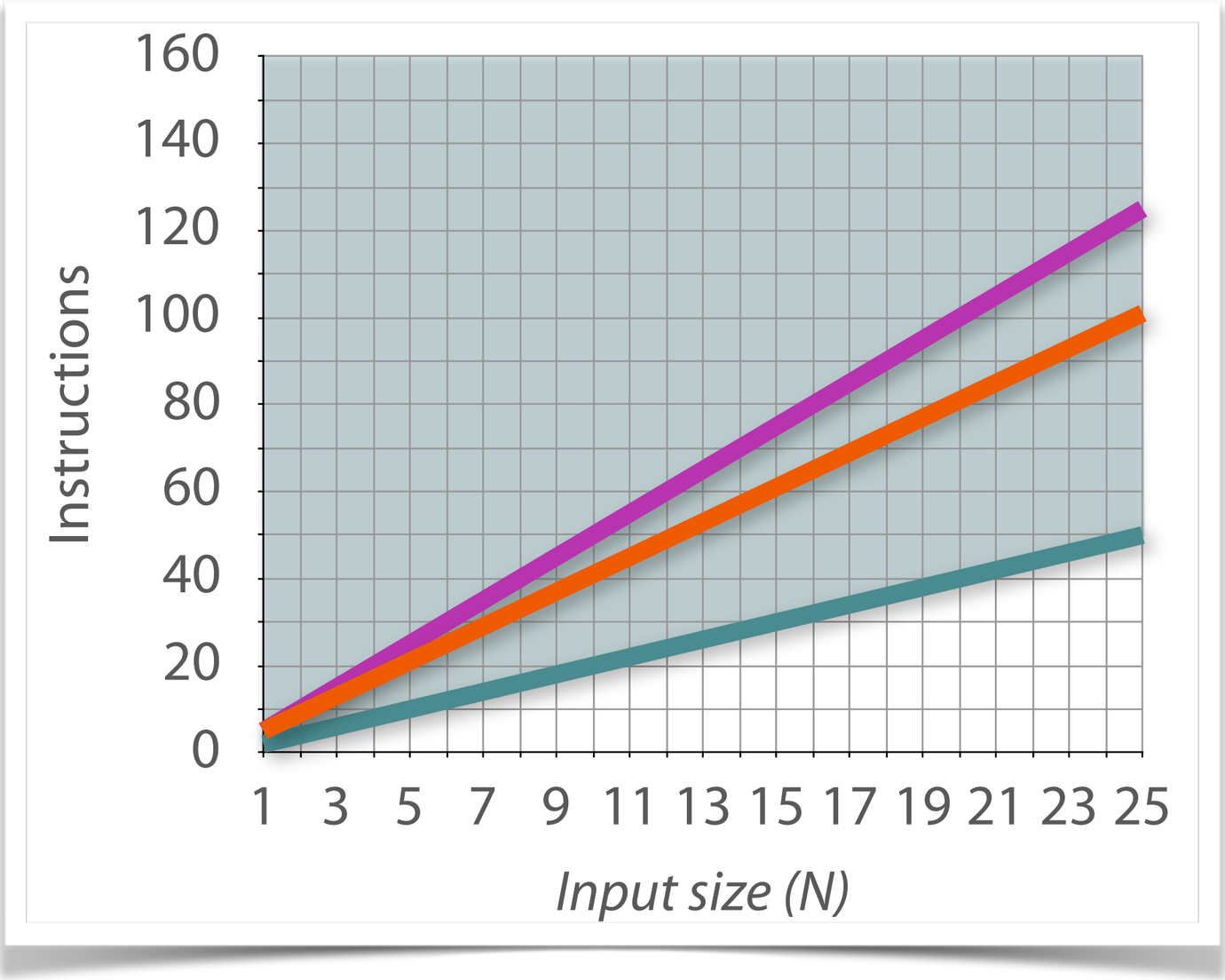
$5g(N) = 5N$

$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)



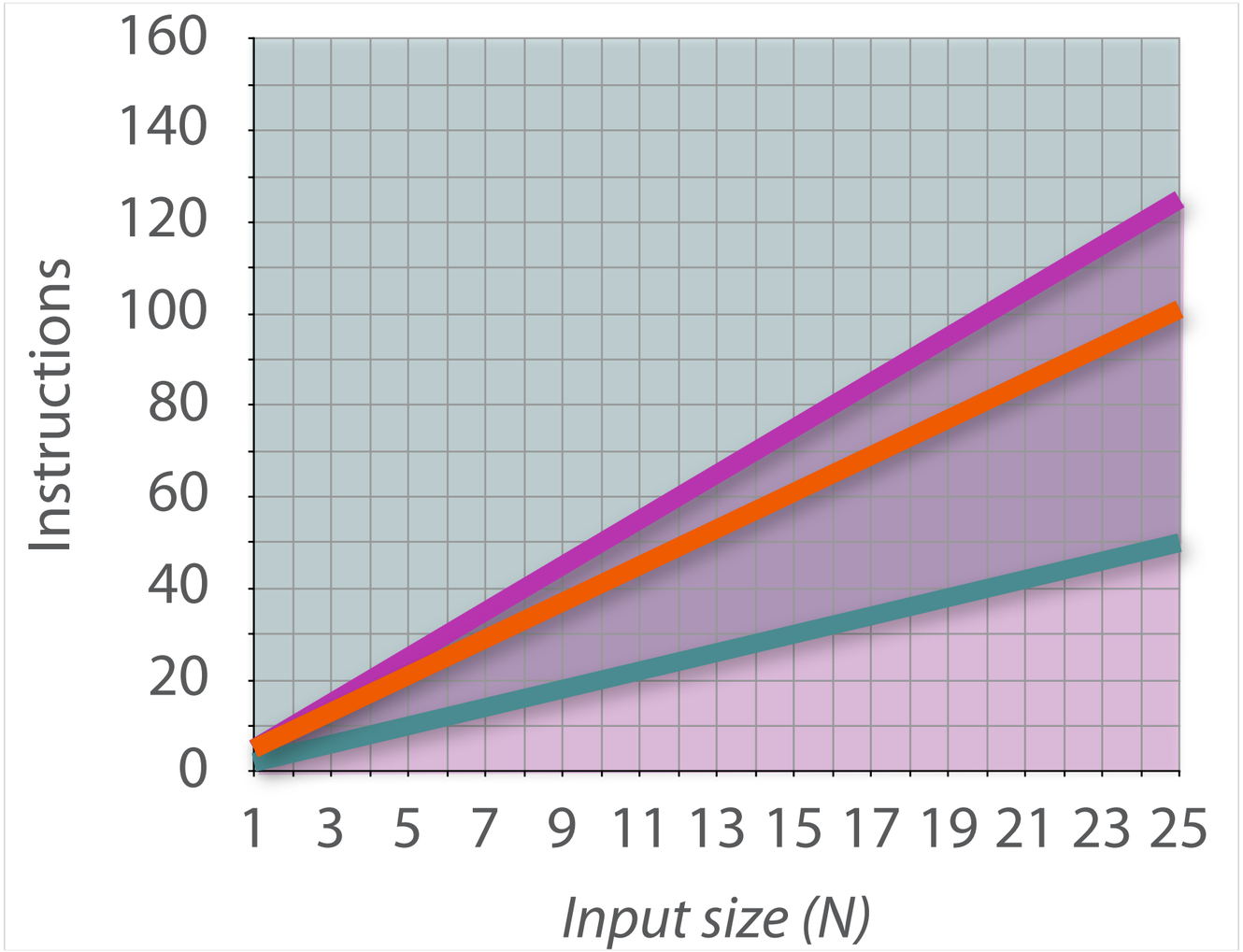
$5g(N) = 5N$

$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)



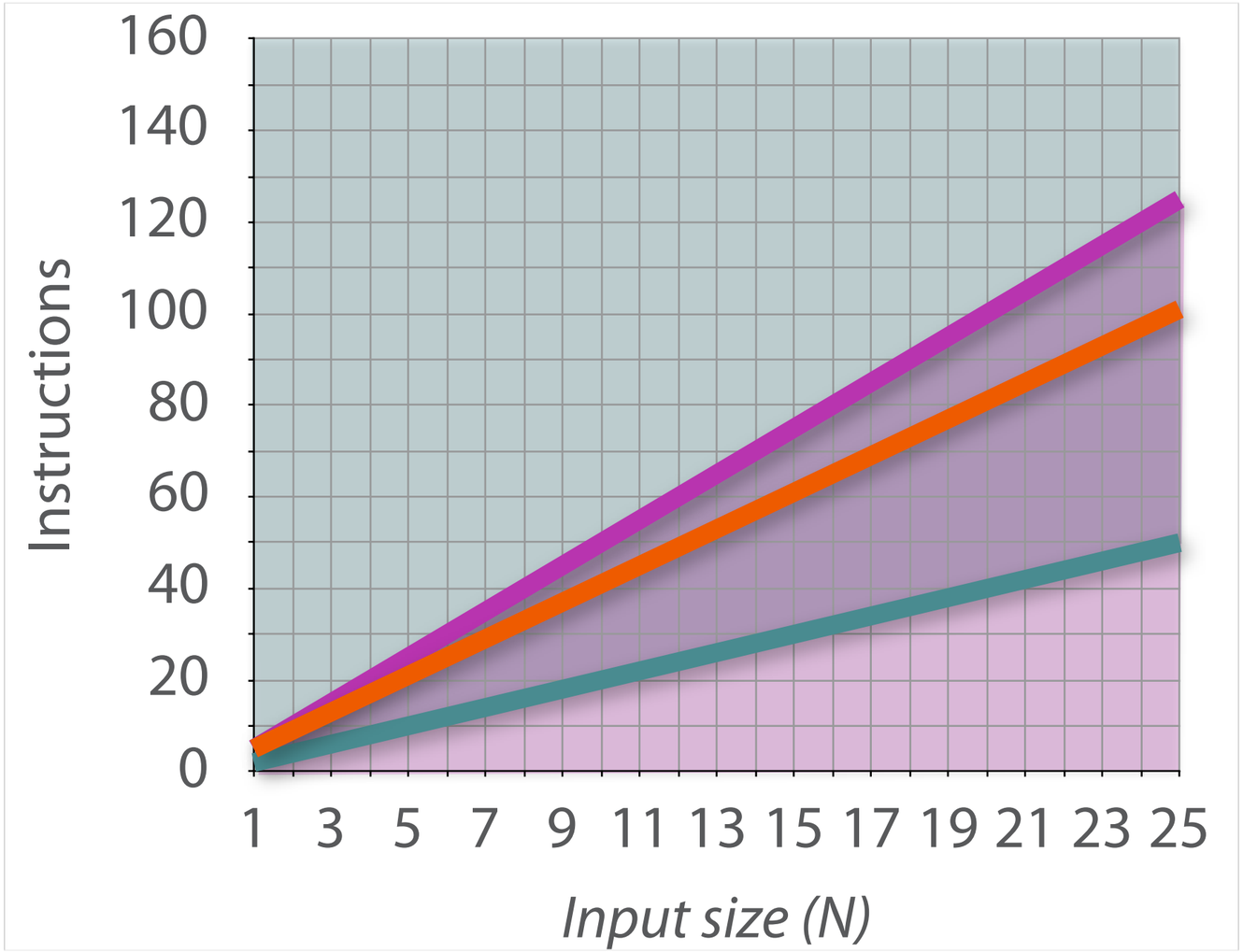
$5g(N) = 5N$

$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

Big Theta (Θ)



$5g(N) = 5N$

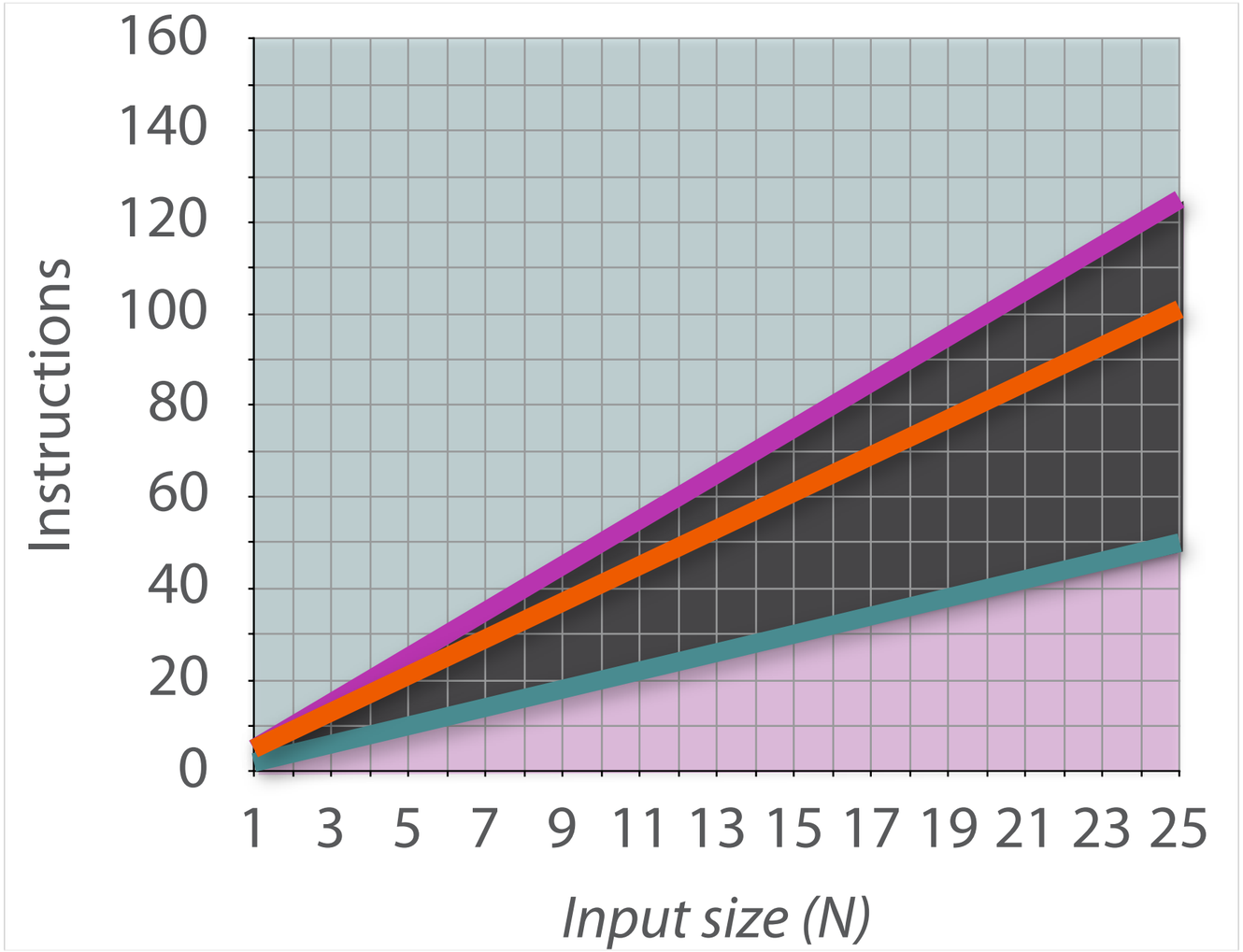
$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

Big Theta (Θ)



$5g(N) = 5N$

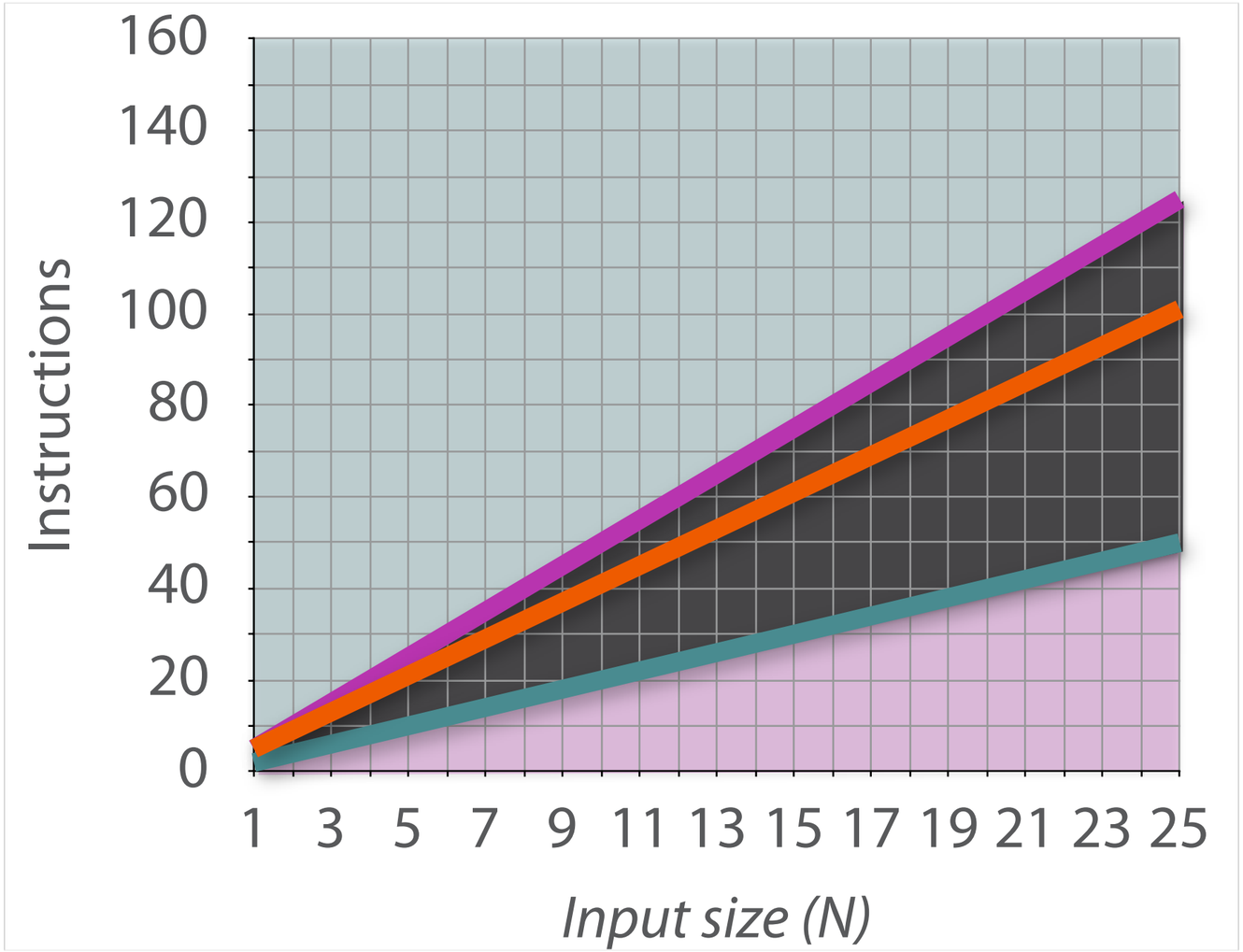
$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

Big Theta (Θ)



$5g(N) = 5N$

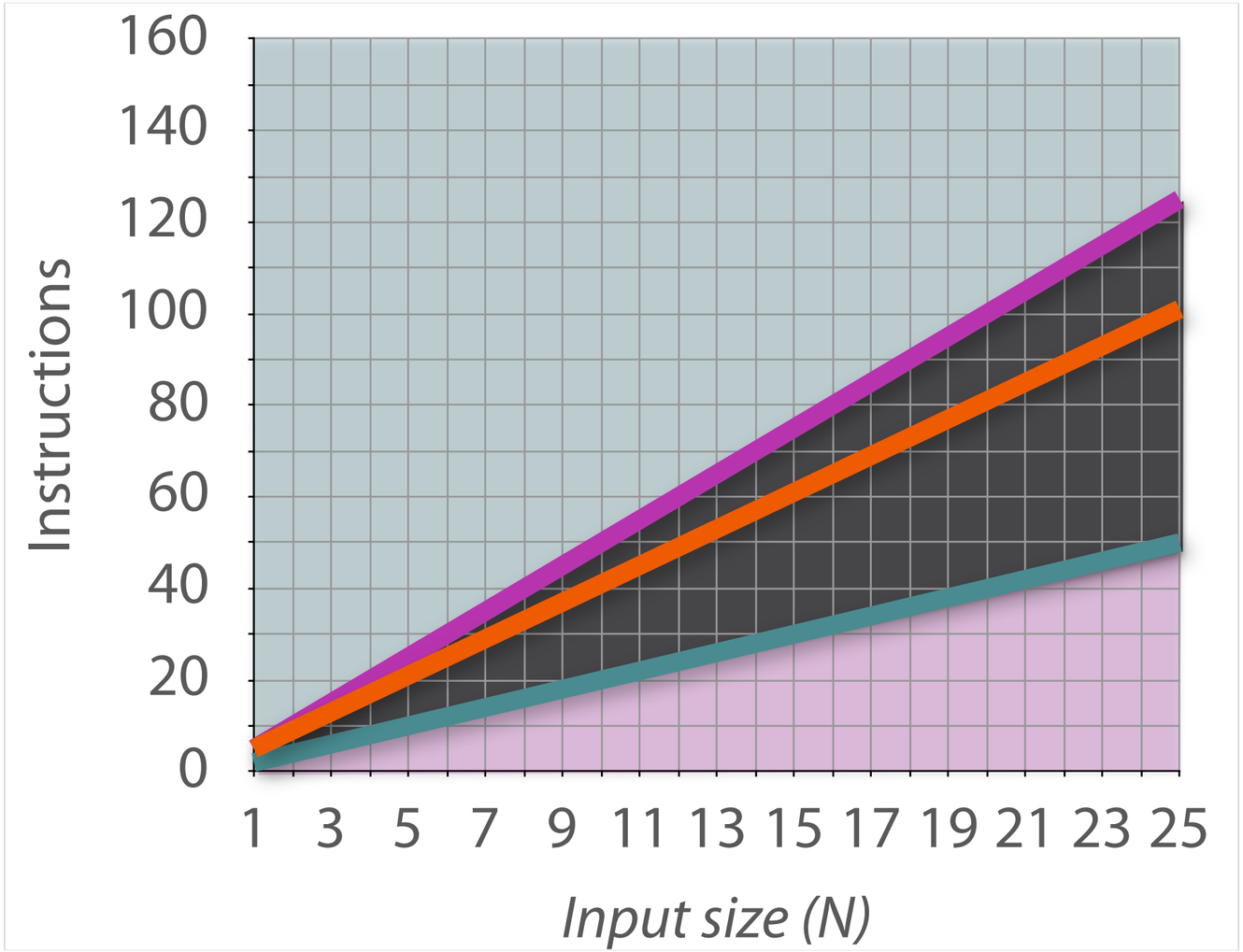
$f(N) = 4N + 1$

$g(N) = N$

$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

Big Theta (Θ)



$5g(N) = 5N$

$f(N) = 4N + 1$

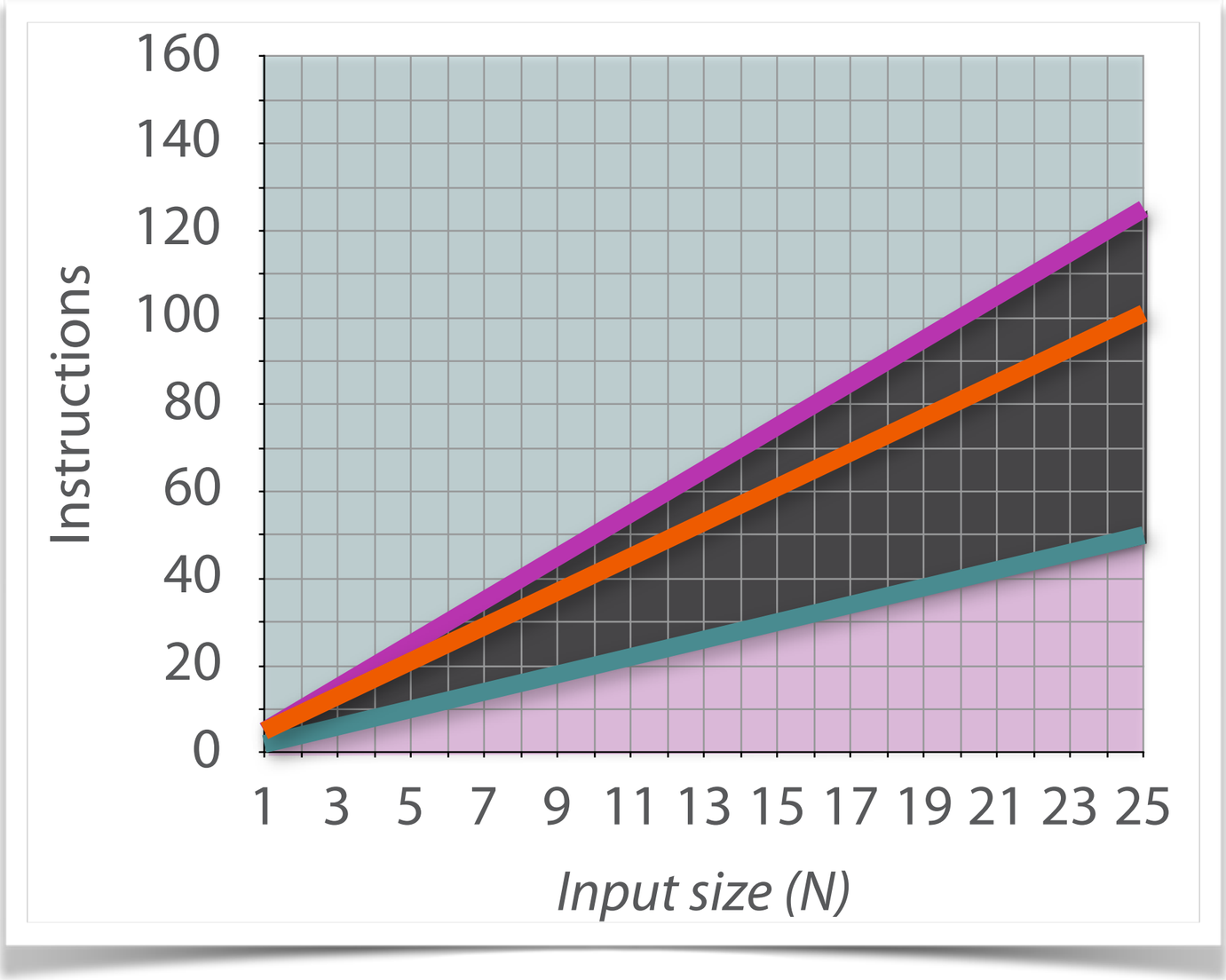
$g(N) = N$

$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

$\Theta(g(N))$

Big Theta (Θ)



$5g(N) = 5N$

$f(N) = 4N + 1$

$g(N) = N$

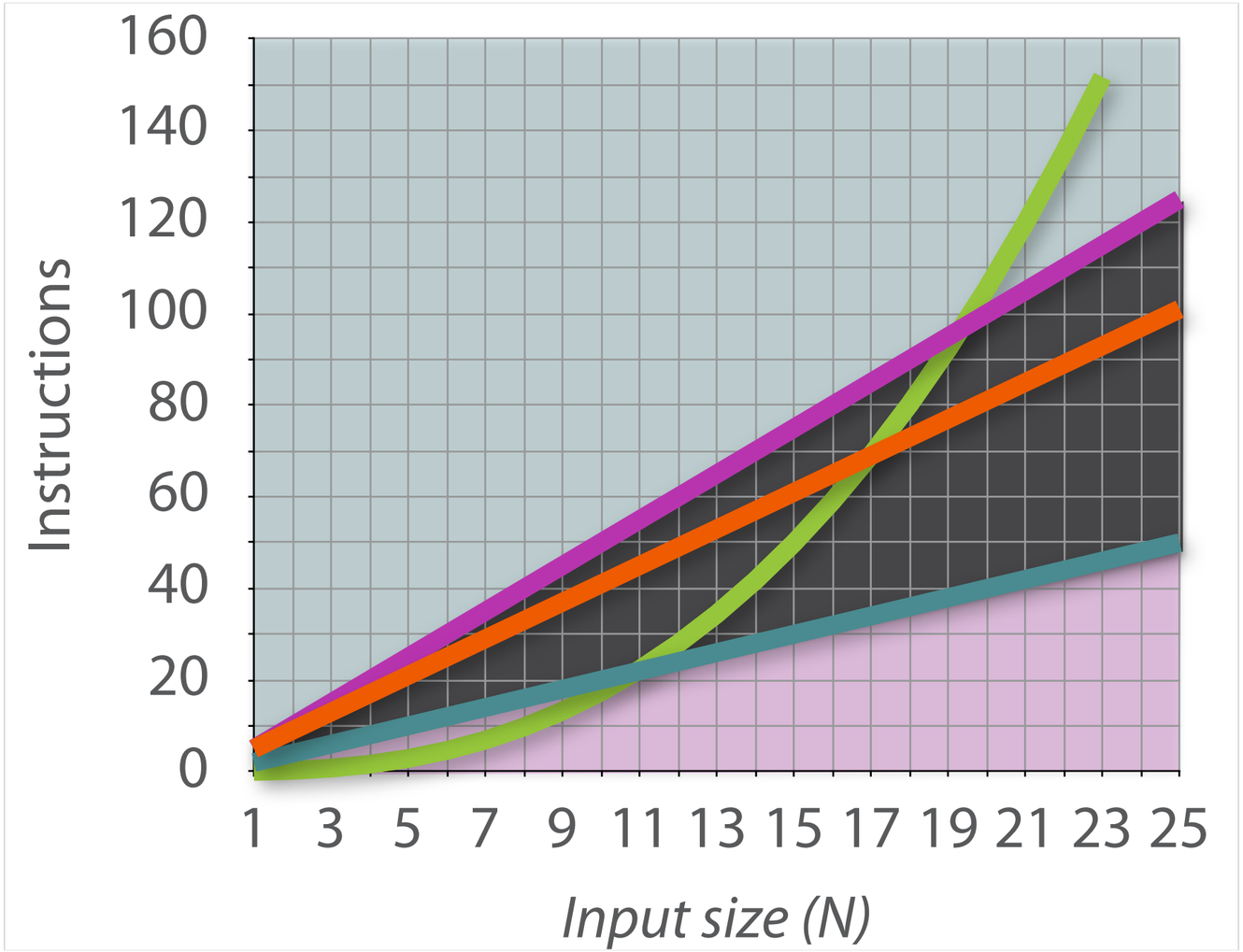
$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

$\Theta(g(N))$

$\Theta(N)$

Big Theta (Θ)



$5g(N) = 5N$

$f(N) = 4N + 1$

$g(N) = N$

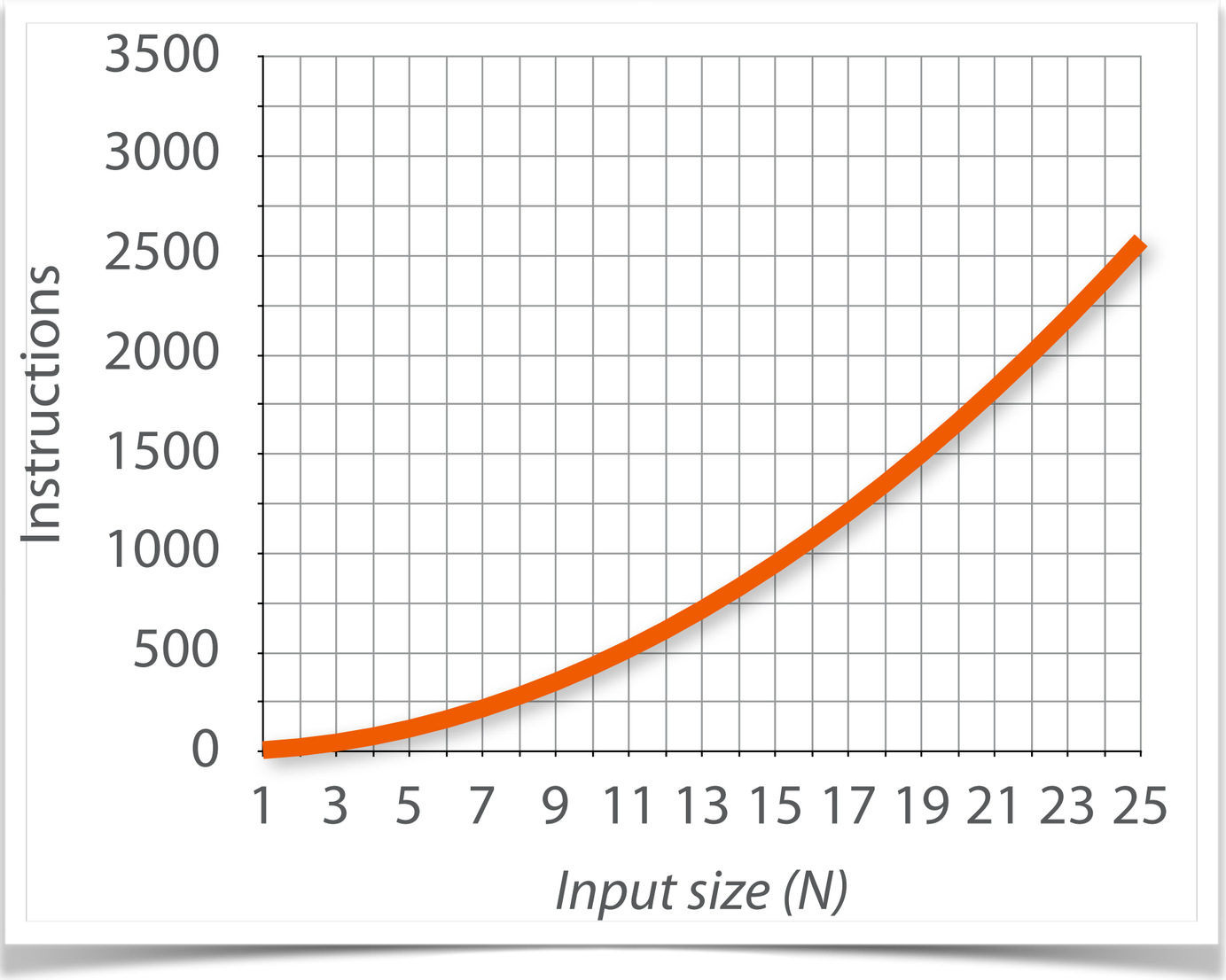
$3g(N) = 3N$

$3g(N) < f(N) < 5g(N)$

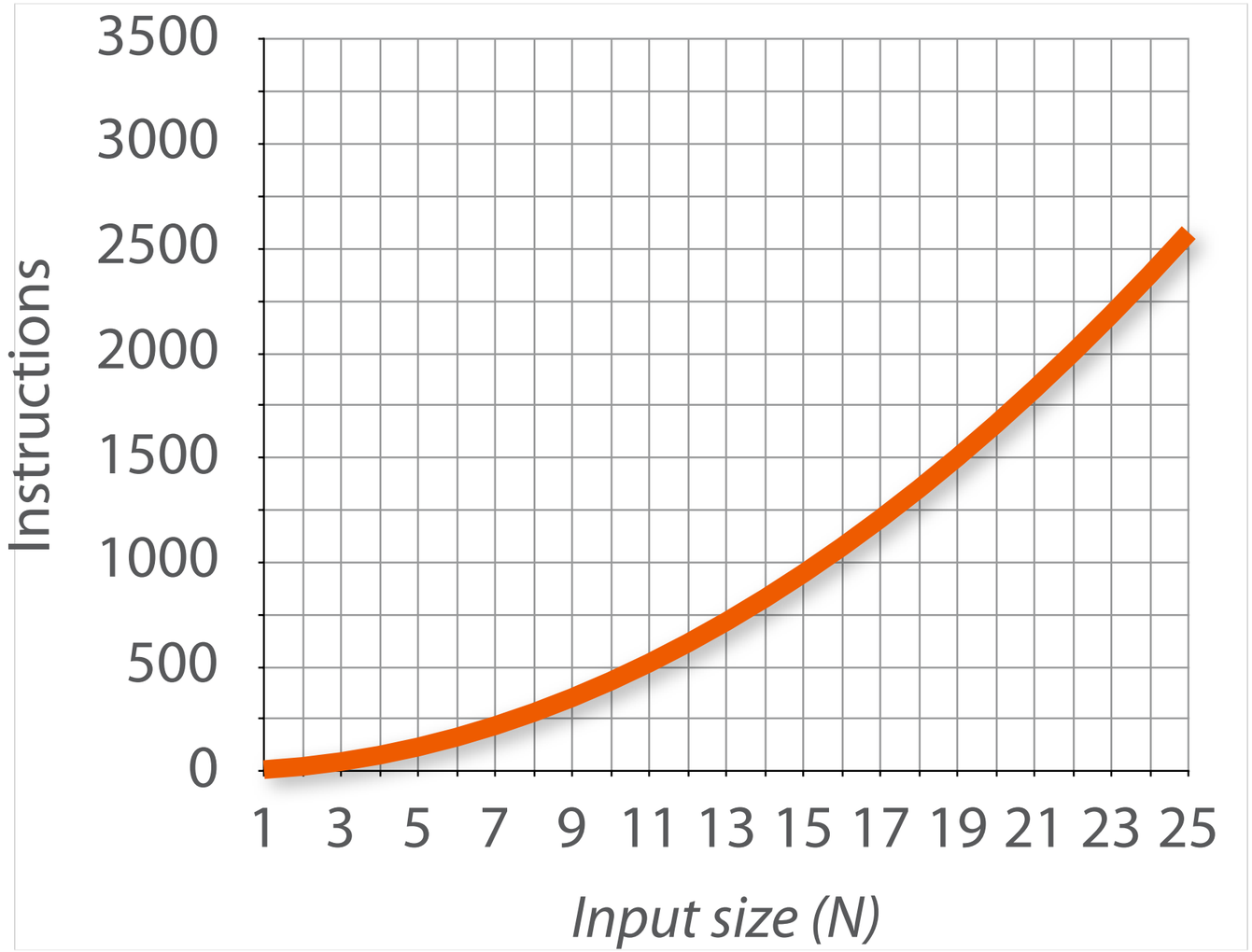
$\Theta(g(N))$

$\Theta(N)$

```
void CreateAllPairs(int N)
{
    var x = 0;
    var y = 0;
    while (x < N)
    {
        while (y < N)
        {
            Console.WriteLine("{0}, {1}", x, y);
            y = y + 1;
        }
        x = x + 1;
    }
}
```

