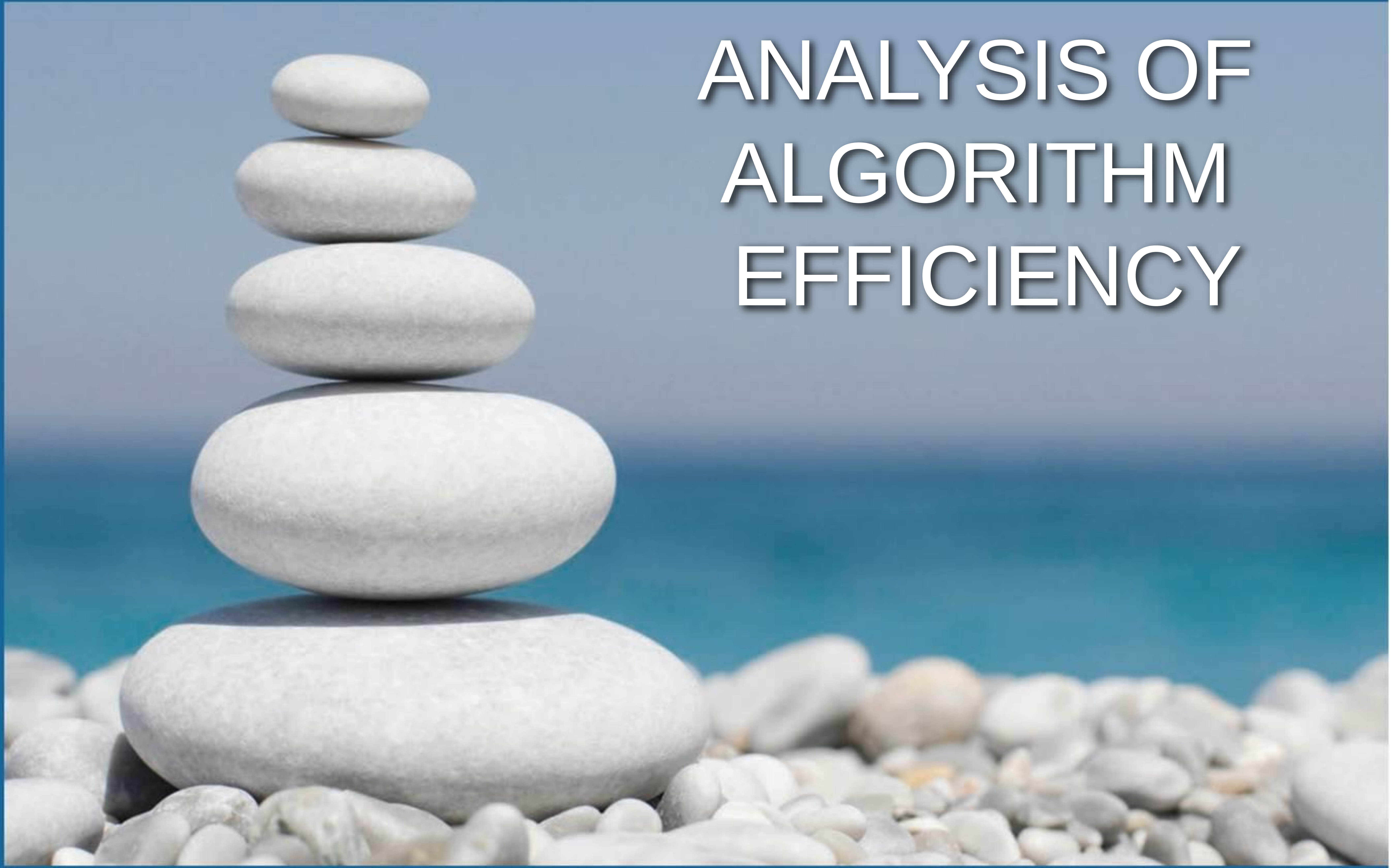
CHAPTER 11

ANALYSIS OF ALGORITHM EFFICIENCY



What makes an algorithm good?

- **Correctness**:
  Does the algorithm always return a correct solution?

- **Ease of understanding:**
  Is it easy to understand what the code is doing by just reading the code? Useful for checking correctness and for code maintenance.

- **Elegance**:
  Does it use a clever or non-obvious approach? (Example: formula for summing n consecutive integers: 1 + 2 + … + n). However, too much elegance may undermine ease of understanding ☺

- **Efficiency**:
  Does it minimize **running time** and **memory space**?

Algorithm Efficiency

The **efficiency of an algorithm** is an **estimate** of the number of **computational resources** used by the algorithm. The computational resources can be:

1. **Running time**: number of computational steps.
2. **Memory space**: amount of memory used.

Running Time

In general the running time depends on many factors, such as:

1. **Machine** used: speed of the CPU (clock), hard-disk speed, etc.
2. **Programming language** and compiler used.
3. **Implementation details**: example, binary search algorithms are faster than sequential search ones.
4. **Amount and type of input data**
    1. Processing an array with 1 million entries takes more than one with 100 entries.
    2. Sorting a pre-sorted array takes less time than an unsorted one, thus we need to distinguish among **best case, average case, and worst case** scenarios.

Algorithm Complexity Analysis

We want to define a way to *compare different algorithms running times* $T(n)$ in a way that only depends on the input size n and is **agnostic to:**

- the **machine** used,
- the **language** and compiler used,
- the **implementation** details.

Solution:

- estimate the efficiency of each algorithm **asymptotically** (for very large input size n)
- measure $T(n)$ as the number of "**elementary operations**" (e.g., additions, subtractions, multiplications, division, comparisons (e.g., <, >, =, etc.), etc..

Algorithm Complexity Analysis - *Example*

As an example, consider the following simple algorithm that only contains a for loop.

For loop	Number of elementary operations
`for ( i = 0; i < n; i++ )`	(n+1)*comparisons + (n)*additions

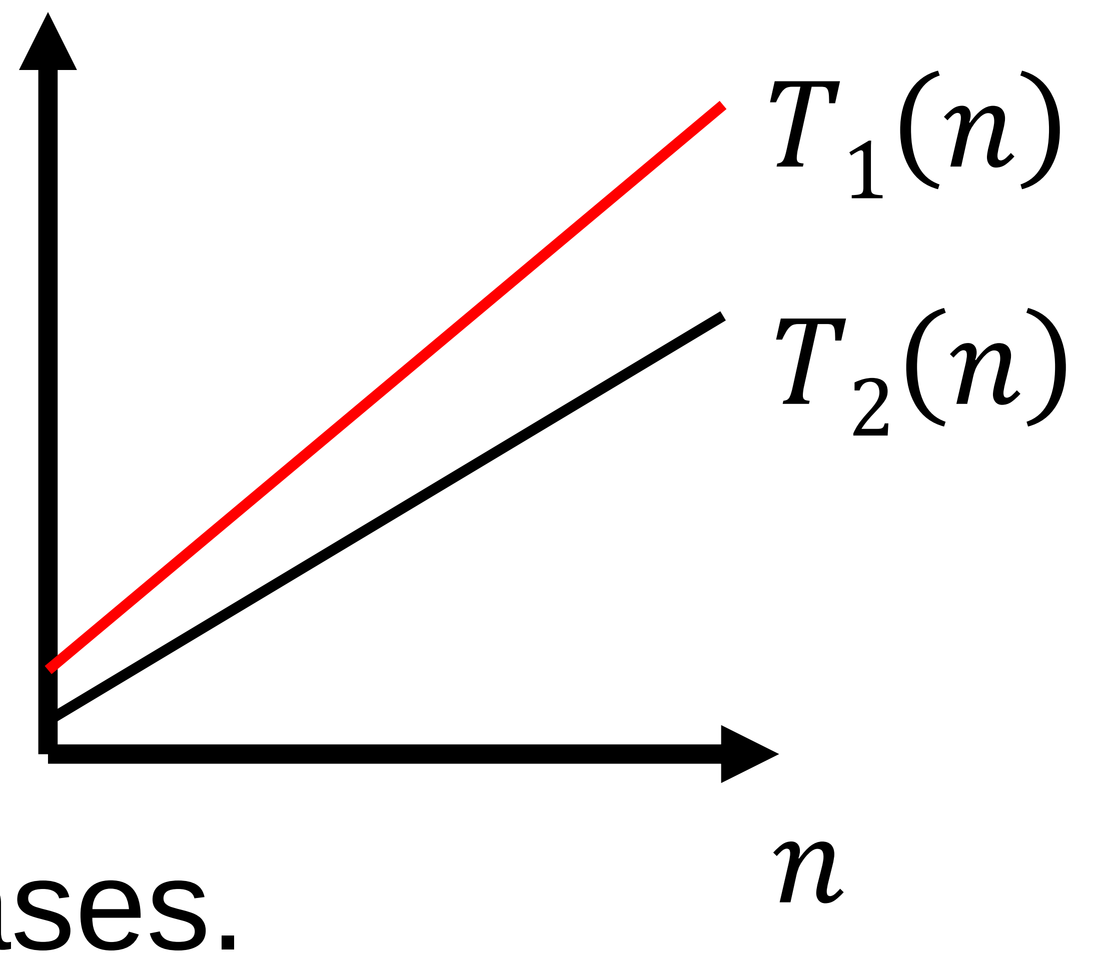
If we assume that the time taken by one addition is the same as the time by one comparison and that this is equal to c, then the running time is:

$$T(n) = c \cdot (2n + 1)$$

Algorithm Complexity Analysis - *Example*

On two different computers, these elementary operations might take different times, say c_1 and c_2, thus the running time on each PC would be

1. $T_1(n) = c_1 \cdot (2n + 1)$ and
2. $T_2(n) = c_2 \cdot (2n + 1)$, respectively.



This shows that different machines result in different slopes, but time $T(n)$ grows **linearly** as input size n increases.

The process of abstracting away details and determining the rate of increase of $T(n)$ in terms of the input size is the goal of **Algorithm Complexity Analysis**.

Asymptotic Performance

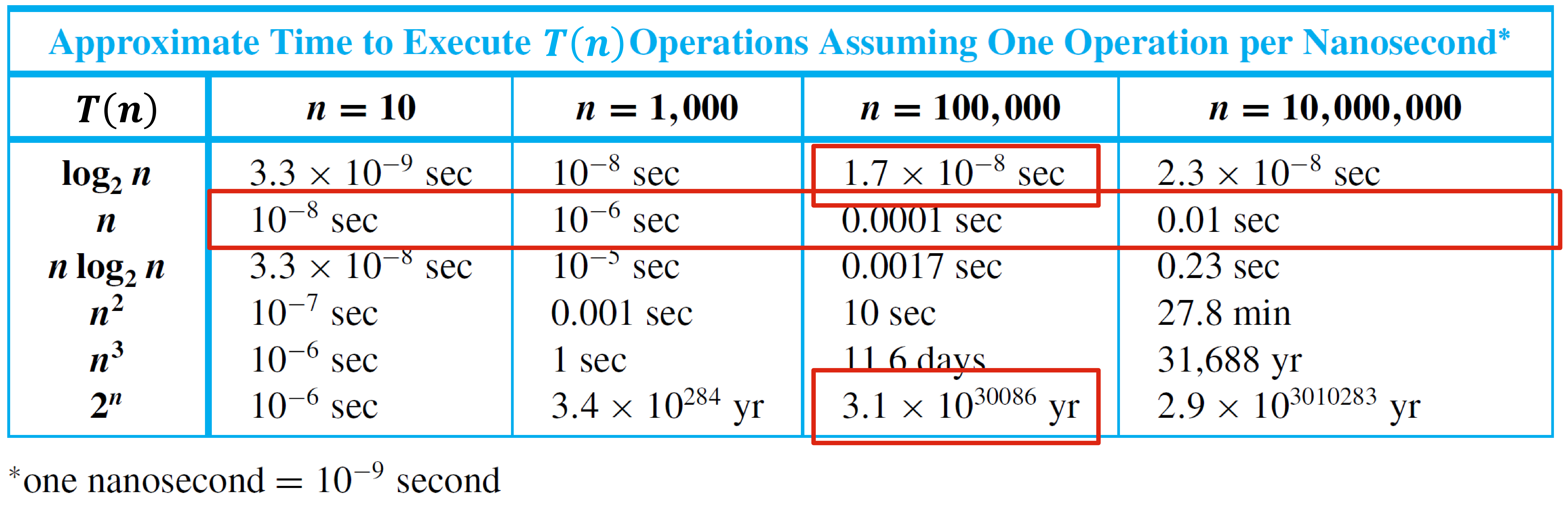
Notice that Algorithm Complexity Analysis only cares about the **asymptotic performance** of the running time:

- How does the algorithm behave as the problem size gets **very large**?

To do so, Algorithm Complexity Analysis disregards the “negligible” operations that don’t happen often and ignore multiplicative factors.

In our previous example, **we say that the algorithm complexity is linear or of order n**, written $\boldsymbol{\Theta(n)}$. $\Theta(n)$ is the class of **all algorithms** that take $An + B$ computational steps for any constants A and B.

Comparisons of some Algorithm Orders



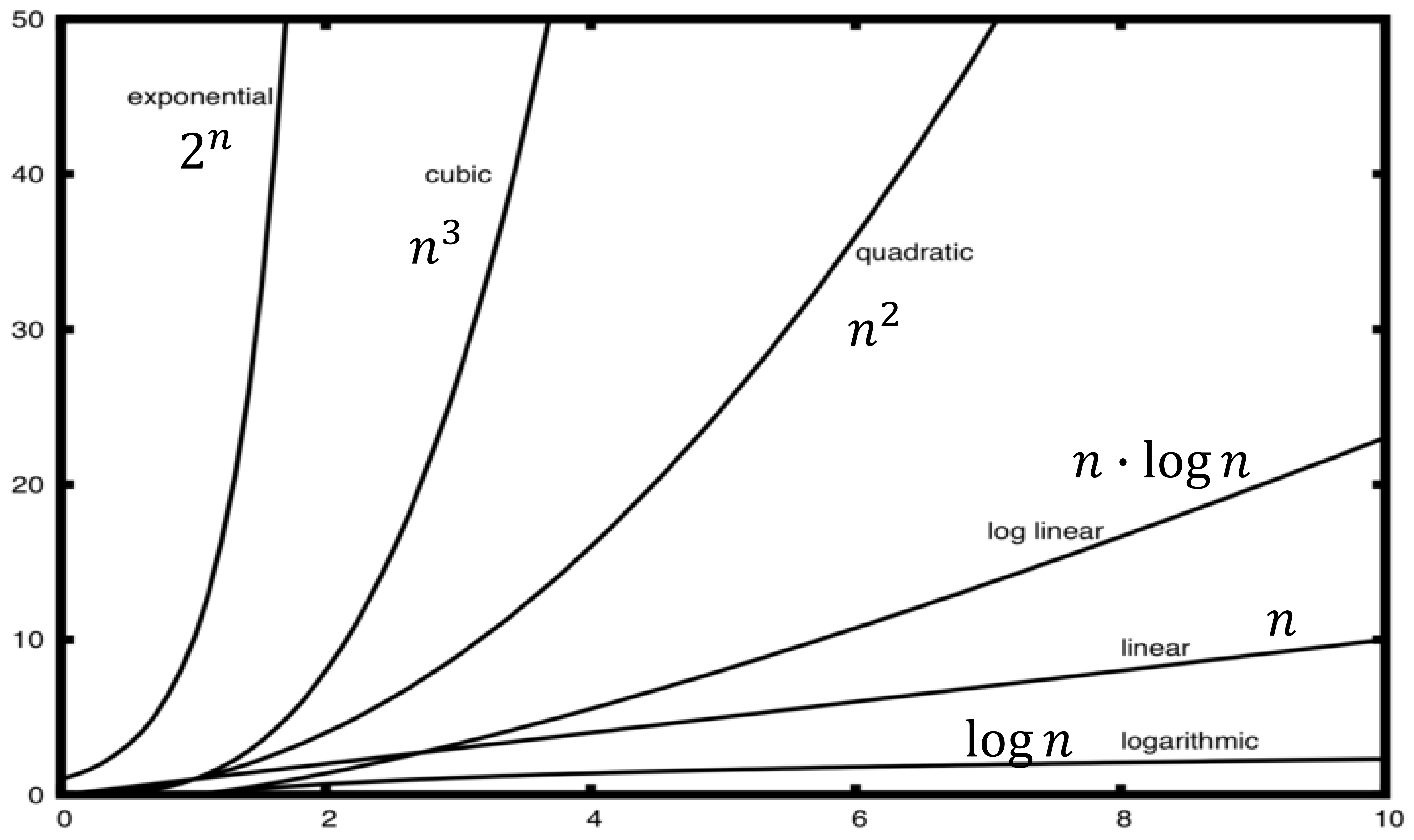
Approximate Time to Execute $T(n)$ Operations Assuming One Operation per Nanosecond*				
$T(n)$	$n = 10$	$n = 1{,}000$	$n = 100{,}000$	$n = 10{,}000{,}000$
$\log_2 n$	3.3×10^{-9} sec	10^{-8} sec	1.7×10^{-8} sec	2.3×10^{-8} sec
n	10^{-8} sec	10^{-6} sec	0.0001 sec	0.01 sec
$n \log_2 n$	3.3×10^{-8} sec	10^{-5} sec	0.0017 sec	0.23 sec
n^2	10^{-7} sec	0.001 sec	10 sec	27.8 min
n^3	10^{-6} sec	1 sec	11.6 days	31,688 yr
2^n	10^{-6} sec	3.4×10^{284} yr	3.1×10^{30086} yr	2.9×10^{3010283} yr

*one nanosecond $= 10^{-9}$ second

The time required for an algorithm of order 2^n to operate on a data set of size 100,000 is approximately $10^{30,076}$ times the estimated 15 billion years since the universe began (according to one theory of cosmology).