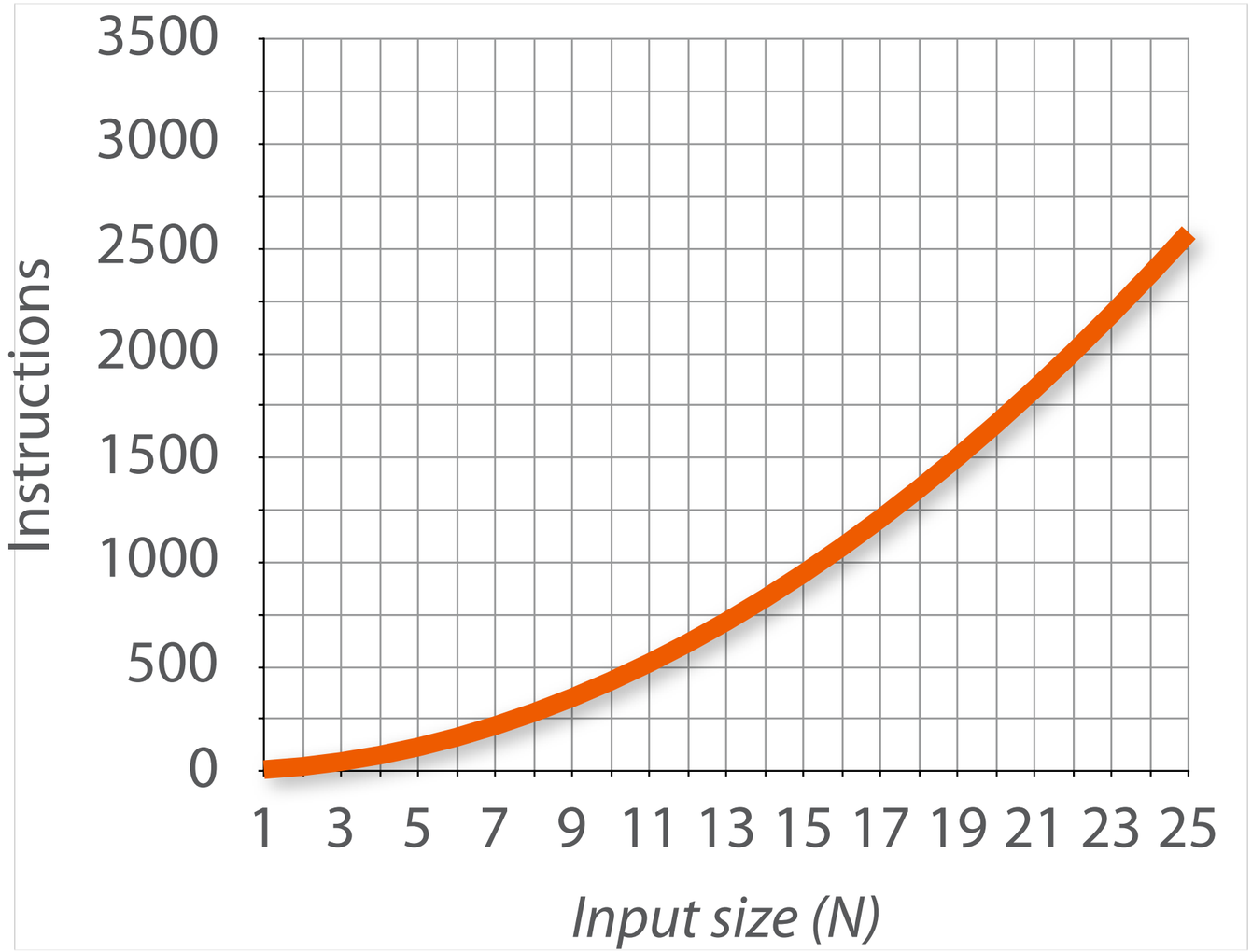

3500
3000
2500
2000
1500
1000
500
0
Instructions
1 3 5 7 9 11 13 15 17 19 21 23 25
Input size (N)

$f(N) = 2 + 3N + 4N^2$



$f(N) = 2 + 3N + 4N^2$

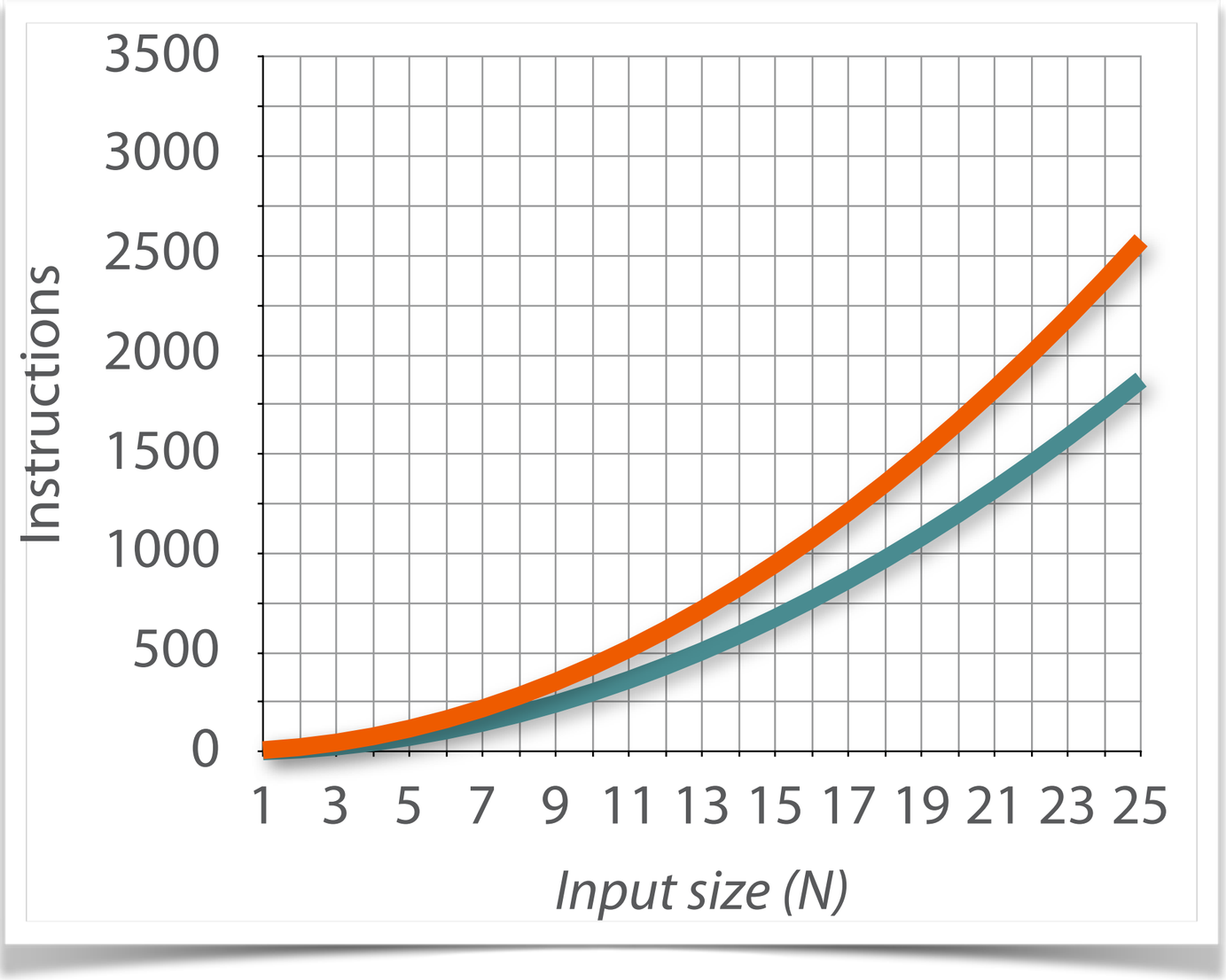
$g(N) = N^2$



$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

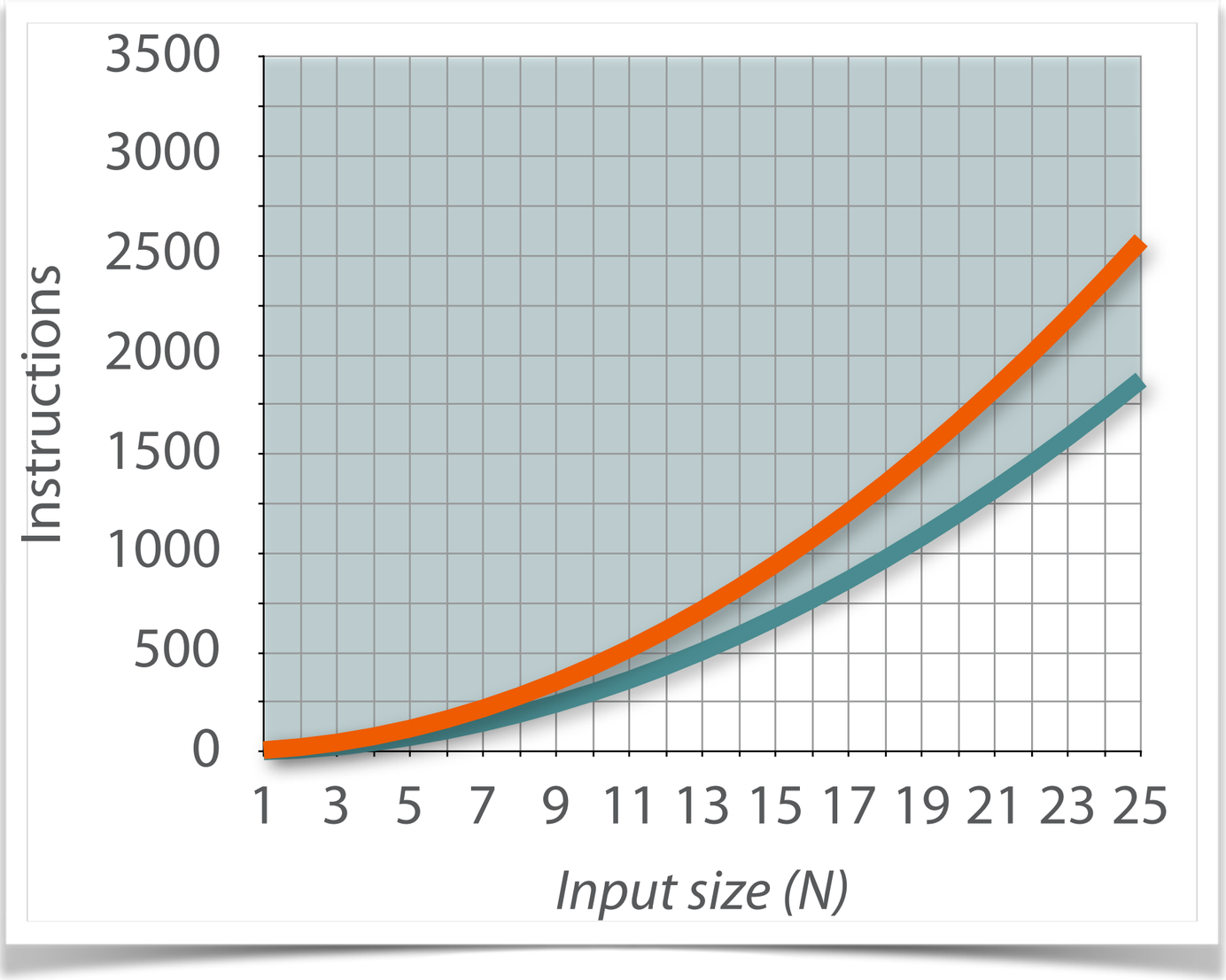
$3g(N) = 3N^2$



$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

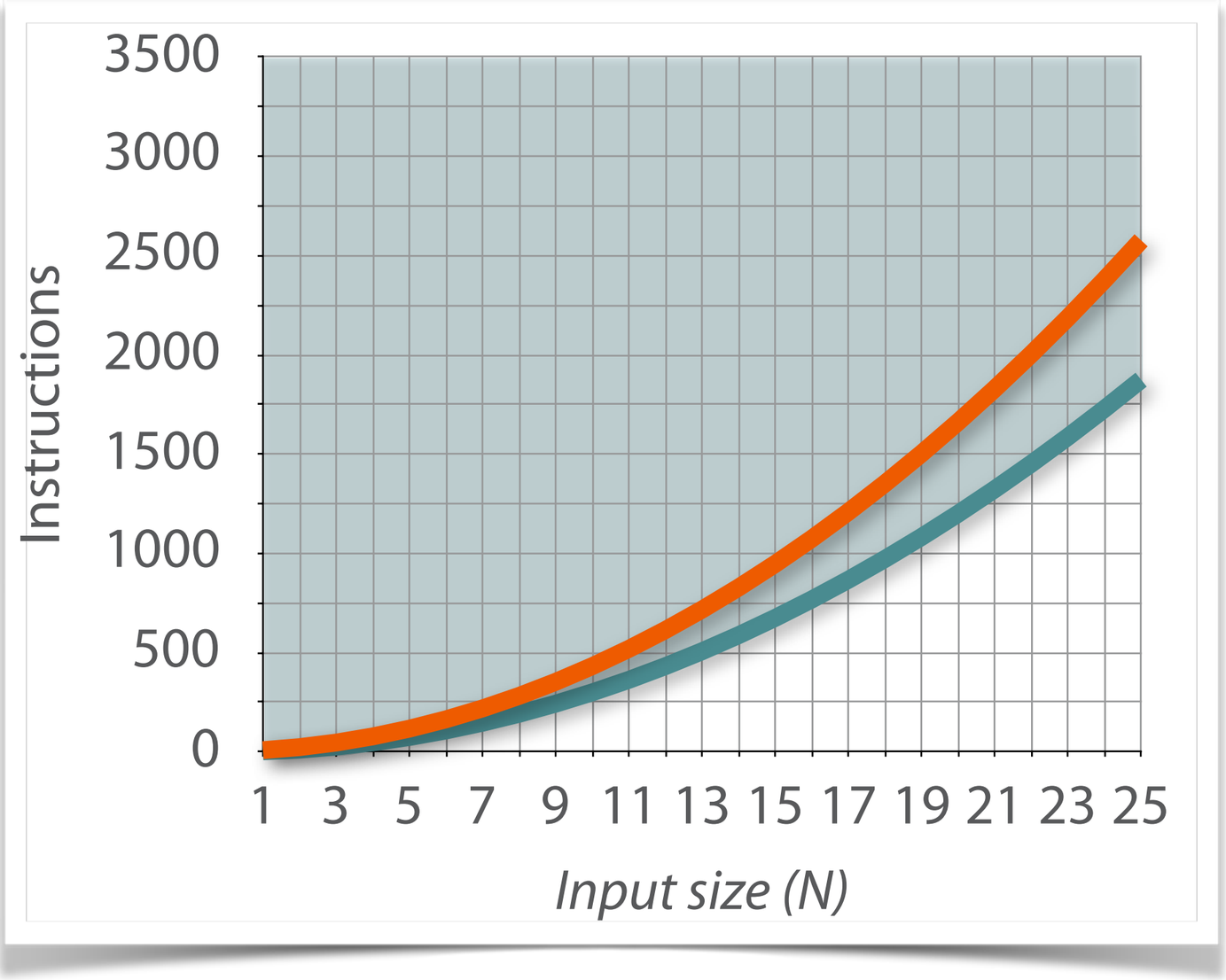
$3g(N) = 3N^2$



$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

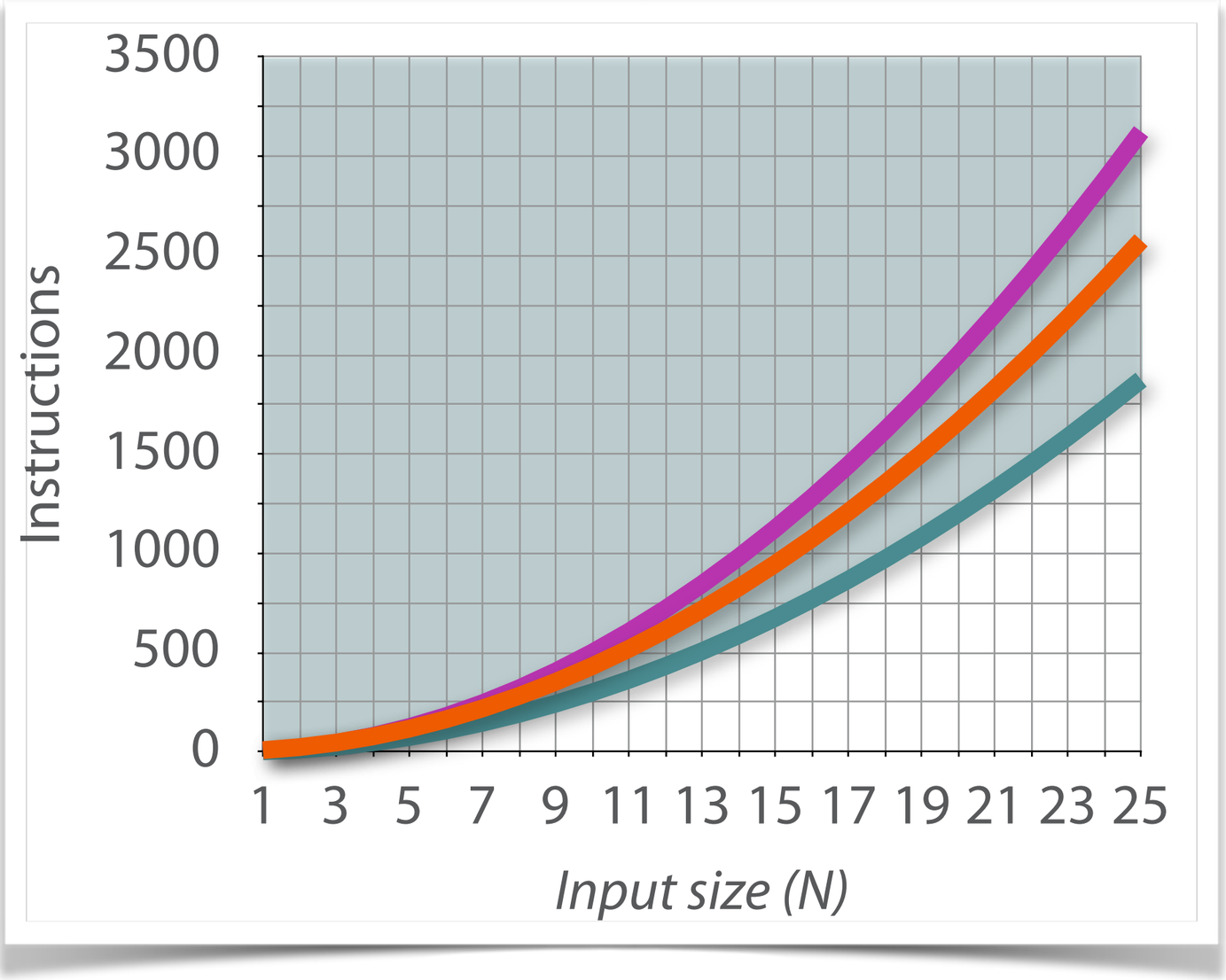


$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

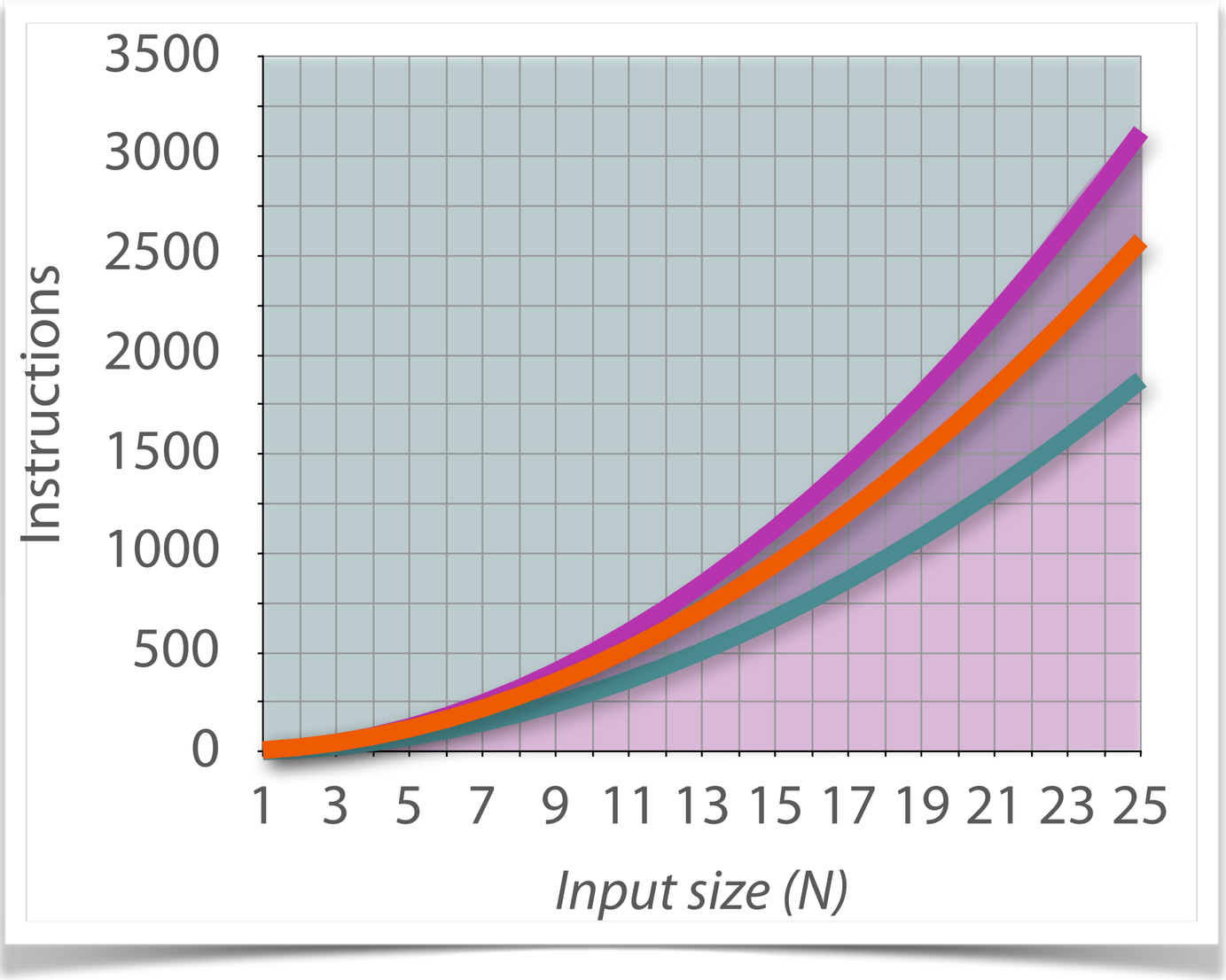


$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

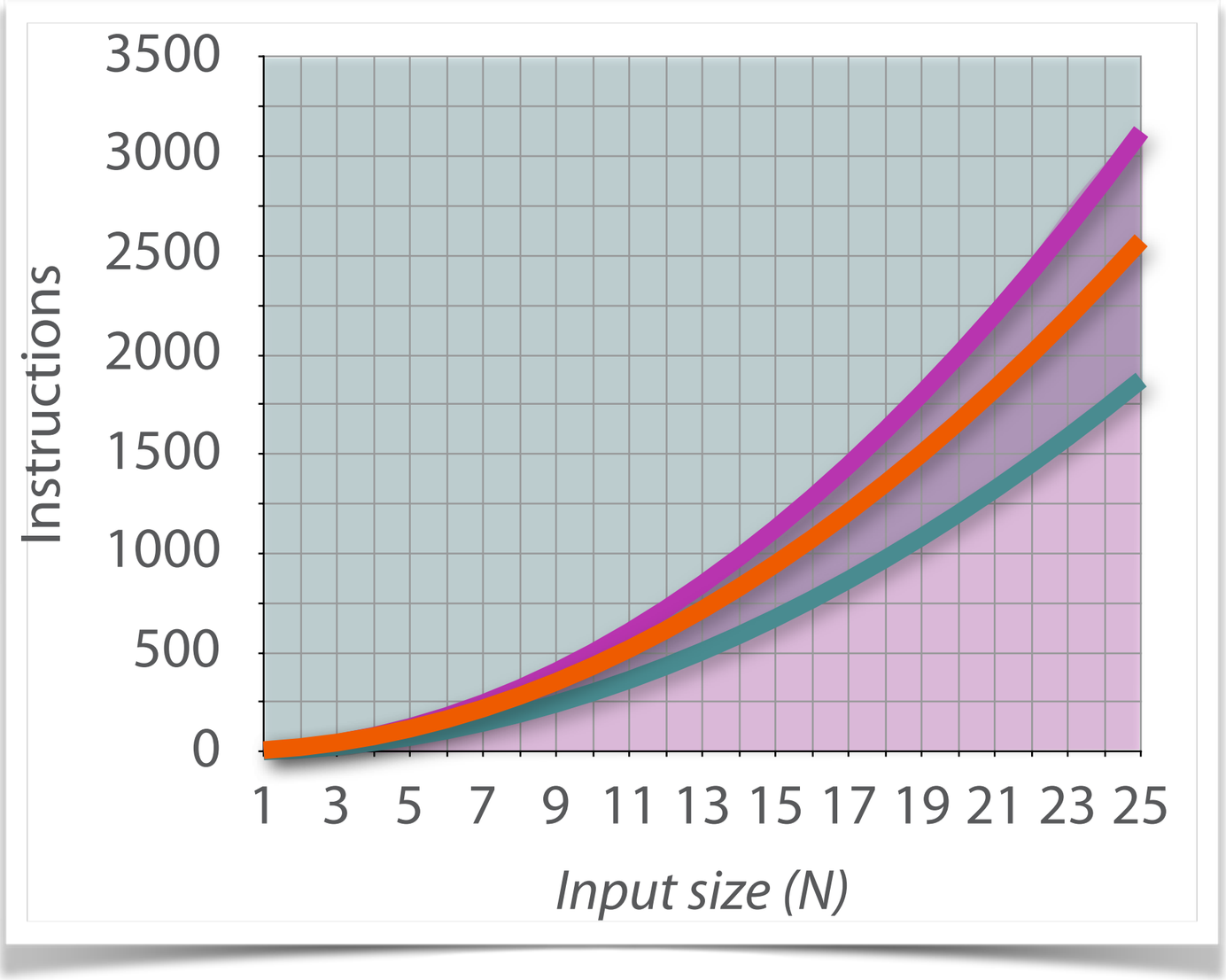


$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$



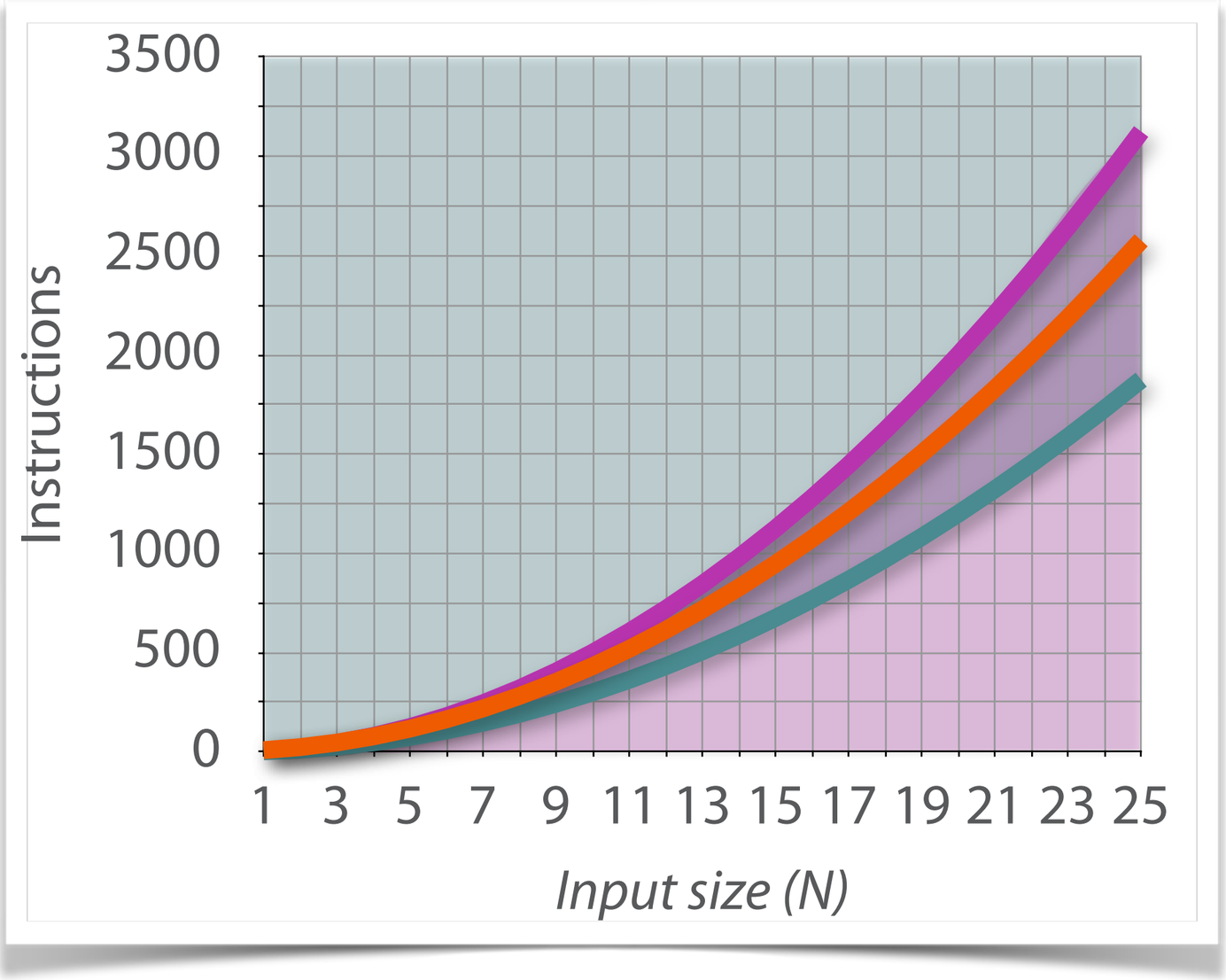
N	1	2	3	4
f(N)	9	24	47	78

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$



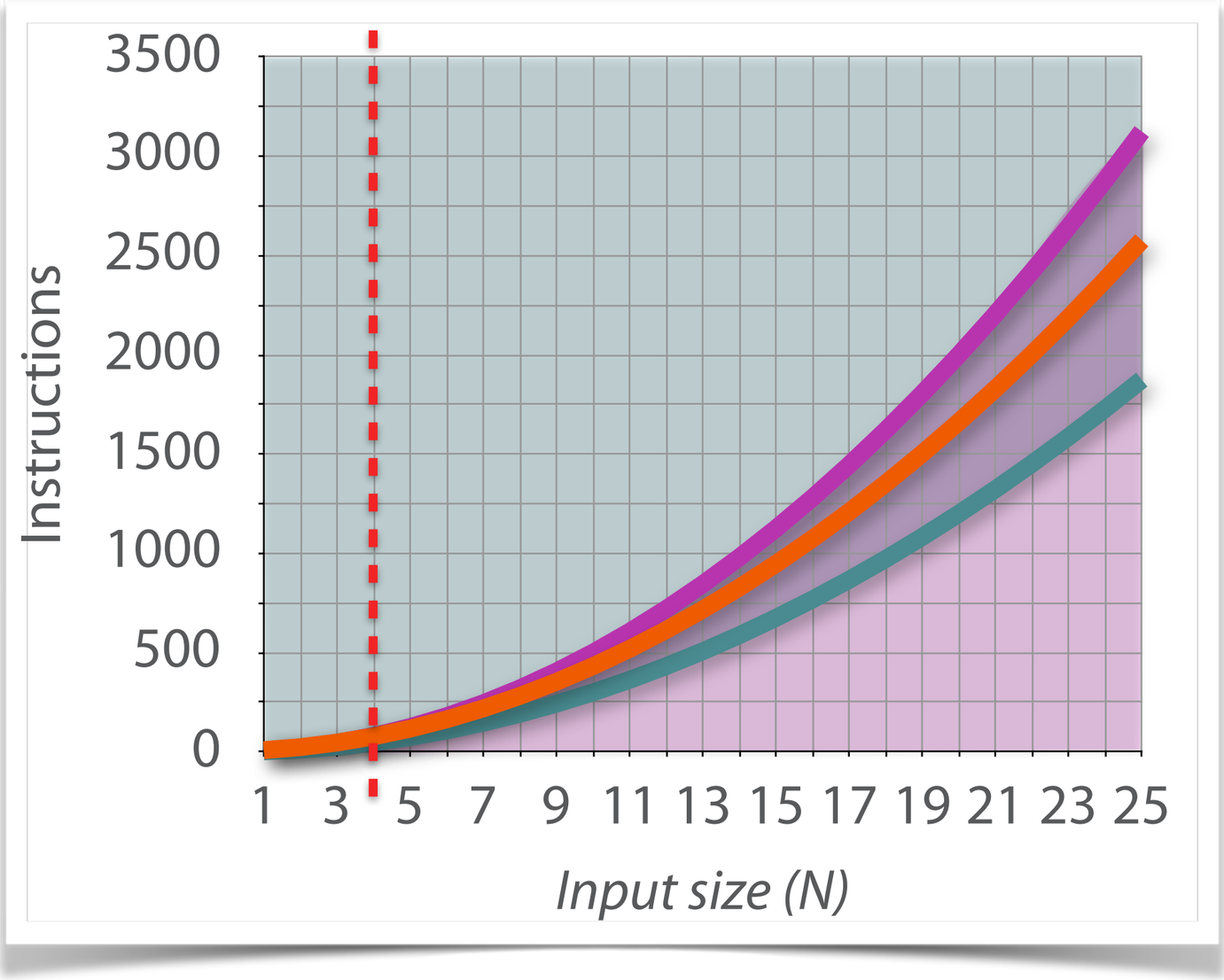
N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$



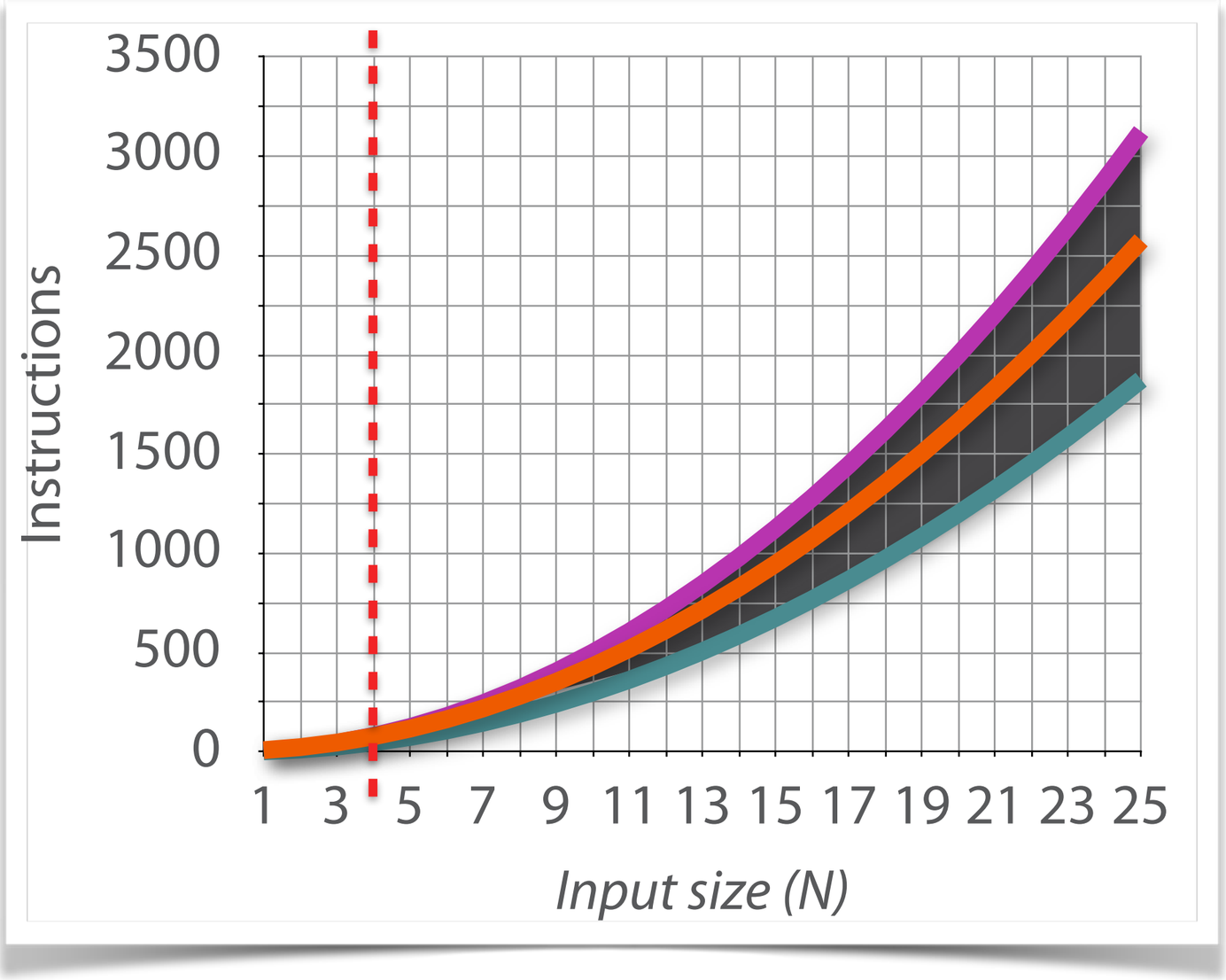
N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$