On the other hand, an algorithm of order $\log_2 n$ needs at most a fraction of a second to process the same data set.

Comparisons of some Algorithm Orders



SECTION 11.2

O-, Ω-, and Θ-Notations

O-, Ω-, and Θ-Notations

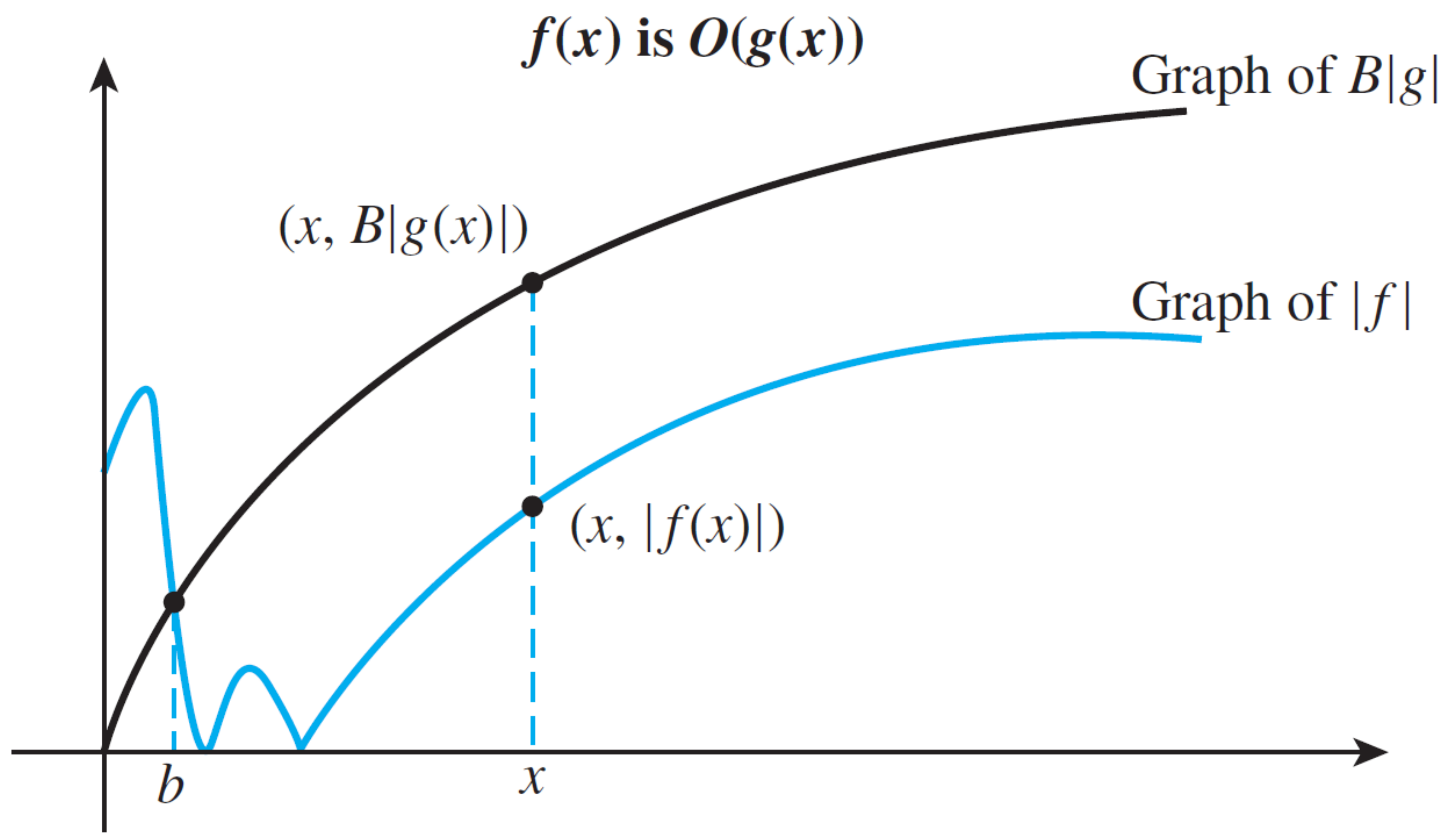
- The O-, Ω-, and Θ-notations provide approximations that make it easy to evaluate large-scale differences in algorithm efficiency.

- They provide differences in **asymptotic performance**, i.e., ignoring differences of a *constant factor* and differences that occur only for *small sets of input data* (e.g., data initialization).

O Notation

Suppose $f(x)$ and $g(x)$ are real-valued functions of a real variable x. If, for sufficiently large values of x:

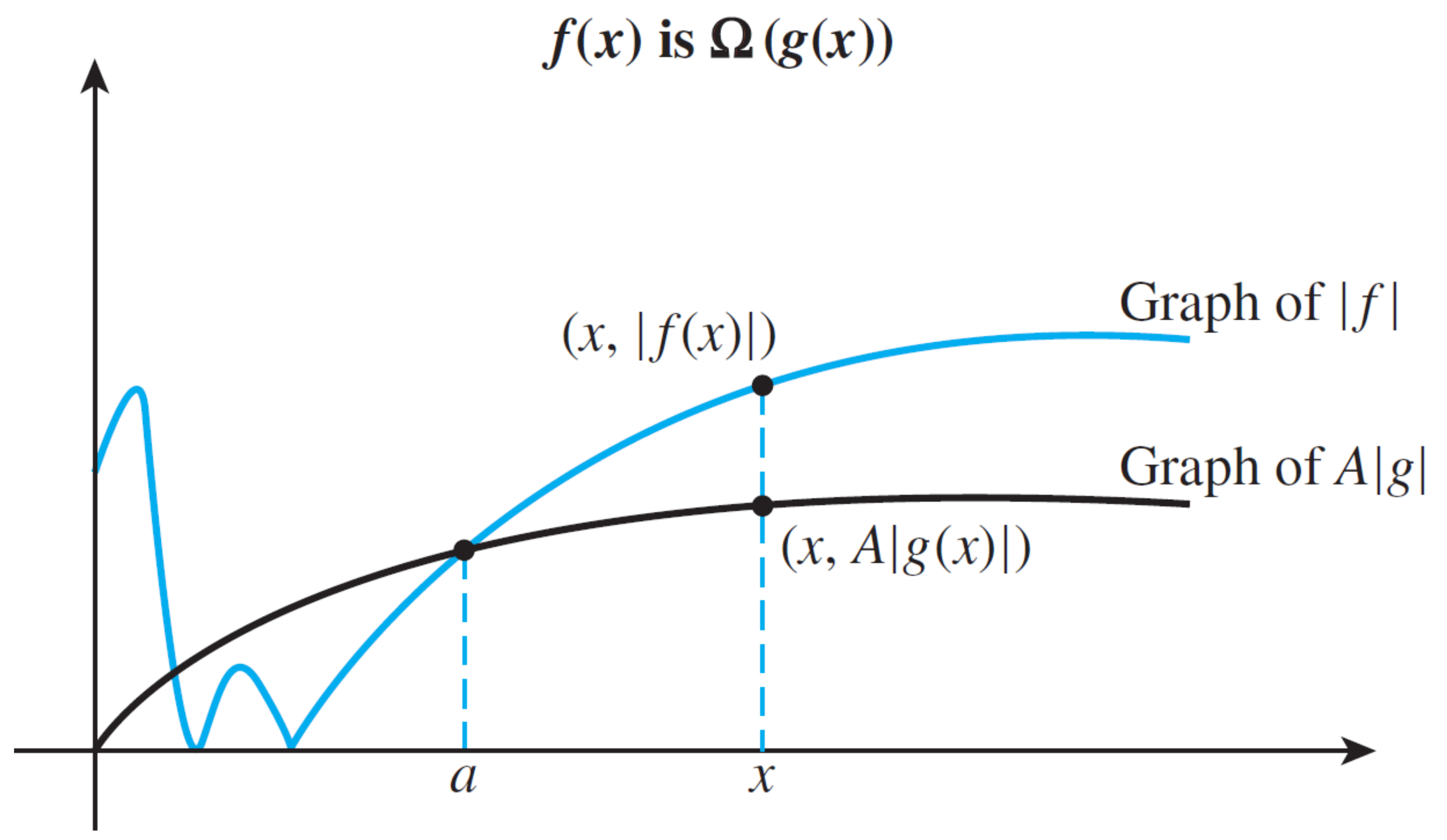
1. the values of $|f|$ are less than those of a multiple of $|g|$, then **f is of order at most g**, or $f(x)$ is $O(g(x))$. (Read O: big-O)



Ω Notation

Suppose $f(x)$ and $g(x)$ are real-valued functions of a real variable x. If, for sufficiently large values of x:

1. the values of $|f|$ are greater than those of a multiple of $|g|$, then ***f* is of order at least *g***, or $f(x)$ is $\Omega(g(x))$. (Read Ω: Omega)

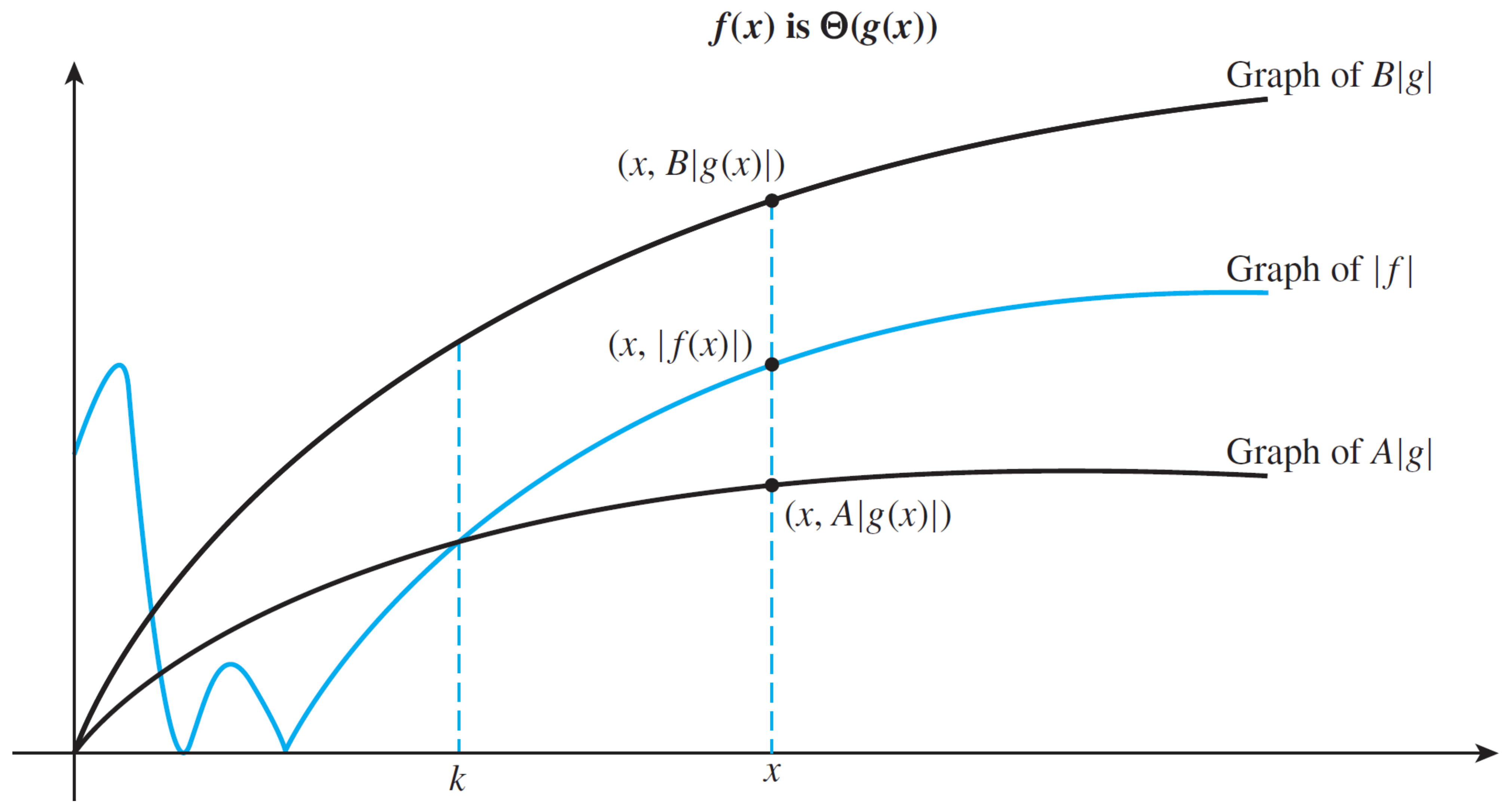


Θ Notation

Suppose $f(x)$ and $g(x)$ are real-valued functions of a real variable x. If, for sufficiently large values of x:

1. the values of $|f|$ are bounded both above and below by those of multiples of $|g|$, then **f is of order g**, or $f(x)$ is $\Theta(g(x))$. (Read Θ: Theta).

The Θ notation is the most used in computer science. If **f is of order g** it means that they behave the same asymptotically, up to a multiplicative factor.



O-, Ω-, and Θ-Notations in terms of Limits

Alternative and more practical definitions of O-, Ω-, Θ-Notations are:

1. $f(x)$ is $O(g(x))$ if $\lim_{x\to\infty}\left|\frac{f(x)}{g(x)}\right| < \infty$ (i.e., 0 or a non-zero constant)
2. $f(x)$ is $\Omega(g(x))$ if $\lim_{x\to\infty}\left|\frac{f(x)}{g(x)}\right| > 0$ (i.e., a non-0 constant or ∞)
3. $f(x)$ is $\Theta(g(x))$ if $\lim_{x\to\infty}\left|\frac{f(x)}{g(x)}\right|$ = non-zero constant

Exercise

Show that:

1. $17x^6 - 45x^3 + 2x + 8$ is $\Theta(x^6)$.

2. $2x^4 + 3x^3 + 5$ is $\Theta(x^4)$.