N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

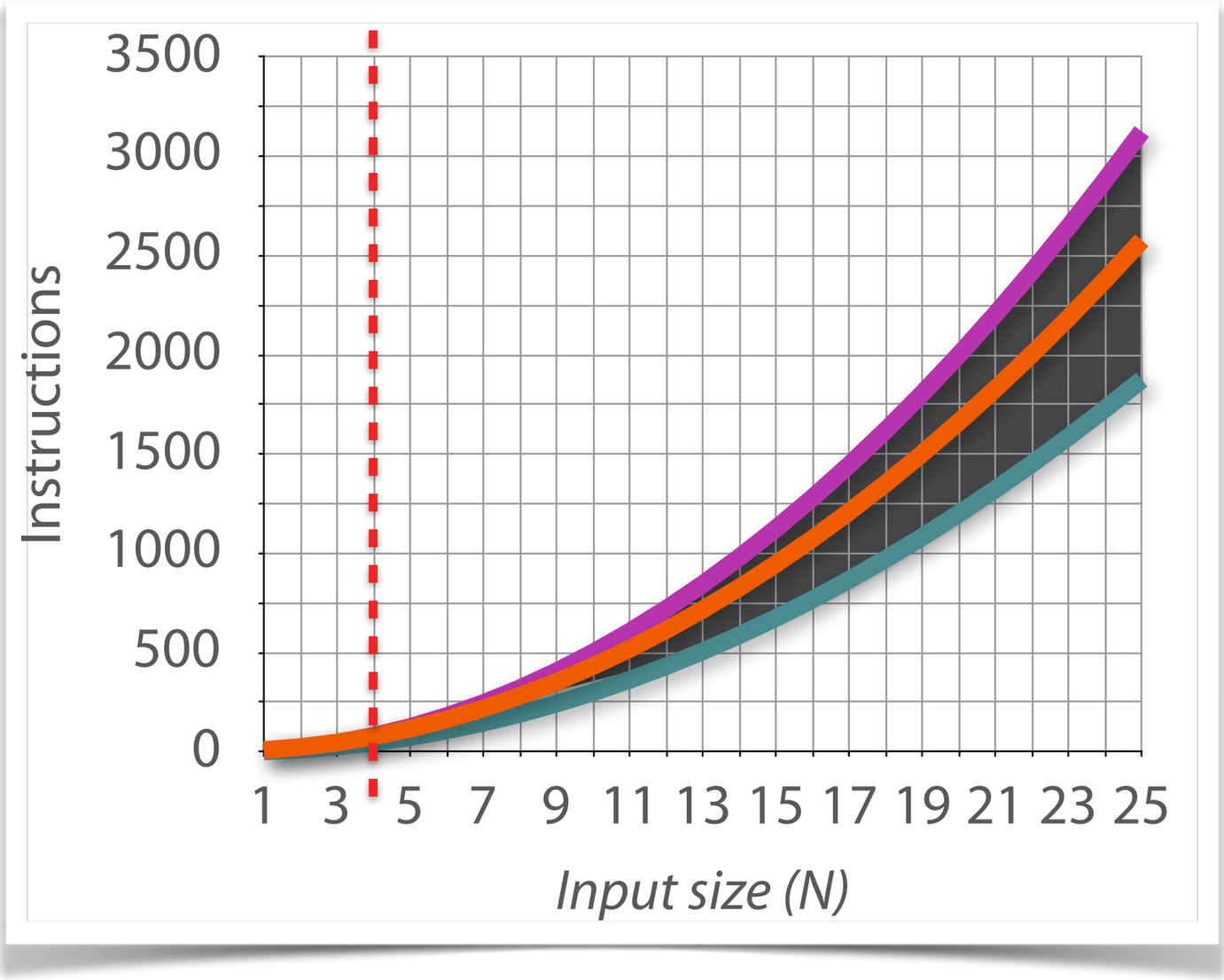
$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

$$3g(N) < f(N) < 5g(N)$$



N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

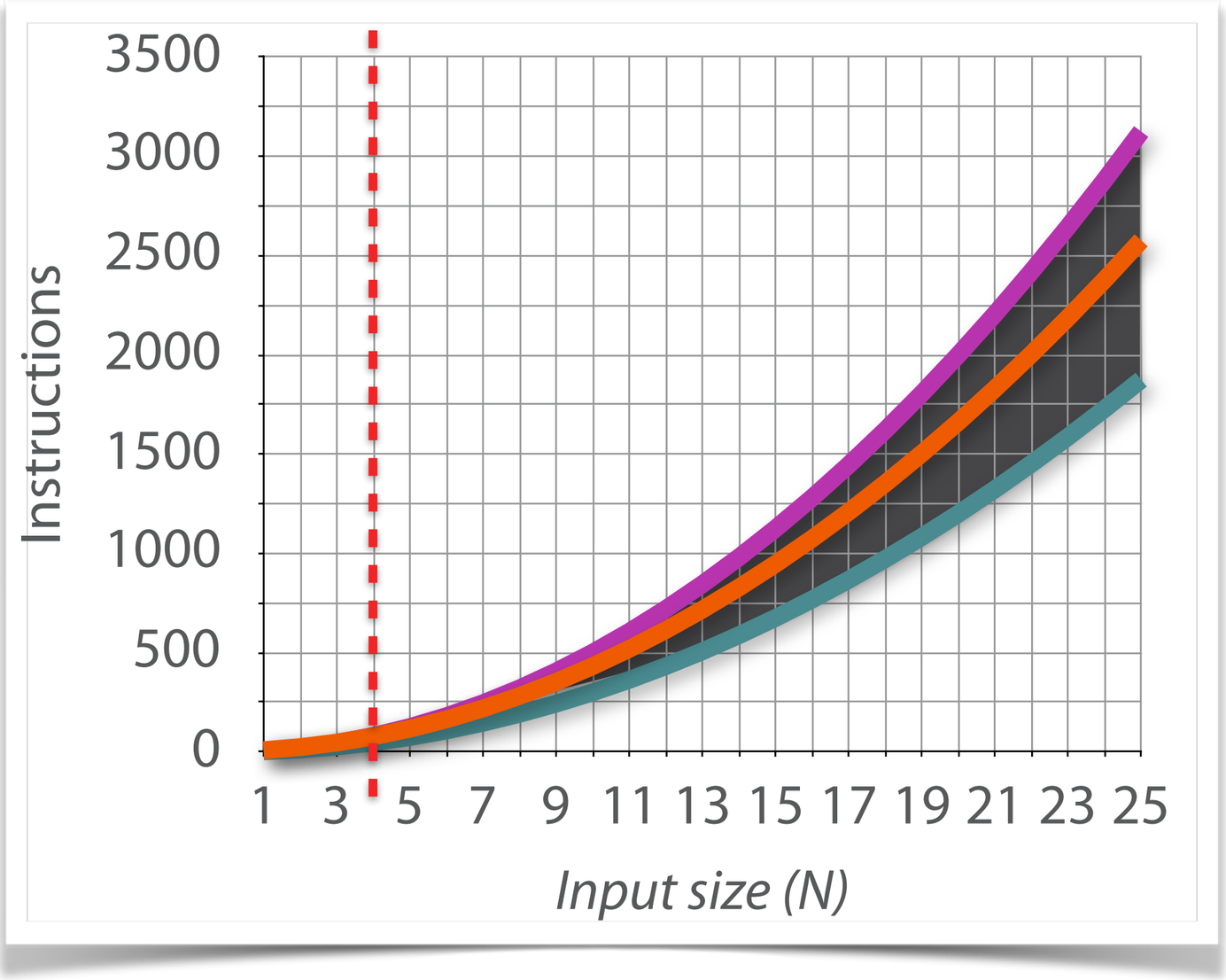
$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

$3g(N) < f(N) < 5g(N)$

when $N > 3$



N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

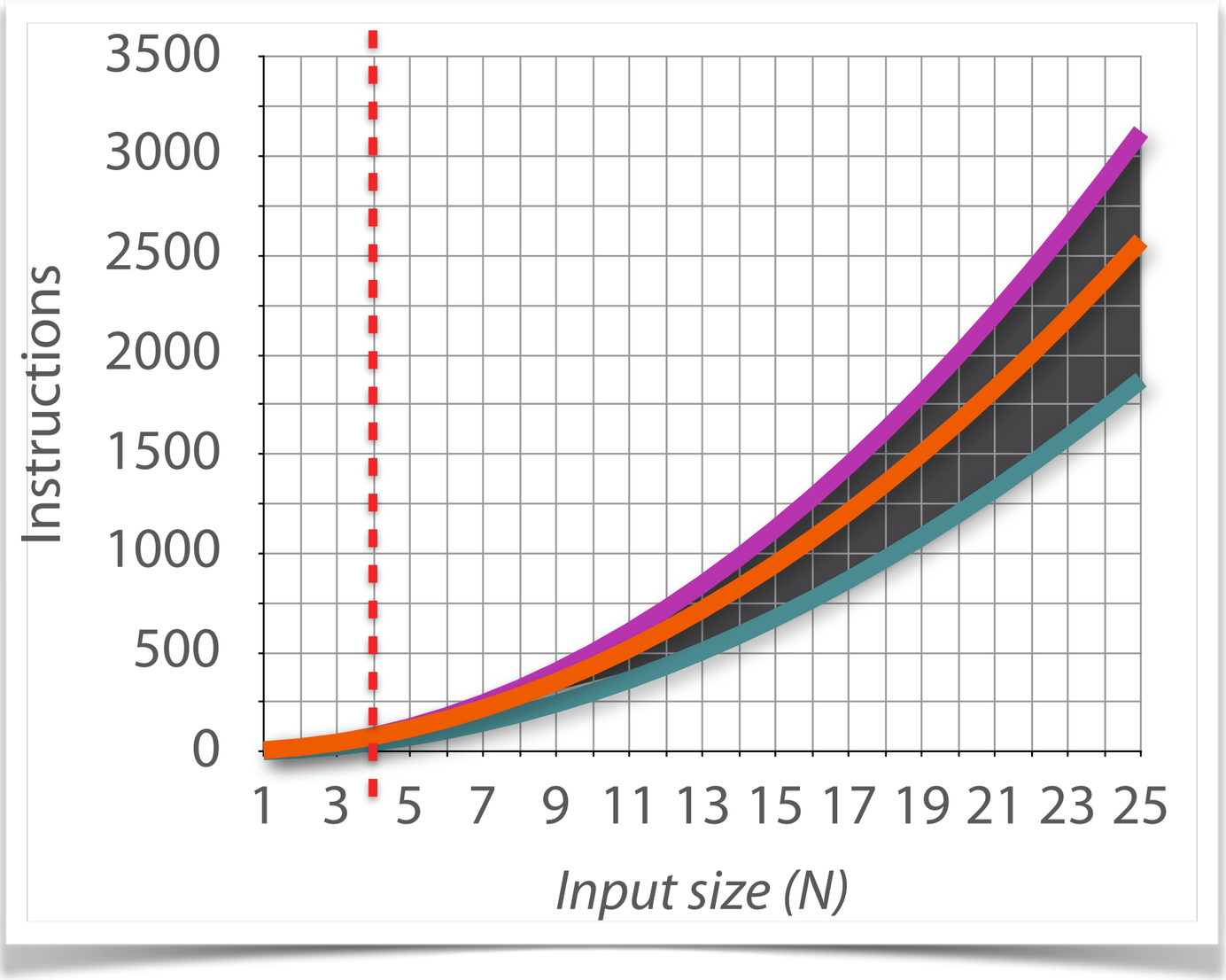
$g(N) = N^2$

$3g(N) = 3N^2$

$3g(N) < f(N) < 5g(N)$

when $N > 3$

$\Theta(g(N))$



N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

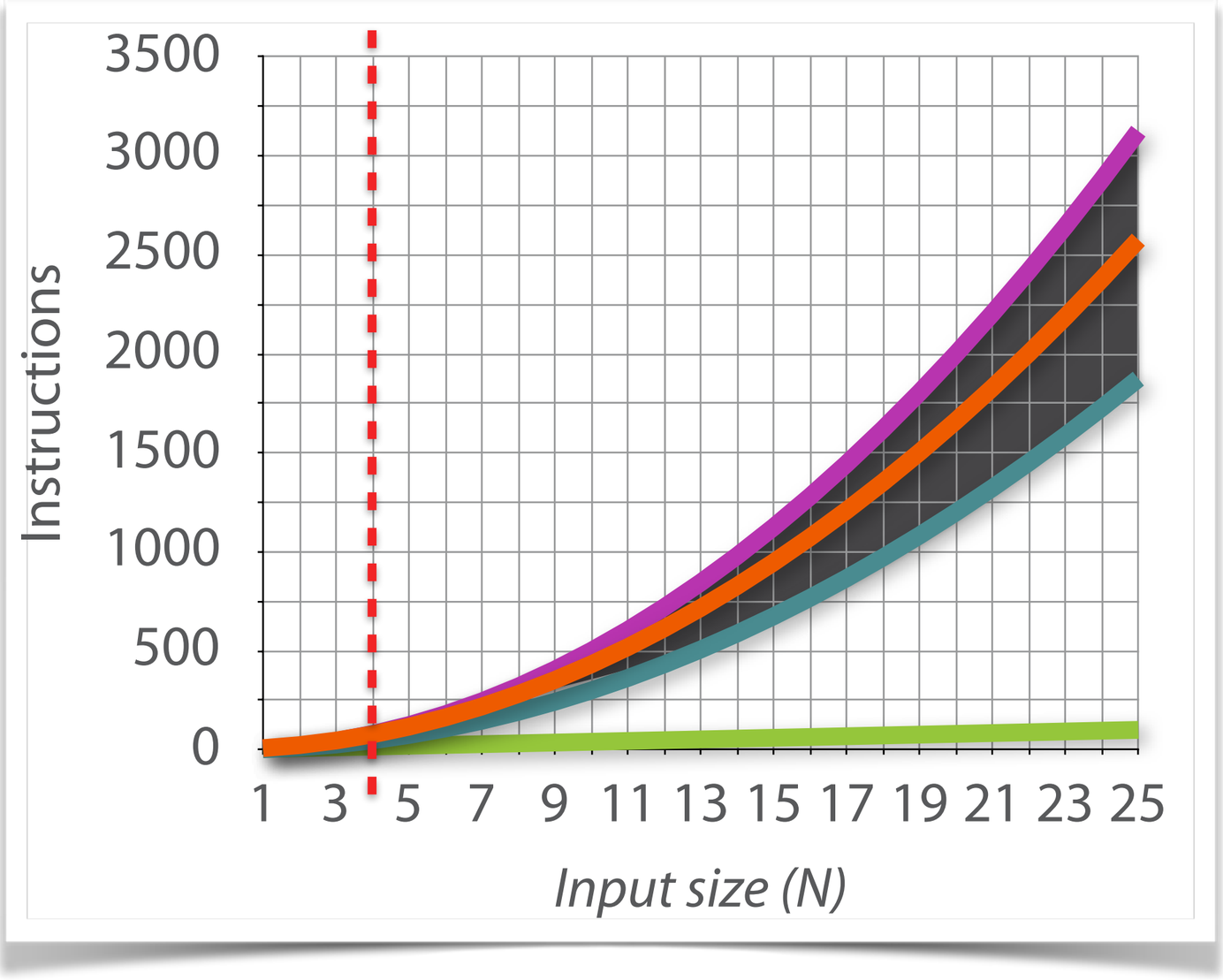
$3g(N) = 3N^2$

$3g(N) < f(N) < 5g(N)$

when $N > 3$

$\Theta(g(N))$

$\Theta(N^2)$



N	1	2	3	4
f(N)	9	24	47	78
5g(N)	5	20	45	80

$5g(N) = 5N^2$

$f(N) = 2 + 3N + 4N^2$

$g(N) = N^2$

$3g(N) = 3N^2$

$3g(N) < f(N) < 5g(N)$

when $N > 3$

$\Theta(g(N))$

$\Theta(N^2)$

Consider a function, $f(N)$…

$$f(N) = 2 + 3N + 4N^2$$

Consider a function, $f(N)$…

$f(N) = 2 + 3N + 4N^2$

If there exist:

Consider a function, $f(N)$… $f(N) = 2 + 3N + 4N^2$

If there exist:

- a function, $g(N)$ $g(N) = N^2$

Consider a function, $f(N)$…

$f(N) = 2 + 3N + 4N^2$

If there exist:

- a function, $g(N)$ $g(N) = N^2$
- two constants, say c_1 and c_2 $3g(N) < f(N) < 5g(N)$

Consider a function, $f(N)$… $f(N) = 2 + 3N + 4N^2$

If there exist:

- a function, $g(N)$ $g(N) = N^2$
- two constants, say c_1 and c_2 $3g(N) < f(N) < 5g(N)$
- another constant, n when $N > 3$

Consider a function, $f(N)$… $f(N) = 2 + 3N + 4N^2$

If there exist:

- a function, $g(N)$ $g(N) = N^2$
- two constants, say c_1 and c_2 $3g(N) < f(N) < 5g(N)$
- another constant, n when $N > 3$

so that $c_1 \cdot g(N) < f(N) < c_2 \cdot g(N)$ when $N > n$

Consider a function, $f(N)$… $f(N) = 2 + 3N + 4N^2$

If there exist:

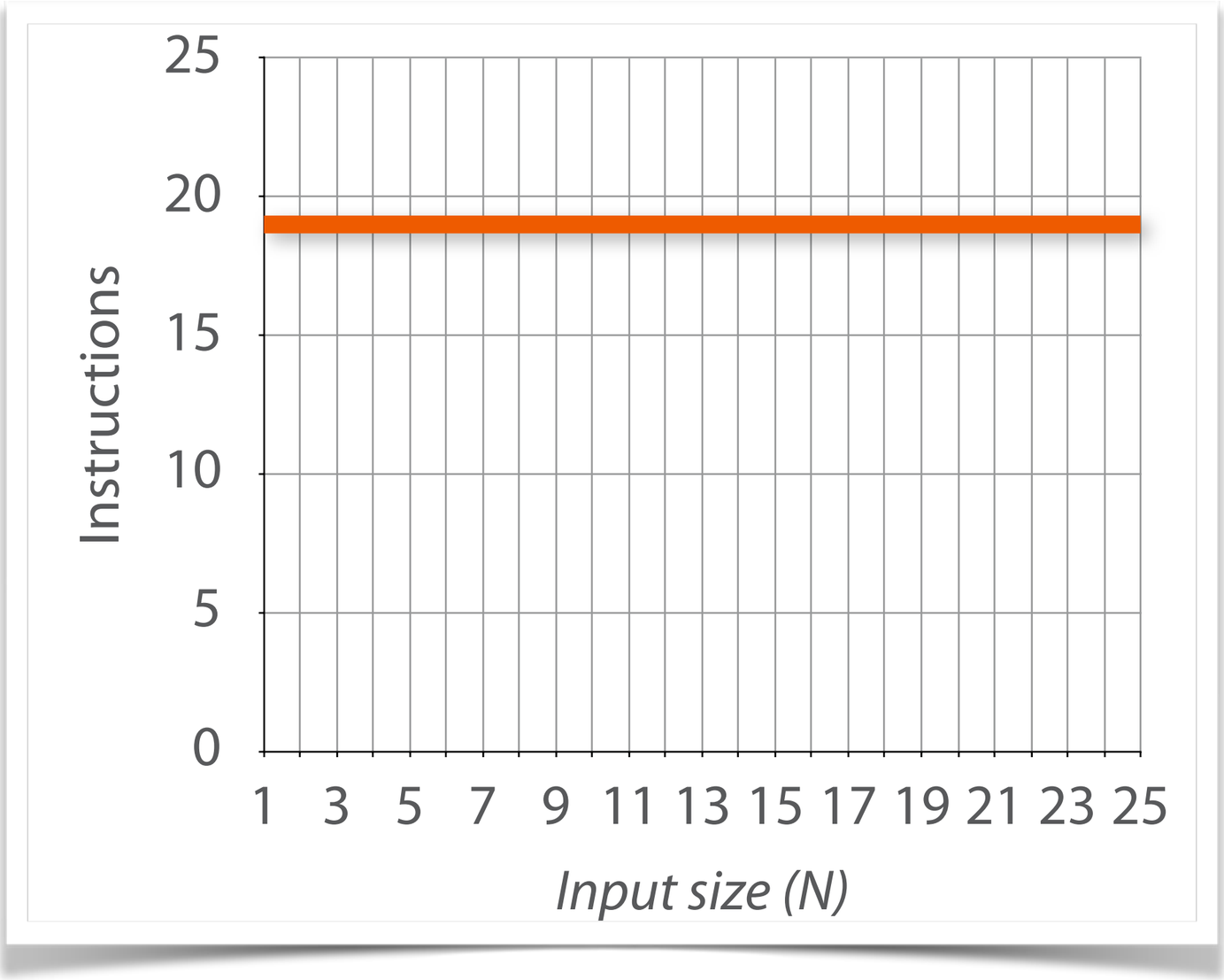
- a function, $g(N)$ $g(N) = N^2$
- two constants, say c_1 and c_2 $3g(N) < f(N) < 5g(N)$
- another constant, n when $N > 3$

so that $c_1 \cdot g(N) < f(N) < c_2 \cdot g(N)$ when $N > n$

…then $f(N)$ is $\Theta(g(N))$

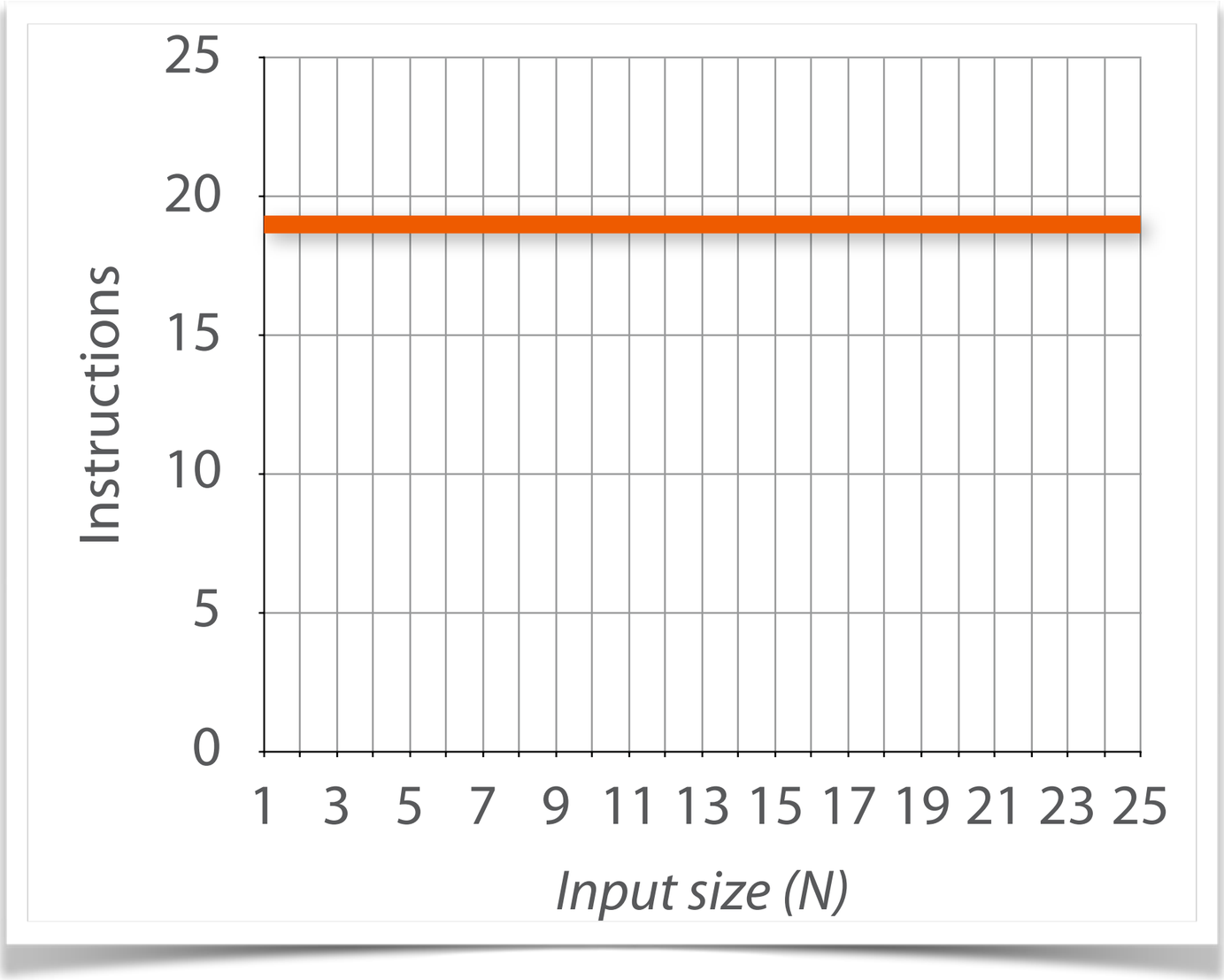
Constant Complexity

Constant Complexity



$f(N) = 19$

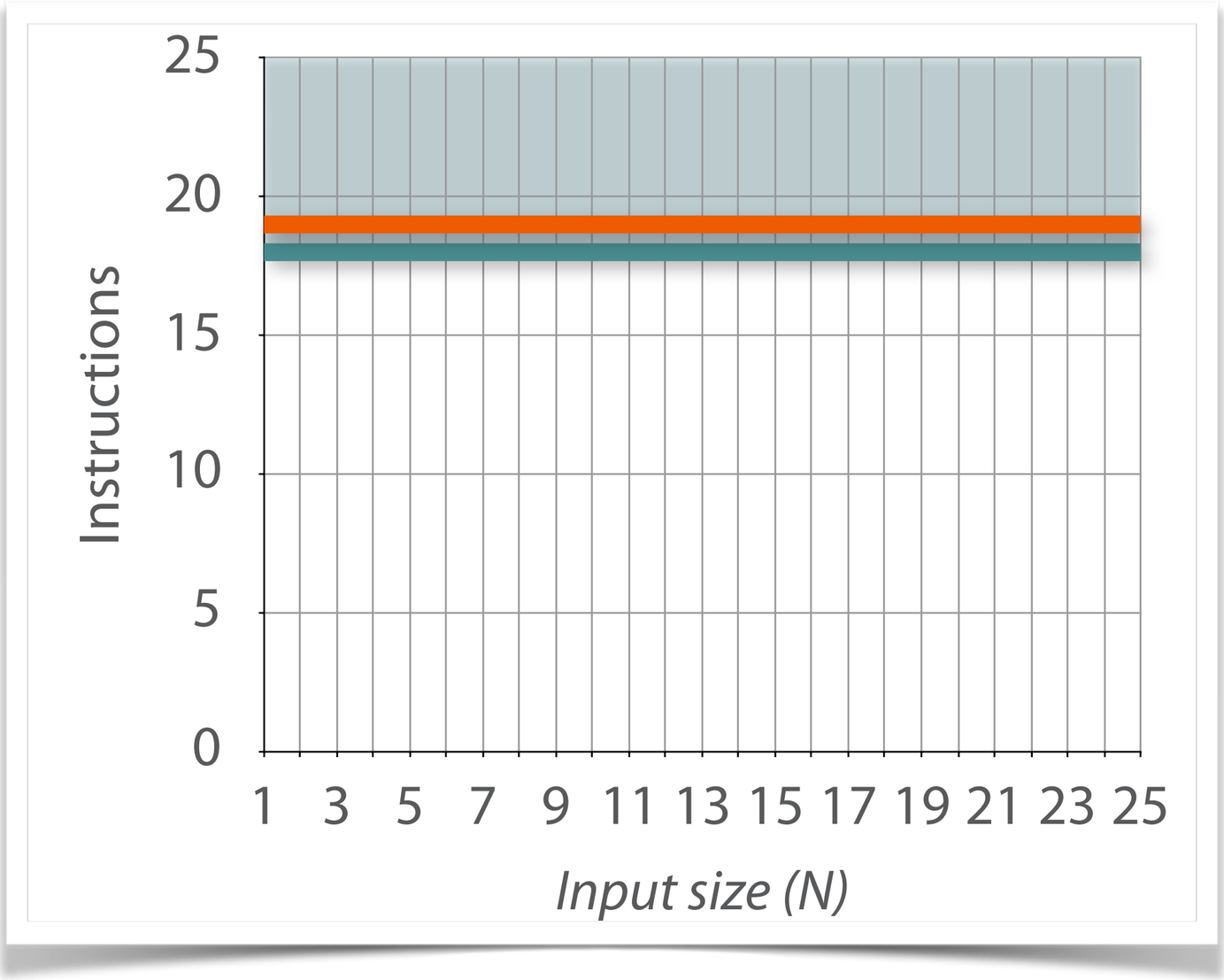
Constant Complexity



$f(N) = 19$

$g(N) = 1$

Constant Complexity

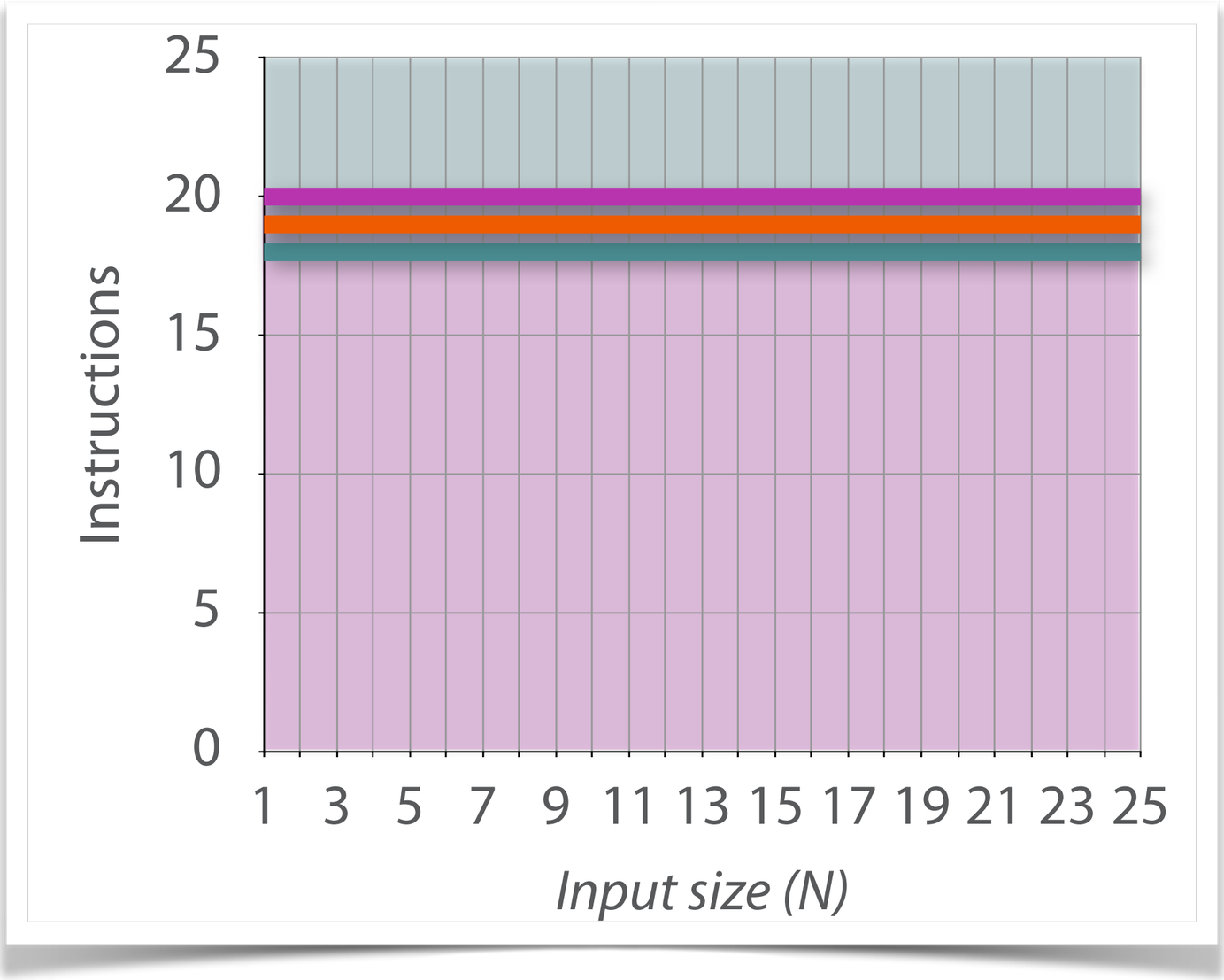


$f(N) = 19$

$g(N) = 1$

$18.999\ldots\ g(N) = 18.999\ldots \cdot 1$

Constant Complexity



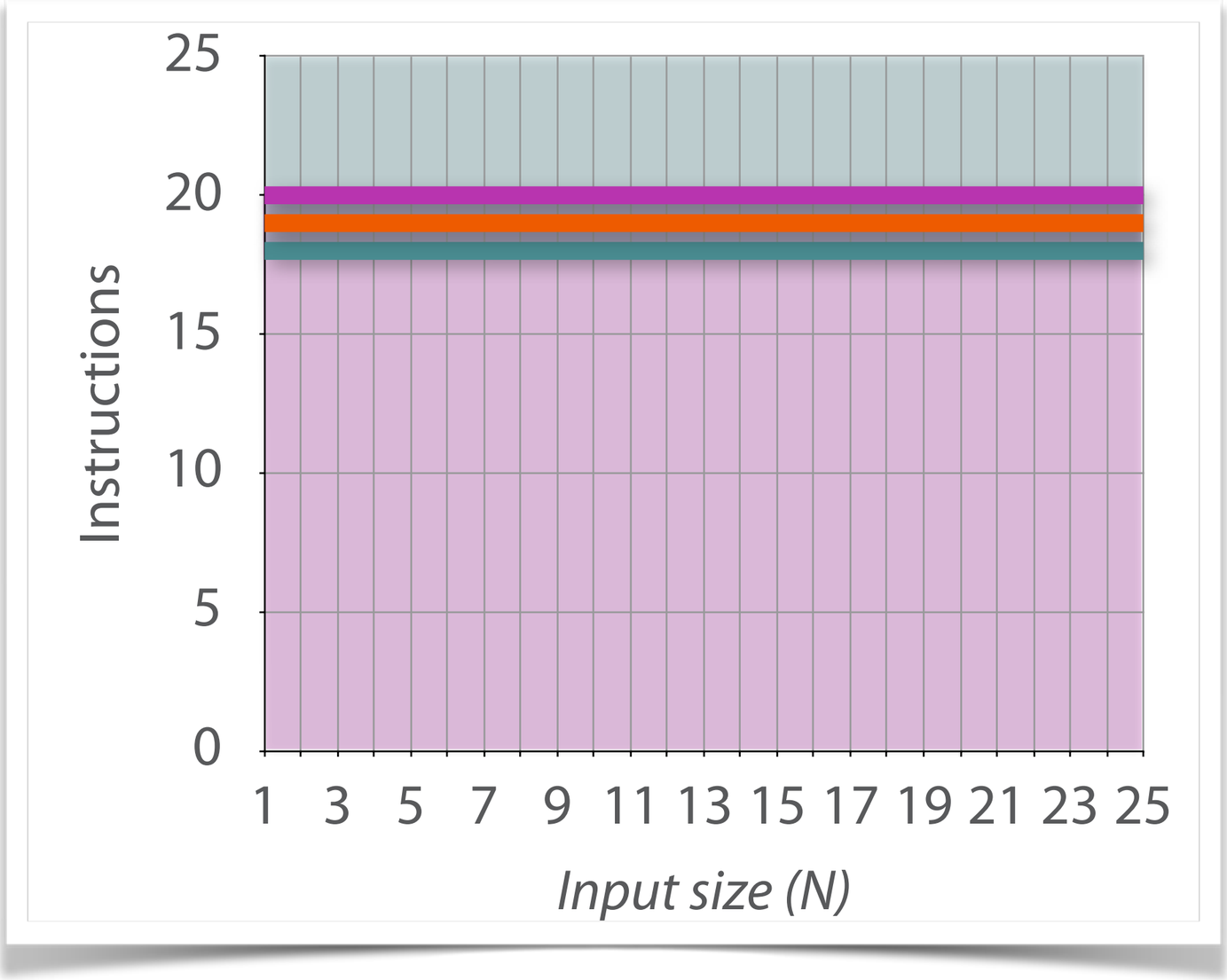
19.00…1 $g(N) = 19.00…1 \cdot 1$

$f(N) = 19$

$g(N) = 1$

18.999… $g(N) = 18.999… \cdot 1$

Constant Complexity



$19.00\ldots1\ g(N) = 19.00\ldots1 \cdot 1$

$f(N) = 19$

$g(N) = 1$

$18.999\ldots\ g(N) = 18.999\ldots \cdot 1$

$\Theta(1)$

Is $\ominus$ Cheating?

Is Θ Cheating?

$\Theta(N)$

Is Θ Cheating?

$\Theta(N)$ $\Theta(N^2)$

Is Θ Cheating?

$\Theta(N)$

$c_1 = 900$
$c_2 = 903$

$\Theta(N^2)$

$c_1 = 1$
$c_2 = 3$

Is Θ Cheating?

$\Theta(N)$

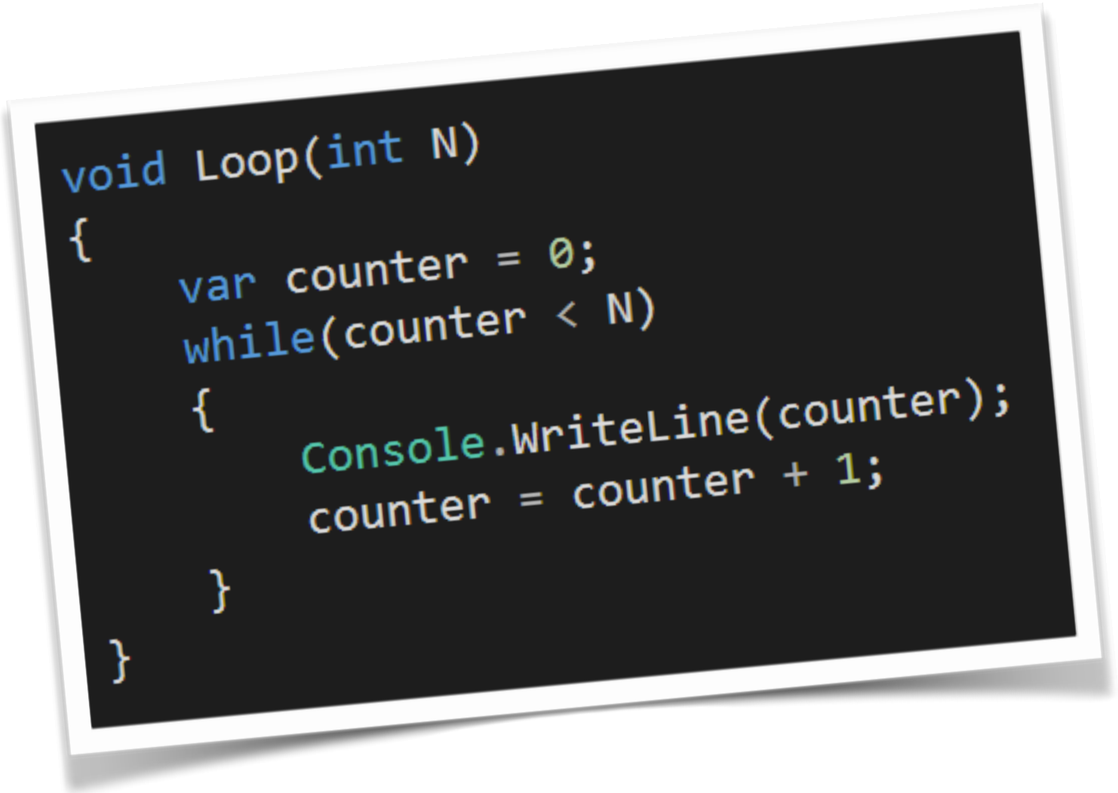
$c_1 = 900$
$c_2 = 903$

$\Theta(N^2)$

$c_1 = 1$
$c_2 = 3$

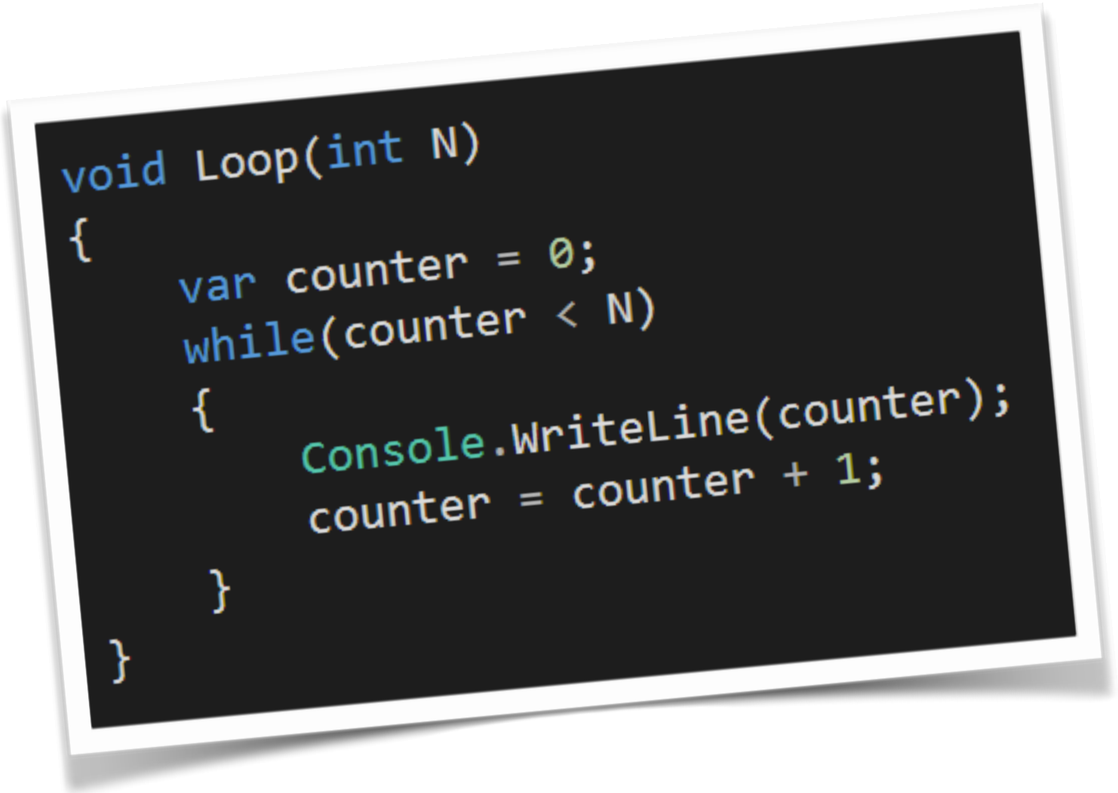
$\Theta(N^2)$ wins until $N > 900$

Big *O*


void Loop(int N)
{
var counter = 0;
while(counter < N)
{
Console.WriteLine(counter);
counter = counter + 1;
}
}

Big *O*

Consistent



```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;

    }
}
```

Big *O*

Consistent

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

Big *O*

Consistent

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

Depends on input content

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

Big *O*

Consistent

Depends on input content

```
void Loop(int N)
{
    var counter = 0;
    while(counter < N)
    {
        Console.WriteLine(counter);
        counter = counter + 1;
    }
}
```

```
bool Contain
t needle, List<int> haystack)
e in haystack)
eedle)
return true;
}
return false;
}
```

Worst-case is interesting

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

At most N.

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

At most $N \cdot 1$

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

At most $N \cdot 1 + 1$

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

At most $N \cdot 1 + 1$

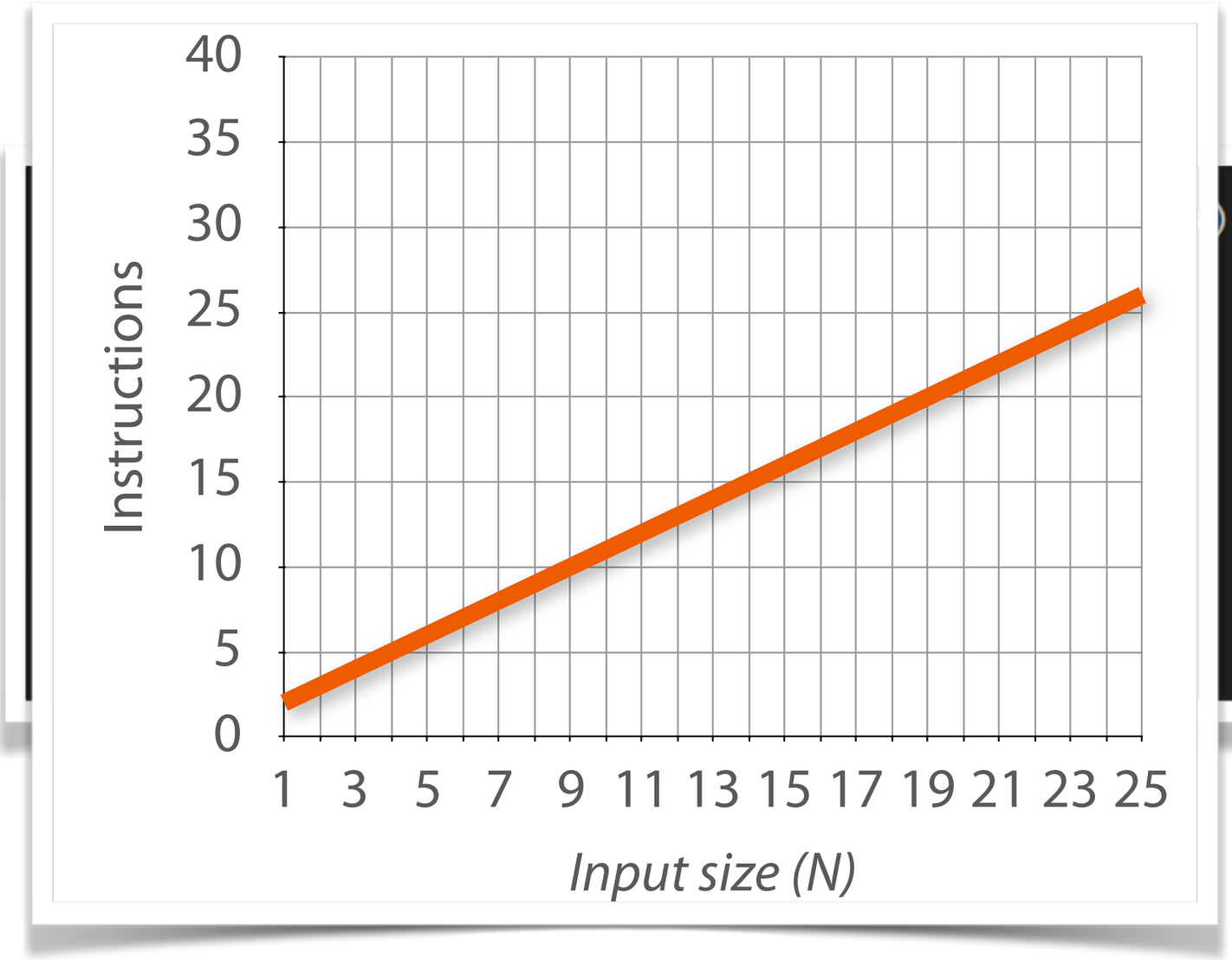
$= N + 1$

Complexity (*N*):

```
bool ContainsNeedle(int needle, List<int> haystack)
{
    foreach (var sample in haystack)
    {
        if (sample == needle)
            return true;
    }
    return false;
}
```

At most $N \cdot 1 + 1$

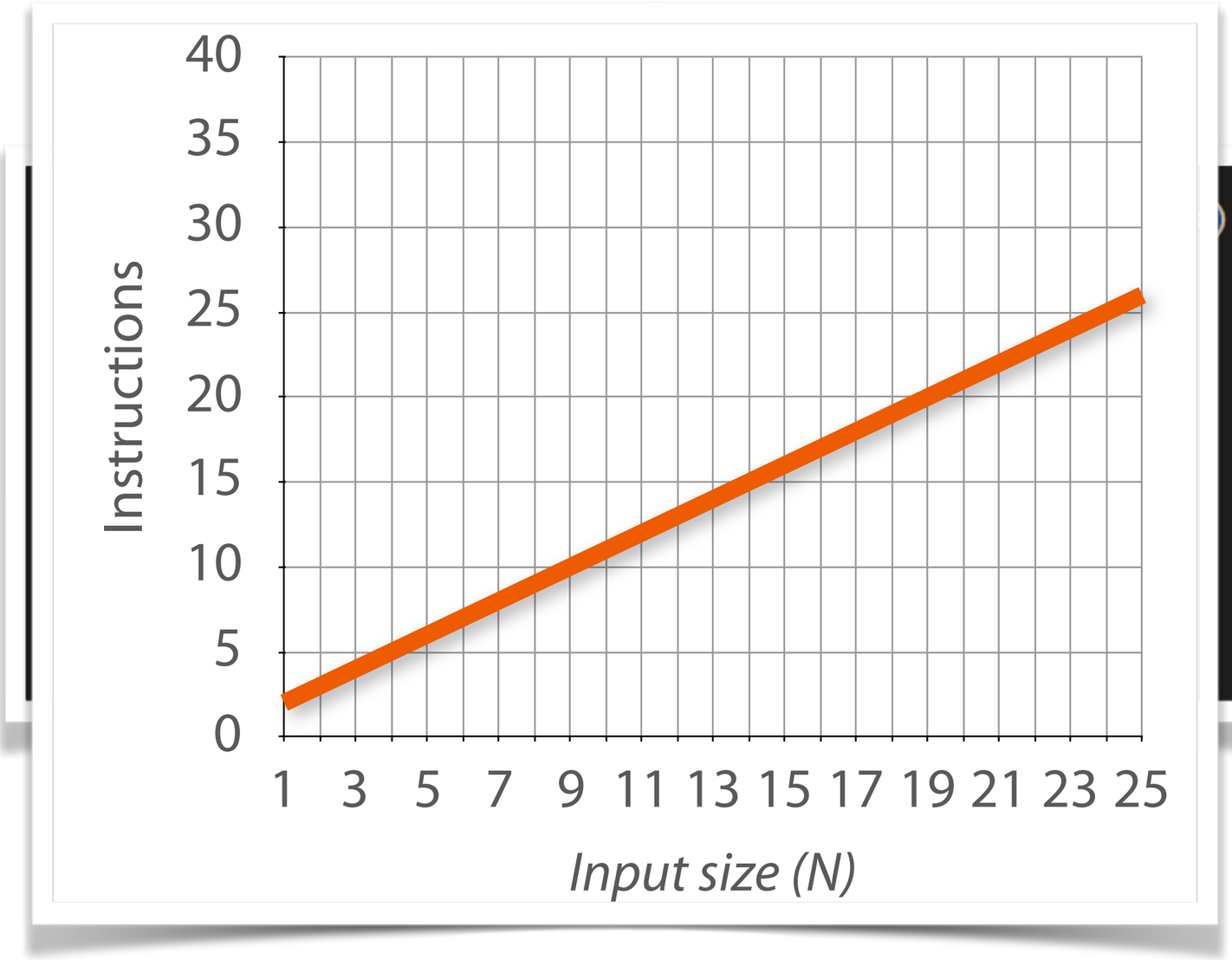
$f(N) = N + 1$



Complexity (*N*):

At most $N \cdot 1 + 1$

$f(N) = N + 1$

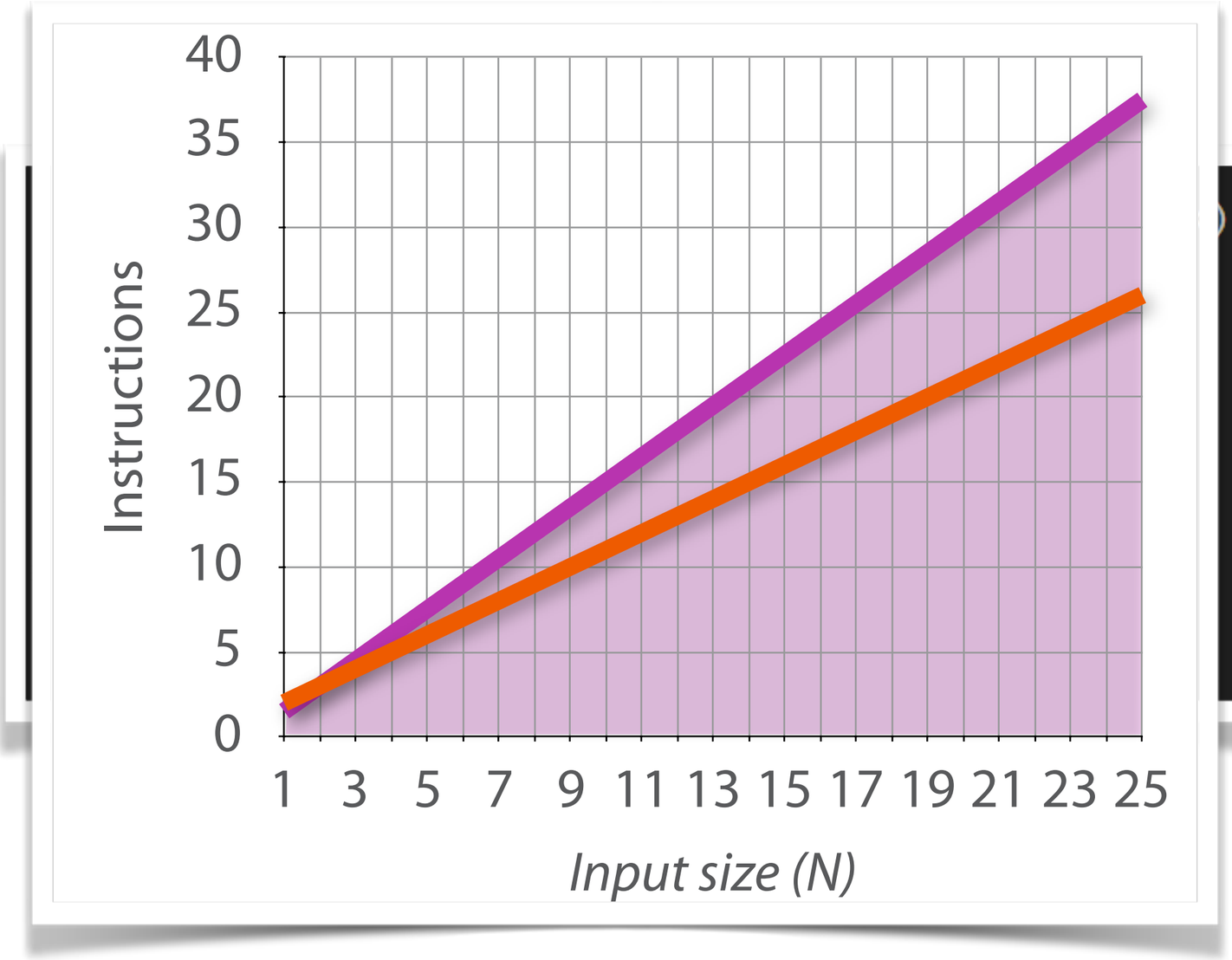


Complexity (*N*):

At most $N \cdot 1 + 1$

$f(N) = N + 1$

$g(N) = N$



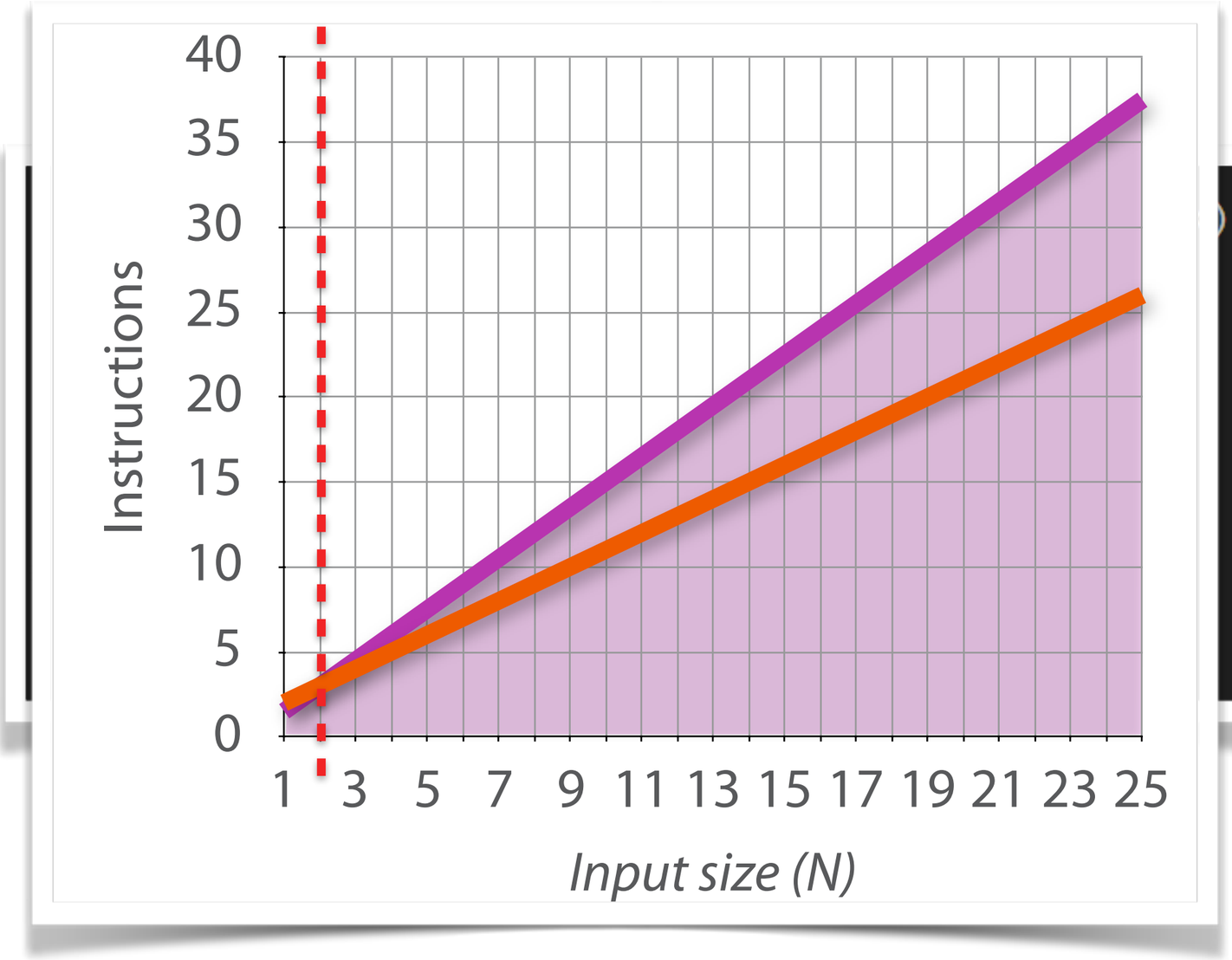
Complexity (*N*):

1.5 $g(N) = 1.5\ N$

At most $N \cdot 1 + 1$

$f(N) = N + 1$

$g(N) = N$



Complexity (*N*):

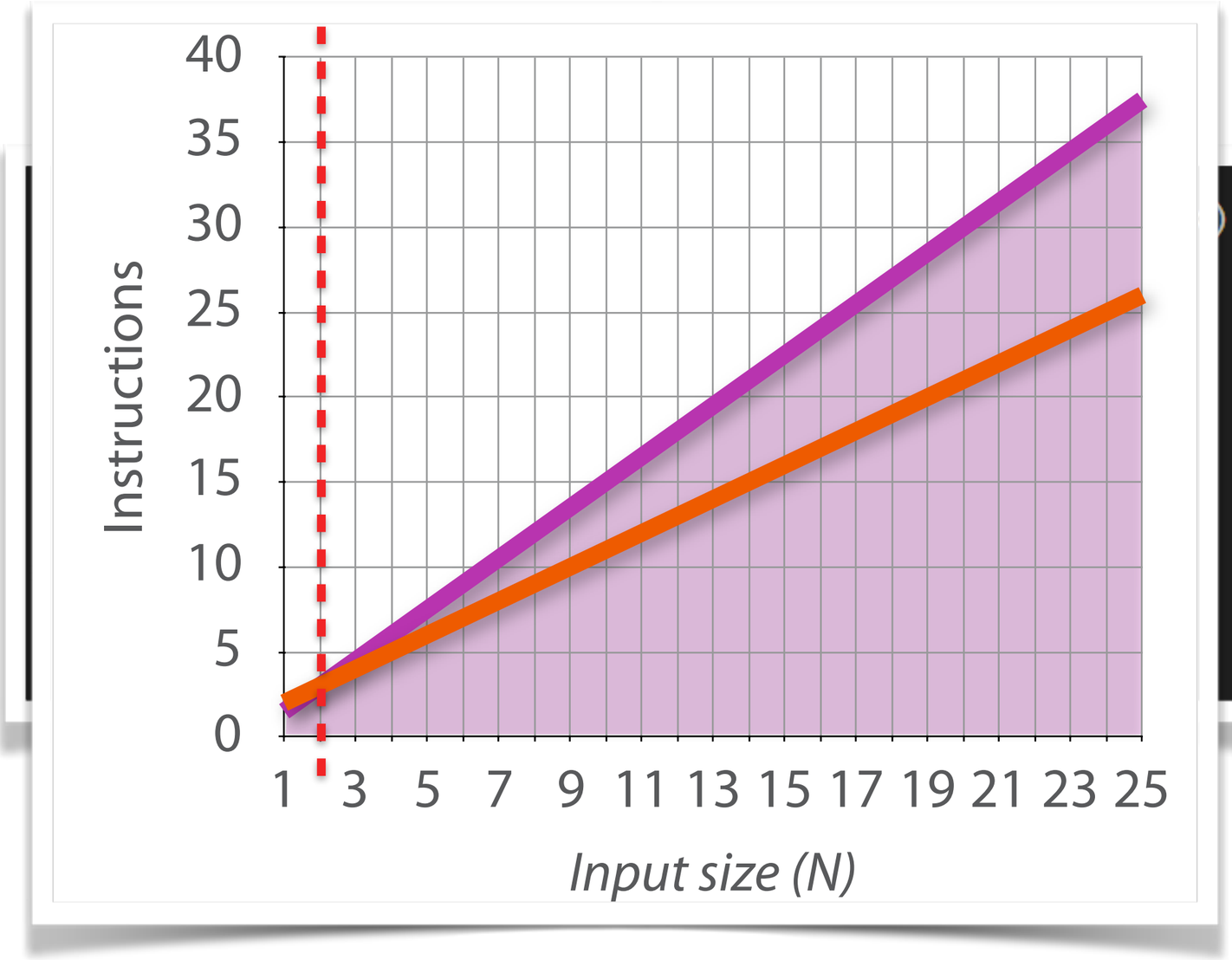
$1.5\ g(N) = 1.5\ N$

At most $N \cdot 1 + 1$

$f(N) = N + 1$

$g(N) = N$

$f(N) < 1.5\ g(N)$
when $N > 1$



Complexity (*N*):

$1.5\ g(N) = 1.5\ N$

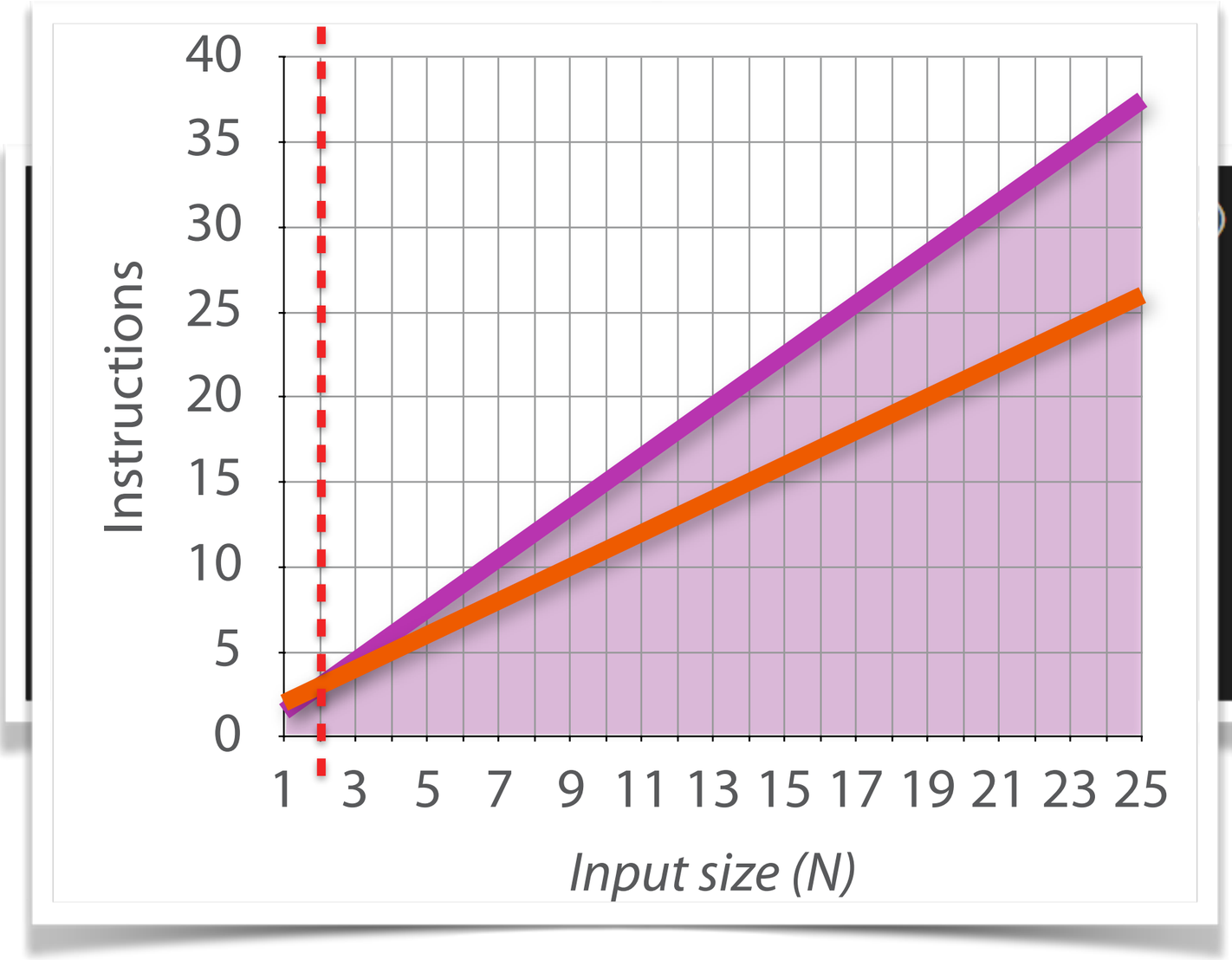
At most $N \cdot 1 + 1$

$f(N) = N + 1$

$g(N) = N$

$f(N) < 1.5\ g(N)$
when $N > 1$

$O(g(N))$



Complexity (*N*):

$1.5\ g(N) = 1.5\ N$

At most $N \cdot 1 + 1$

$f(N) = N + 1$

$g(N) = N$

$f(N) < 1.5\ g(N)$
when $N > 1$

$O(g(N))$

$O(N)$

Consider a function, $f(N)$…

$f(N) = N + 1$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

Consider a function, $f(N)$… $f(N) = N + 1$

If there exist:

- a function, $g(N)$ $g(N) = N$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$
- a constant, say c

$g(N) = N$

$f(N) < 1.5\ g(N)$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) < 1.5\ g(N)$
- another constant, n — when $N > 1$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) < 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) < 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Binary Search

Binary Search

1	*3*	*3*	*7*	*10*	*11*	*16*	*17*	*20*	*21*	*25*	*26*	*30*	*32*	*34*	*35*

Binary Search

Needle: 26

1	***3***	***3***	***7***	***10***	***11***	***16***	***17***	***20***	***21***	***25***	***26***	***30***	***32***	***34***	***35***

Binary Search

Needle: 26

1	***3***	***3***	***7***	***10***	***11***	***16***	***17***	***20***	***21***	***25***	***26***	***30***	***32***	***34***	***35***

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

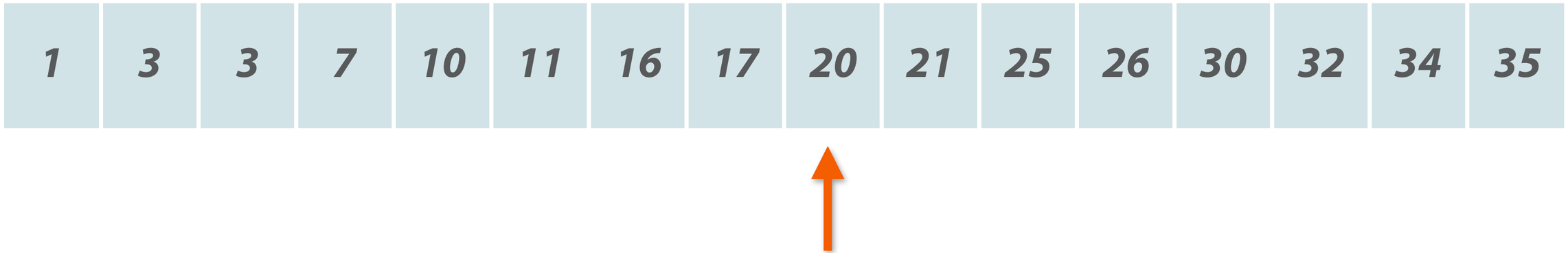
1	***3***	***3***	***7***	***10***	***11***	***16***	***17***	***20***	***21***	***25***	***26***	***30***	***32***	***34***	***35***

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack



Binary Search

Needle: 26

Linear search: *O*(*N*)

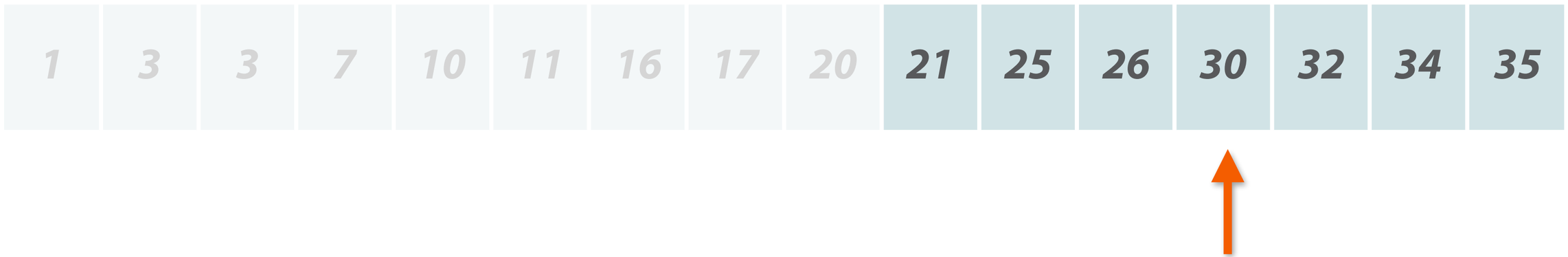
N entries in haystack

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

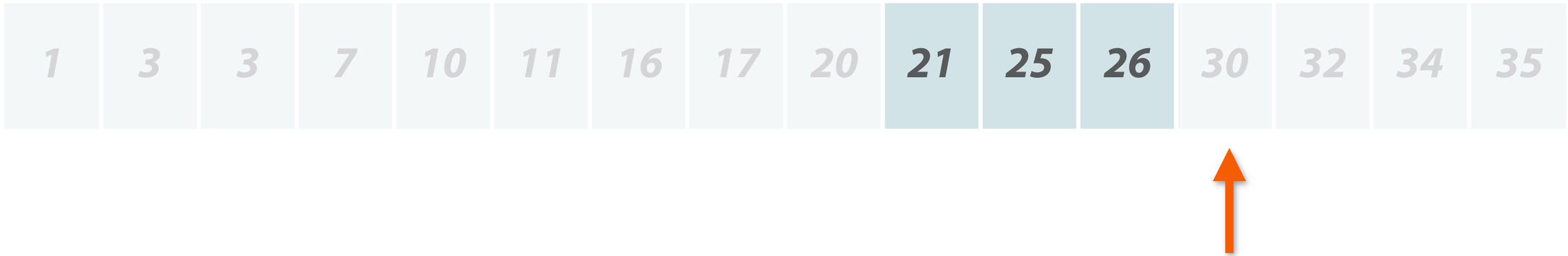


Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

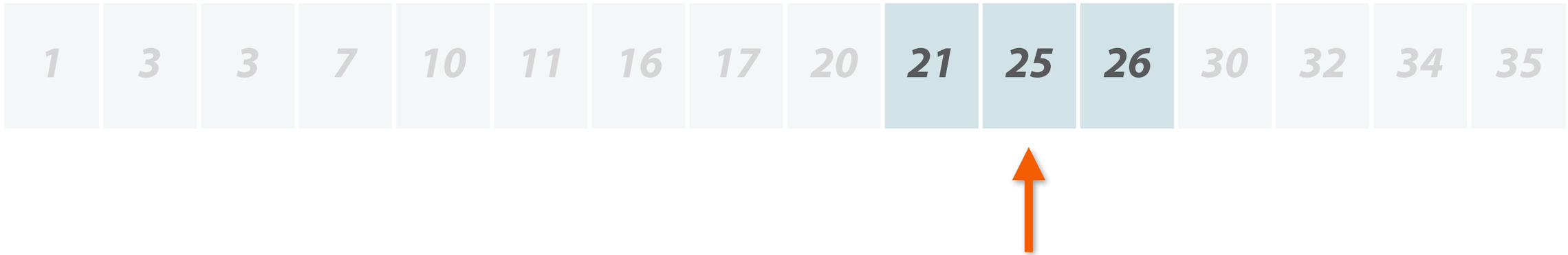


Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

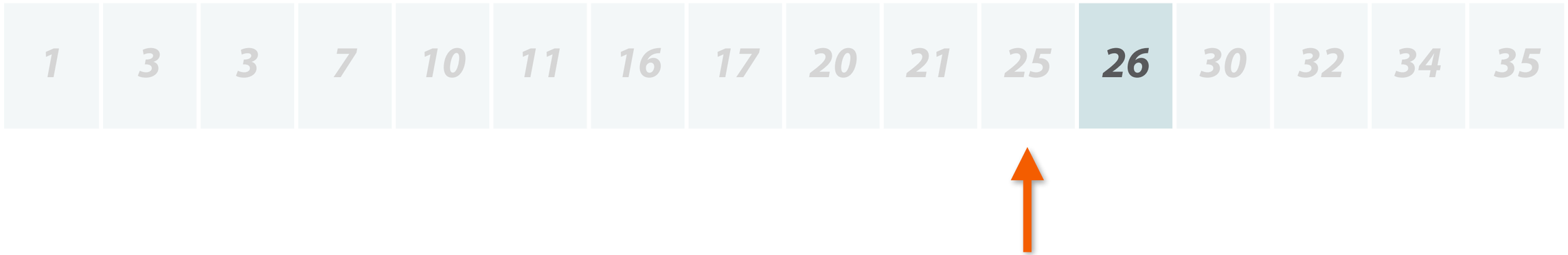


Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

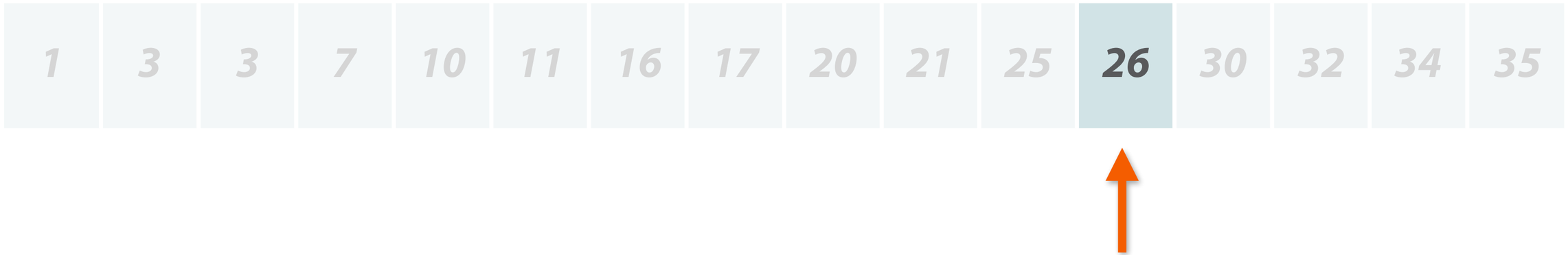


Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack



Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

1	3	3	7	10	11	16	17	20	21	25	**26**	30	32	34	35

Number of comparisons =

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

Number of comparisons = Number of halvings of *N*

Logarithms

Logarithms

$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

Logarithms

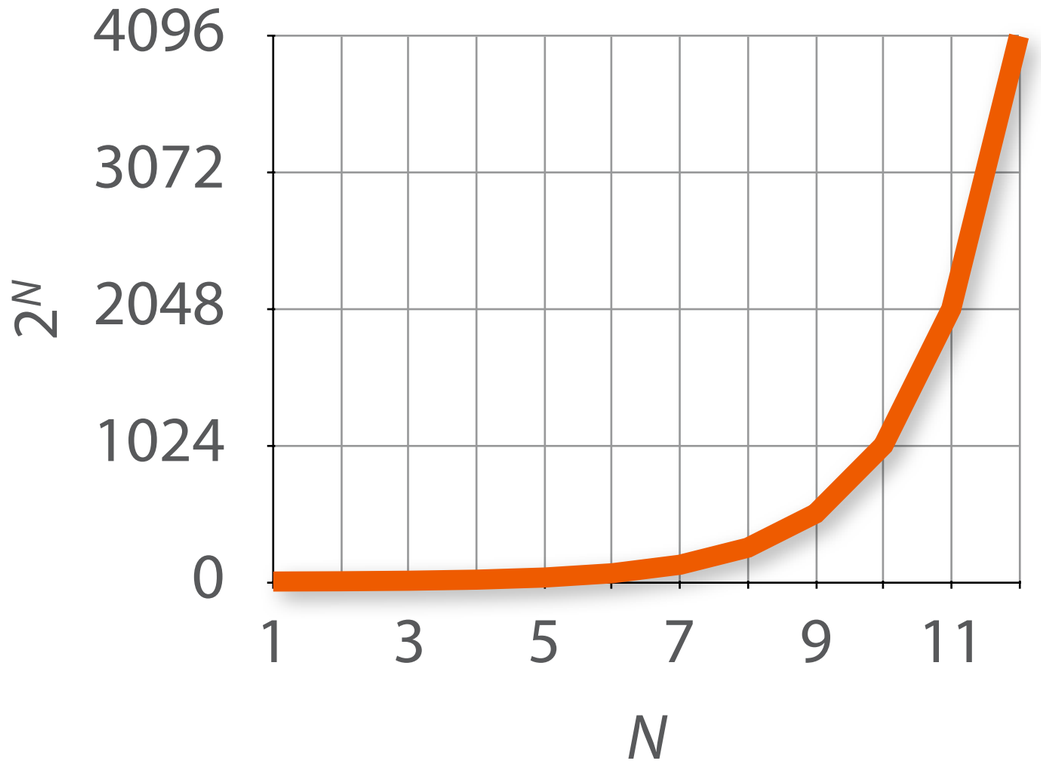
$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$
$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$
$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N$$

Exponential function

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$
$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$
$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$



2^N

Exponential function

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N = 4096$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N = 4096$$

What should N be?