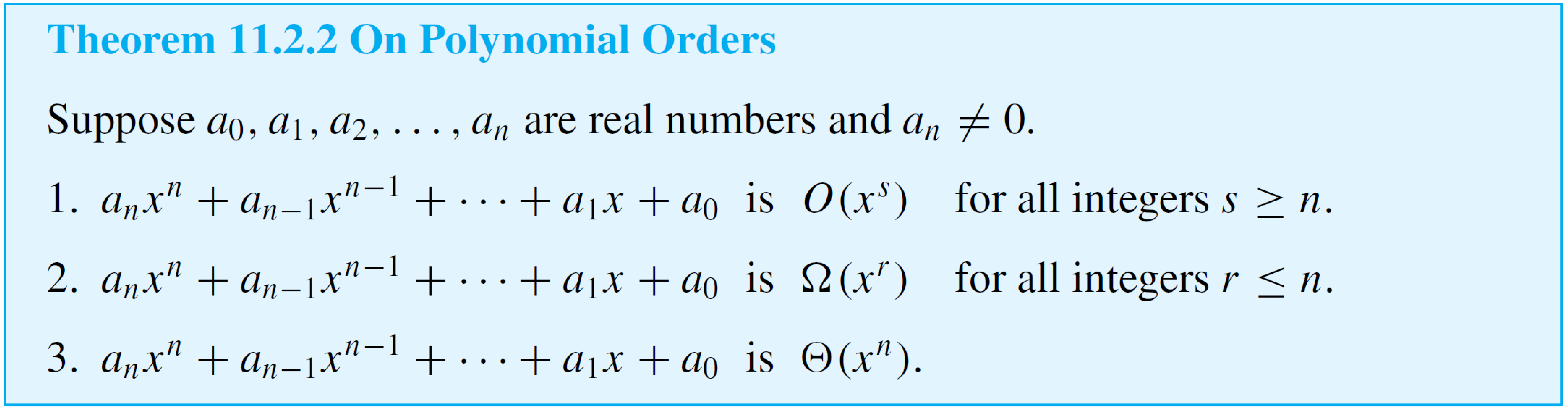
3. n^2 is $\Omega(n)$

4. $n^2 + 3\mathrm{n} + 4$ is $\Theta(n^2)$

5. $n^2 - 2\mathrm{n} + 5$ is $O(n^3)$

Orders of Polynomial Functions

As we observe, a simple way to get the Θ- notation is to drop the low-order terms and constants. Therefore:

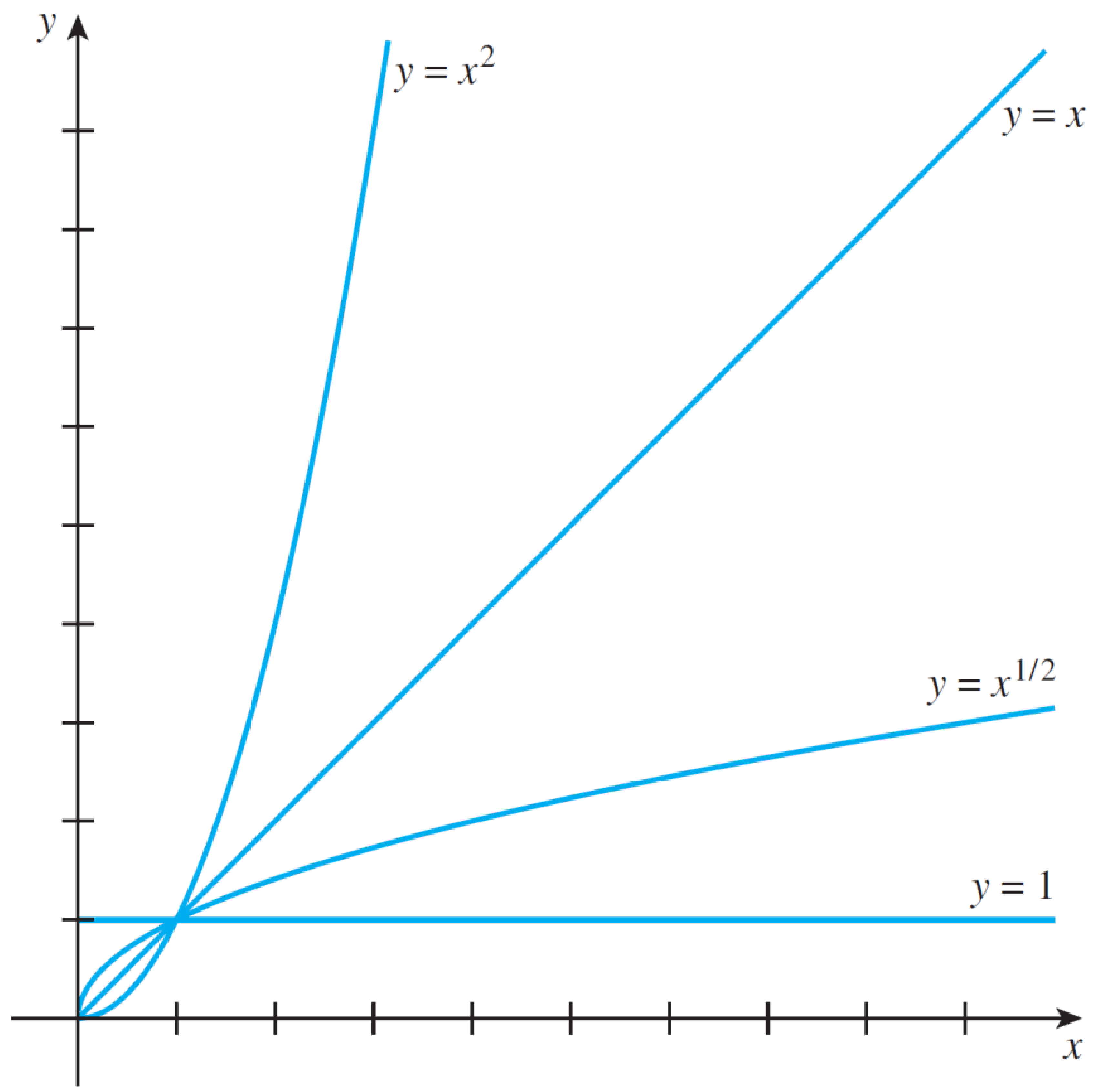


Theorem 11.2.2 On Polynomial Orders

Suppose $a_0, a_1, a_2, \ldots, a_n$ are real numbers and $a_n \neq 0$.

1. $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ is $O(x^s)$ for all integers $s \geq n$.
2. $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ is $\Omega(x^r)$ for all integers $r \leq n$.
3. $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ is $\Theta(x^n)$.

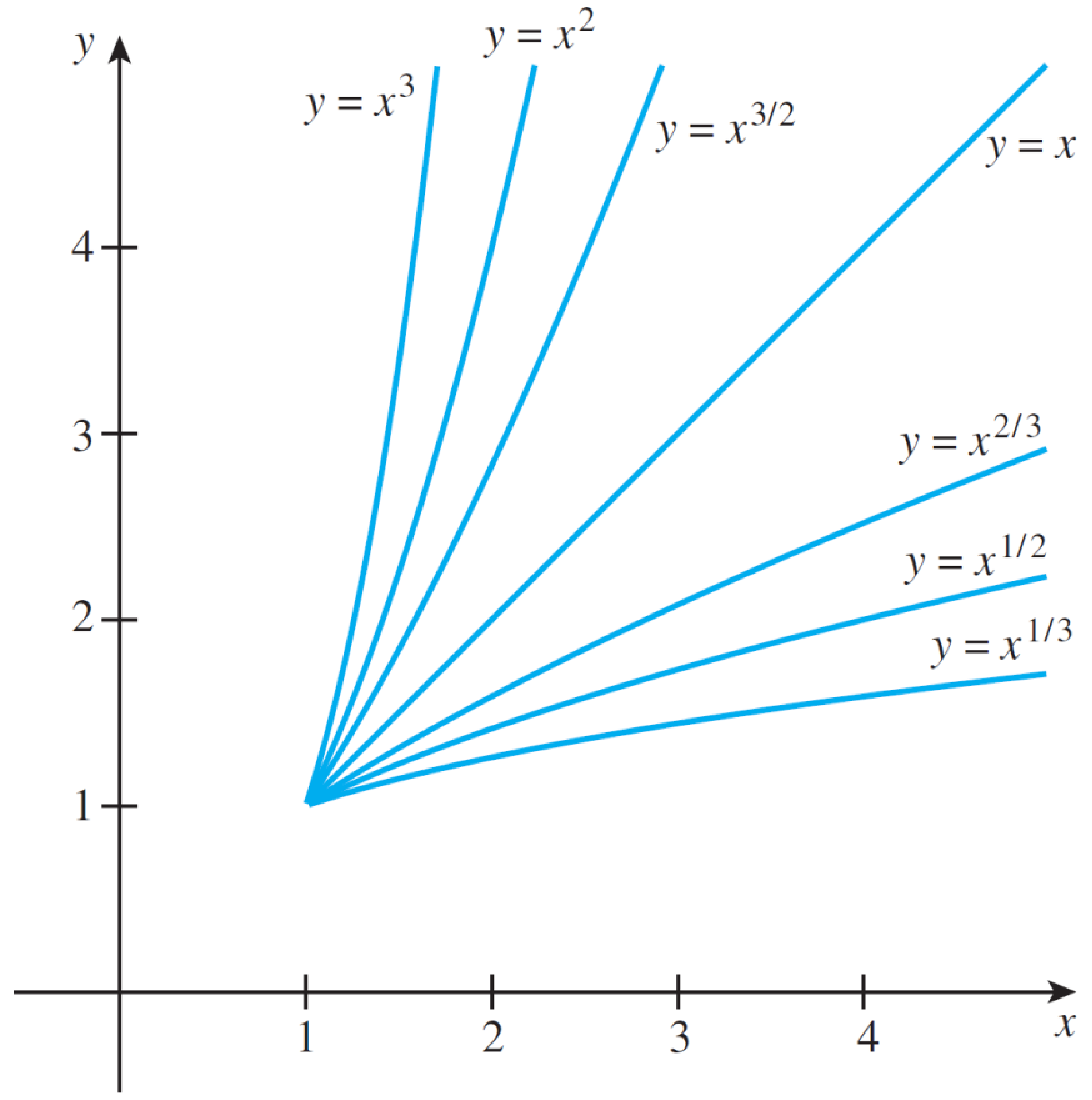
Power Functions: *Example*



Example: Orders of Power Functions

For any rational numbers r and s,

$$\text{if } x > 1 \text{ and } r < s, \text{ then } x^r < x^s. \qquad 11.2.1$$



If $r < s$, the graph of $y = x^r$ lies underneath the graph of $y = x^s$ for $x > 1$.

Example – *An Order for the Sum of the First n Integers*

Sums of the form $1 + 2 + 3 + \cdots + n$ arise in the analysis of computer algorithms such as Selection Sort.

Show that for a positive integer variable n:

$$1 + 2 + 3 + \cdots + n \quad \text{is} \quad \Theta(n^2).$$

Example – *An Order for the Sum of the First n Integers*

Solution:

By the formula for the sum of the first n integers:

Theorem 5.2.2 Sum of the First *n* Integers

For all integers $n \geq 1$,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

And thus: $\frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$, which is $\Theta(n^2)$.

Exercise: Rational Power Functions

Show that:

1. $\frac{15\sqrt{x}(2x+9)}{x+1}$ is $\Theta(\sqrt{x})$

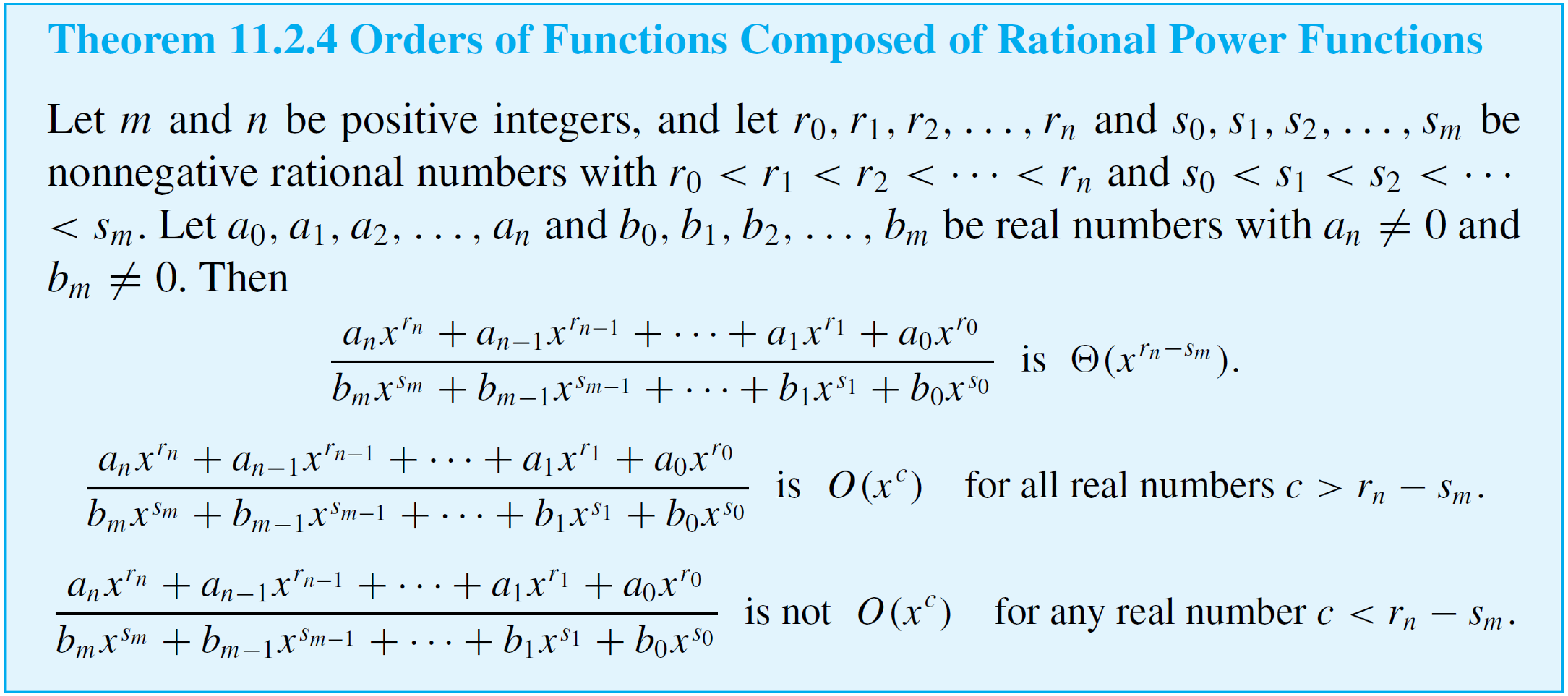
2. $\frac{2x^4-2x^3+3x^2+\sqrt{x}(2x+9)}{x^3+x+1}$ is $\Theta(x)$

3. $\frac{x^4-5x^3+3x^2}{x^3+x+1}$ is $\mathrm{O}(x^2)$

4. $\frac{2x^4-5x^3+3x^2}{5x^4+x+1}$ is $\Theta(1)$

Extension to Functions Composed of Rational Power Functions

As we observe, a simple way to get the Θ- notation is to drop the low-order terms and constants in the nominator and denominator. Therefore:



Theorem 11.2.4 Orders of Functions Composed of Rational Power Functions

Let m and n be positive integers, and let $r_0, r_1, r_2, \ldots, r_n$ and $s_0, s_1, s_2, \ldots, s_m$ be nonnegative rational numbers with $r_0 < r_1 < r_2 < \cdots < r_n$ and $s_0 < s_1 < s_2 < \cdots < s_m$. Let $a_0, a_1, a_2, \ldots, a_n$ and $b_0, b_1, b_2, \ldots, b_m$ be real numbers with $a_n \neq 0$ and $b_m \neq 0$. Then

$$\frac{a_n x^{r_n} + a_{n-1}x^{r_{n-1}} + \cdots + a_1 x^{r_1} + a_0 x^{r_0}}{b_m x^{s_m} + b_{m-1}x^{s_{m-1}} + \cdots + b_1 x^{s_1} + b_0 x^{s_0}} \quad \text{is} \quad \Theta(x^{r_n - s_m}).$$

$$\frac{a_n x^{r_n} + a_{n-1}x^{r_{n-1}} + \cdots + a_1 x^{r_1} + a_0 x^{r_0}}{b_m x^{s_m} + b_{m-1}x^{s_{m-1}} + \cdots + b_1 x^{s_1} + b_0 x^{s_0}} \quad \text{is} \quad O(x^c) \quad \text{for all real numbers } c > r_n - s_m.$$

$$\frac{a_n x^{r_n} + a_{n-1}x^{r_{n-1}} + \cdots + a_1 x^{r_1} + a_0 x^{r_0}}{b_m x^{s_m} + b_{m-1}x^{s_{m-1}} + \cdots + b_1 x^{s_1} + b_0 x^{s_0}} \quad \text{is not} \quad O(x^c) \quad \text{for any real number } c < r_n - s_m.$$

SECTION 11.3

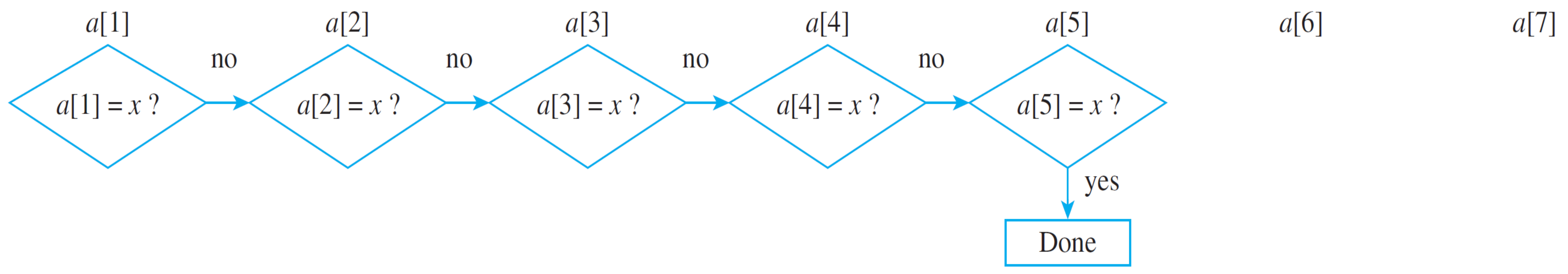
Application: Analysis of Algorithm Efficiency I

The Sequential Search Algorithm

Examine an array of data in an attempt to find a particular item x.

In a sequential search, x is compared to the first item in the array, then to the second, then to the third, and so on.

The search is stopped if a match is found at any stage.



Sequential Search of $a[1], a[2], \ldots, a[7]$ for x where $x = a[5]$

Example – *Best- and Worst-Case Orders for Sequential Search*

Find best- and worst-case orders for the sequential search algorithm from among the set of power functions.

Solution:

In the best case, the algorithm requires only one comparison between x and the first item $a[1]$.