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N = 4096$$

$$\log_2 4096$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N = 4096$$

$$\log_2 4096$$

$$\log_2 4096 = 12$$

$$2^{12} = 4096$$

Logarithms

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$$

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$$

$$2^N = 4096$$

$$\log_2 4096$$

$$\log_2 4096 = 12$$

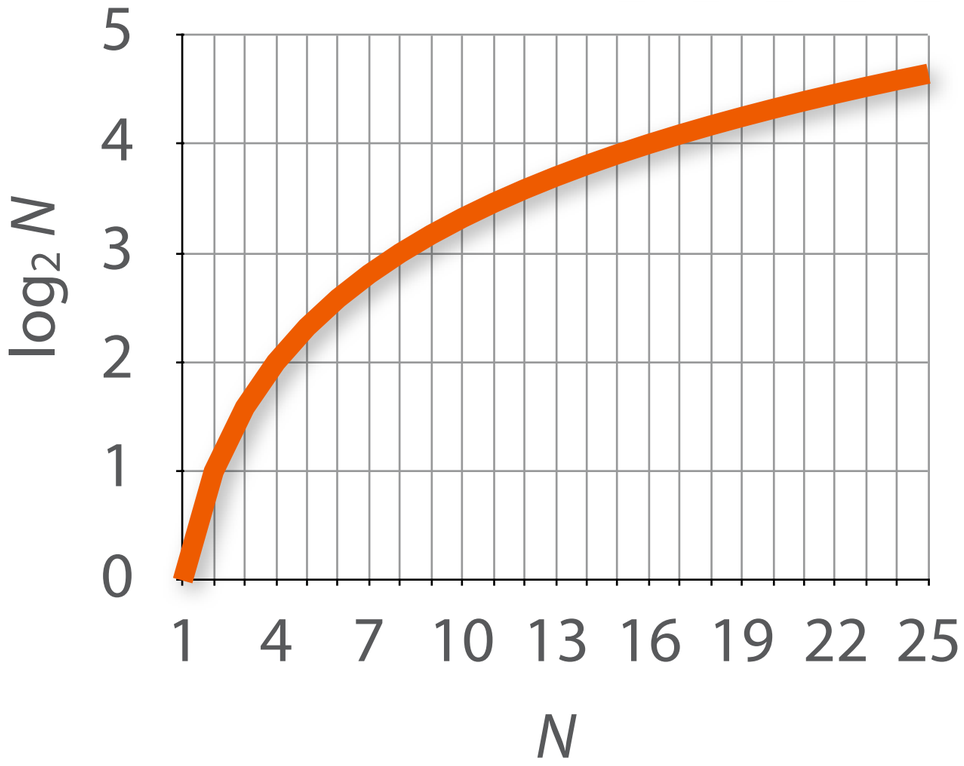
$$2^{12} = 4096$$

Logarithms

$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7 = 128$

$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$

$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9 = 512$



$2^N = 4096$

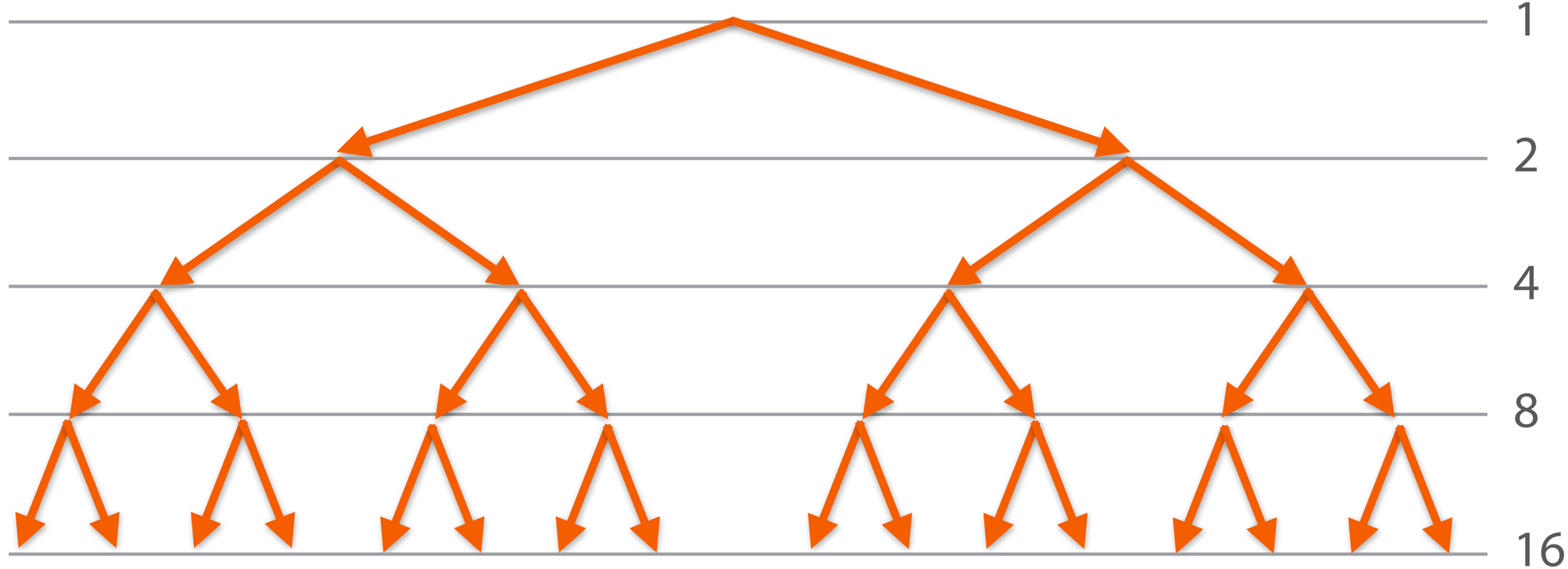
$\log_2 4096$

$\log_2 4096 = 12$

$2^{12} = 4096$

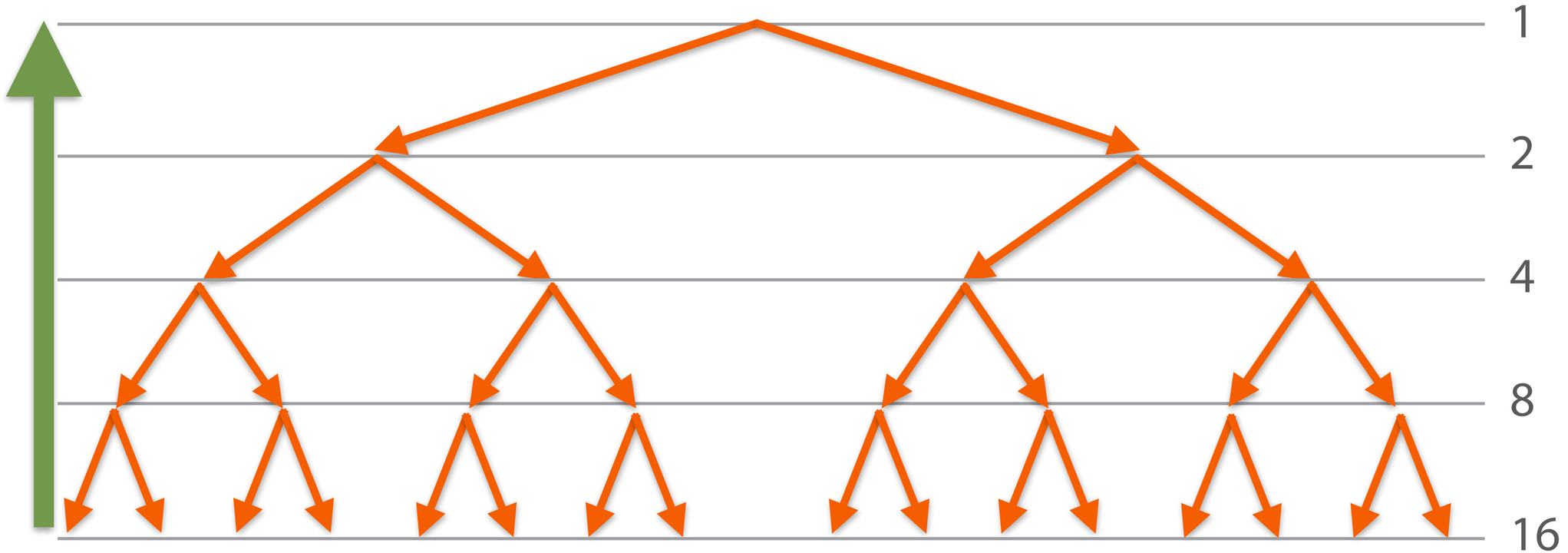
Logarithms

Logarithms



Total number after 4 doublings: exponential function ($2^4 = 16$)

Logarithms



Total number after 4 doublings: exponential function ($2^4 = 16$)

Total number of *halvings* of 16: $\log_2 16 = 4$

Binary Search

Needle: 26

Linear search: *O*(*N*)

N entries in haystack

Number of comparisons = Number of halvings of *N*

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

Number of comparisons = Number of halvings of N = $\log_2 N$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

Number of comparisons = Number of halvings of N = $\log_2 N$

$\log_2 N$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

Number of comparisons = Number of halvings of N = $\log_2 N$

$\log_2 N$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

1	3	3	7	10	11	16	17	20	21	25	**26**	30	32	34	35

Number of comparisons = Number of halvings of N $= \log_2 N$

$$c \cdot \log_2 N$$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

Number of comparisons = Number of halvings of N = $\log_2 N$

Complexity: $O(\log_2 N)$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

1	3	3	7	10	11	16	17	20	21	25	**26**	30	32	34	35

Number of comparisons = Number of halvings of

10,000 entries: 14 comparisons

Complexity: $O(\log_2 N)$

Binary Search

Needle: 26

Linear search: $O(N)$

N entries in haystack

1	3	3	7	10	11	16	17	20	21	25	**26**	30	32	34	35

Number of comparisons = Number of halvings of

Complexity: $O(\log_2 N)$

10,000 entries: 14 comparisons

1,000,000 entries: 20 comparisons

A Few Points About Big *O*

A Few Points About Big *O*

$O(999^N)$ rules almost everything

A Few Points About Big *O*

$O(999^N)$ rules almost everything

A Few Points About Big *O*

$O(999^N)$ rules almost everything

A Few Points About Big *O*

$O(999^N)$ rules almost everything

Find the *lowest possible* worst-case

A Few Points About Big *O*

$O(999^N)$ rules almost everything

Find the *lowest possible* worst-case

$N^2 + N$ is $O(N^3)$

A Few Points About Big *O*

$O(999^N)$ rules almost everything

Find the *lowest possible* worst-case

$N^2 + N$ is $O(N^3)$

$O(N^2)$ is more interesting

A Few Points About Big *O*

$O(999^N)$ rules almost everything

Find the *lowest possible* worst-case

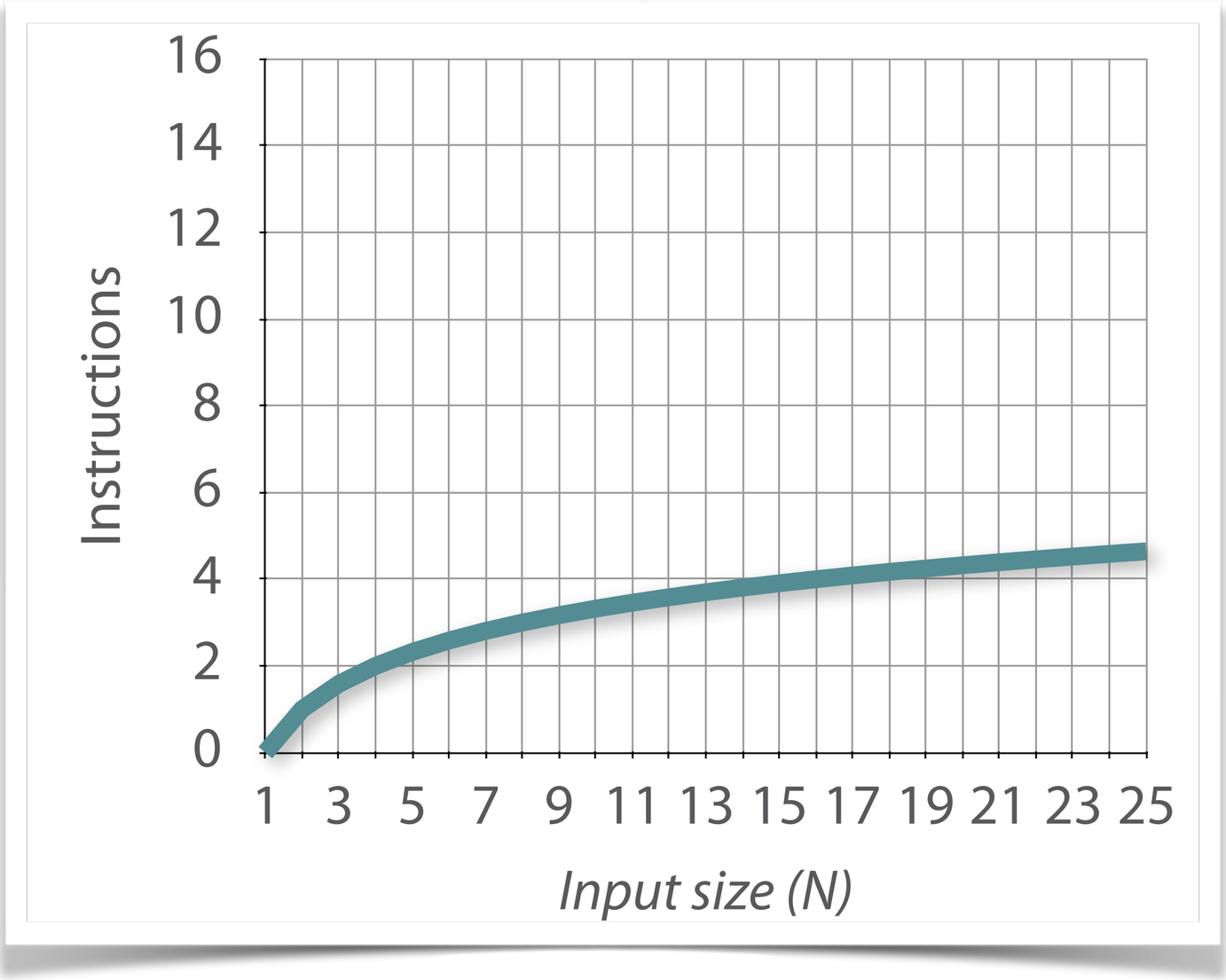
Tight bound

$N^2 + N$ is $O(N^3)$

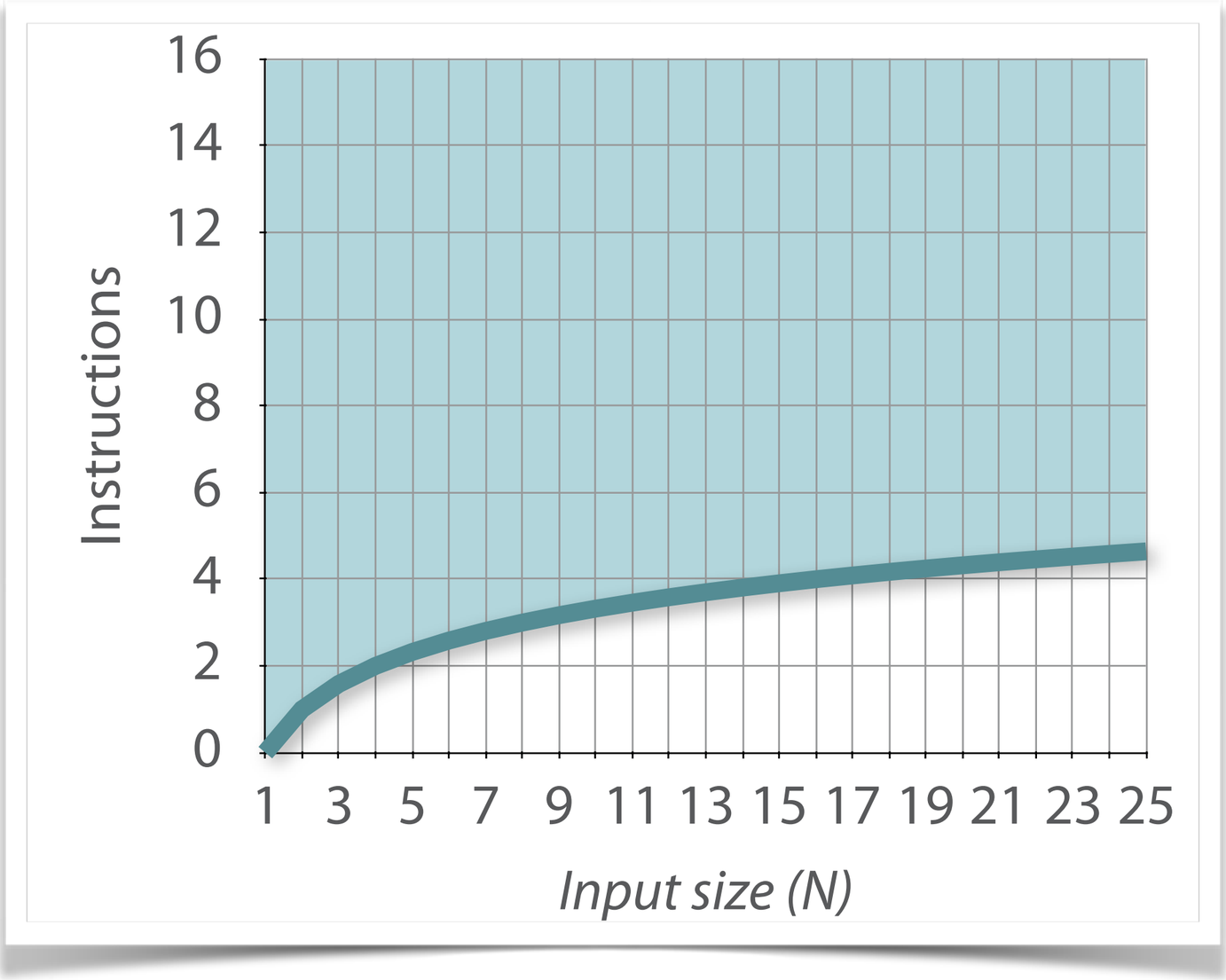
$O(N^2)$ is more interesting

Big Omega (Ω)

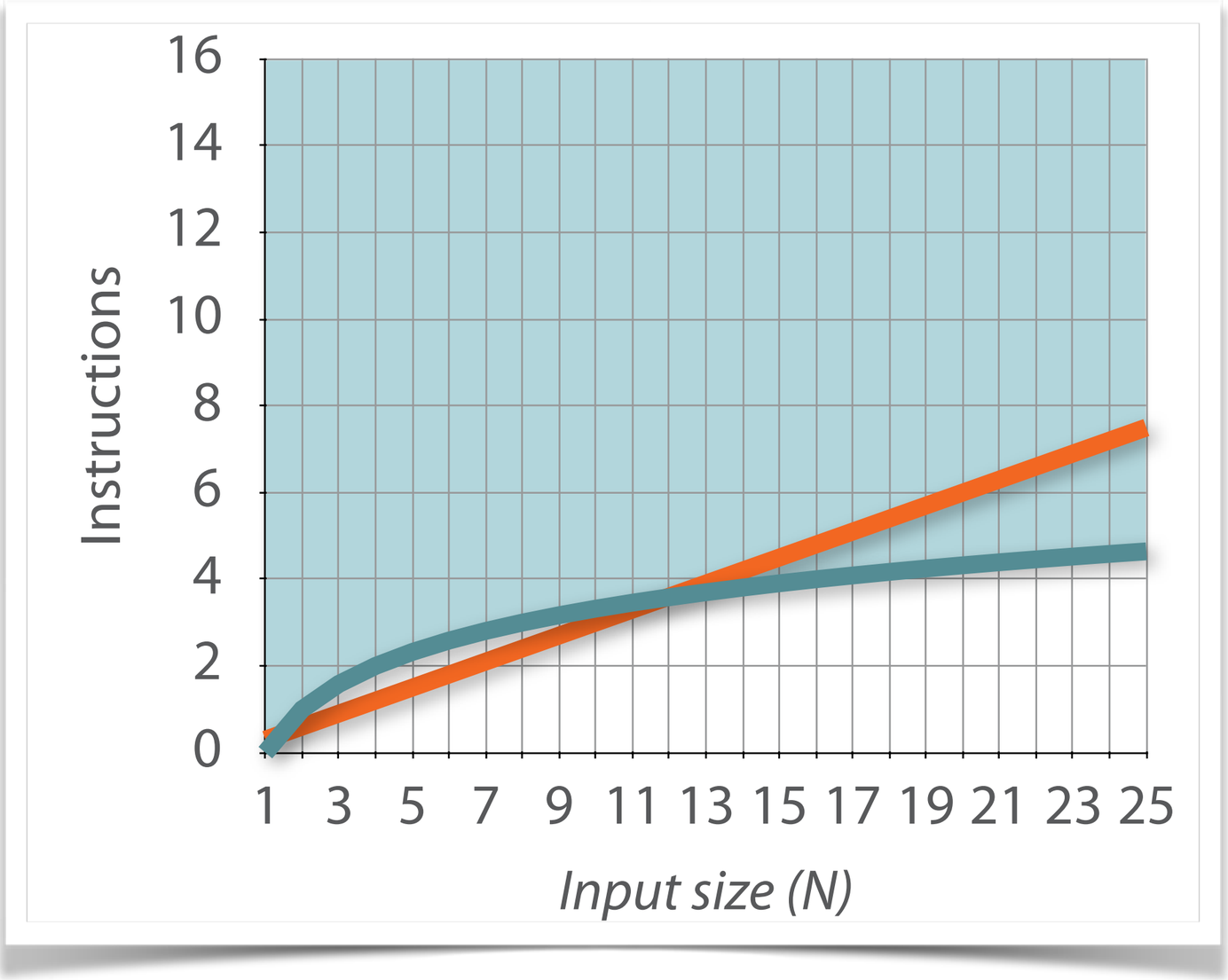
Big Omega (Ω)


16
14
12
10
8
6
4
2
0
Instructions
1 3 5 7 9 11 13 15 17 19 21 23 25
Input size (N)

Big Omega (Ω)


16
14
12
10
8
6
4
2
0
Instructions
1 3 5 7 9 11 13 15 17 19 21 23 25
Input size (N)

Big Omega (Ω)


16
14
12
10
8
6
4
2
0
Instructions
1 3 5 7 9 11 13 15 17 19 21 23 25
Input size (N)

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ $g(N) = N$
- a constant, say c $f(N) < 1.5\ g(N)$
- another constant, n when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

§

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) < 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ $g(N) = N$
- a constant, say c
- another constant, n when $N > 1$

$f(N) < 1.5\ g(N)$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c
- another constant, n — when $N > 1$

$f(N) < 1.5\ g(N)$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) < 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) > 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) > 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c
- another constant, n — when $N > 1$

$f(N) > 1.5\ g(N)$

so that $f(N) < c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c
- another constant, n — when $N > 1$

$f(N) > 1.5\ g(N)$

so that $f(N) > c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) > 1.5\ g(N)$
- another constant, n — when $N > 1$

so that $f(N) > c \cdot g(N)$ when $N > n$

…then $f(N)$ is $O(g(N))$

Consider a function, $f(N)$…

$f(N) = N + 1$

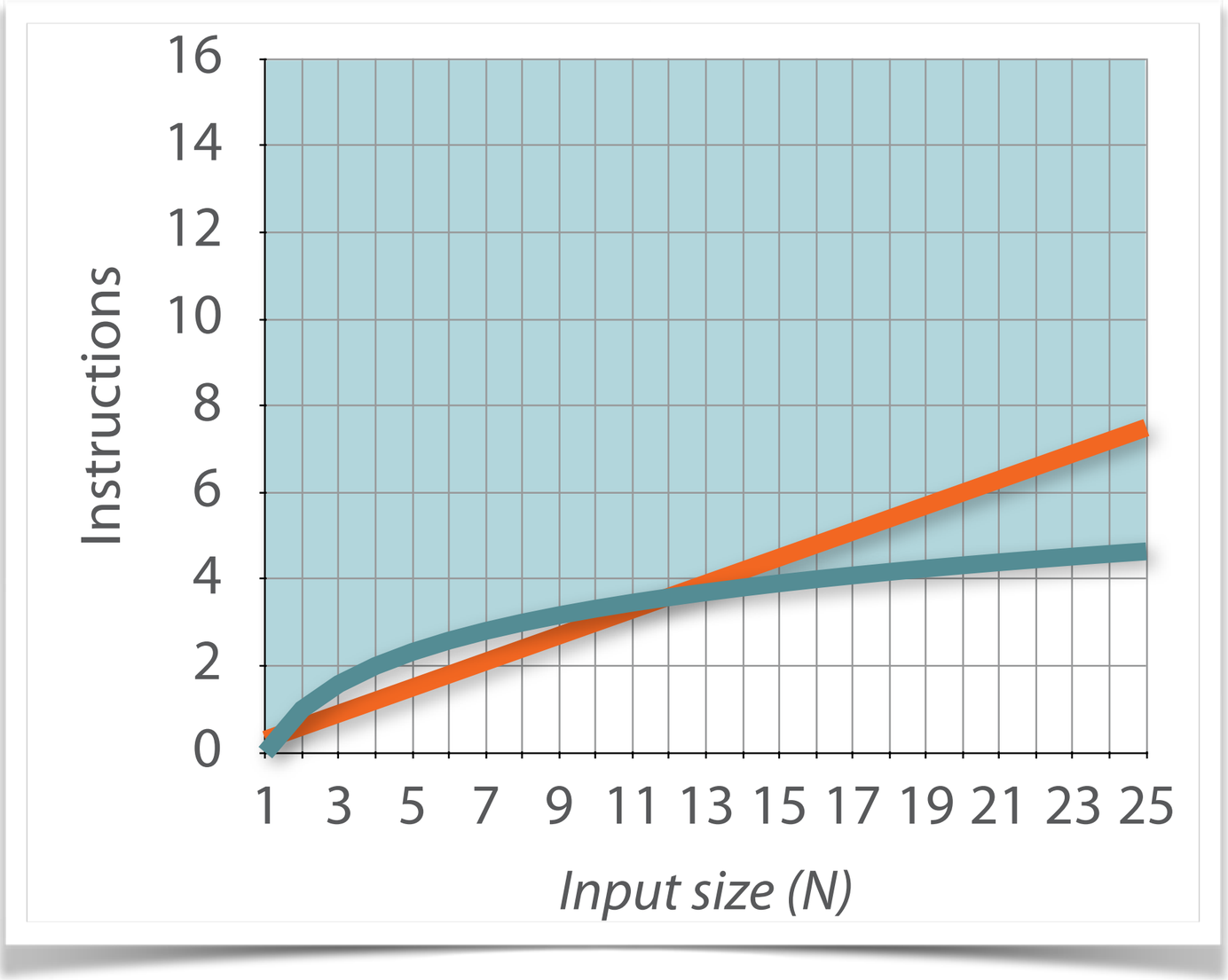
If there exist:

- a function, $g(N)$ — $g(N) = N$
- a constant, say c — $f(N) > 1.5\ g(N)$
- another constant, n — when $N > 1$

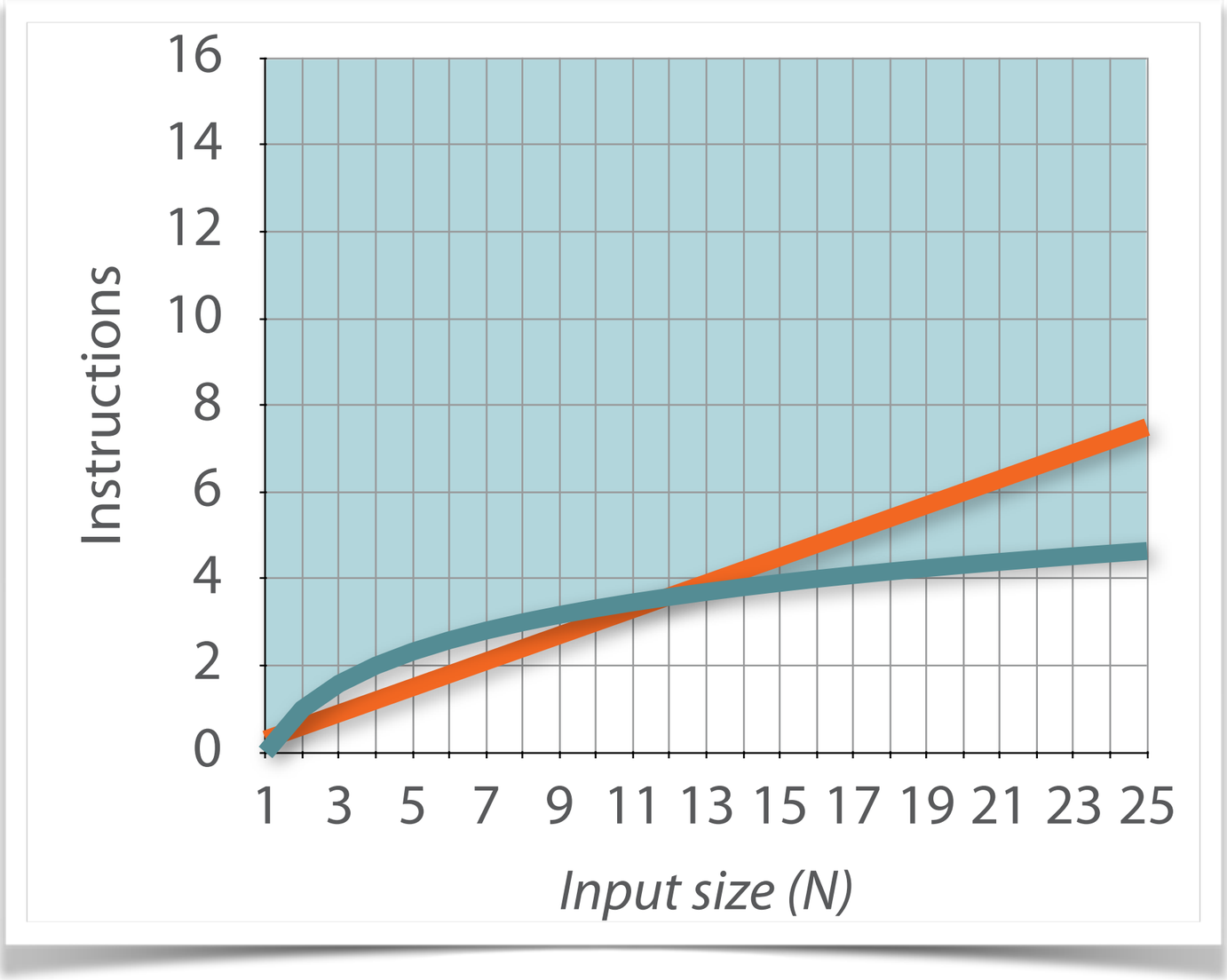
so that $f(N) > c \cdot g(N)$ when $N > n$

…then $f(N)$ is $\Omega(g(N))$

Big Omega (Ω)

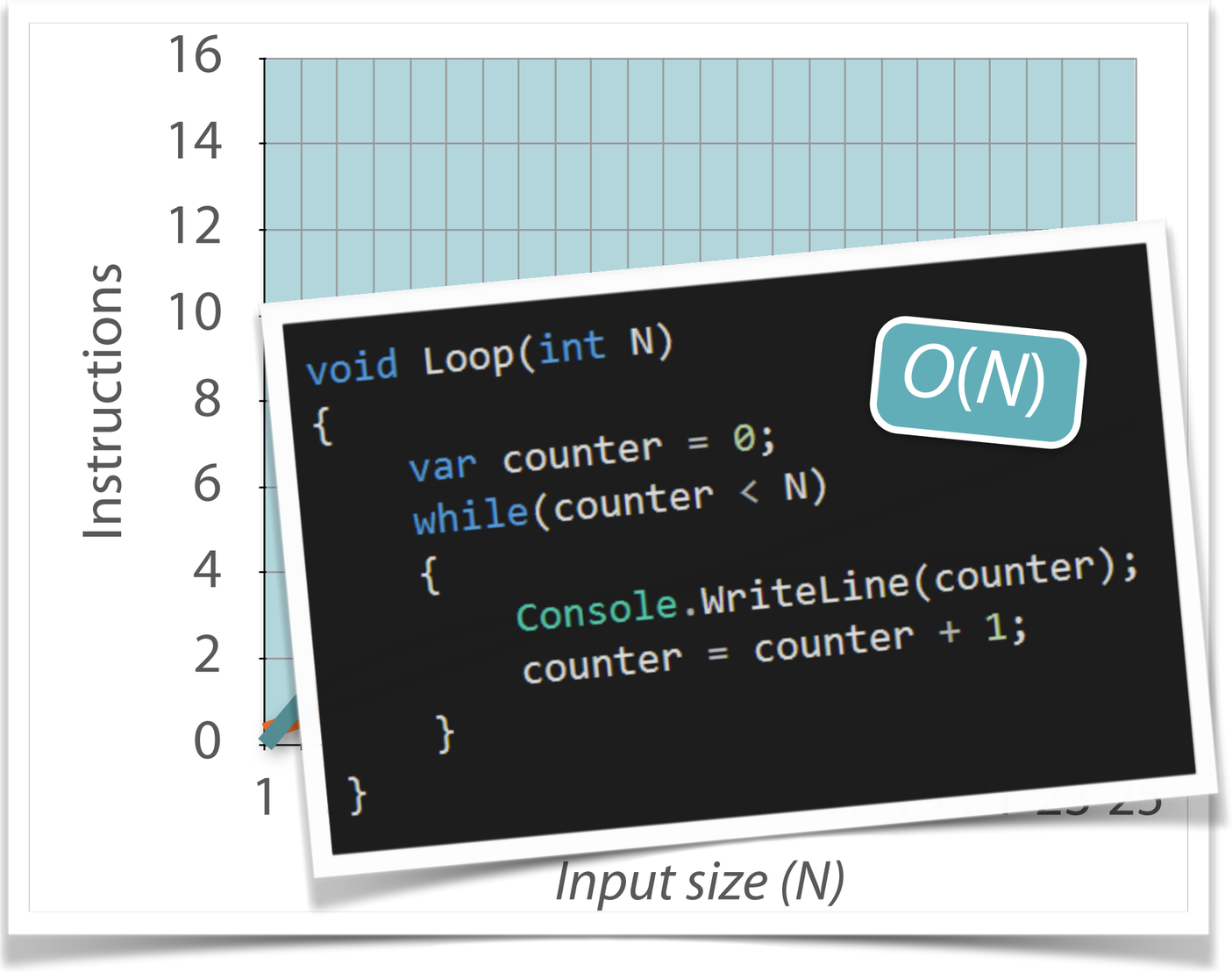

16
14
12
10
8
6
4
2
0
Instructions
1 3 5 7 9 11 13 15 17 19 21 23 25
Input size (N)

Big Omega (Ω)



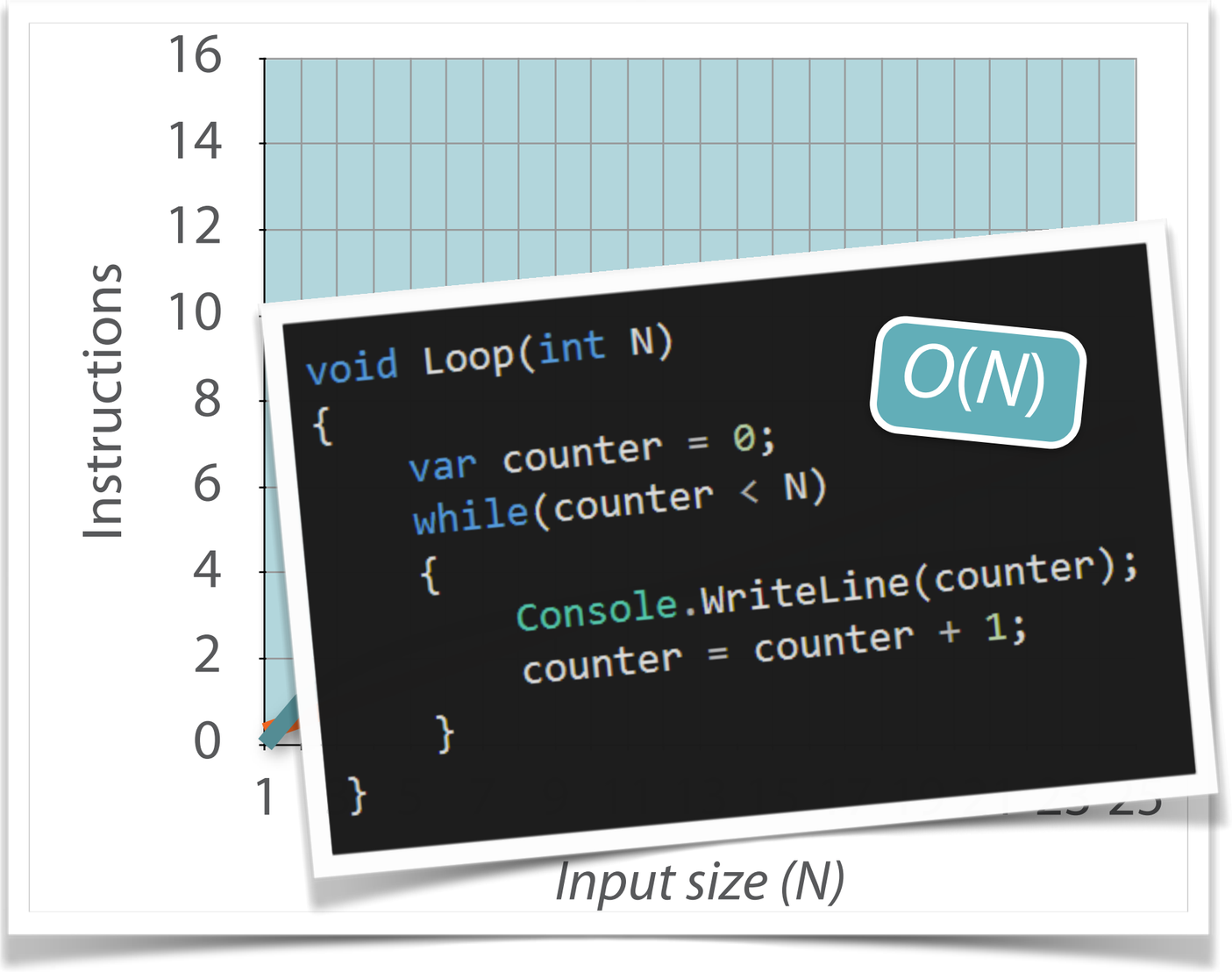
Search algorithms: Ω(1)

Big Omega (Ω)



Search algorithms: Ω(1)

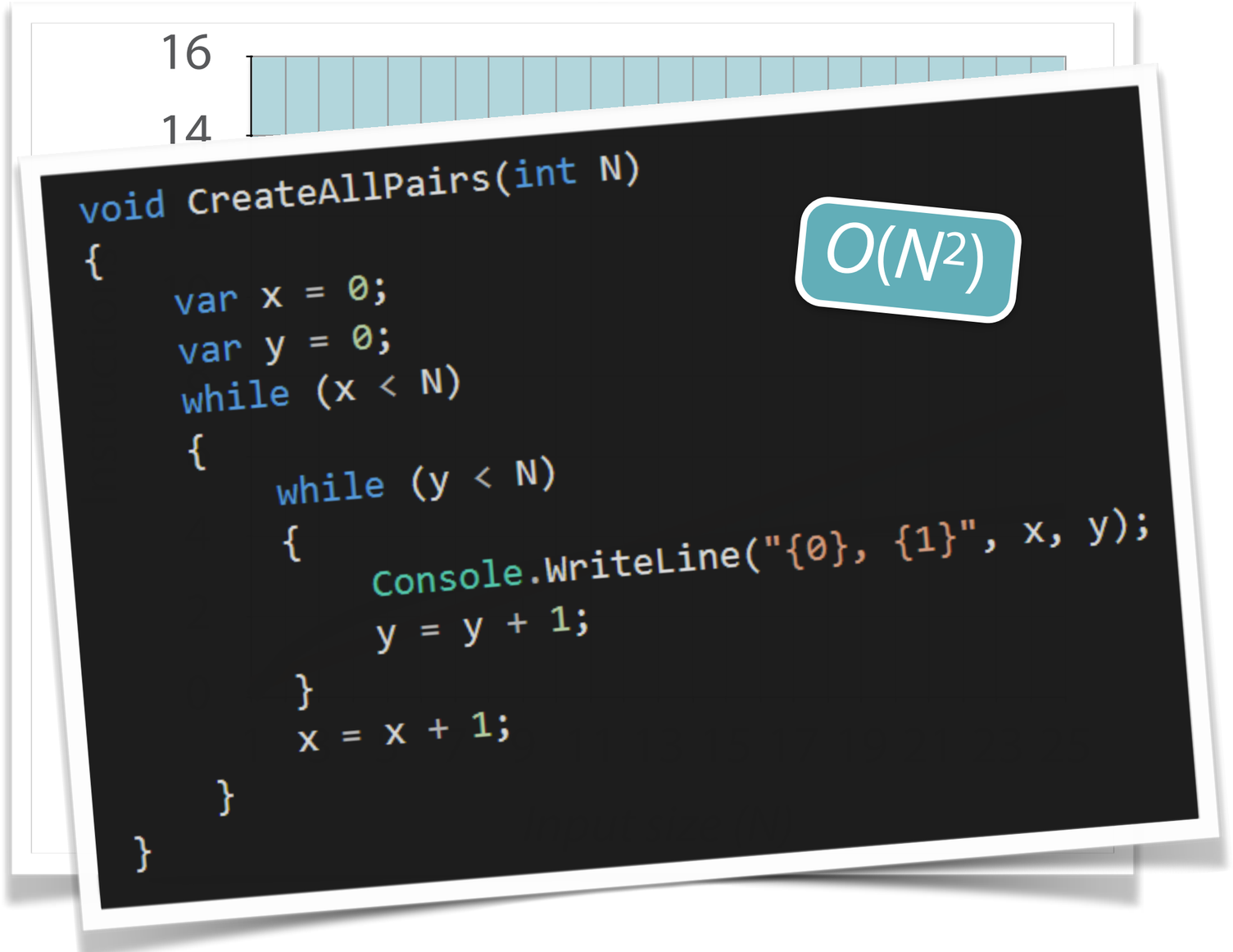
Big Omega (Ω)



Search algorithms: Ω(1)

Simple loop: Ω(*N*)

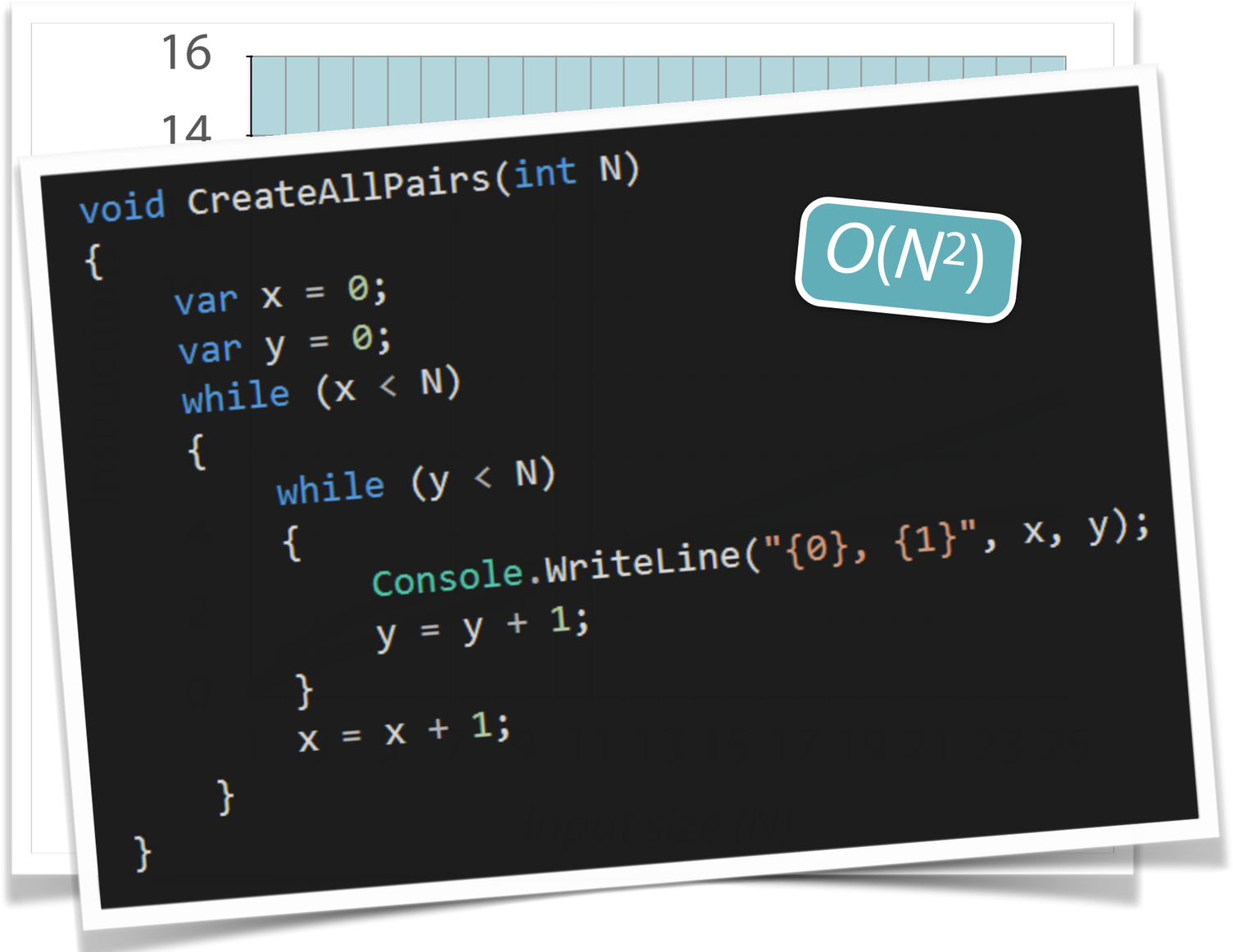
Big Omega (Ω)



Search algorithms: Ω(1)

Simple loop: Ω(*N*)

Big Omega (Ω)



Search algorithms: Ω(1)

Simple loop: Ω(*N*)

Pair generation: Ω(N^2)

Recursive Methods

Recursive Methods

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Recursive Methods

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

$$Faculty(8) = 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 40{,}320$$

Recursive Methods

Faculty(N) = N · Faculty(N-1)

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

2 · Faculty(1)

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

2 · Faculty(1)

1

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

N recursion steps

…

2 · Faculty(1)

1

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

2 · Faculty(1)

1

N recursion steps

For each step:
only a *constant* amount
of instructions, say c

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

2 · Faculty(1)

1

N recursion steps

For each step:
only a *constant* amount
of instructions, say c

Complexity(N) = $c \cdot N$

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Faculty(8) = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1 = 40,320

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

1

N recursion steps

For each step:
only a *constant* amount
of instructions, say c

Complexity(N) = $c \cdot N$

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

Upper bound constant: $c + 1$

$8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 40{,}320$

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

N recursion steps

For each step:
only a *constant* amount
of instructions, say c

Complexity(N) = $c \cdot N$

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

$8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 40{,}320$

Upper bound constant: $c + 1$

Lower bound constant: $c - 1$

Recursive Methods

Faculty(N) = N · Faculty(N-1)

(N-1) · Faculty(N-2)

(N-2) · Faculty(N-3)

…

N recursion steps

For each step:
only a *constant* amount
of instructions, say c

Complexity(N) = $c \cdot N$

$\Theta(N)$

```
int Faculty(int n)
{
    if (n == 1)
        return 1;
    return n * Faculty(n - 1);
}
```

$8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 40{,}320$

Upper bound constant: $c + 1$

Lower bound constant: $c - 1$

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5

```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55

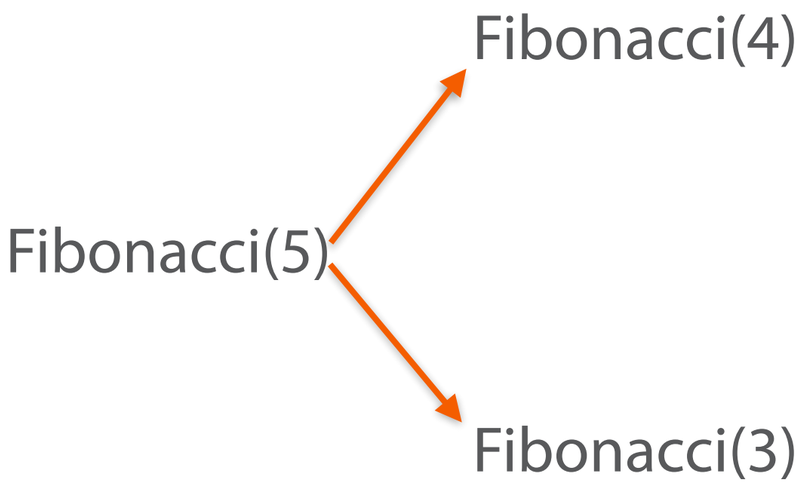
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55

Fibonacci(5)

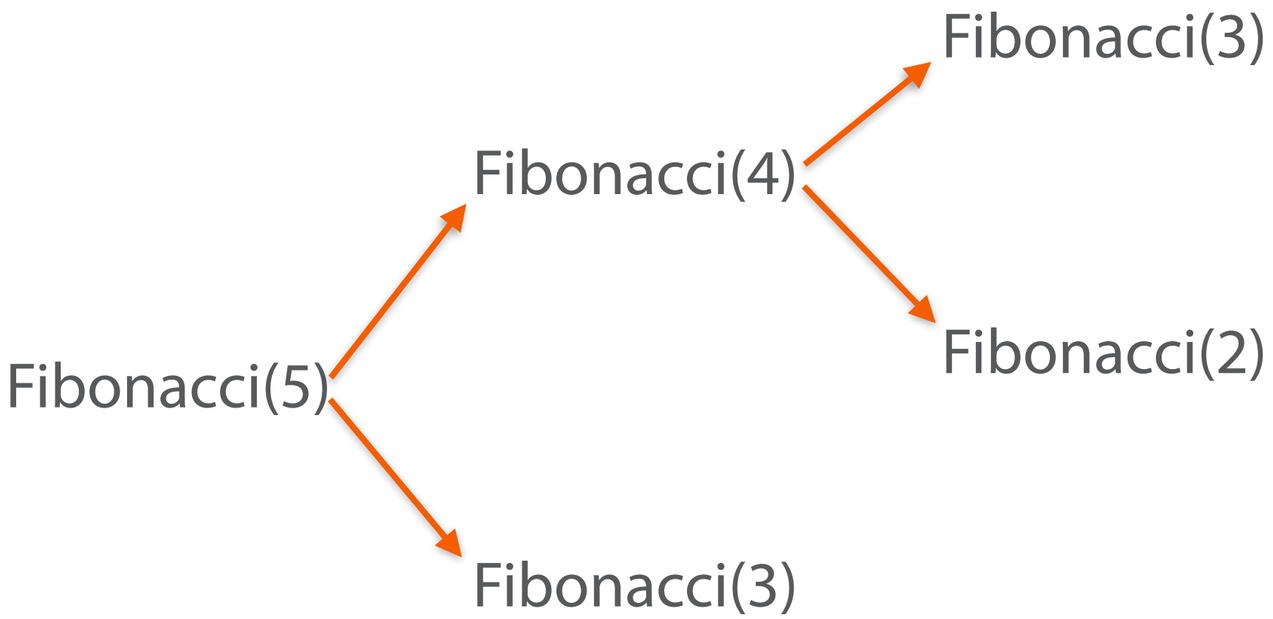
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



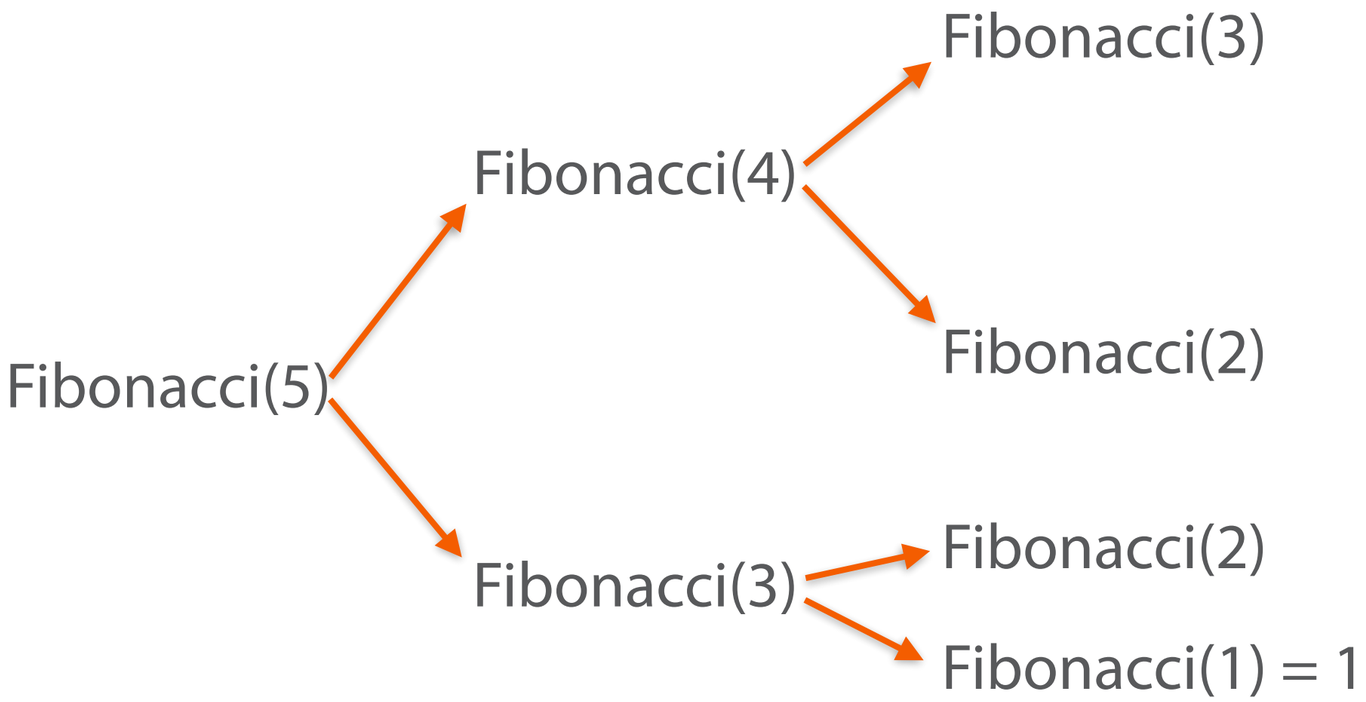
```c
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



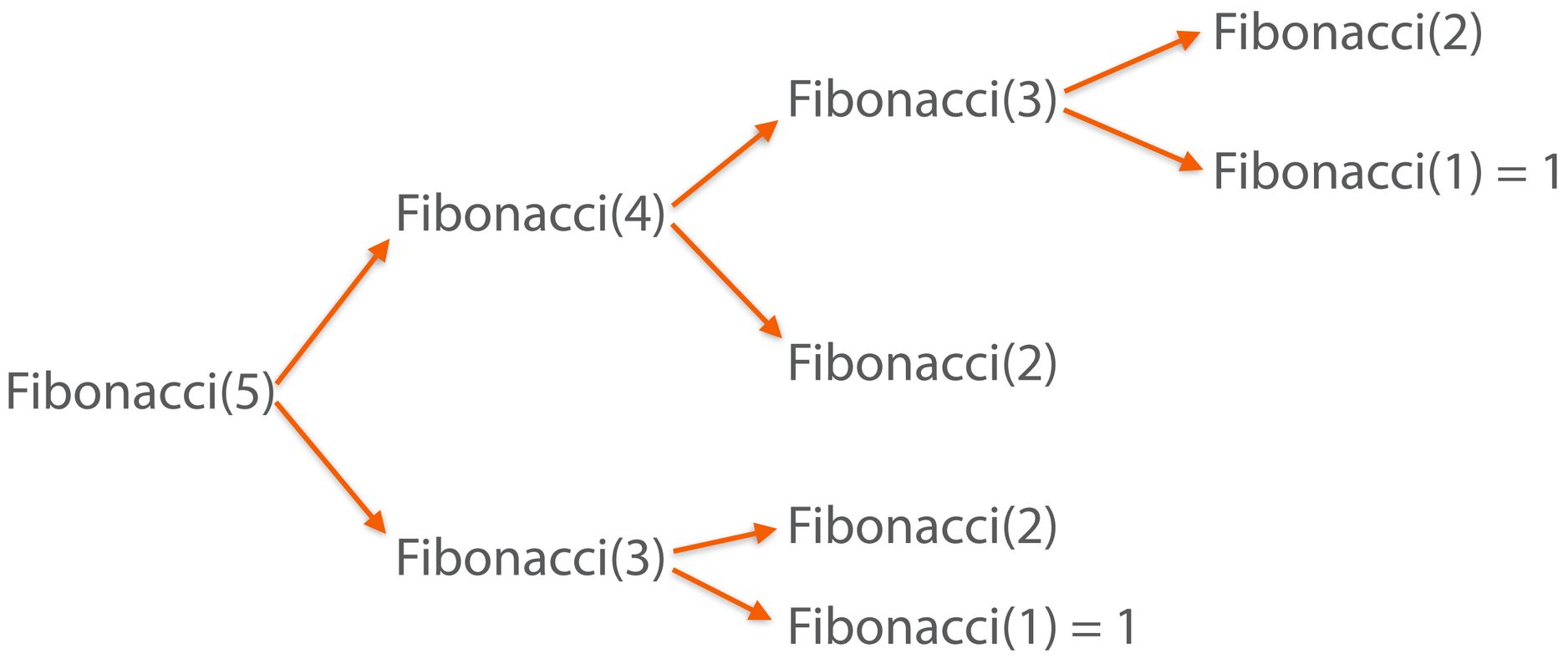
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



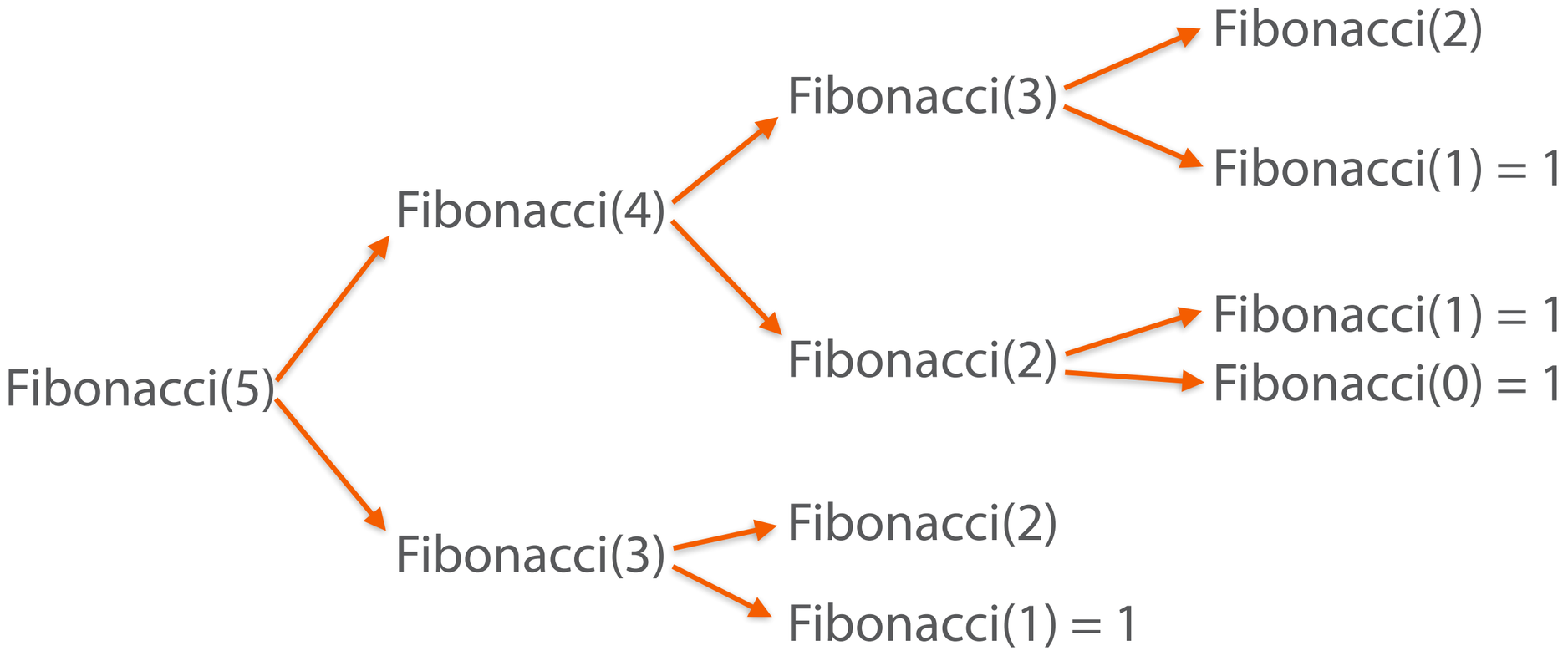
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



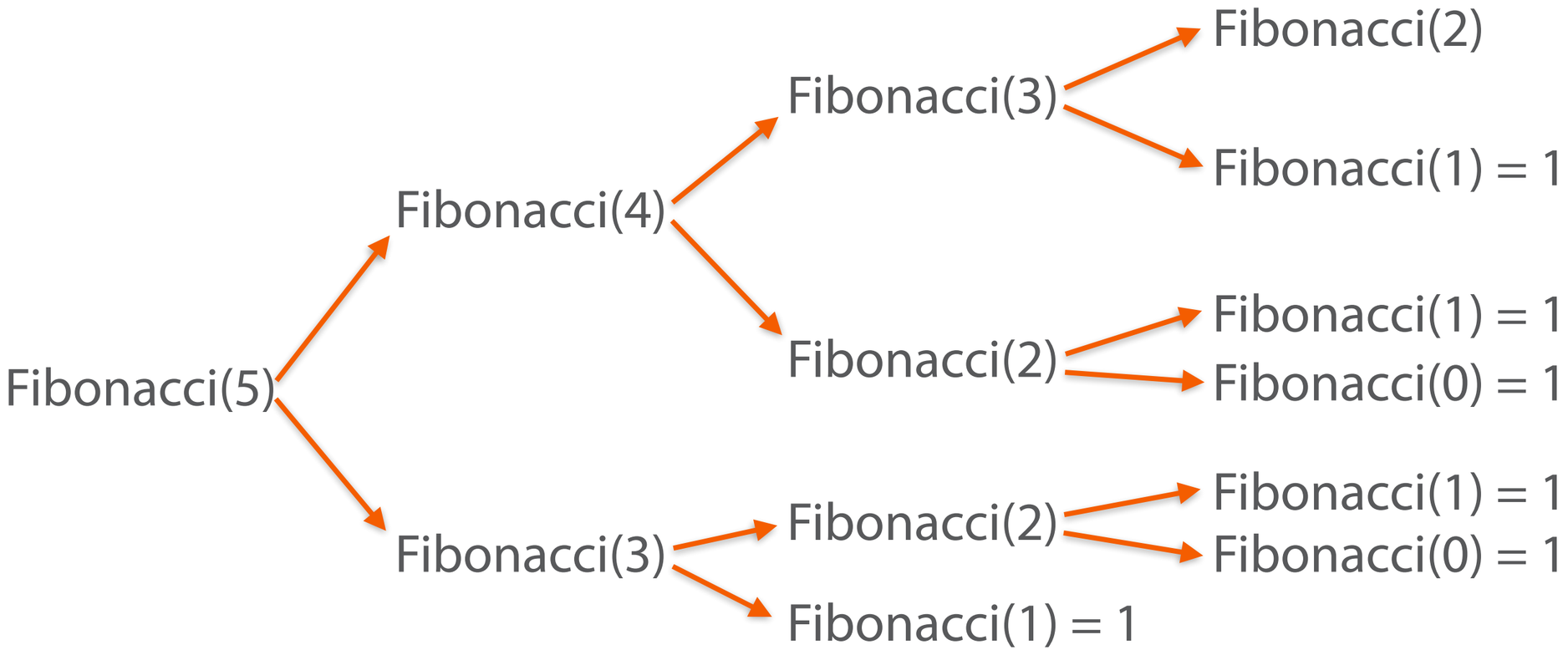
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



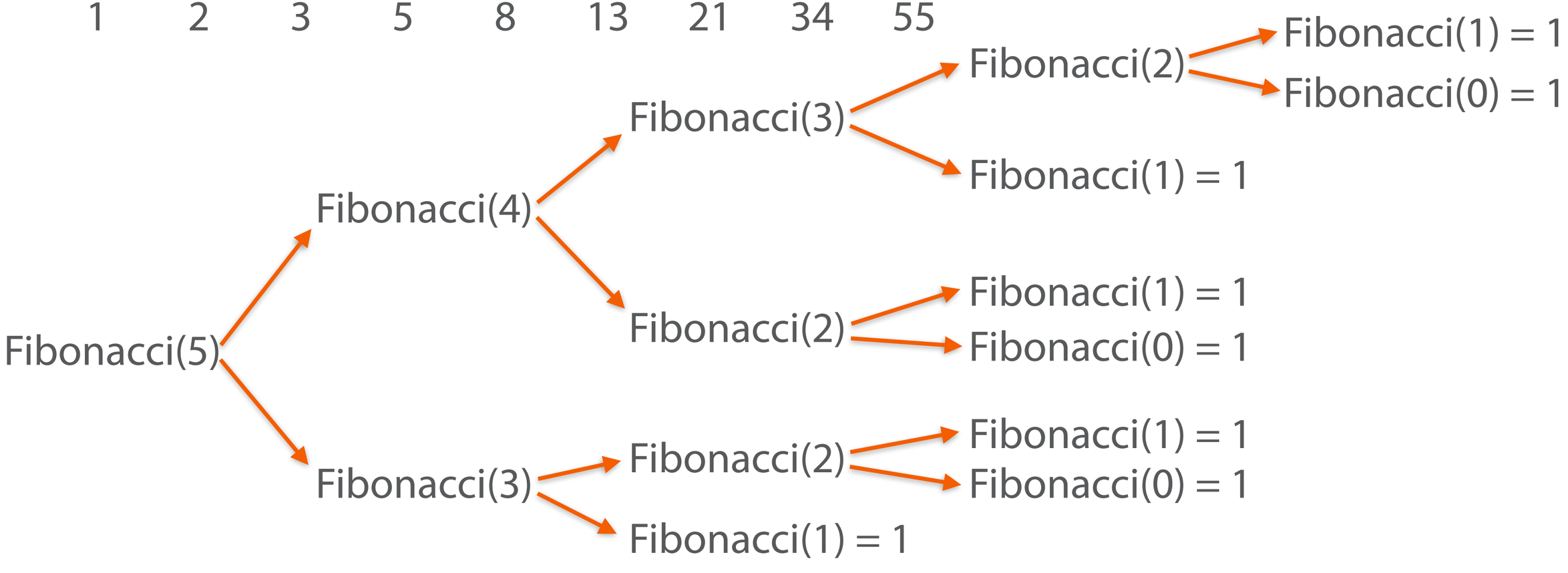
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



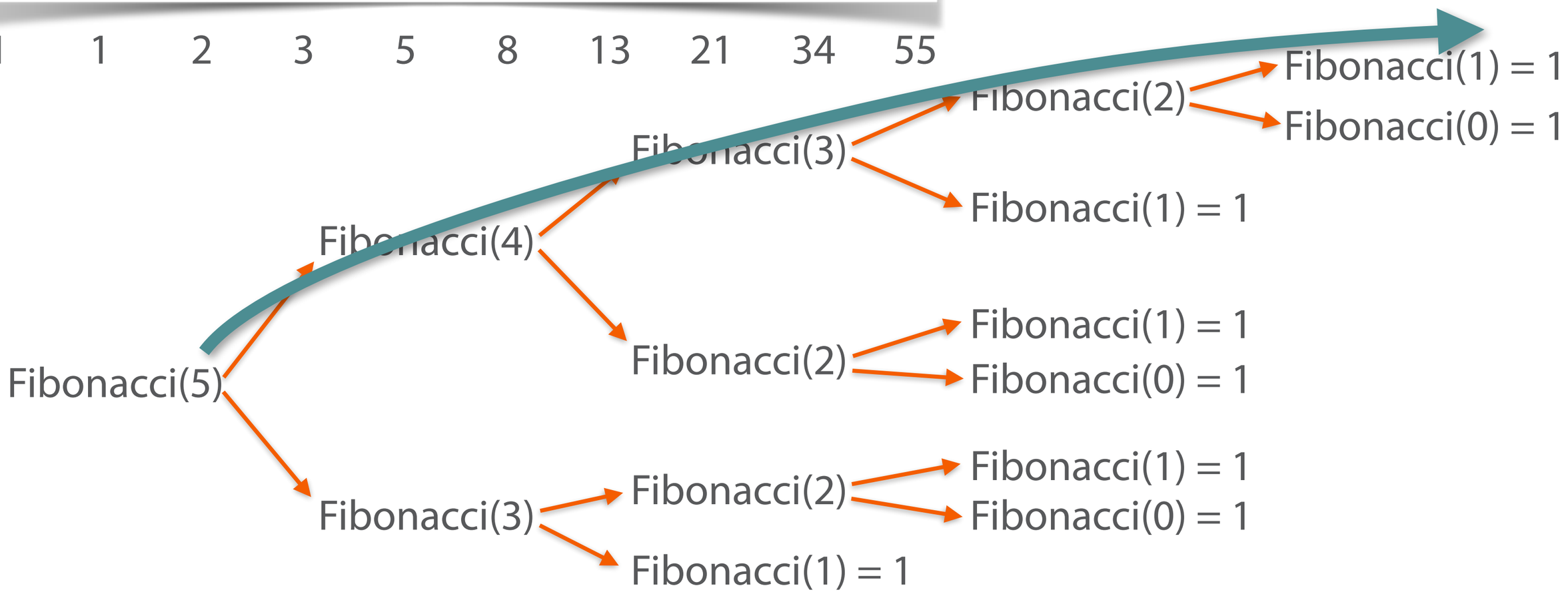
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 1 2 3 5 8 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

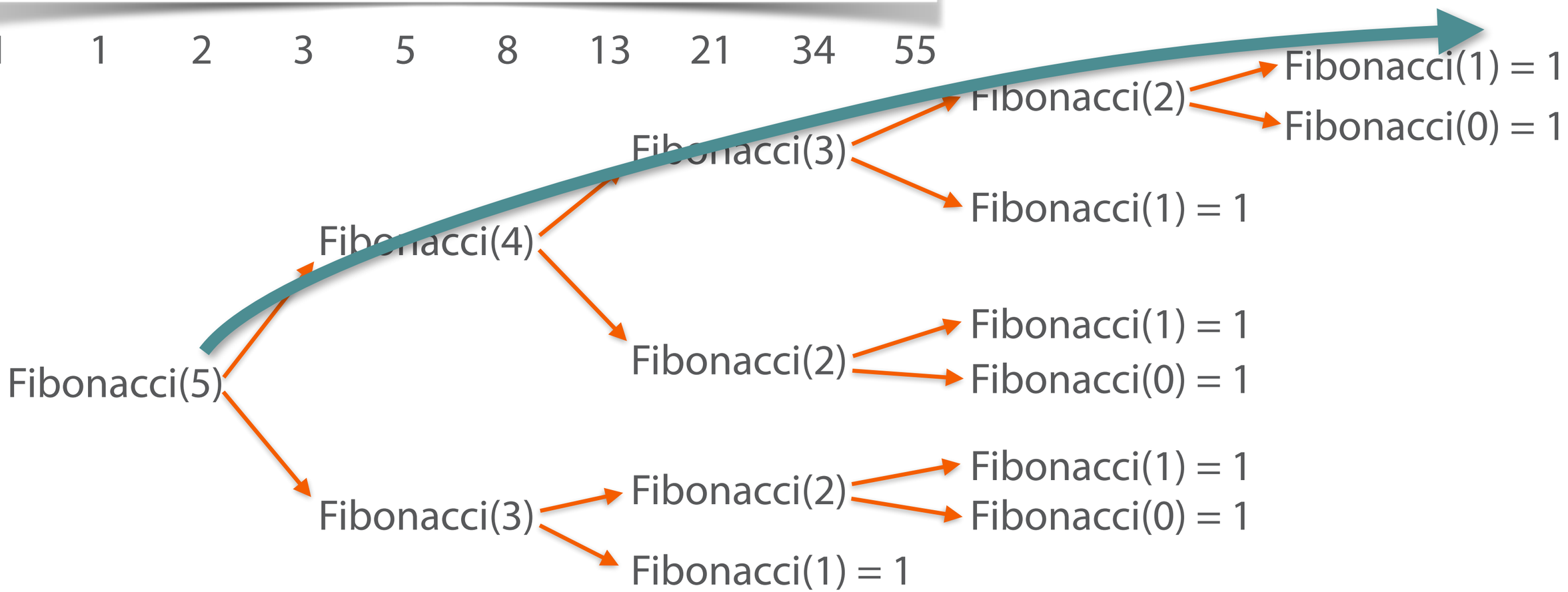
1 1 2 3 5 8 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Fibonacci(*N*): *N* recursion steps