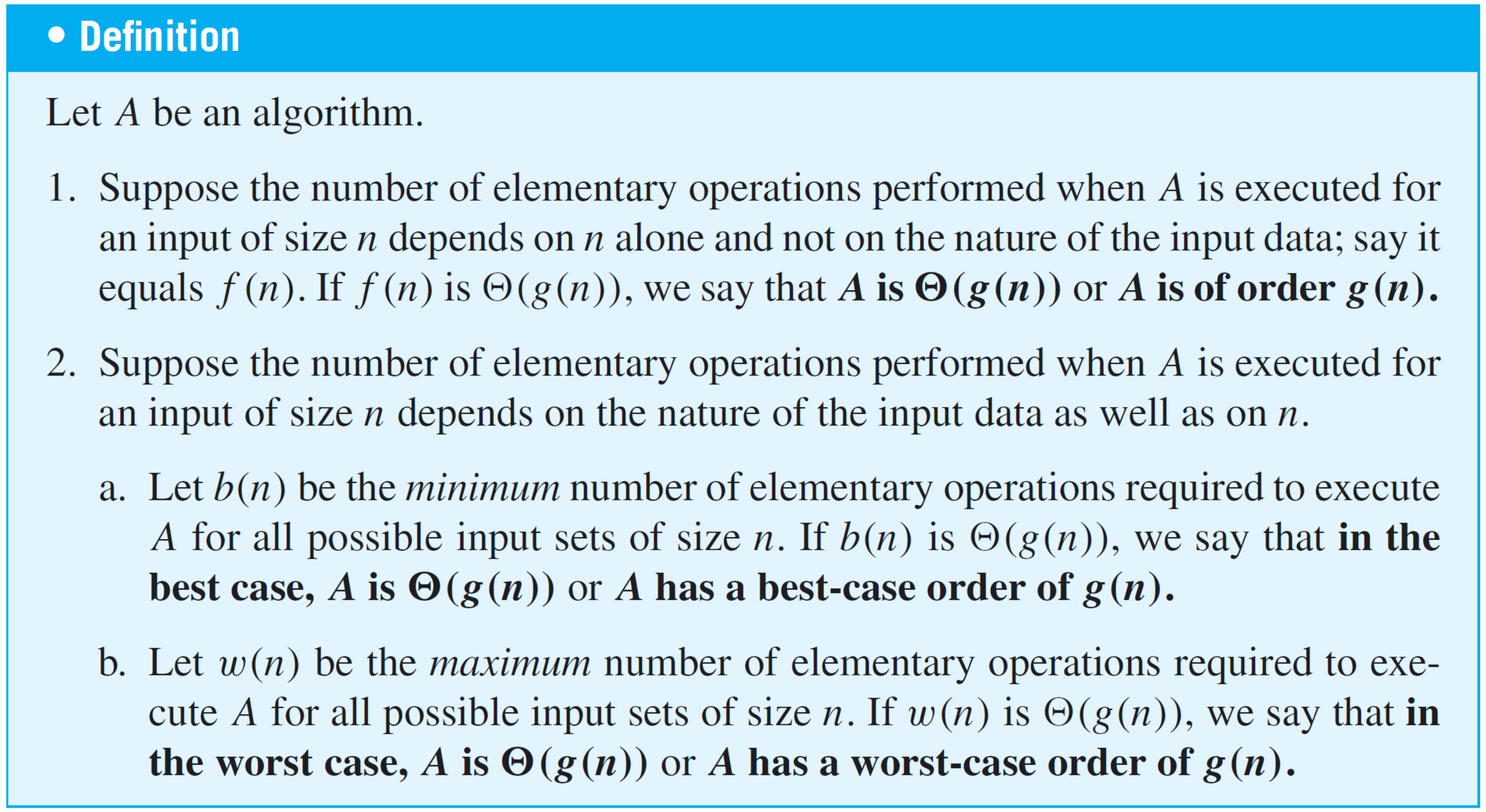
$\Rightarrow \Theta(1) = \Theta(n^0)$ **constant time**

In the worst case, the algorithm requires to compare x with all n items to find x being $a[n]$ or not in the array at all.

$\Rightarrow \Theta(n) = \Theta(n^1)$ **linear time**

Time Efficiency of an Algorithm

Since each elementary operation is executed in time no longer than the slowest, the time efficiency of an algorithm is approximately proportional to the number of elementary operations required to execute the algorithm.



• **Definition**

Let A be an algorithm.

1. Suppose the number of elementary operations performed when A is executed for an input of size n depends on n alone and not on the nature of the input data; say it equals $f(n)$. If $f(n)$ is $\Theta(g(n))$, we say that **A is $\Theta(g(n))$** or **A is of order $g(n)$.**

2. Suppose the number of elementary operations performed when A is executed for an input of size n depends on the nature of the input data as well as on n.

   a. Let $b(n)$ be the *minimum* number of elementary operations required to execute A for all possible input sets of size n. If $b(n)$ is $\Theta(g(n))$, we say that **in the best case, A is $\Theta(g(n))$** or **A has a best-case order of $g(n)$.**

   b. Let $w(n)$ be the *maximum* number of elementary operations required to execute A for all possible input sets of size n. If $w(n)$ is $\Theta(g(n))$, we say that **in the worst case, A is $\Theta(g(n))$** or **A has a worst-case order of $g(n)$.**

Exercise – *An Order for an Algorithm with a Nested Loop*

a. Compute the number of additions, subtractions, and multiplications that are performed when this algorithm segment is executed.

b. Find an order for this algorithm segment.

```
$s := 0$
for $i := 1$ to $n$
    for $j := 1$ to $i$
        $s := s + j \cdot (i - j + 1)$
    next $j$
next $i$
```

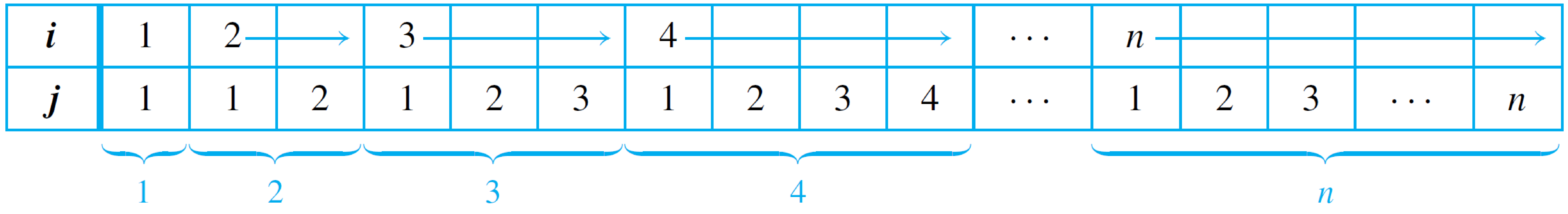
Solution – *An Order for an Algorithm with a Nested Loop*

a. There are **2 additions**, **1 multiplication**, and **1 subtraction** for each iteration of the inner loop, so the total number of additions, multiplications, and subtractions is **4**, times the number of iterations of the inner loop. (NB: we discard the comparisons and additions to run each for loop).

Number of iterations of the inner loop:

- 1 time when $i = 1$,
- 2 times when $i = 2$,
- 3 times when $i = 3$,
- …
- n times when $i = $ n.

```
s := 0
for i := 1 to n
    for j := 1 to i
        s := s + j·(i − j + 1)
    next j
next i
```



i	1	2		3			4				⋯	n				
j	1	1	2	1	2	3	1	2	3	4	⋯	1	2	3	⋯	n

$$\Rightarrow 1 + 2 + 3 + \cdots + n$$

Solution – *An Order for an Algorithm with a Nested Loop* cont’d

Hence the total number of iterations of the inner loop is

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

and so the number of operations is

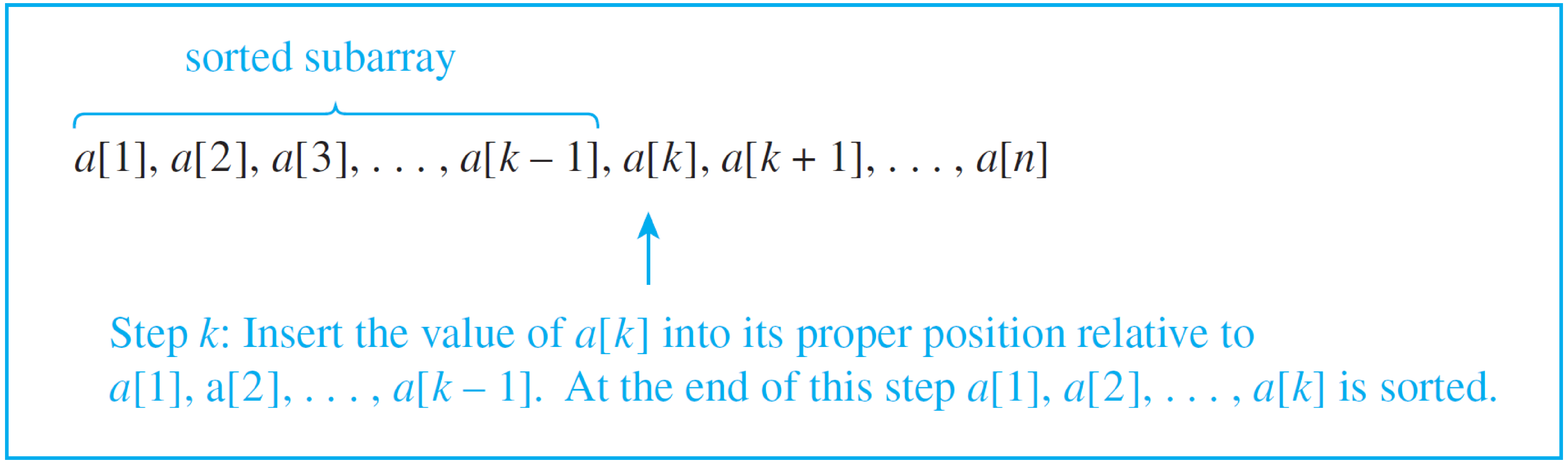
$$4 \cdot \frac{n(n+1)}{2} = 2n(n+1).$$

b. Polynomial order: $2n\,(n+1) = 2n^2 + 2n$ which is $\boldsymbol{\Theta(n^2)}$.
Thus the algorithm is $\Theta(n^2)$.

The Insertion Sort Algorithm

Sort algorithms arrange items in an array into ascending order.

Starting with the first item as the initial known sorted subarray, the next element of the remaining unsorted subarray is inserted into the correct position relative to the preceding ones.



$$\overbrace{a[1], a[2], a[3], \ldots, a[k-1]}^{\text{sorted subarray}}, a[k], a[k+1], \ldots, a[n]$$

Step k: Insert the value of $a[k]$ into its proper position relative to $a[1]$, a[2], . . . , $a[k-1]$. At the end of this step $a[1], a[2], \ldots, a[k]$ is sorted.

Step *k* of Insertion Sort

The Insertion Sort Algorithm

Algorithm Body:

- Compare $a[k]$ to previous sorted elements $a[1], a[2], \ldots, a[k-1]$, starting from the largest and moving downward. If $a[k]$ is greater than or equal to $a[k-1]$ then leave array unchanged.
- Whenever $a[k]$ is less than a preceding array item $a[j]$, copy that element one position to the right, to $a[j+1]$.
- As soon as $a[k]$ is greater than or equal to $a[j]$, insert $a[k]$ to the right of that, thus to $a[j+1]$.

$$\overbrace{a[1], a[2], a[3], \ldots, a[k-1]}^{\text{sorted subarray}}, \underset{\uparrow}{a[k]}, a[k+1], \ldots, a[n]$$

Step k: Insert the value of $a[k]$ into its proper position relative to $a[1]$, a$[2], \ldots, a[k-1]$. At the end of this step $a[1], a[2], \ldots, a[k]$ is sorted.

Step *k* of Insertion Sort

The Insertion Sort Algorithm

```
**for** $k := 2$ **to** $n$
    $x := a[k]$
    $j := k - 1$
    **while** $(j \neq 0)$
     **if** $x < a[j]$ **then**
        $a[j + 1] := a[j]$
        $j := j - 1$
    **end if**
    **end while**
    $a[j + 1] := x$
**next** $k$
```

Output: $a[1], a[2], a[3], \ldots, a[n]$ (in ascending order)

Exercise – *Finding Worst-Case Order for Insertion Sort*

a. What is the maximum number of comparisons that are performed when insertion sort is applied to the array $a[1], a[2], a[3], \ldots, a[n]$?

b. Find the worst-case polynomial order for insertion sort.

```
for $k := 2$ to $n$
    $x := a[k]$
    $j := k - 1$
    while $(j \neq 0)$
     if $x < a[j]$ then
        $a[j + 1] := a[j]$
        $j := j - 1$
    end if
    end while
    $a[j + 1] := x$
next $k$
```

Solution:

a. In each iteration of the inner while loop, **2 explicit comparisons** are made: test $j \neq 0$ and test $x < a[j]$.

To find where $a[k]$ belongs within the sorted subarray $a[1], \ldots, a[k-1]$, the maximum number of iterations of the while loop is k. When $a[k]$ is less then any of these elements, $j = 0$, it is placed at $a[1]$.

$$\overbrace{a[1], a[2], a[3], \ldots, a[k-1]}^{\text{sorted subarray}}, a[k], a[k+1], \ldots, a[n]$$

Thus the **maximum number of comparisons** for a given value of k is $\mathbf{2}\boldsymbol{k}$.

Solution – *Finding Worst-Case Order for Insertion Sort* cont'd

Because k goes from 2 to n ($a[1]$ alone is sorted), it follows that the maximum number of comparisons occurs when the items in the array are in reverse order, thus, considering 2 comparisons for each value of k:

$$\begin{aligned}
2\cdot 2 + 2\cdot 3 + \cdots + 2\cdot n &= 2(2+3+\cdots+n) && \text{by factoring out the 2} \\
&= 2[(1+2+3+\cdots+n)-1] && \text{by adding and subtracting 1} \\
&= 2\left(\frac{n(n+1)}{2}-1\right) && \text{by Theorem 5.2.2} \\
&= n(n+1)-2 \\
&= n^2+n-2 && \text{by algebra.}
\end{aligned}$$

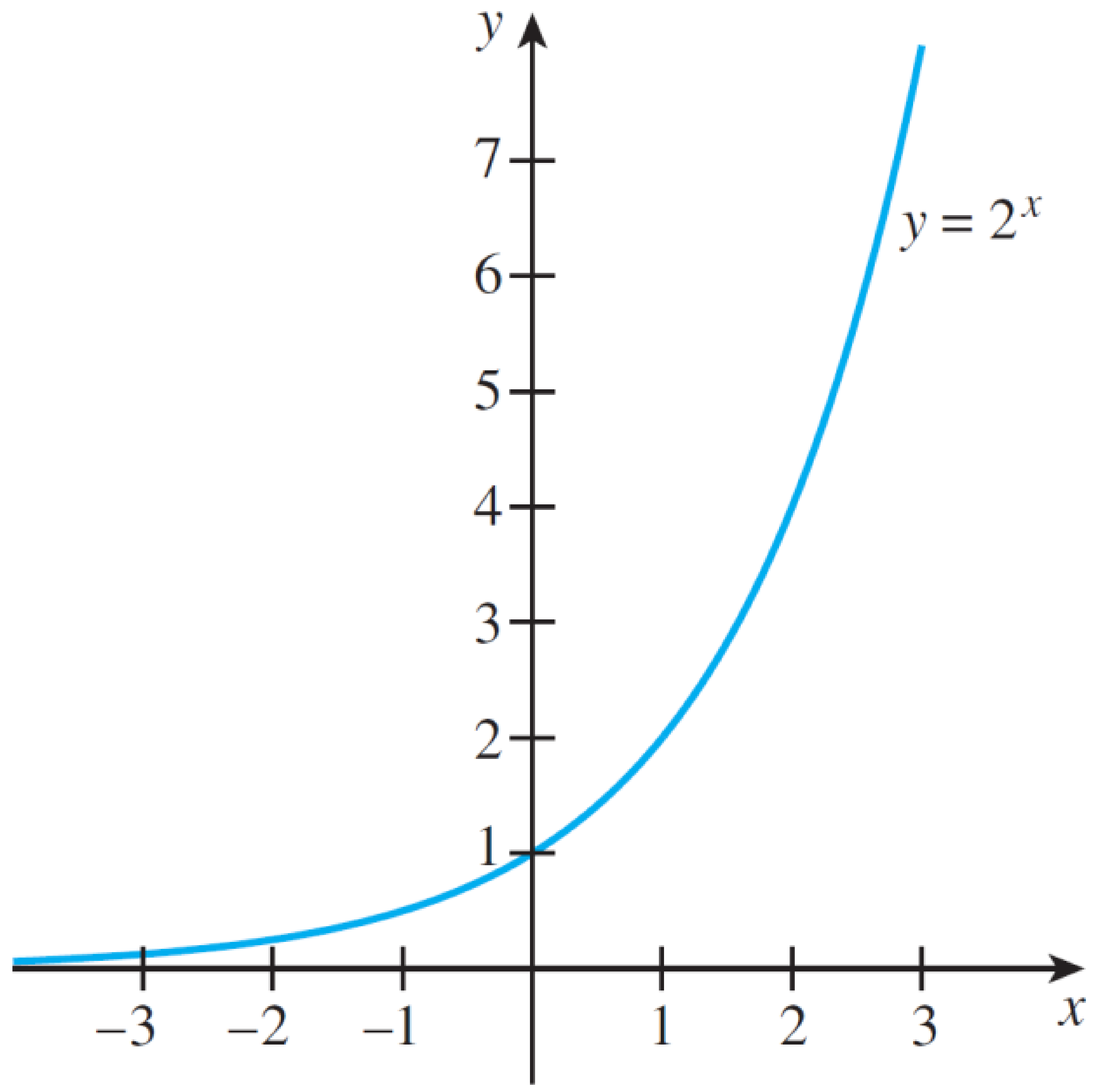
b. By the theorem on polynomial orders, $2n\,(n+1) = 2n^2 + 2n$ is $\Theta(n^2)$, and so the **Insertion Sort algorithm** has **worst-case order $\boldsymbol{\Theta(n^2)}$**.

SECTION 11.4

Exponential and Logarithmic Functions: Graphs and Orders

Graphs of Exponential Functions

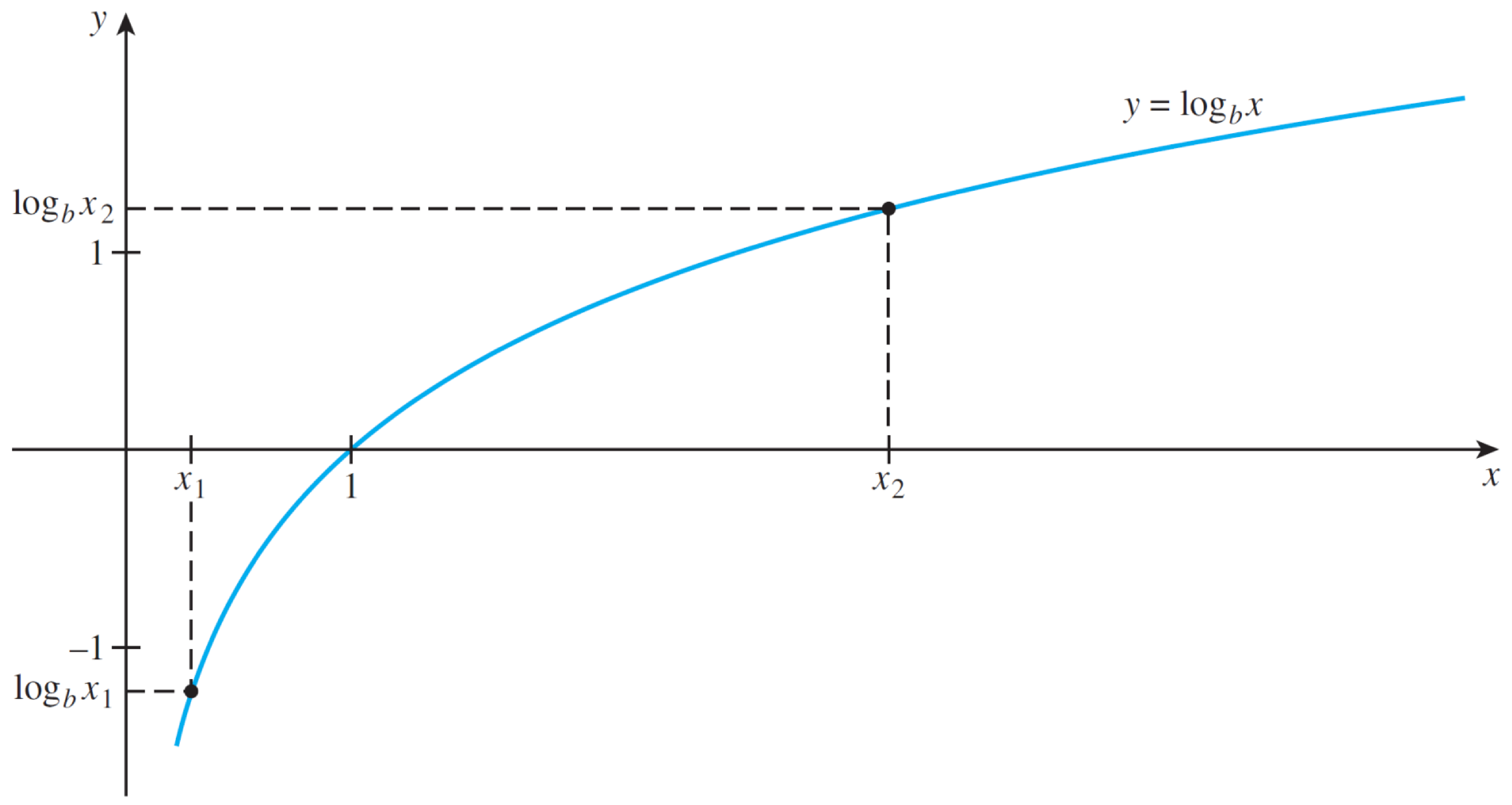
The graph of any exponential function with base $b > 1$ is similar to 2^x.



The Exponential Function with Base 2

Graphs of Logarithmic Functions

The graph of the logarithmic function with base $b > 0$ and $b \neq 1$ is:



The Graph of the Logarithmic Function with Base $b > 1$

Exponential and Logarithmic Orders

Ratios of logarithm to power as well as power to exponential functions:

These have the following implications for O-notation.

For all real numbers b and r with $b > 1$ and $r > 0$,

$\log_b x \leq x^r$ for all sufficiently large real numbers x. 11.4.9

and $x^r \leq b^x$ for all sufficiently large real numbers x. 11.4.10

For all real numbers b and r with $b > 1$ and $r > 0$,

$\log_b x$ is $O(x^r)$ 11.4.11

and x^r is $O(b^x)$ 11.4.12

Exponential and Logarithmic Orders

Another important function in the analysis of algorithms is the function f defined by the formula $f(x) = x \log_b x$.

For large values of x, $f(x)$ fits in between the graph of the identity function x and the squaring function x^2.

The O-notation versions of these facts are as follows:

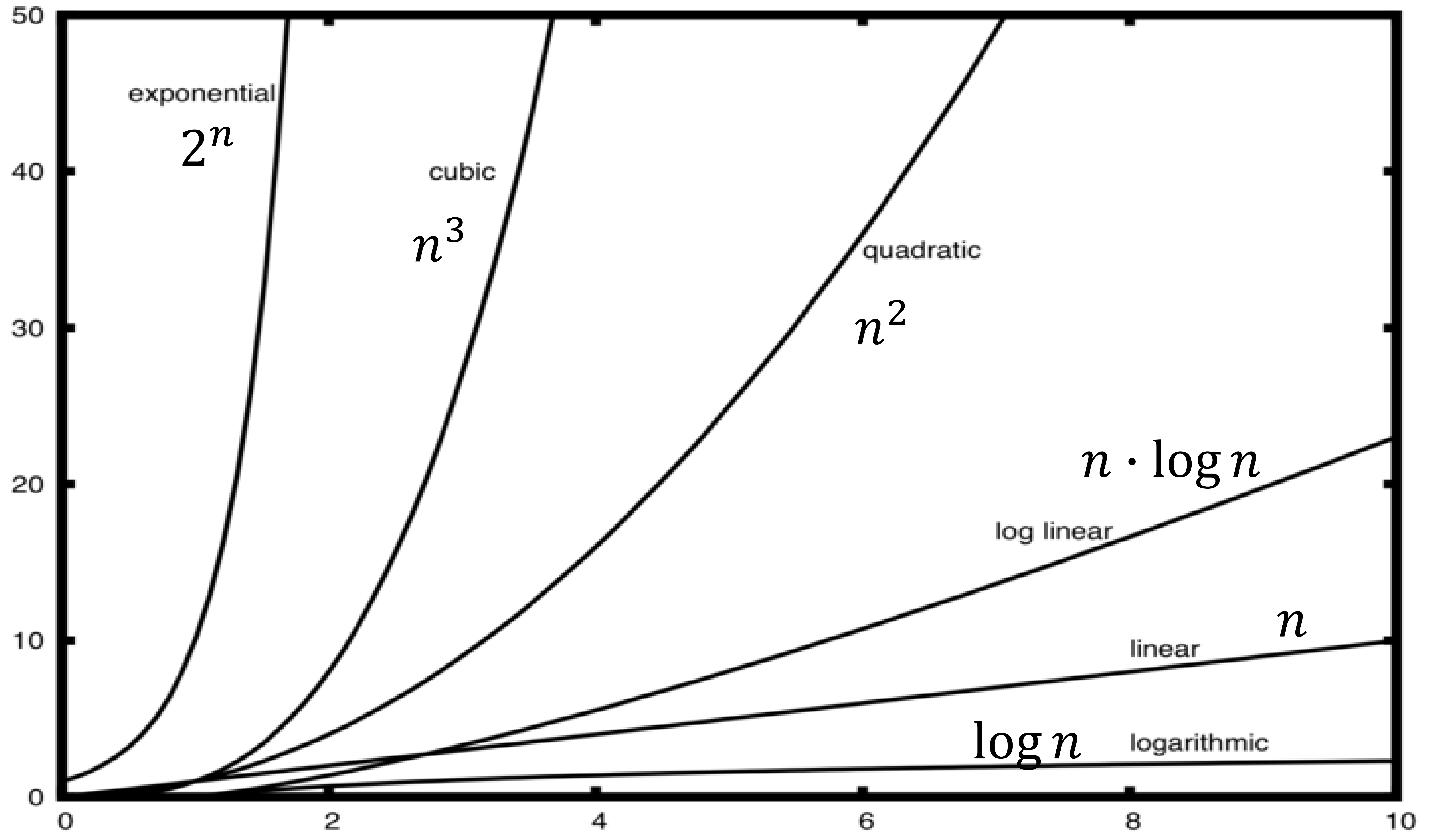
For all real numbers b with $b > 1$ and for all sufficiently large real numbers x,

$$x \leq x \log_b x \leq x^2. \tag{11.4.13}$$

For all real numbers $b > 1$,

$$x \text{ is } O(x \log_b x) \quad \text{and} \quad x \log_b x \text{ is } O(x^2). \tag{11.4.14}$$

Comparisons of Popular Algorithm Orders



Exercise

Show that:

1. $2\,n^2 + 3n \cdot \log n$ is $\Theta(n^2)$

2. $5\log n + n$ is $\Theta(n)$

3. $5\log n + n + n^3 + 2^n$ is $\Theta(2^n)$

SECTION 11.5

Application: Analysis of Algorithm Efficiency II

Binary Search Algorithm

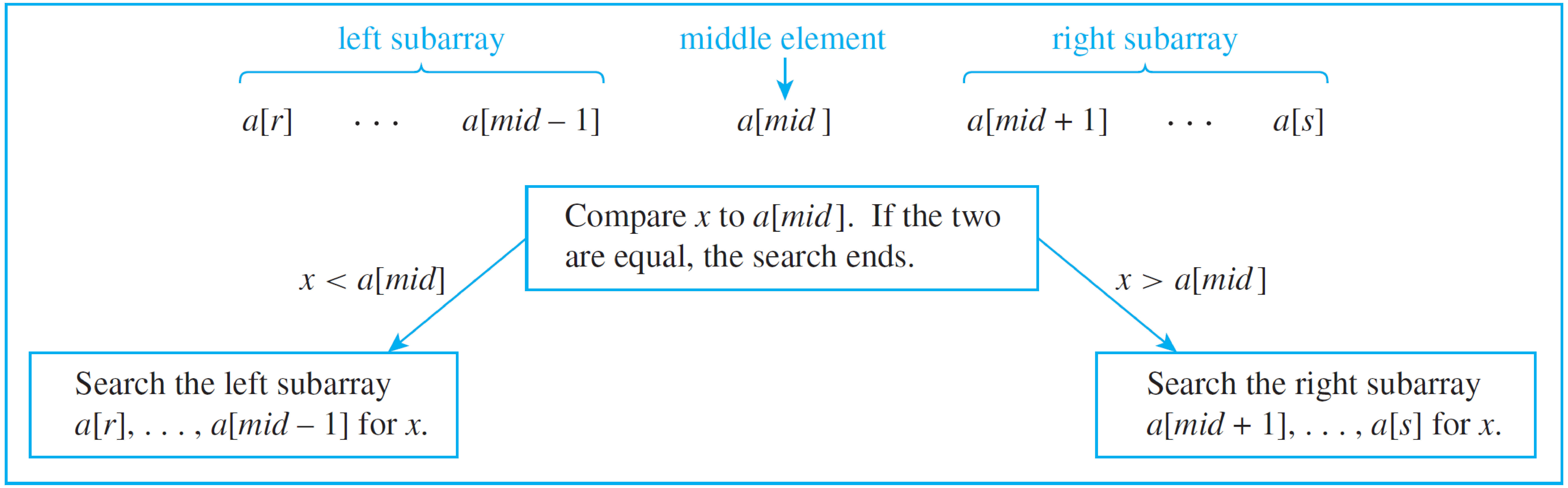
Goal:

- Given a sorted array of ascending elements $a[1], a[2], \ldots, a[n]$ find a particular element x in the array.

Binary Search Algorithm

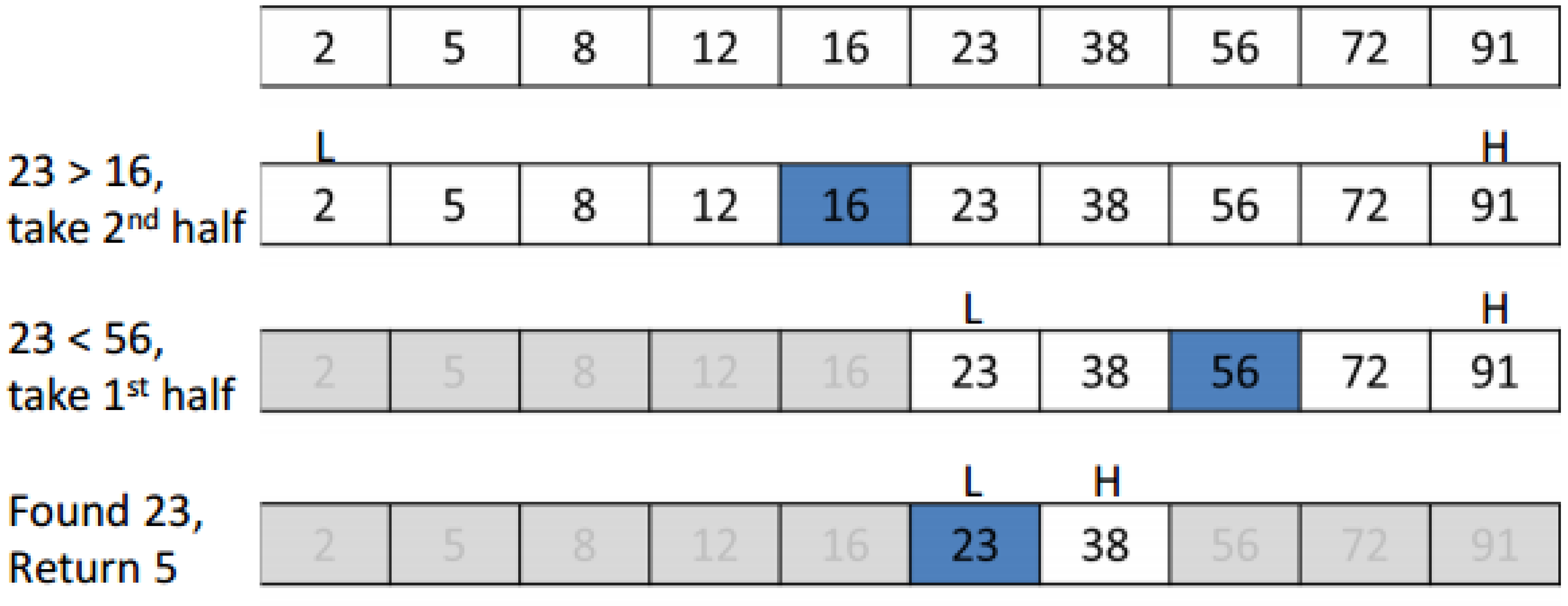
Algorithm:

1. First compare x to the “middle element” $a[m]$ of the array. If the two are equal, then the search is successful and terminate.
2. If x is smaller, then recursively continue the search in the left subarray $a[1]$, ..., $a[m\text{-}1]$, else continue in the right subarray $a[m], \ldots, a[n]$.
3. Recursion stops if no element is left in either sub interval.


left subarray
middle element
right subarray
$a[r]$ . . . $a[mid-1]$
$a[mid]$
$a[mid+1]$ . . . $a[s]$
Compare x to $a[mid]$. If the two are equal, the search ends.
$x < a[mid]$
$x > a[mid]$
Search the left subarray $a[r], \ldots, a[mid-1]$ for x.
Search the right subarray $a[mid+1], \ldots, a[s]$ for x.

Binary Search Algorithm: *Example*

Consider searching 23 in this 10-element sorted array:



Observe that we only need to make **3 comparisons**.

Efficiency of the Binary Search Algorithm

We observe that in each recursive step the length of the new subarray is approximately half of the previous. And in the worst case every subarray down to a single element must be searched.