1 1 2 3 5 8 13 21 34 55

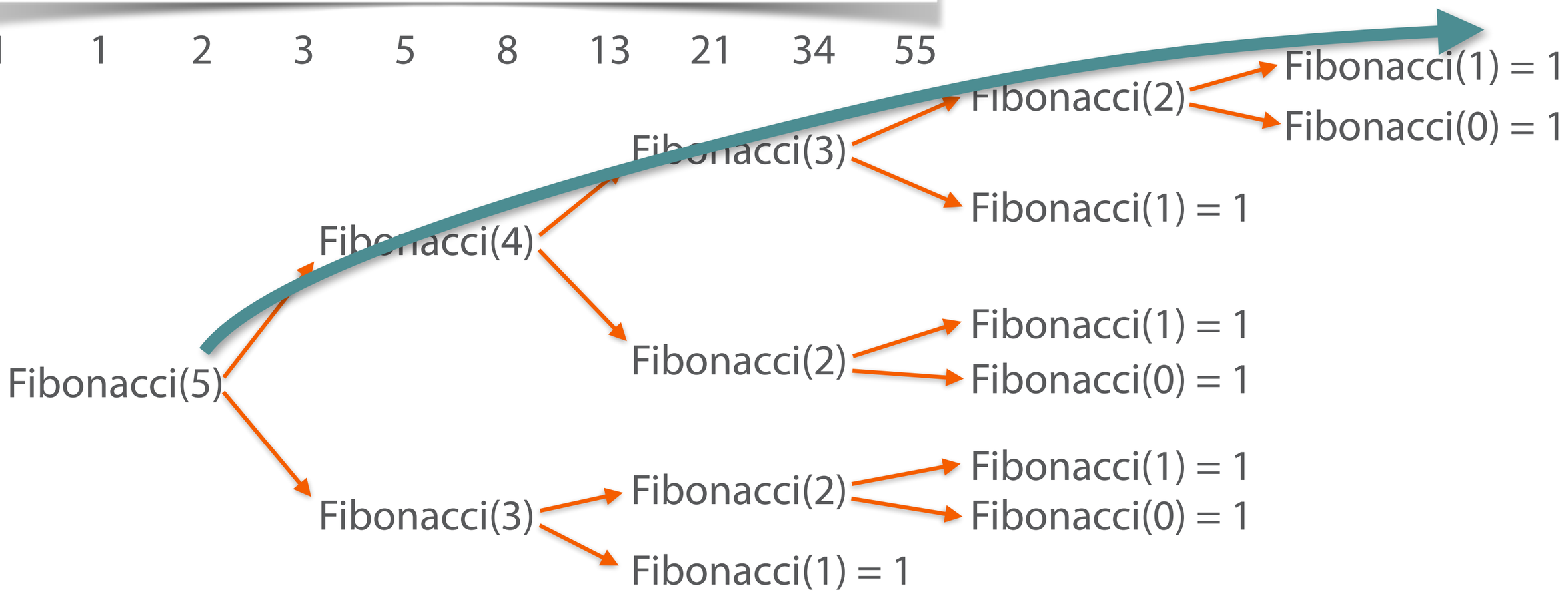


```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Fibonacci(*N*): *N* recursion steps

Each step *at most* doubles

1 1 2 3 5 8 13 21 34 55



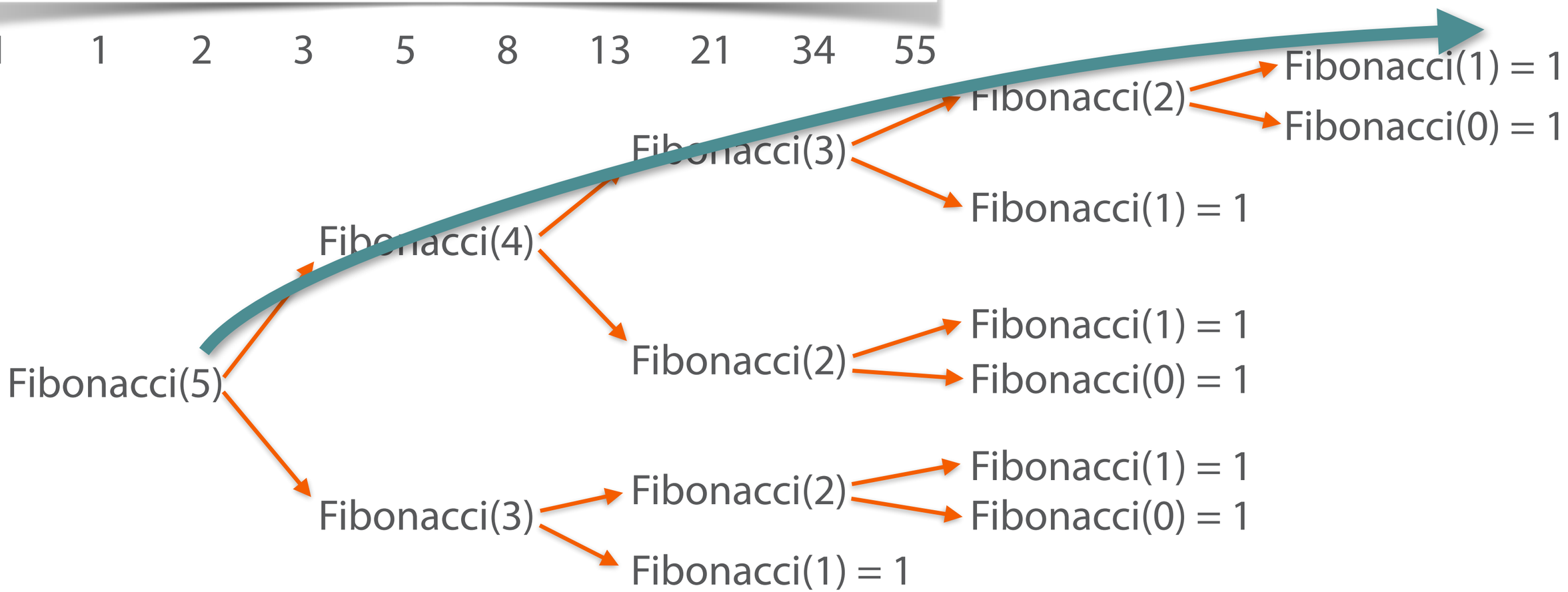
```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Fibonacci(*N*): *N* recursion steps

Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ \ldots \ \cdot 2 = c \cdot 2^{N-1}$

1 1 2 3 5 8 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n
}
```

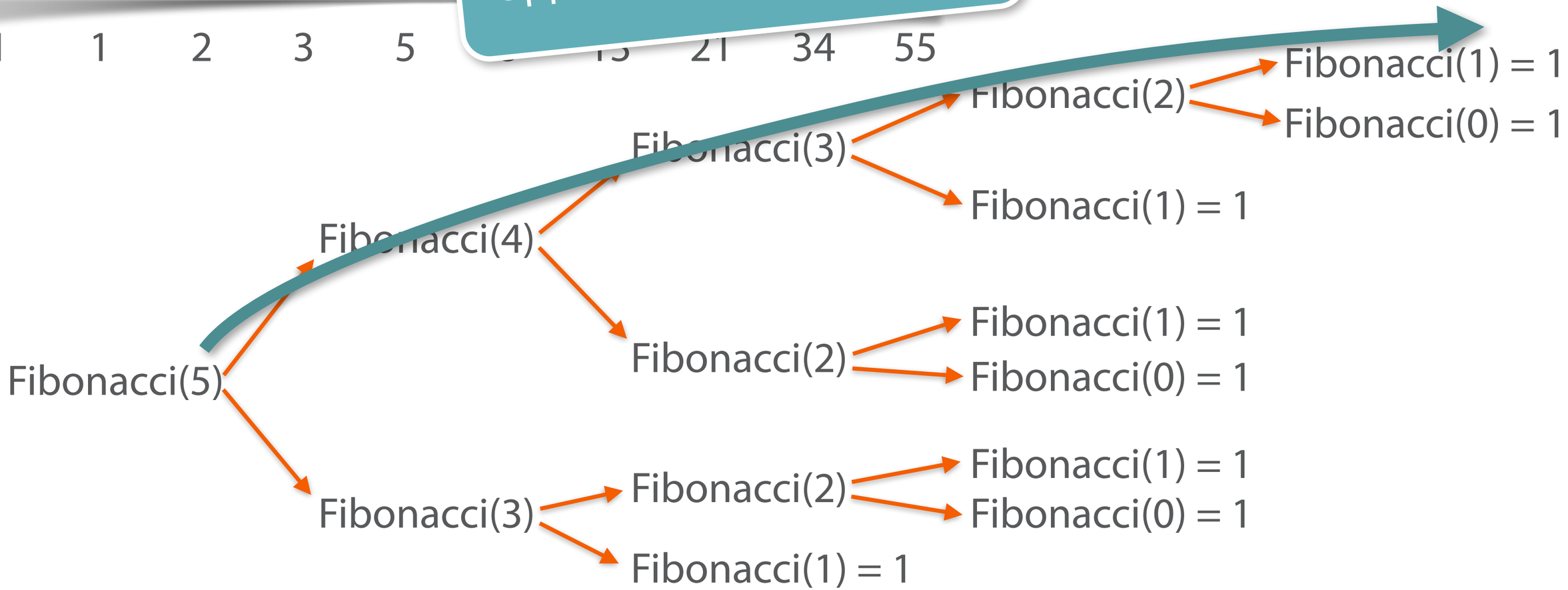
Fibonacci(*N*): *N* recursion steps

Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = c \cdot 2^{N-1}$

Upper bound constant: $c \cdot 2$

1 1 2 3 5 ... 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n
}
```

Fibonacci(*N*): *N* recursion steps

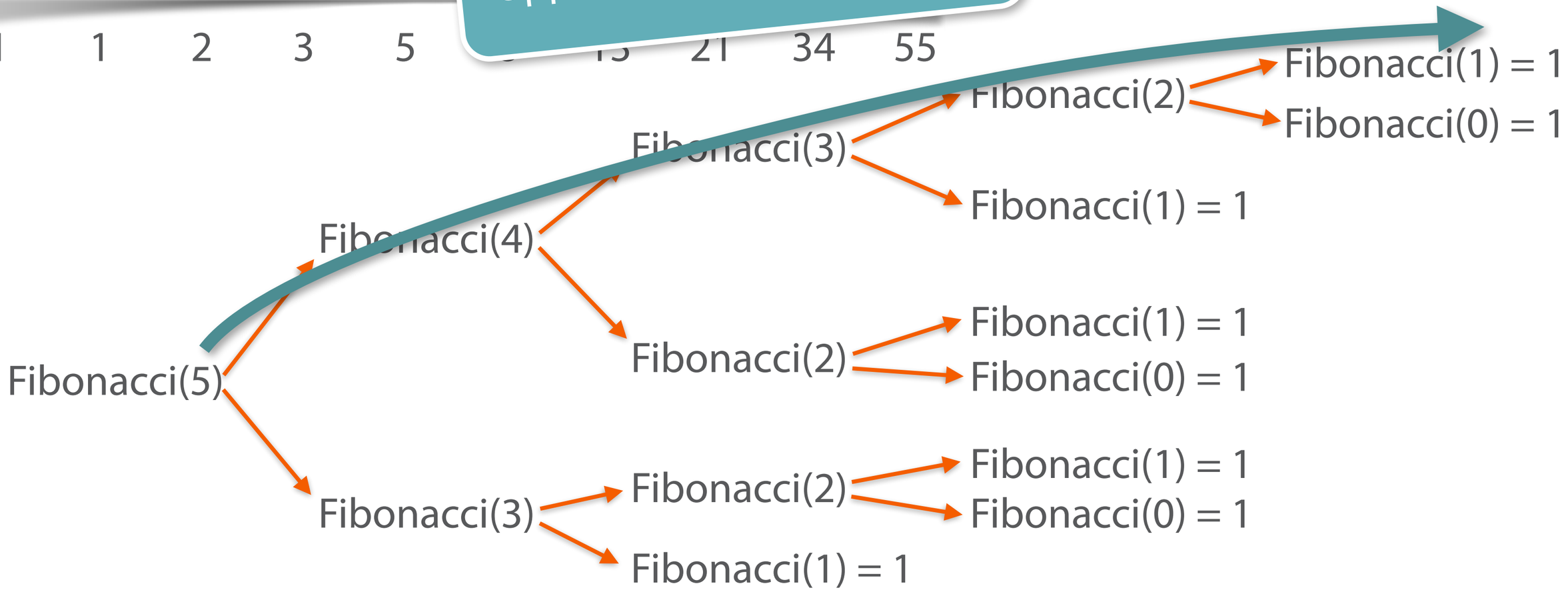
Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = c \cdot 2^{N-1}$

Upper bound: $c \cdot 2 \cdot 2^{N-1}$

Upper bound constant: $c \cdot 2$

1 1 2 3 5 ... 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n
}
```

Fibonacci(*N*): *N* recursion steps

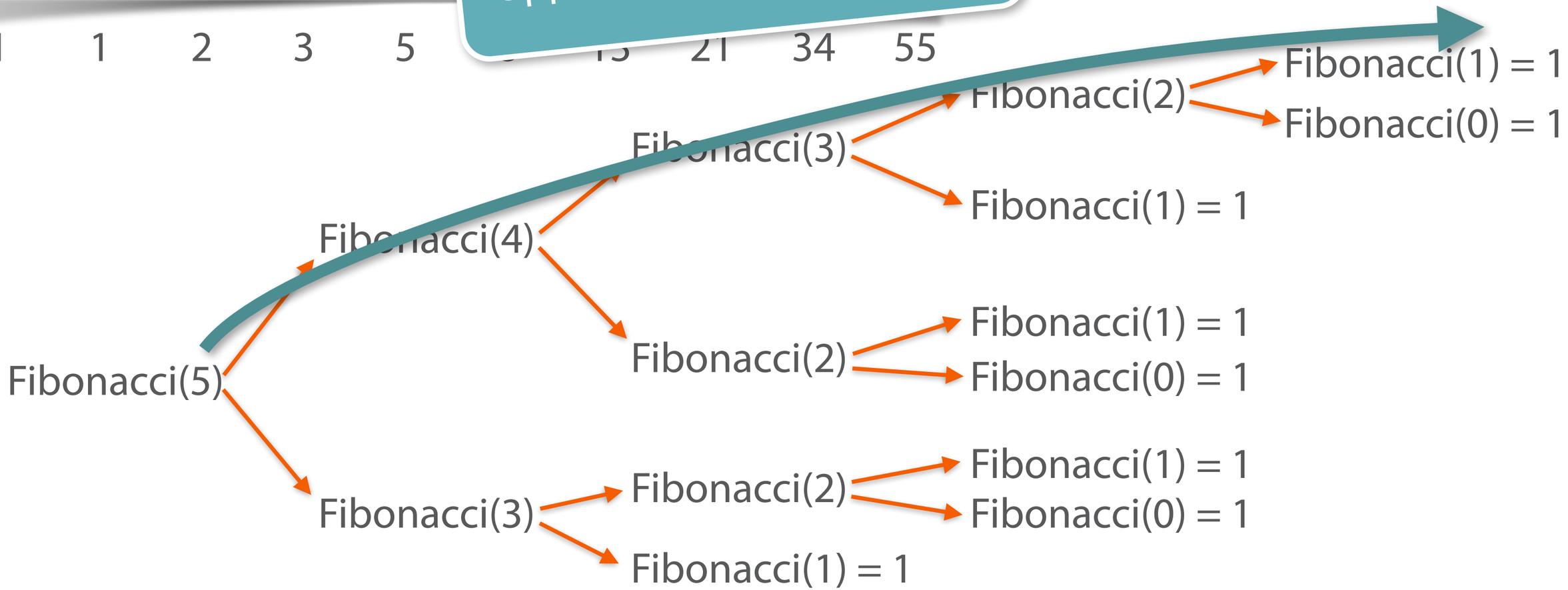
Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = c \cdot 2^{N-1}$

Upper bound: $c \cdot 2 \cdot 2^{N-1} = c \cdot 2^{N}$

Upper bound constant: $c \cdot 2$

1 1 2 3 5 ... 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n
}
```

Upper bound constant: $c \cdot 2$

Fibonacci(*N*): *N* recursion steps

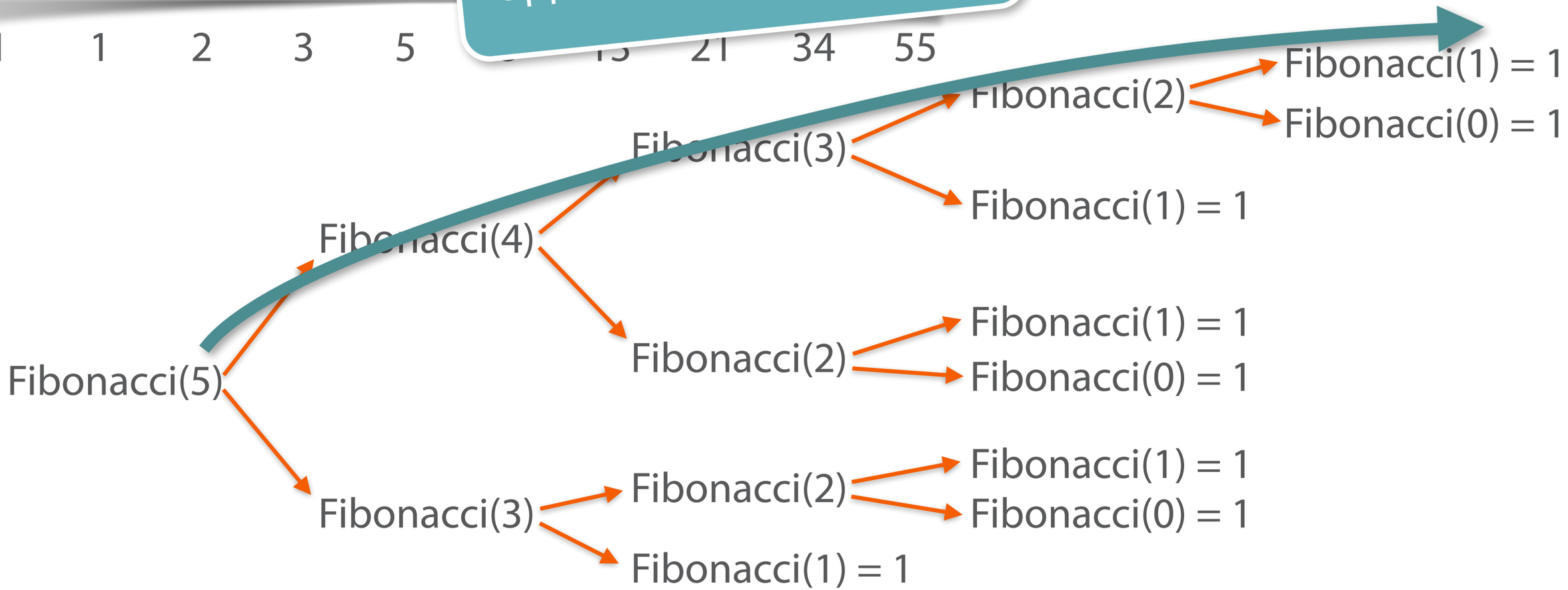
Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = c \cdot 2^{N-1}$

Upper bound: $c \cdot 2 \cdot 2^{N-1} = c \cdot 2^{N}$

$O(2^N)$

1 1 2 3 5 ... 13 21 34 55



```
int Fibonacci(int n)
{
    if (n <= 1) // throw error if n < 0
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n
}
```

Fibonacci(*N*): *N* recursion steps

Each step *at most* doubles

$c \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = c \cdot 2^{N-1}$

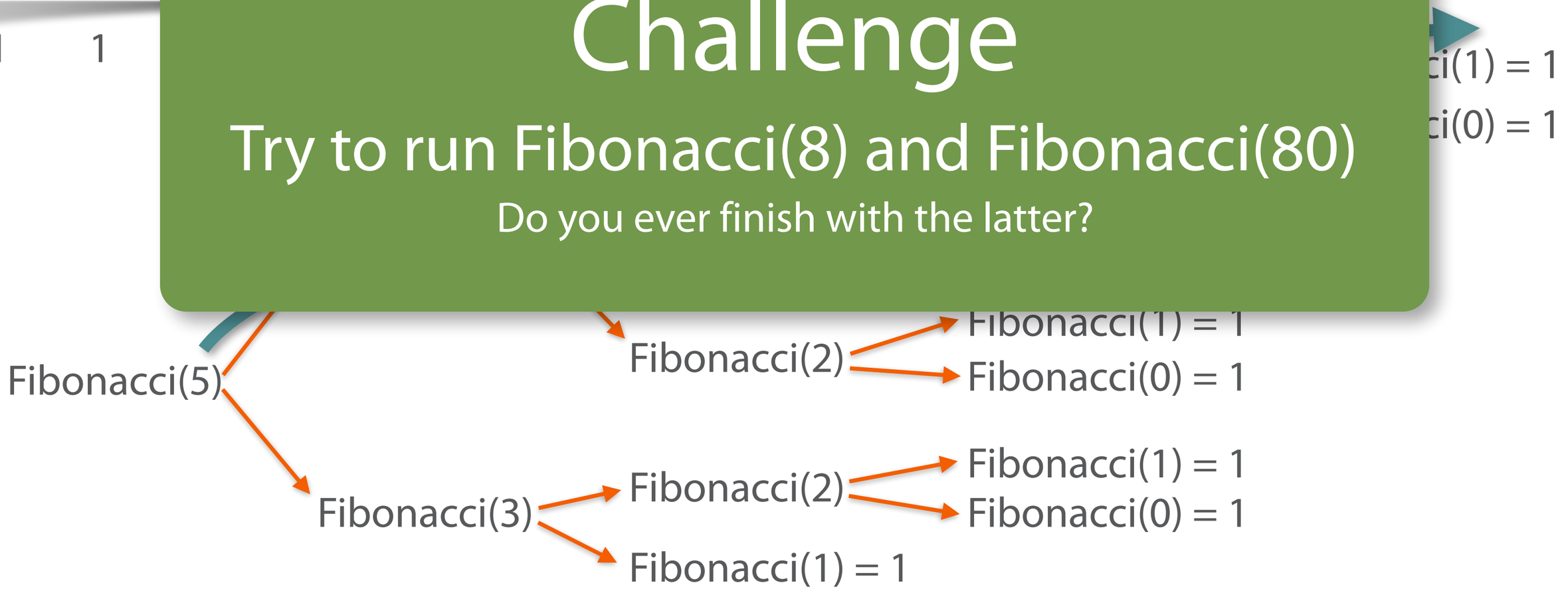
Upper bound: $c \cdot 2 \cdot 2^{N-1} = c \cdot 2^{N}$

1 1

Challenge

Try to run Fibonacci(8) and Fibonacci(80)

Do you ever finish with the latter?



```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i < n; i++)
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i < n; i++)
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Complexity(*N*):

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i < n; i++)
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Complexity(*N*):

c_1 +

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i < n; i++)
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Complexity(*N*):

$$c_1 + N \cdot$$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i < n; i++)
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Complexity(*N*):

$$c_1 + N \cdot c_2$$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Upper bound constant: $c_2 + 1$

Complexity(N):

$$c_1 + N \cdot c_2$$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i
    {
        var oldHigh = high;
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Upper bound constant: $c_2 + 1$

Complexity(N):

$$c_1 + N \cdot c_2$$

$$O(N)$$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i
    {
        var oldHigh =
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Upper bound constant: $c_2 + 1$

Lower bound constant: $c_2 - 1$

Complexity(N):

$$c_1 + N \cdot c_2$$

$$O(N)$$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i
    {
        var oldHigh =
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Upper bound constant: $c_2 + 1$

Lower bound constant: $c_2 - 1$

Complexity(N):

$$c_1 + N \cdot c_2$$

$O(N)$

$\Omega(N)$

```
internal int Fibonacci2(int n)
{
    int low = 1;
    int high = 1;
    for (int i = 0; i
    {
        var oldHigh =
        high = low + high;
        low = oldHigh;
    }
    return low;
}
```

Upper bound constant: $c_2 + 1$

Lower bound constant: $c_2 - 1$

Complexity(N):

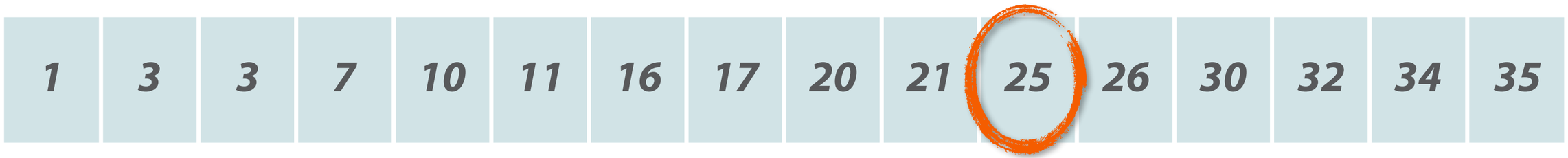
$$c_1 + N \cdot c_2$$

$O(N)$

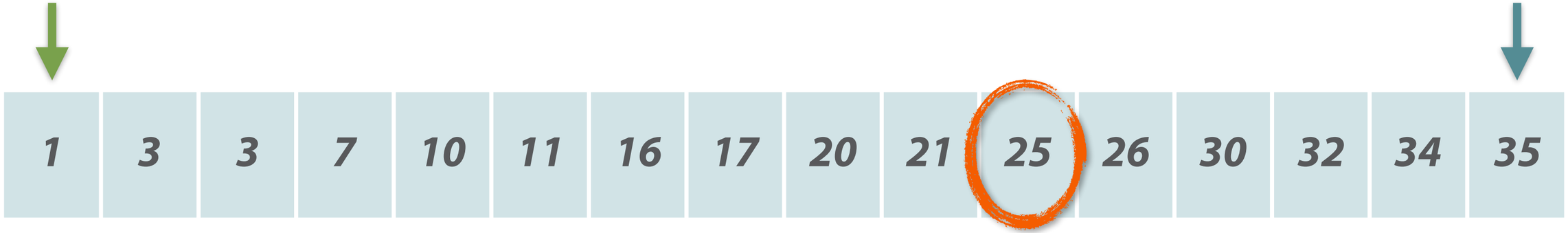
$\Theta(N)$

$\Omega(N)$

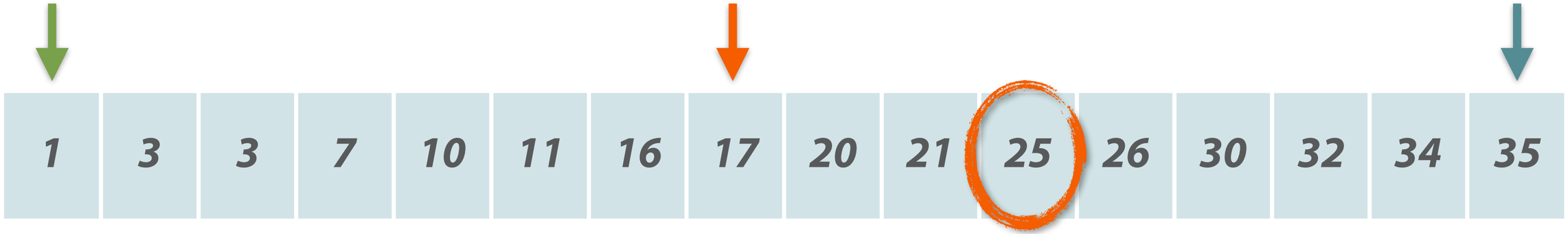
```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



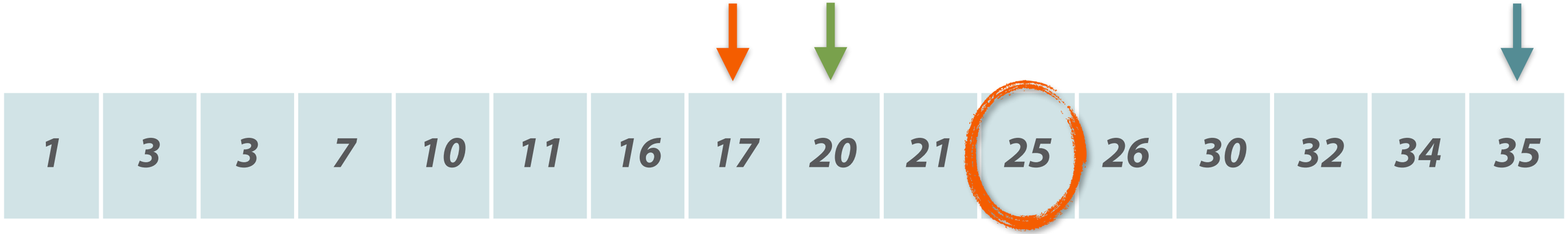

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



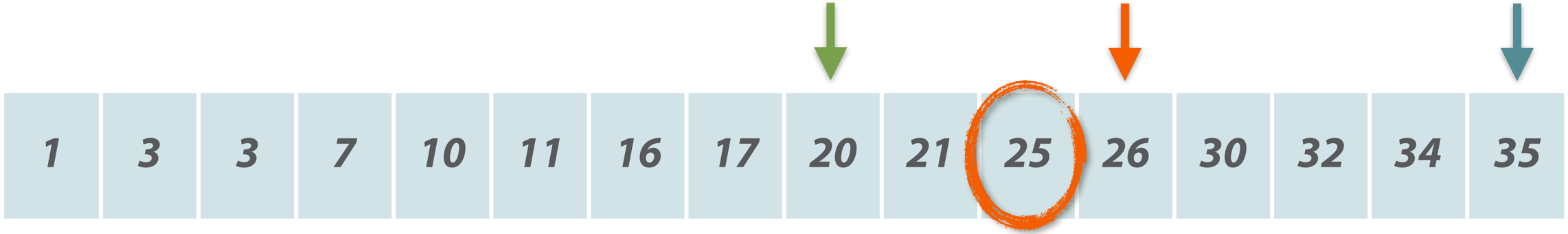

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



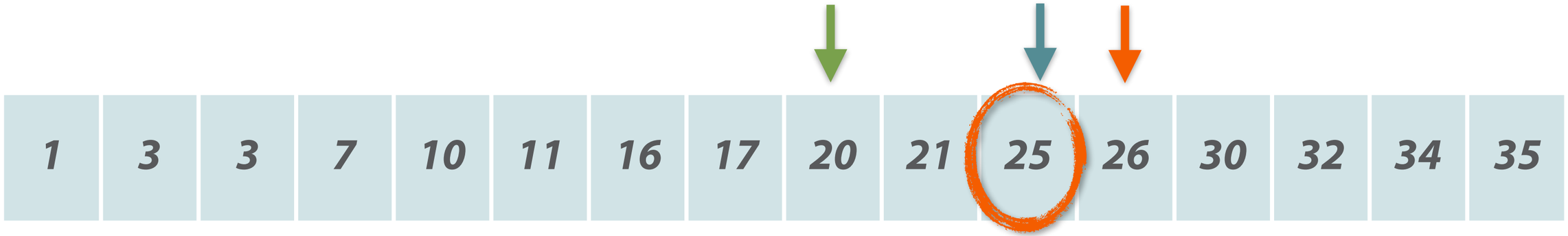

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



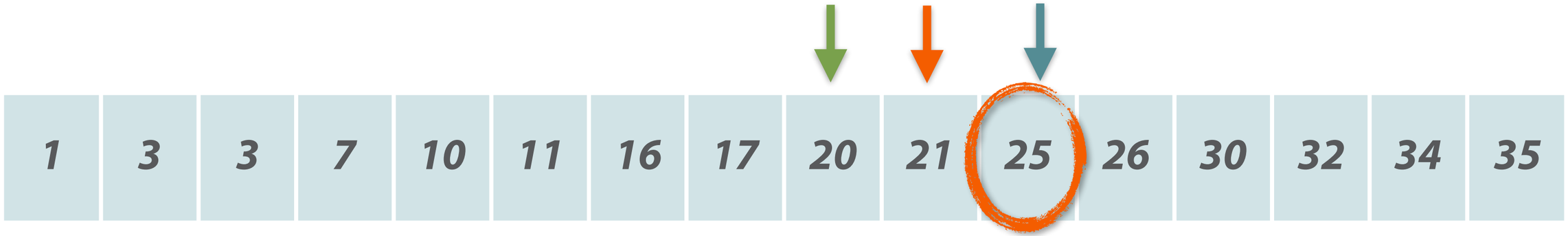

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



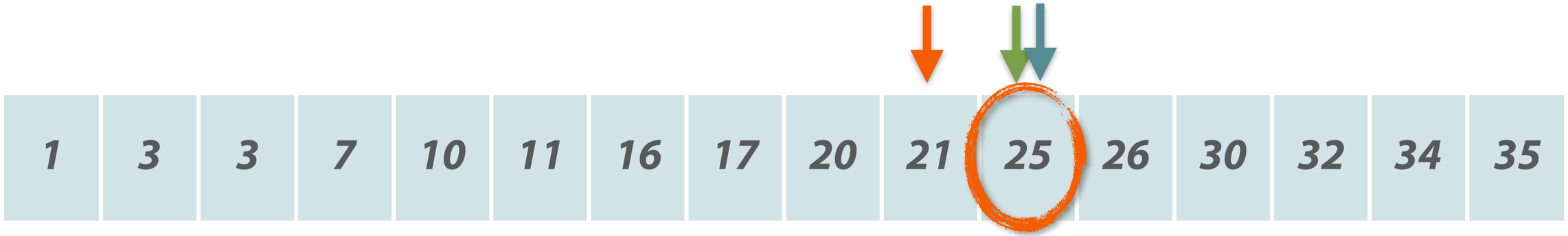

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```



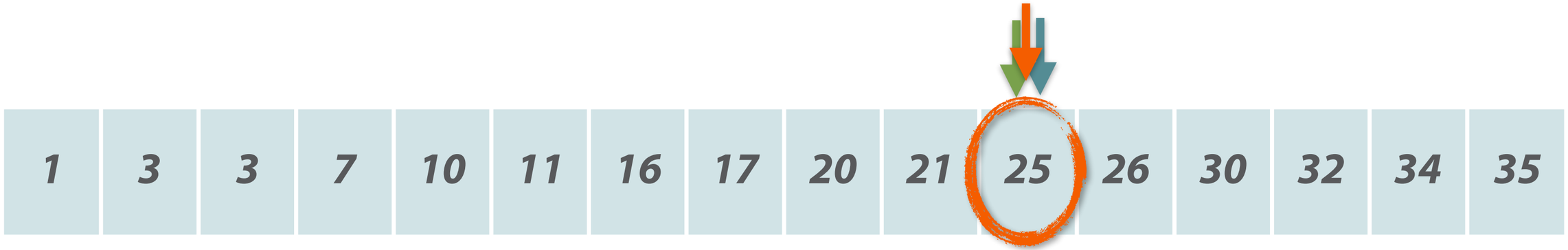

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```




```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```




```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```




```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

1	3	3	7	10	11	16	17	20	21	25	26	30	32	34	35

Haystack size: *N*

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case:

1	3	3	7	10	11	16	17	20	21	25	26	30	32	34	35

Haystack size: *N*

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps

1	3	3	7	10	11	16	17	20	21	25	26	30	32	34	35

Haystack size: *N*

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps

1	3	3	7	10	11	16	17	20	21	25	26	30	32	34	35

Haystack size: *N*

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps max c instructions per step

1	3	3	7	10	11	16	17	20	21	25	26	30	32	34	35

Haystack size: N

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps max c instructions per step

$c \cdot \log_2 N$ instructions in total

Haystack size: N

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps max c instructions per step

$c \cdot \log_2 N$ instructions in total

$O(\log_2 N)$

Haystack size: N

```
int? BinarySearch(List<int> haystack, int needle, int min, int max)
{
    var midpoint = (max + min) / 2;
    if (haystack.Count > 0 && haystack[midpoint] == needle)
        return midpoint;
    if (min >= max)
        return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