So how many times (k) do we need to divide n by 2 until we have only one element?

Answer:

$$\frac{n}{2^k} = 1 \;\rightarrow\; k = \log_2 n$$

Thus, in the worst case, the number of comparisons is $\Theta(\log_2 n)$.

Efficiency of Binary vs Sequential Search

To recap:

- The sequential search algorithm has order $\Theta(n)$
- The binary search algorithm has order $\Theta(\log_2 n)$

This difference in efficiency becomes more important as n gets larger and larger.

Assume each algorithm iteration takes 1 nanosecond, the running time for different n is:

	$n = 10^8$	$n = 10^{11}$	$n = 10^{14}$
Sequential Search $\Theta(\mathrm{n})$	$0.1\ \mathrm{s}$	$1.67\ \mathrm{min}$	$27.8\ \mathrm{hours}$
Binary Search $\Theta(\log_2 n)$	$\mathbf{2.7 \cdot 10^{-8}\ s}$	$\mathbf{3.7 \cdot 10^{-8}\ s}$	$\mathbf{4.7 \cdot 10^{-8}\ s}$

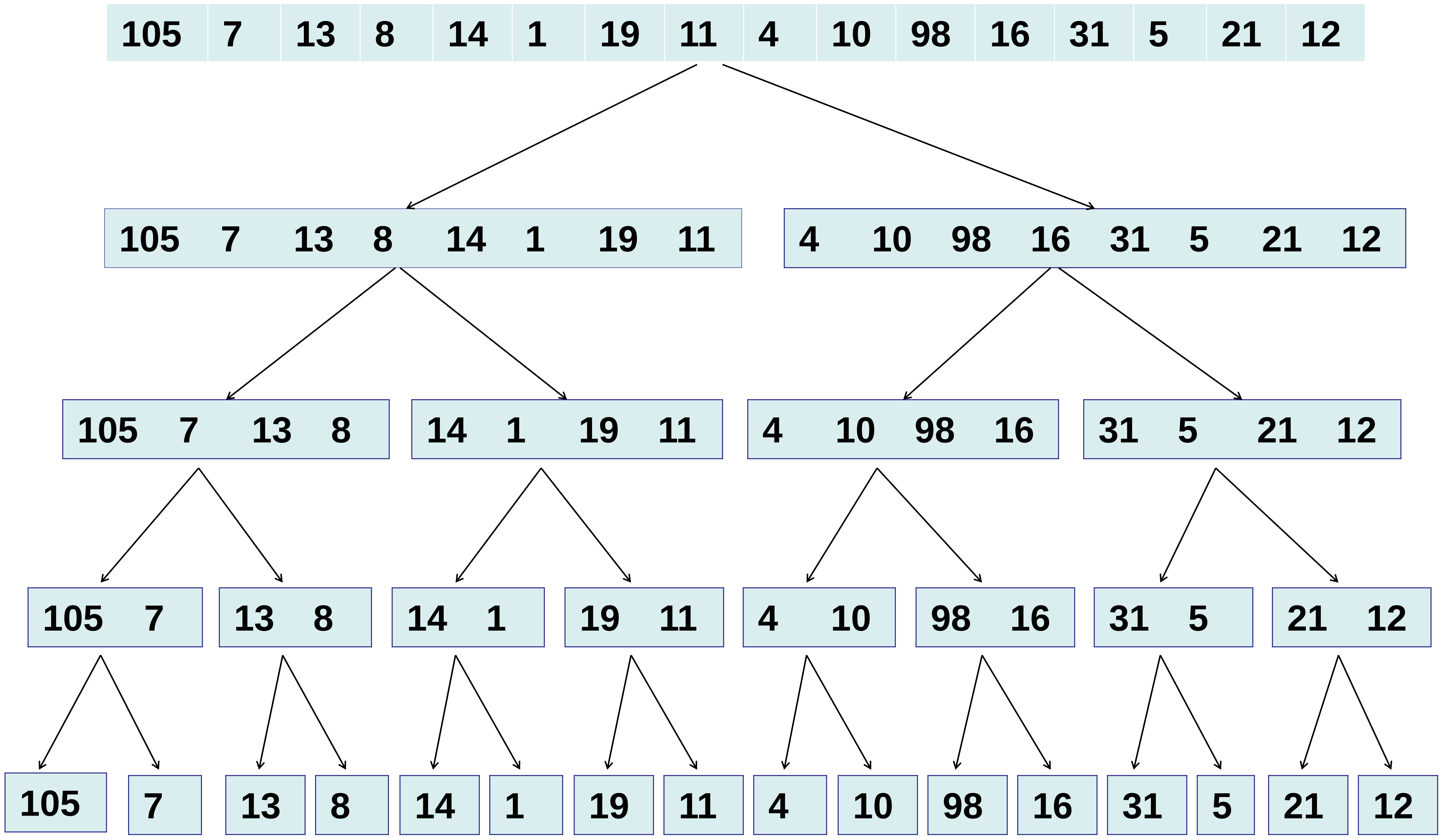
Merge Sort Algorithm

Merge Sort is an algorithm for sorting data with $\Theta(n \cdot \log_2 n)$ complexity.

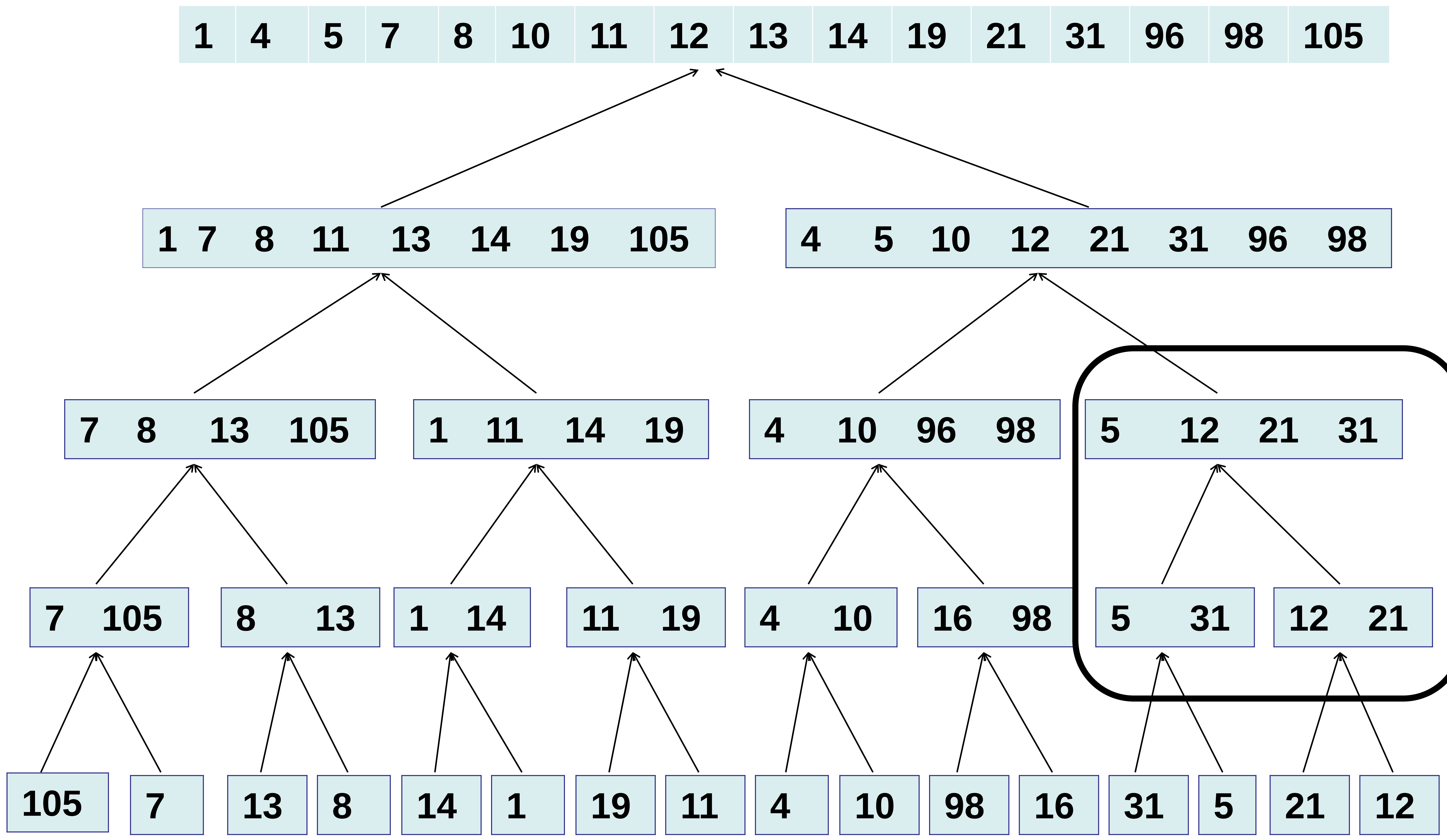
It is a recursive, *divide-and-conquer* algorithm. It consists of three steps:

- split the array into left and right subarrays (until min subarray of length 1 is reached),
- sort each subarray recursively,
- finally **merge** the sorted subarrays into a single **sorted** array.

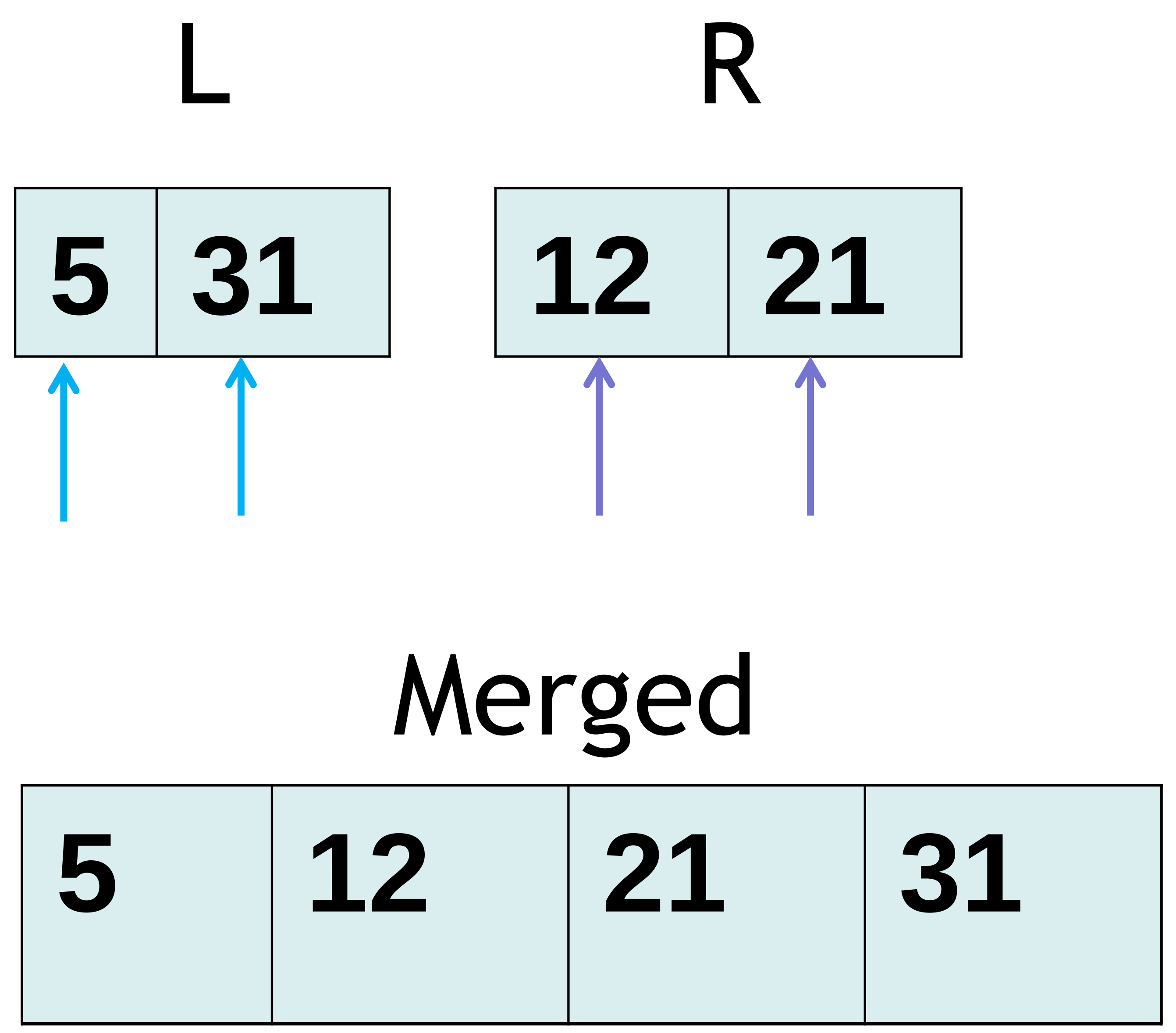
Applying Merge Sort: *Divide Step*



Applying Merge Sort: *Sort and Merge*



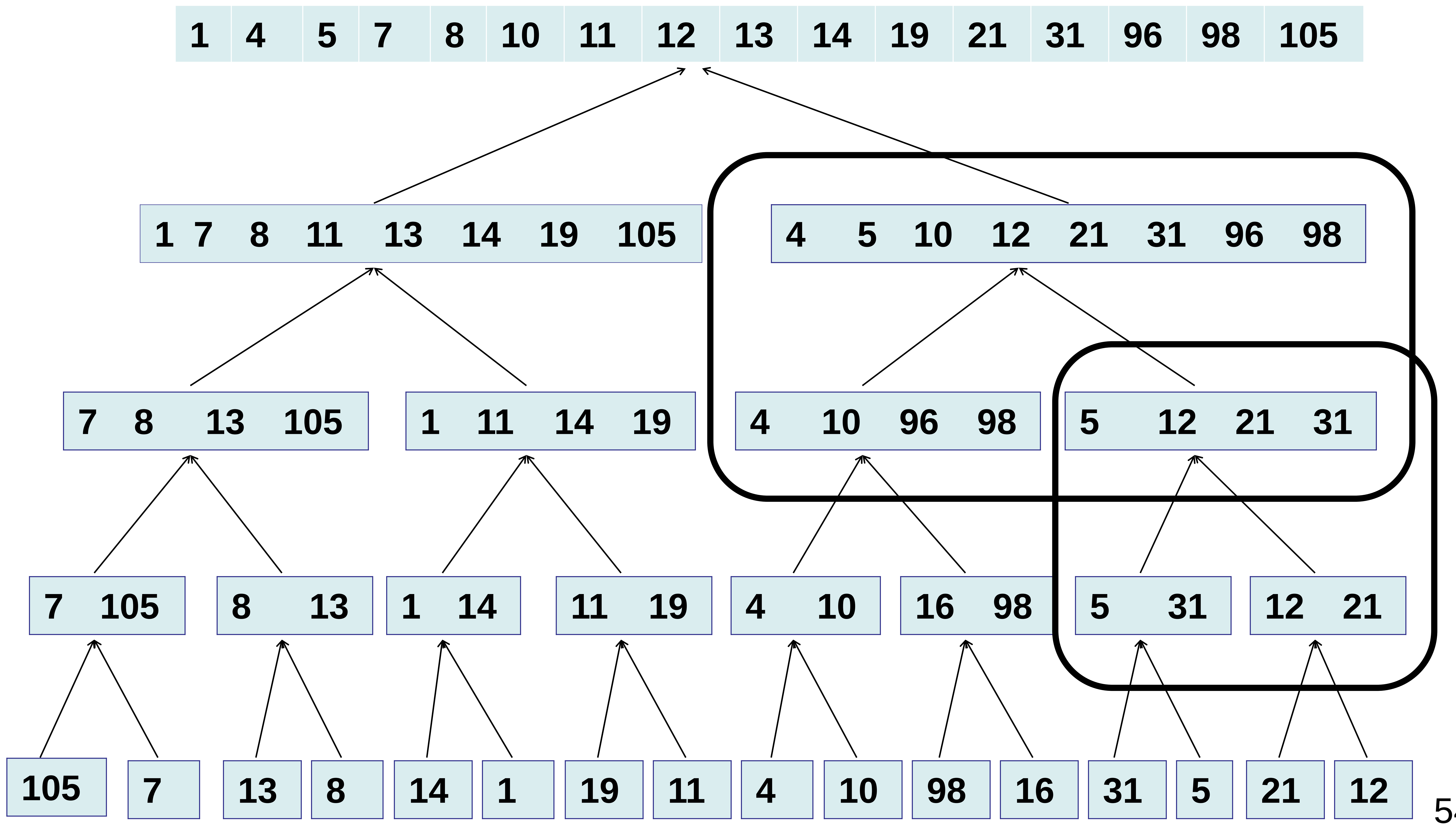
Applying Merge Sort: *Sort and Merge example 1*



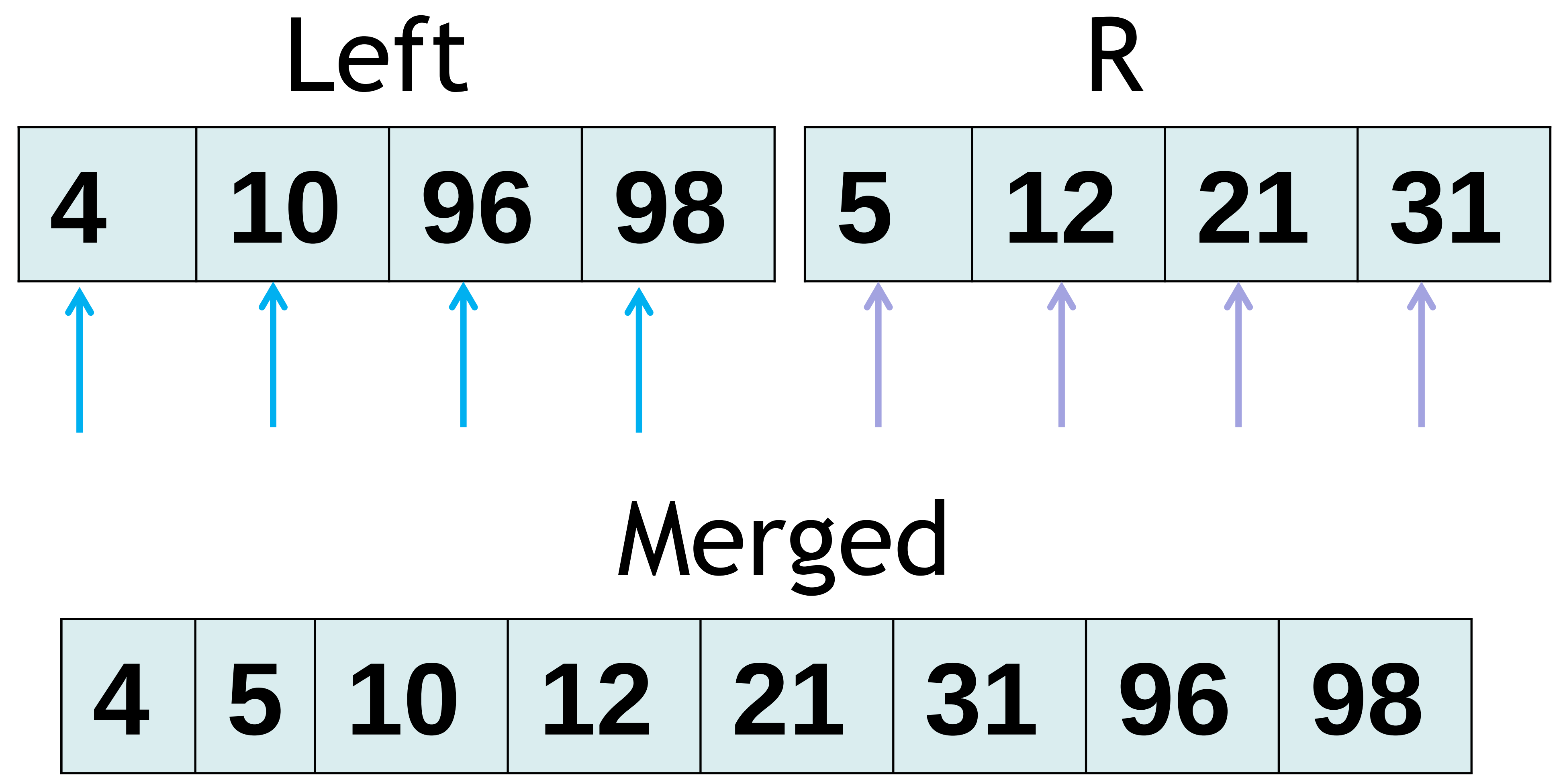
Elementary operations:

- 3 comparisons
- 4 write operations

Applying Merge Sort: *Sort and Merge example 2*



Applying Merge Sort: *Sort and Merge example 2*



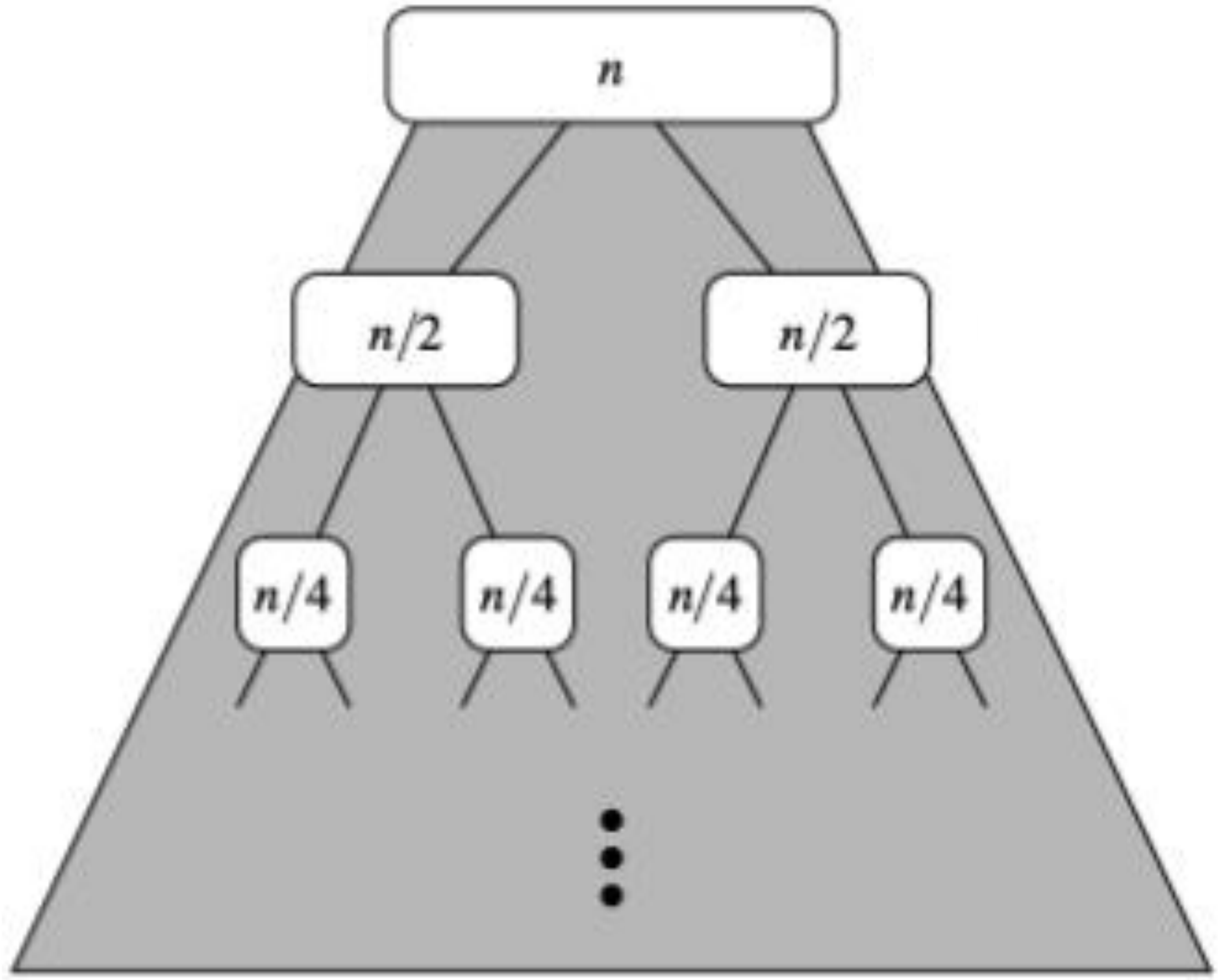
Elementary operations:

- 6 comparisons
- 8 write operations

We can conclude that merging two sorted subarrays of length $k/2$ into one of combined length k needs at most $k-1$ comparisons and k write operations and thus it has $\Theta(k)$.

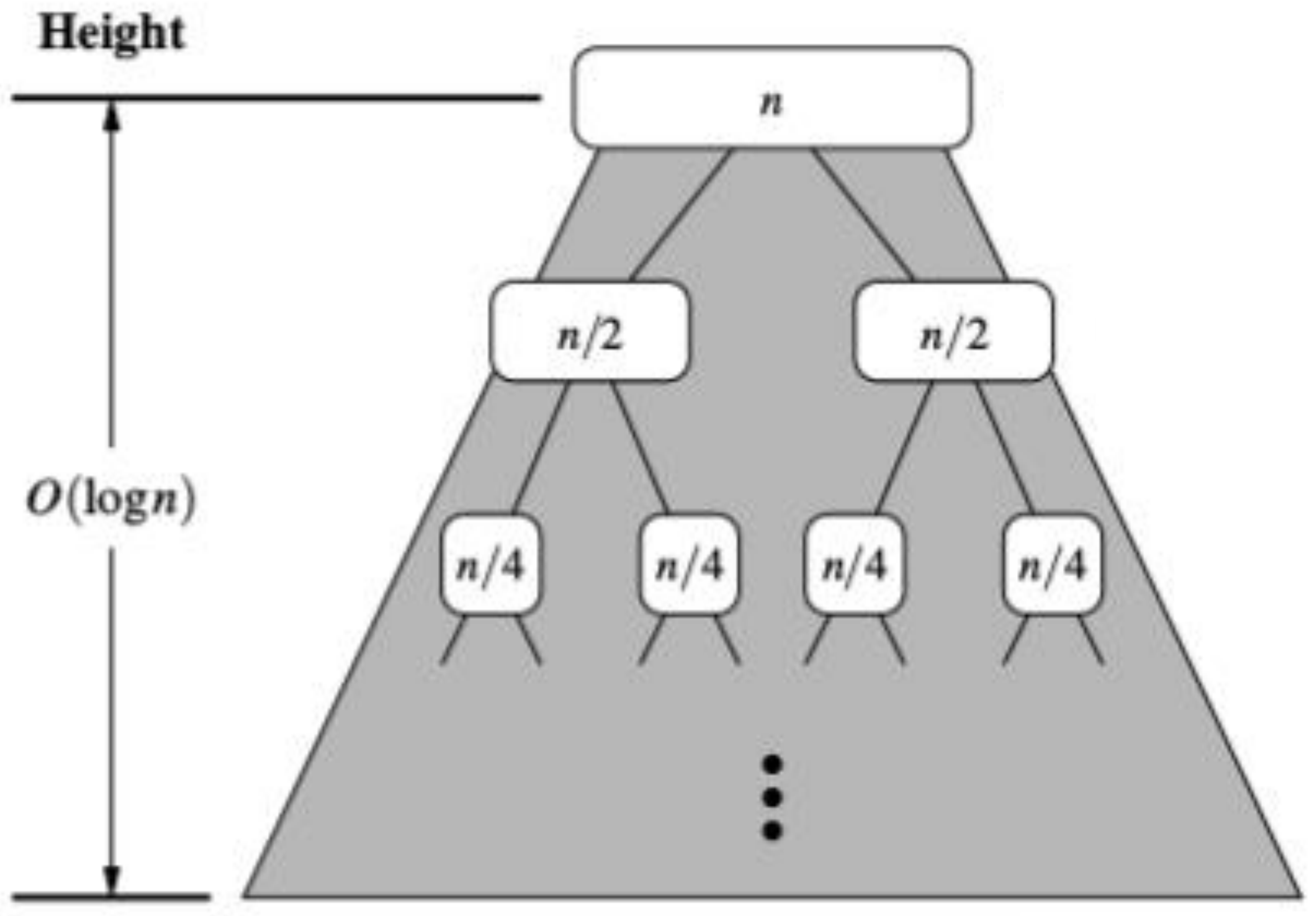
Efficiency of Merge Sort

We can represent Merge Sort as a **recursive tree**, where each *descendant of the root* (the original array with n elements) represents a *recursive call* of the sorting function.



Efficiency of Merge Sort: *Divide Step*

The elementary operation of the split is the cost to compute the mid point of each array. Therefore, the total number of splits is 1+2+4+…+$2^{h-1} = 2^h$-1 (geometric series). Since the height of the tree is $h = \log_2 n$, then the total cost is $n - 1$. So, the Divide step is $\Theta(n)$.