Worst-case: $\log_2 N$ recursion steps max c instructions per step

$c \cdot \log_2 N$ instructions in total

$O(\log_2 N)$

Haystack size: N

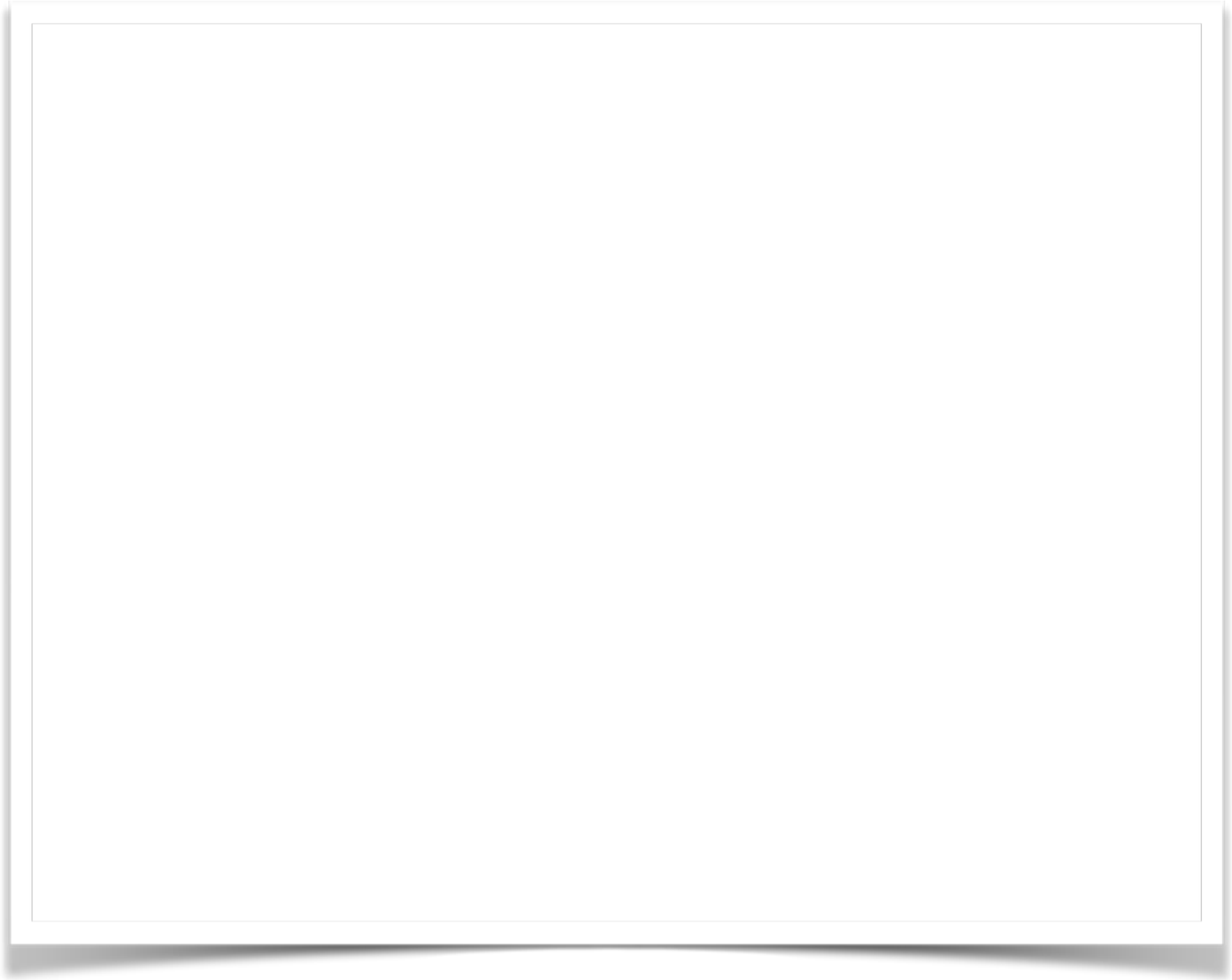
General strategy

1: Fold out and count steps

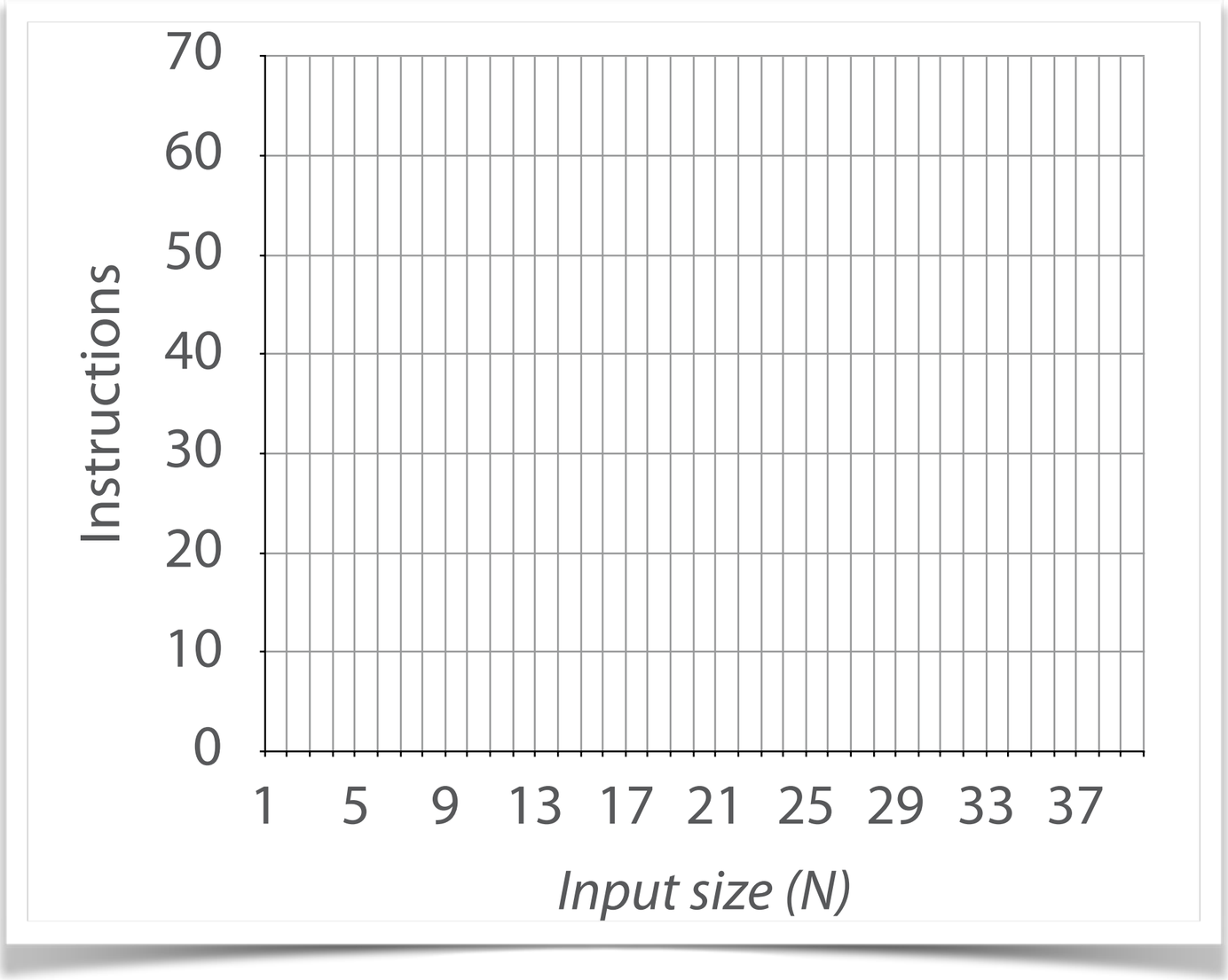
2: Count size per step

```
i                                                    x)
{
    return null;
    if (haystack[midpoint] > needle)
        return BinarySearch(haystack, needle, min, midpoint - 1);
    return BinarySearch(haystack, needle, midpoint + 1, max);
}
```

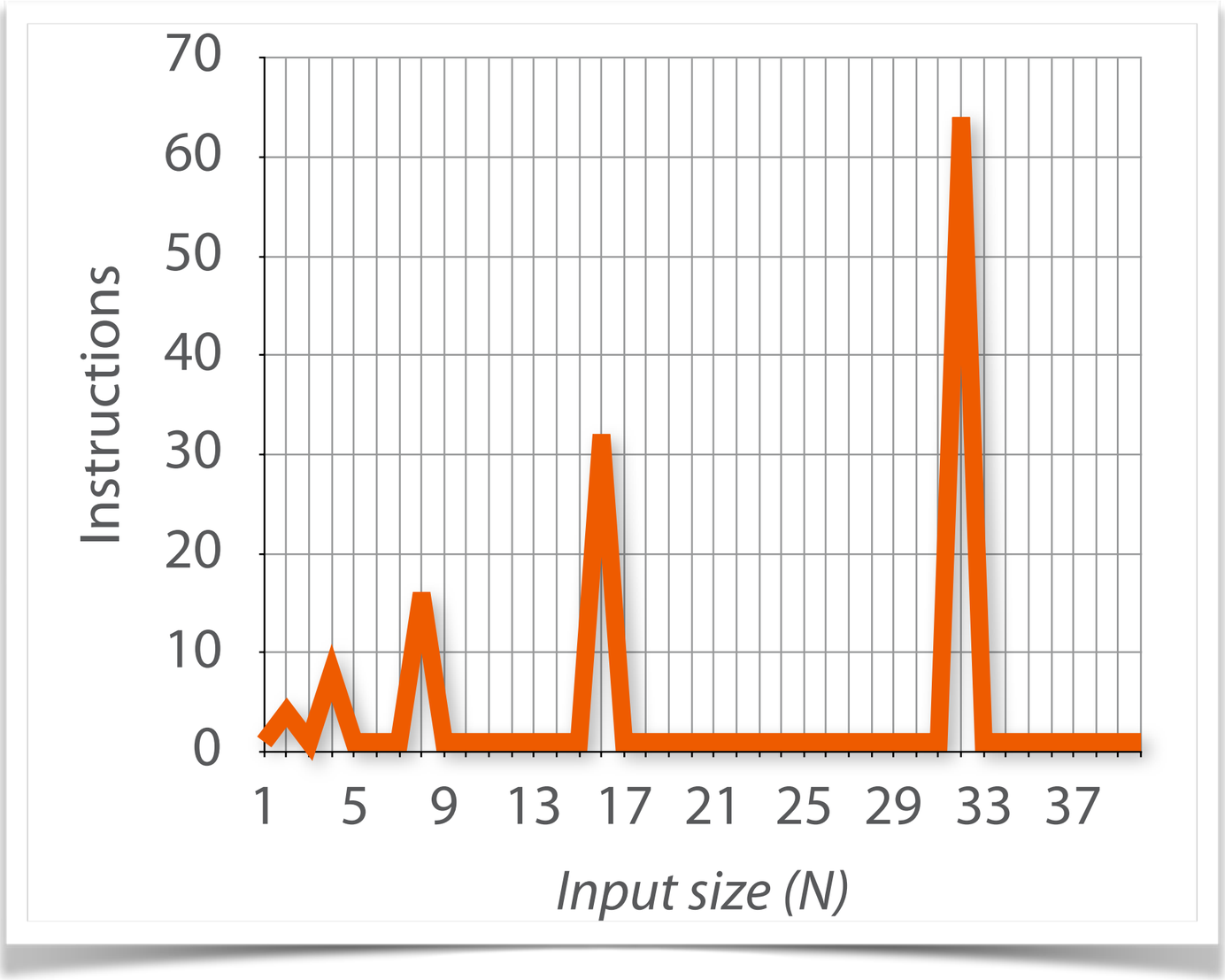
Amortized Complexity



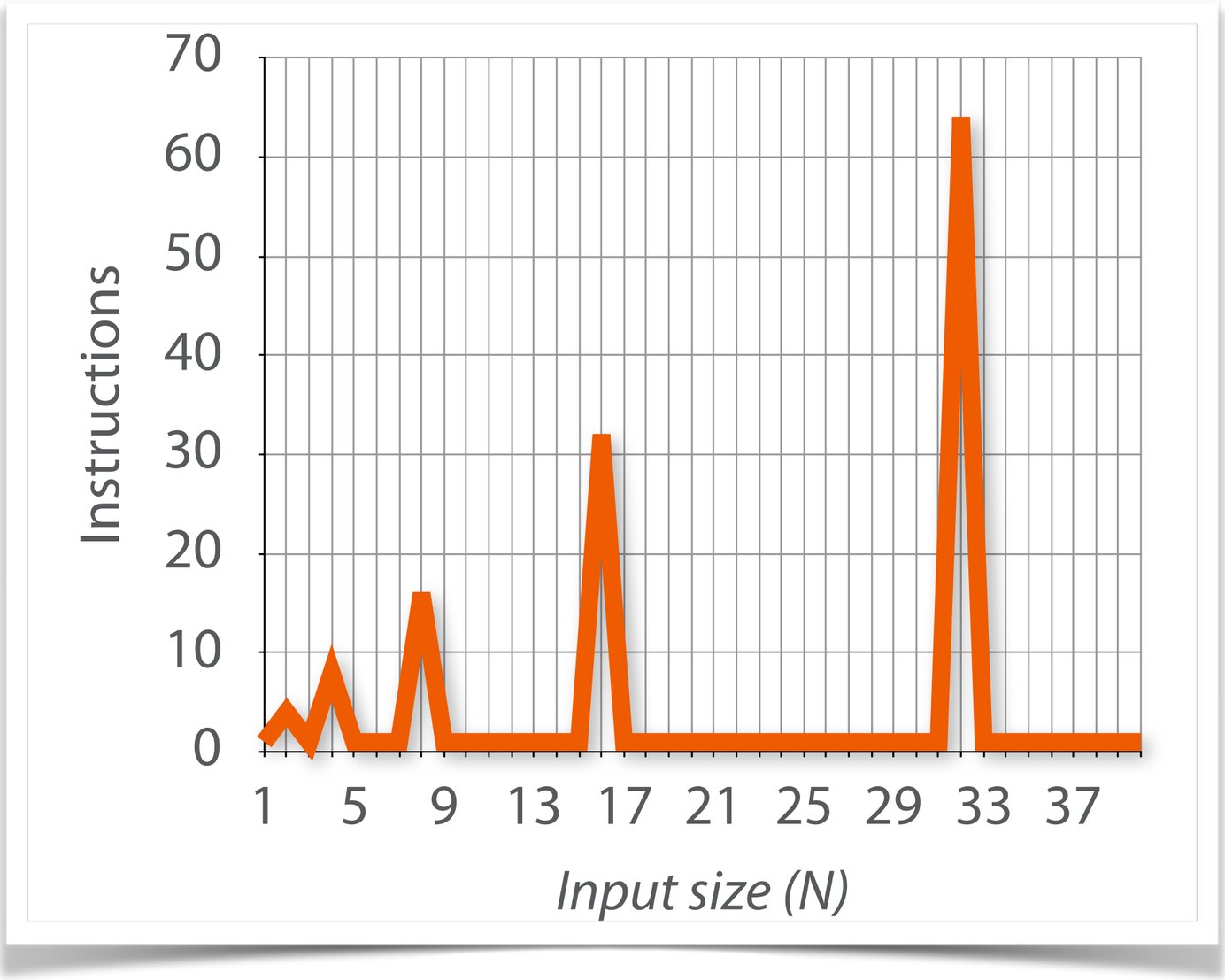
Amortized Complexity



Amortized Complexity

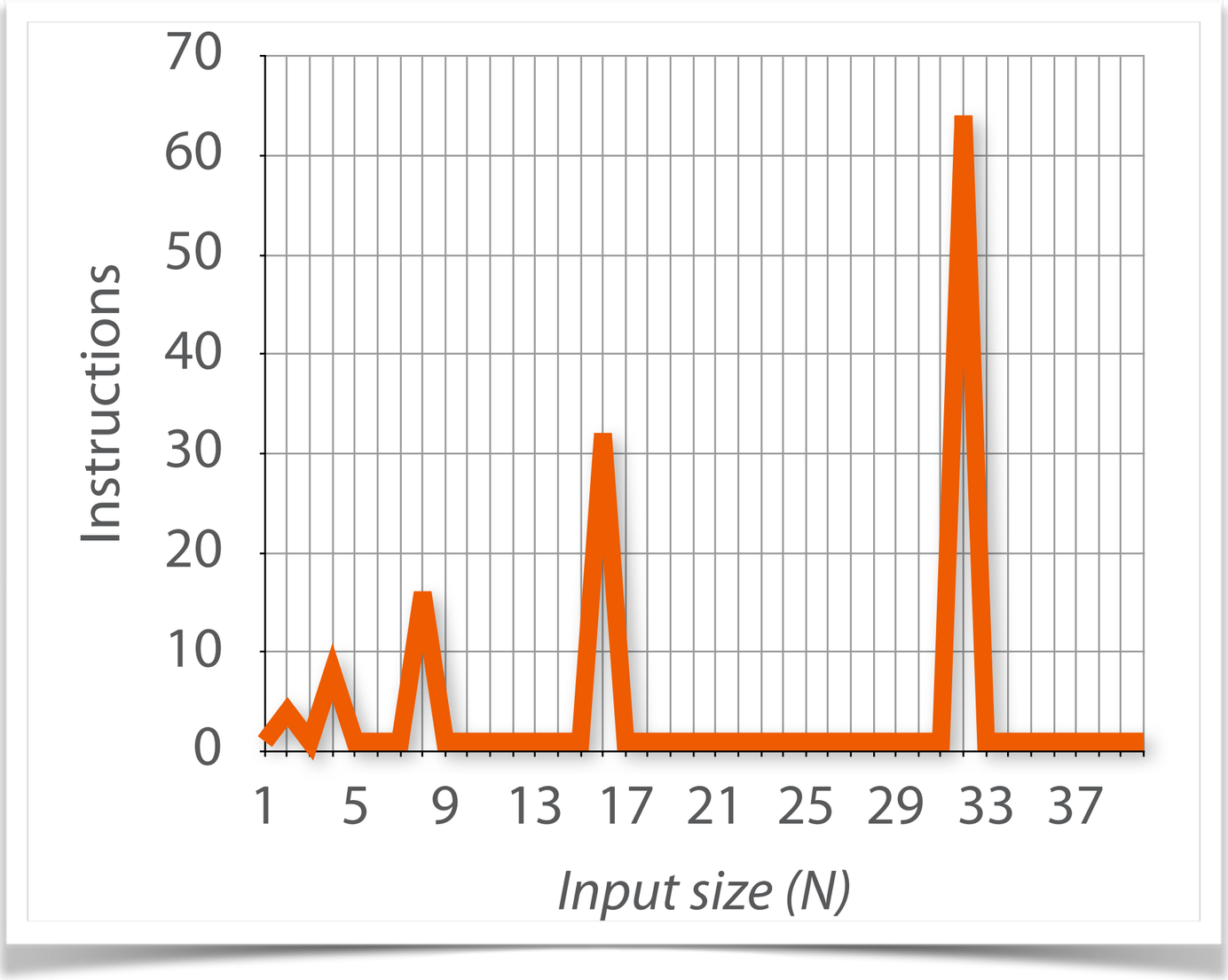


Amortized Complexity



Complexity of multiple calls

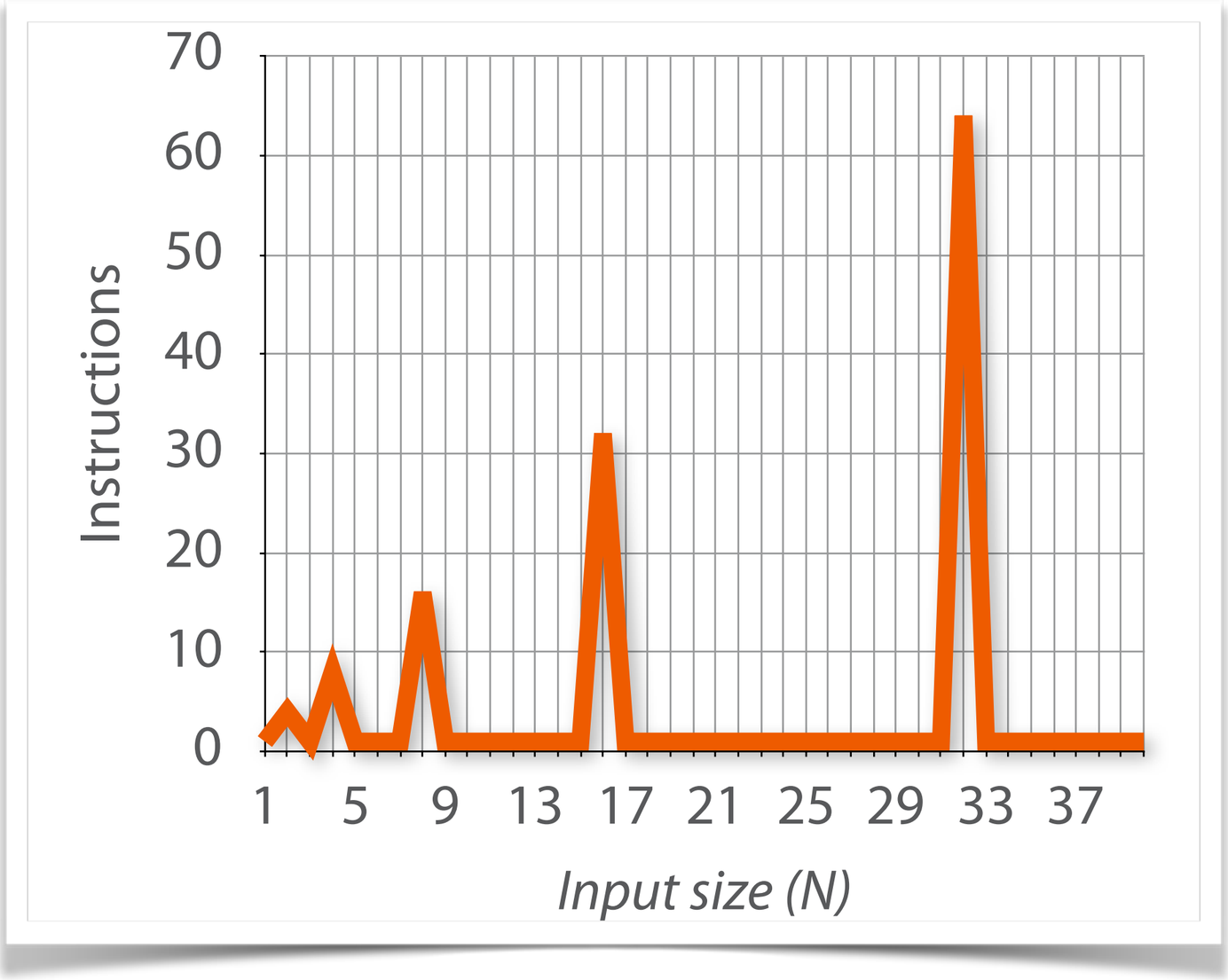
Amortized Complexity



Complexity of multiple calls

Typically: $O(1)$ per call

Amortized Complexity

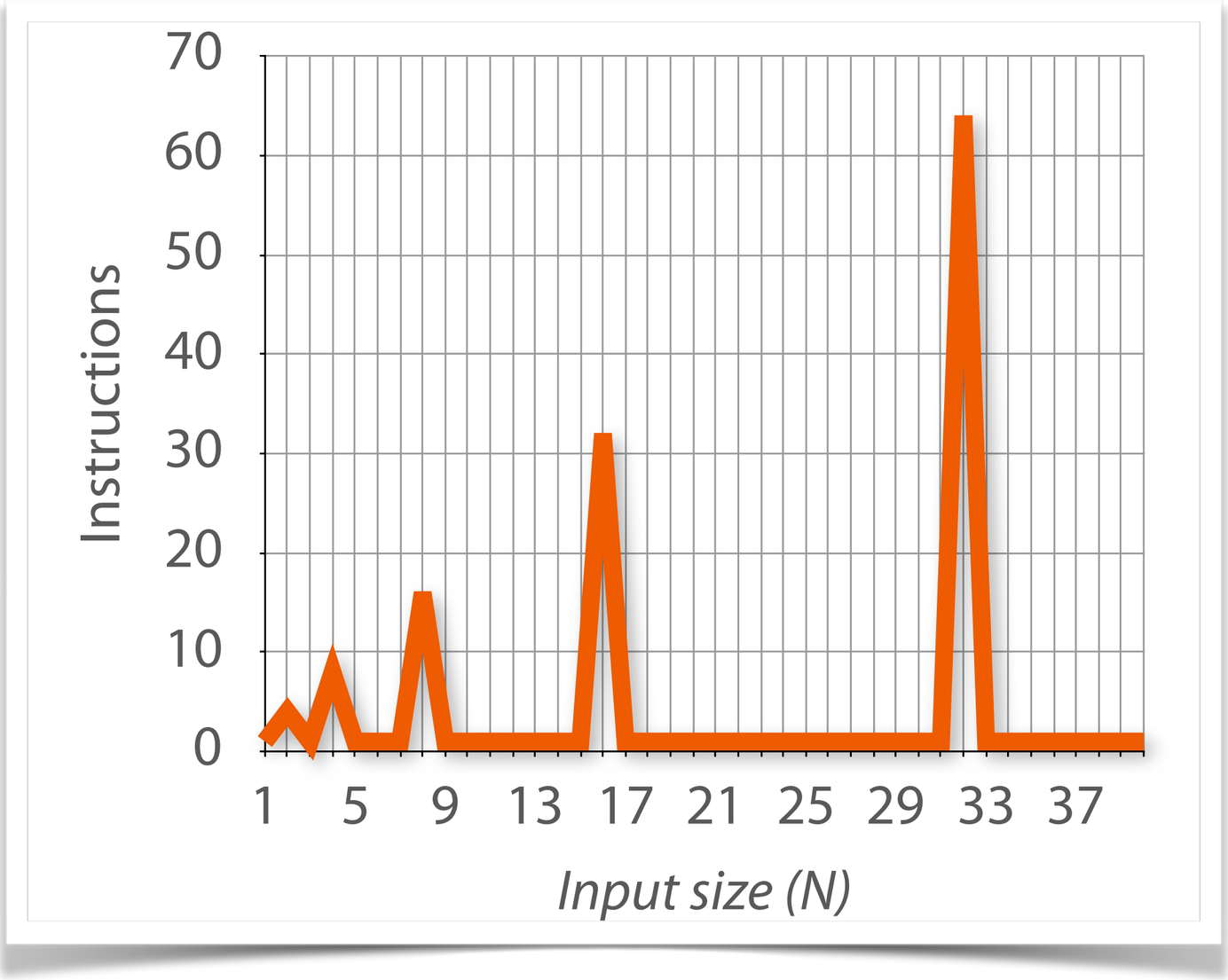


Complexity of multiple calls

Typically: $O(1)$ per call

Housekeeping: $O(N)$

Amortized Complexity



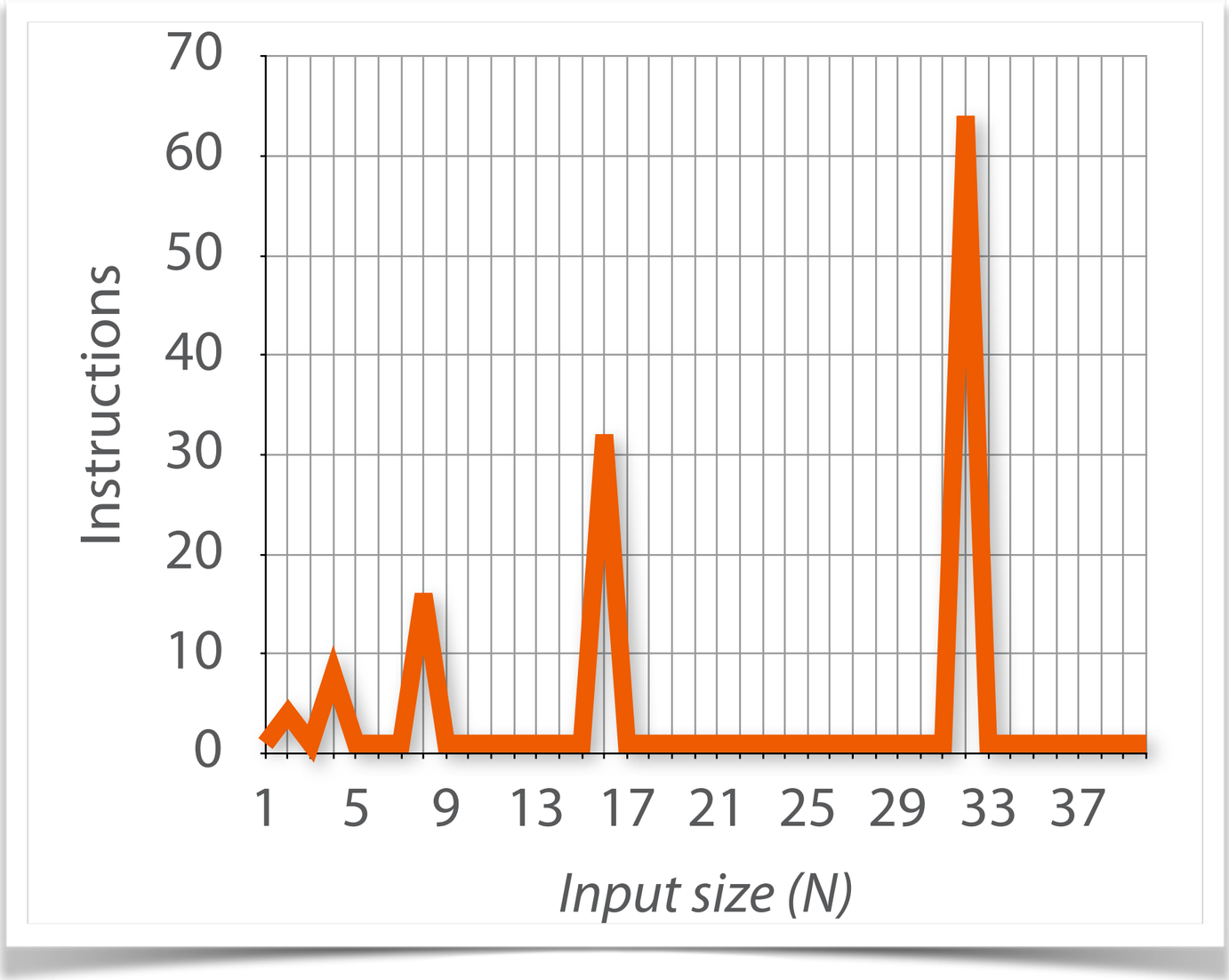
Complexity of multiple calls

Typically: $O(1)$ per call

Housekeeping: $O(N)$

M calls:

Amortized Complexity



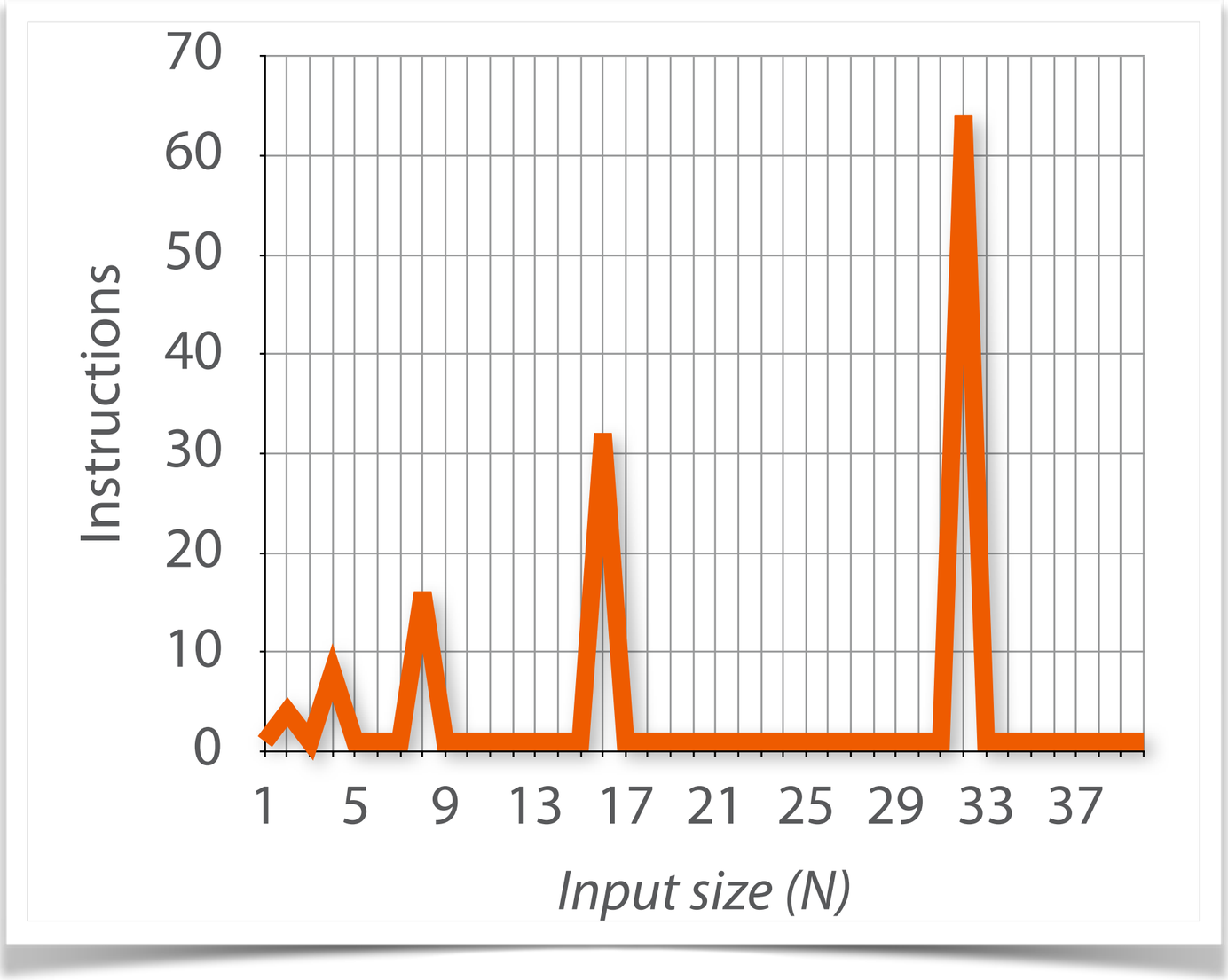
Complexity of multiple calls

Typically: $O(1)$ per call

Housekeeping: $O(N)$

M calls: $O(M)$ or $O(M \cdot N)$?

Amortized Complexity



Complexity of multiple calls

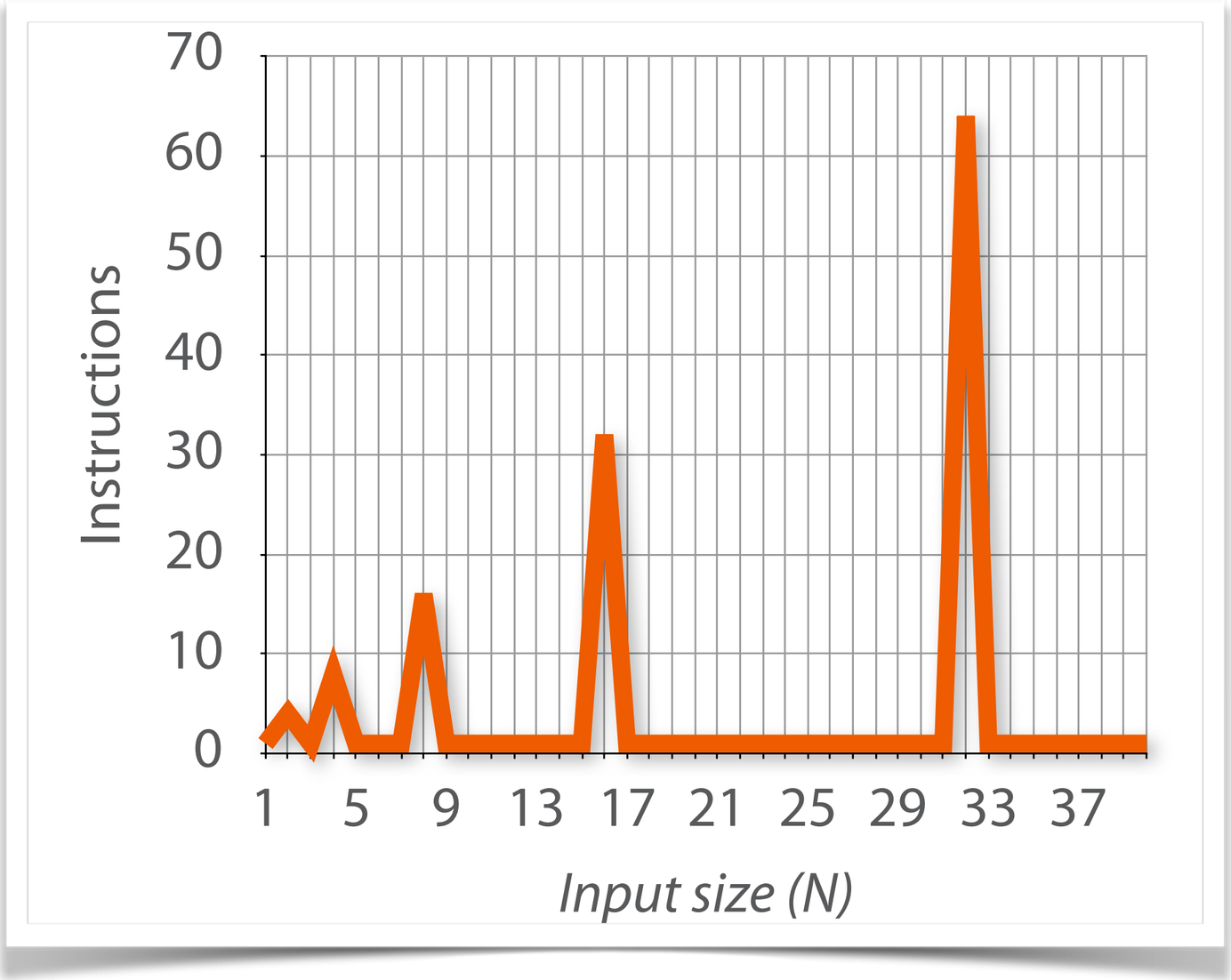
Typically: $O(1)$ per call

Housekeeping: $O(N)$

M calls: $O(M)$ or $O(M \cdot N)$?

Housekeeping frequency?

Amortized Complexity



Complexity of multiple calls

Typically: $O(1)$ per call

Housekeeping: $O(N)$

M calls: $O(M)$ or $O(M \cdot N)$?

Housekeeping frequency?

Housekeeping complexity?

Lessons Learned

Lessons Learned

Count instructions

Lessons Learned

Count instructions

Look at the *nature* of the curve

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part:
Ignore constants

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part: Ignore constants

$O(\ldots)$

$\Theta(\ldots)$

$\Omega(\ldots)$

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part:
Ignore constants

$O(\ldots)$

$\Theta(\ldots)$

$\Omega(\ldots)$

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part:
Ignore constants

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part:
Ignore constants

Recursive methods:
Fold out

Lessons Learned

Count instructions

Look at the *nature* of the curve

Hard part:
Ignore constants

Recursive methods:
Fold out

Amortized analysis:
Even out the housekeeping