1 split

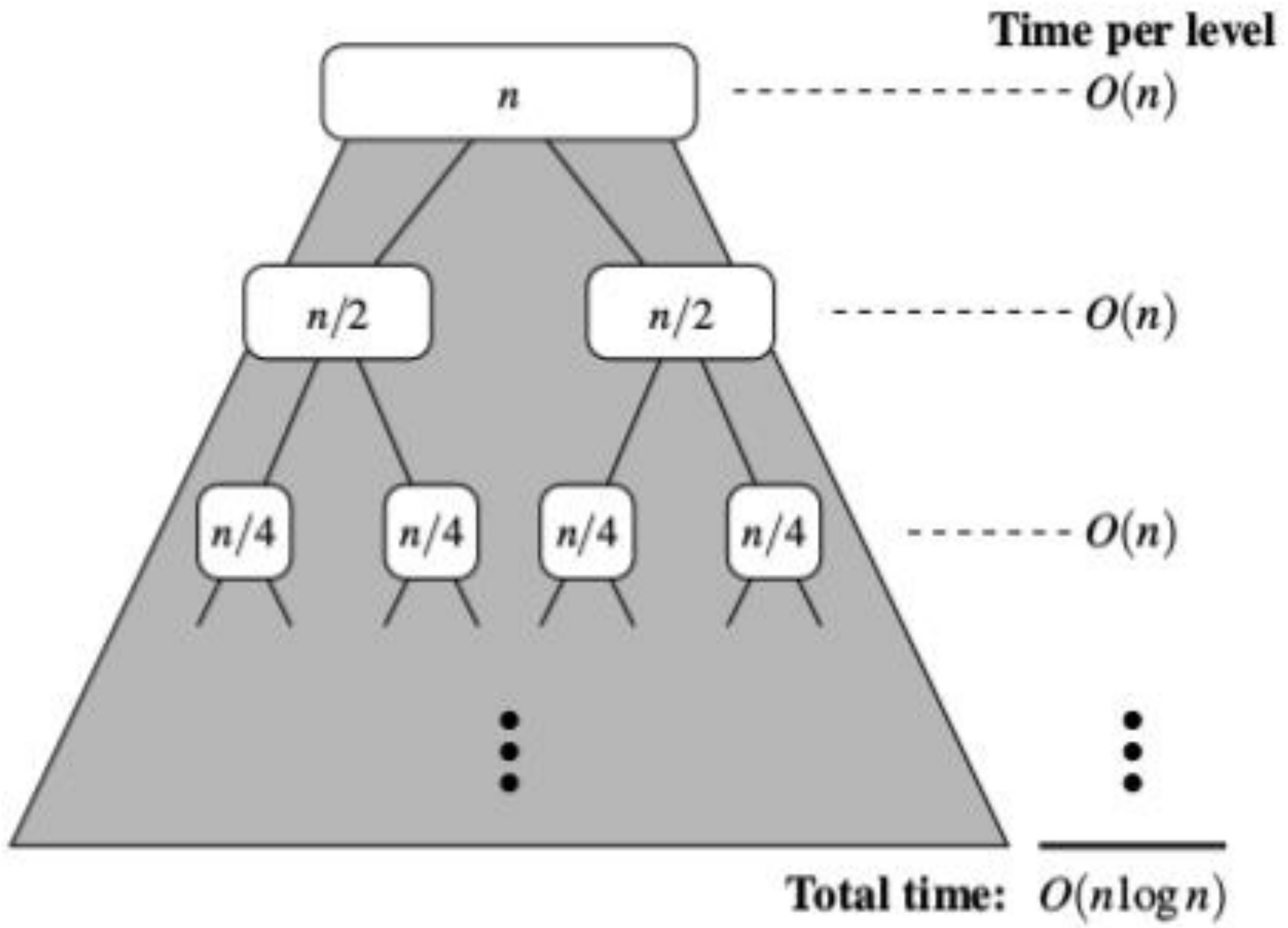
2 splits

4 splits

2^i splits

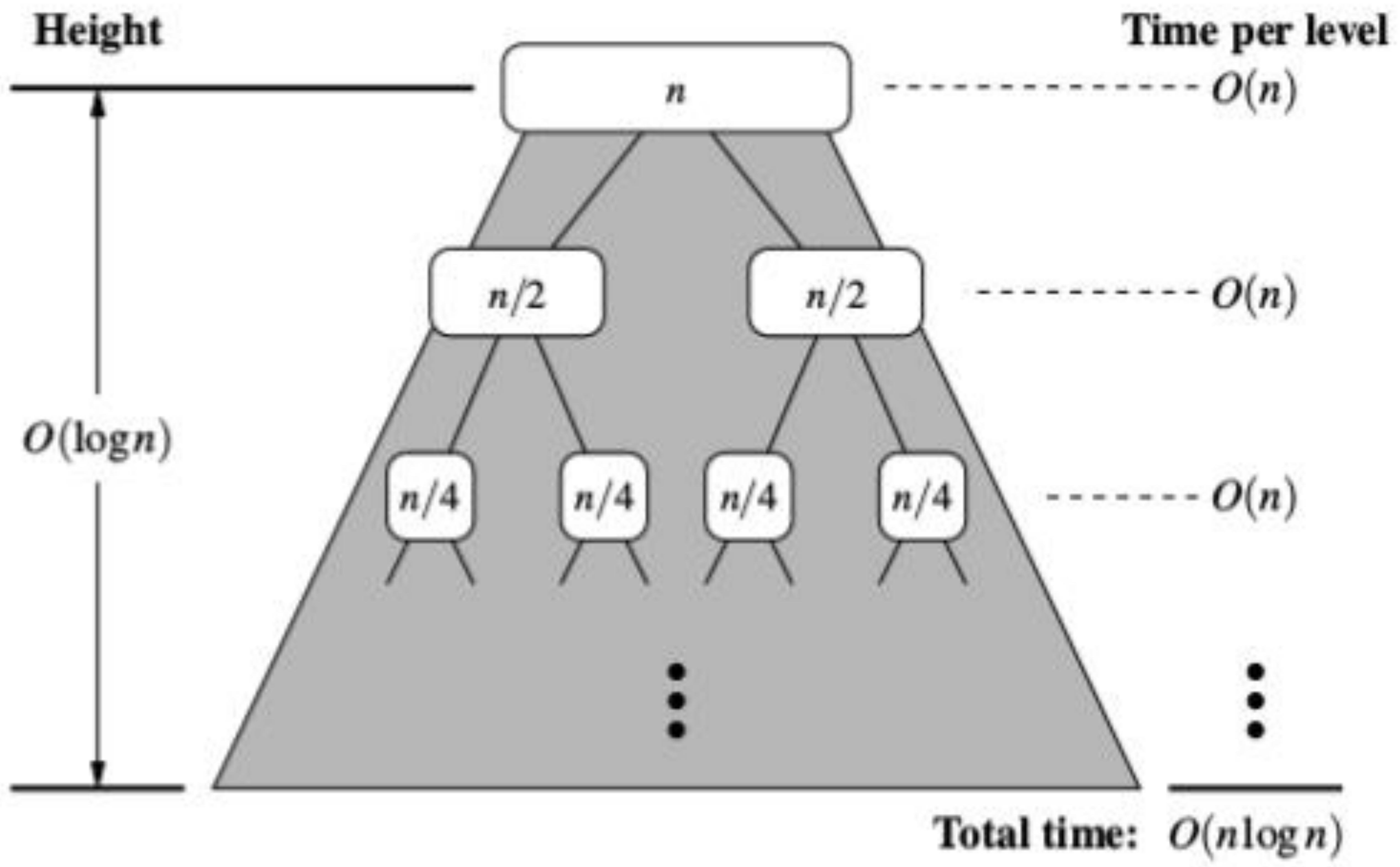
Efficiency of Merge Sort: *Sort and Combine Step*

Each i-th level of the tree contains 2^i subarrays, each of length $\frac{n}{2^i}$. Thus, the number of comparisons and write operations is $\Theta(2^i \frac{n}{2^i}) = \Theta(n)$.



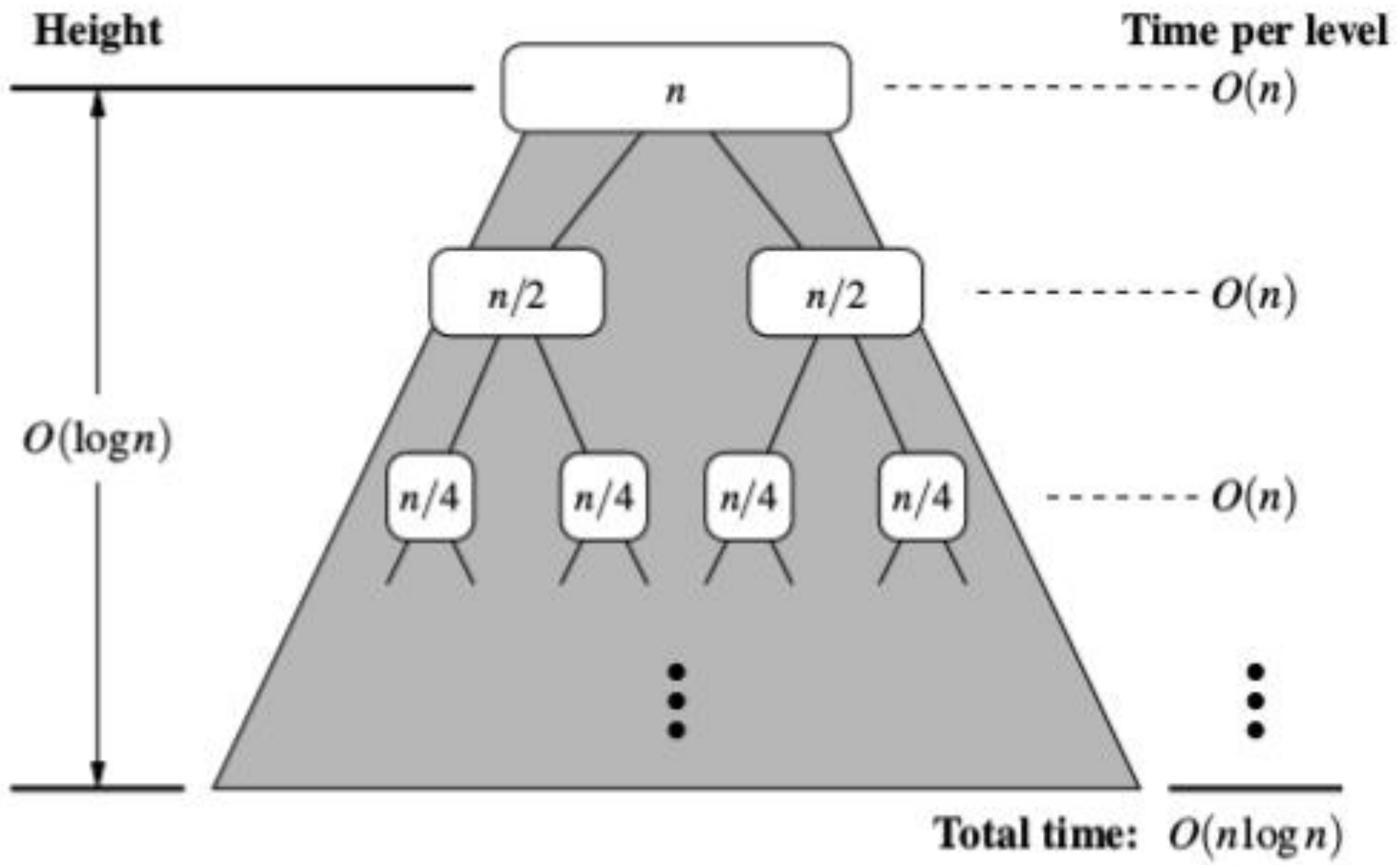
Efficiency of Merge Sort: *Sort and Combine Step*

Since the max number of levels of the tree is $\log_2 n$, then the total number of comparisons and write operations is $\Theta(\mathrm{n} \cdot \log_2 n)$



Efficiency of Merge Sort: *Sort and Combine Step*

If we now add the complexity of the Divide Step ($\Theta(n)$), the overall algorithm efficiency is then $\Theta(n + \mathrm{n} \cdot \log_2 n) = \Theta(\mathrm{n} \cdot \log_2 n)$



Efficiency of Merge Sort vs Insertion Sort

To recap:

- The Insertion Sort algorithm is $\Theta(n^2)$
- The Merge Sort algorithm is $\Theta(\mathrm{n} \cdot \log_2 n)$

This difference in efficiency becomes more important as n gets larger and larger.
Assume each algorithm iteration takes 1 nanosecond, the running time for different n is:

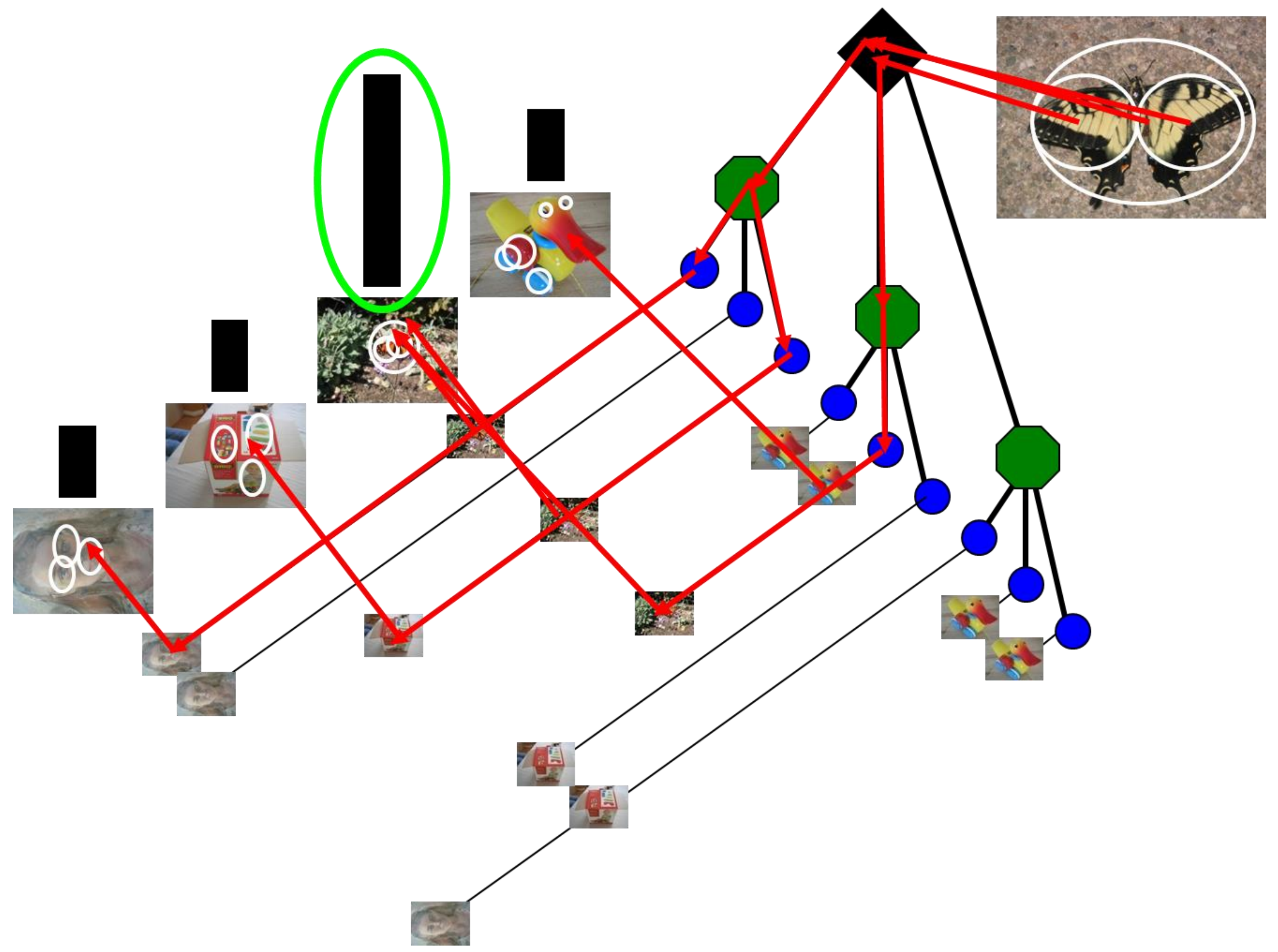
	$n = 10^8$	$n = 10^{11}$	$n = 10^{14}$
Insertion Sort $\Theta(n^2)$	$115\ days$	$317{,}000\ years$	317 billion years
Merge Sort $\Theta(\mathrm{n} \cdot \log_2 n)$	$\mathbf{2.7\ s}$	$\mathbf{1\ hour}$	$\mathbf{54\ days}$

Using information theory, **it can be shown that there cannot be any sorting algorithm with a better worst-case time-complexity than O(n log n)**.

Bag of Words (Google Image Search)

At the end of Chapter 10, we saw that, thanks to hierarchical clustering, we can reduce the time to look up a feature in the visual vocabulary from linear to logarithmic:

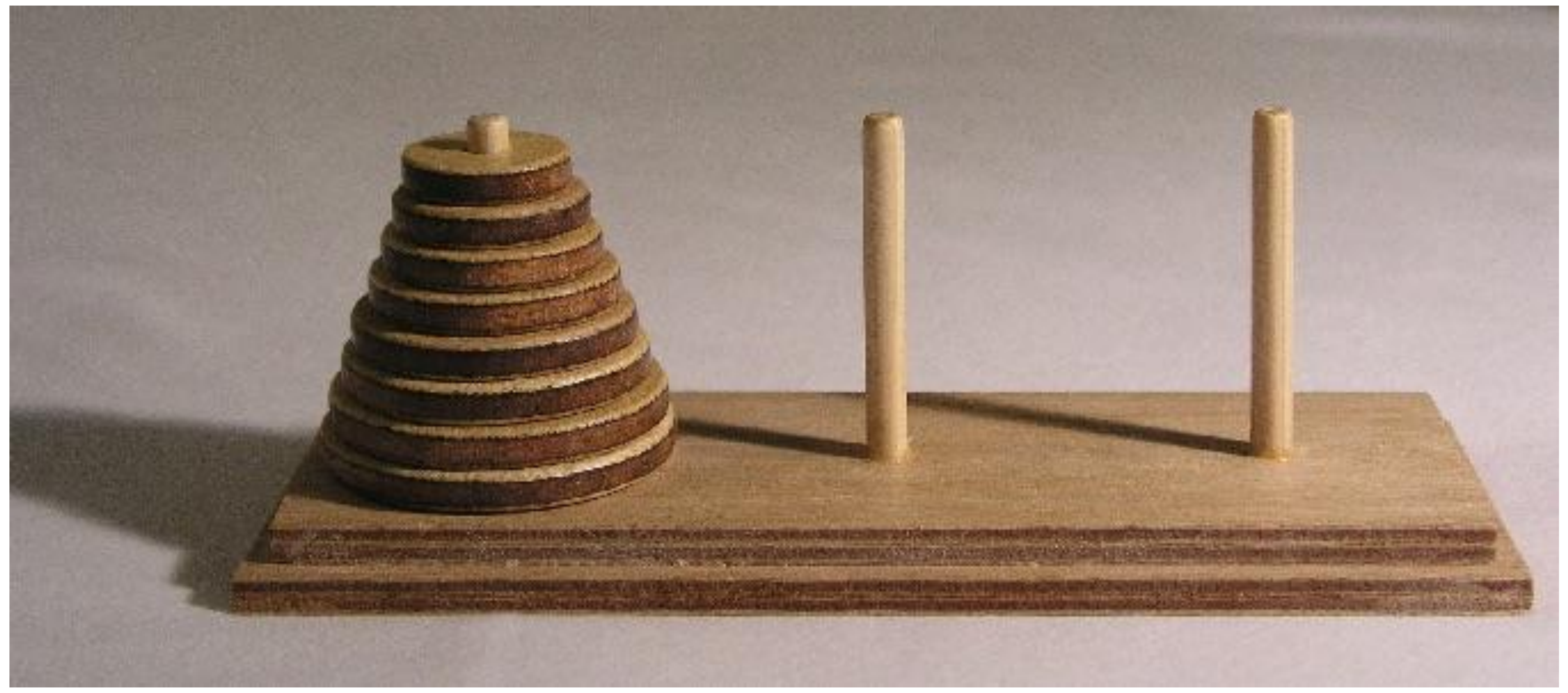
- if n is the number of visual words in the vocabulary, then the number of comparisons per query feature is $\mathrm{b} \cdot \mathrm{L}$, where b is the number of branches and L the number of levels of the tree. Since $\mathrm{L} = \log_b n$, then the number of comparisons is O($\log_b n$)



Tower of Hanoi

In Chapter 5, we showed that the 64 disk Tower of Hanoi puzzle requires $2^{64} - 1$ steps (if a computer took 1 nanosecond to calculate each transfer step, the total time to calculate all the steps would be 584 years!).

We call this types of algorithms exponential algorithms, and they quickly become intractable if the amount of data to process is very large!



Algorithm Efficiency – Recap Table

These are the algorithms seen thus far and their complexities:

- Sequential Search: n
- Binary Search: $\log_2 n$
- Insertion Sort: n^2
- Merge Sort: $n \cdot \log_2 n$
- Bag-of-Words feature look-up: $\log_b n$
- Tower of Hanoi: 2^n

Tractable and Intractable Problems

Problems whose solutions can be found with algorithms whose worst-case order with respect to time is a polynomial, or $O(n^c)$, are said to belong to class **P**.

- Polynomial-time algorithms and are said to be tractable.
- Problems that cannot be solved in polynomial time are called intractable.

For some problems, it is possible to check the correctness of a solution with a polynomial-time algorithm, but it may not be possible to find a solution in polynomial time.

Such problems are said to belong to class **NP**.

Tractable and Intractable Problems

The biggest open question in theoretical computer science is whether every problem in class NP belongs to class P.

Known as the **P** vs. **NP** problem.