Complexity Analysis

Analyzing Algorithms

- ❑ Analyzing an algorithm has come to mean predicting the **resources** that the algorithm requires.
  - ❑ Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern,
  - ❑ but most often it is **computational time** that we want to measure.

- ❑ **Simplistic Assumption**
  - ❑ assume a generic one processor, random-access machine (RAM) model of computation
    - ❑ instructions are executed one after another, with no concurrent operations

Experimental (Empirical) Studies

- we can determine the elapsed time by recording the time just before the algorithm and the time just after the algorithm, and computing their difference

```
from time import time
start_time = time( )                    # record the starting time
run algorithm
end_time = time( )                      # record the ending time
elapsed = end_time − start_time         # compute the elapsed time
```

- Many processes share use of a computer's CPU, the elapsed time will depend on what other processes are running on the computer when the test is performed.

Challenges of Experimental Analysis

While experimental studies of running times are valuable, especially when fine-tuning production-quality code, there are three major limitations to their use for algorithm analysis:

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without need for implementation.
3. Takes into account all possible inputs.

Input Size of an Algorithm

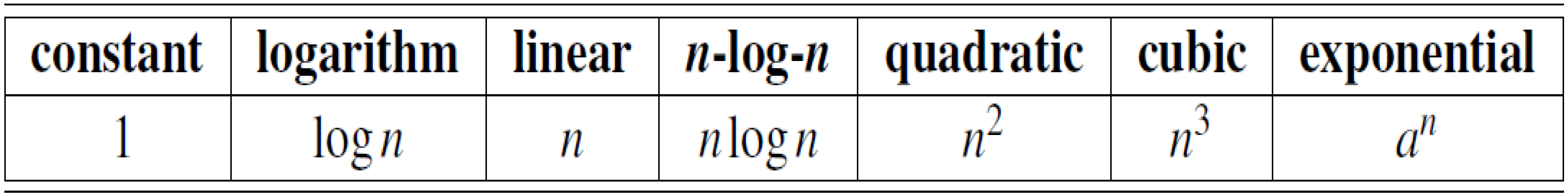
- The input size of an algorithm is a measure of the amount of data that the algorithm needs to process in order to produce its output.
- The best notion for input size **depends on the problem** being studied

    - if an algorithm takes a list of numbers as input, the input size could be measured as the number of elements in the list.
    - if the algorithm takes a string as input, the input size could be measured as the length of the string.
    - if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

Running time of an Algorithm

- The running time of an algorithm is a measure of how long the algorithm takes to produce its output given a particular input size.
- Running time is usually measured in terms of **the number of operations that the algorithm performs**, and can be expressed as a **function of the input size**.

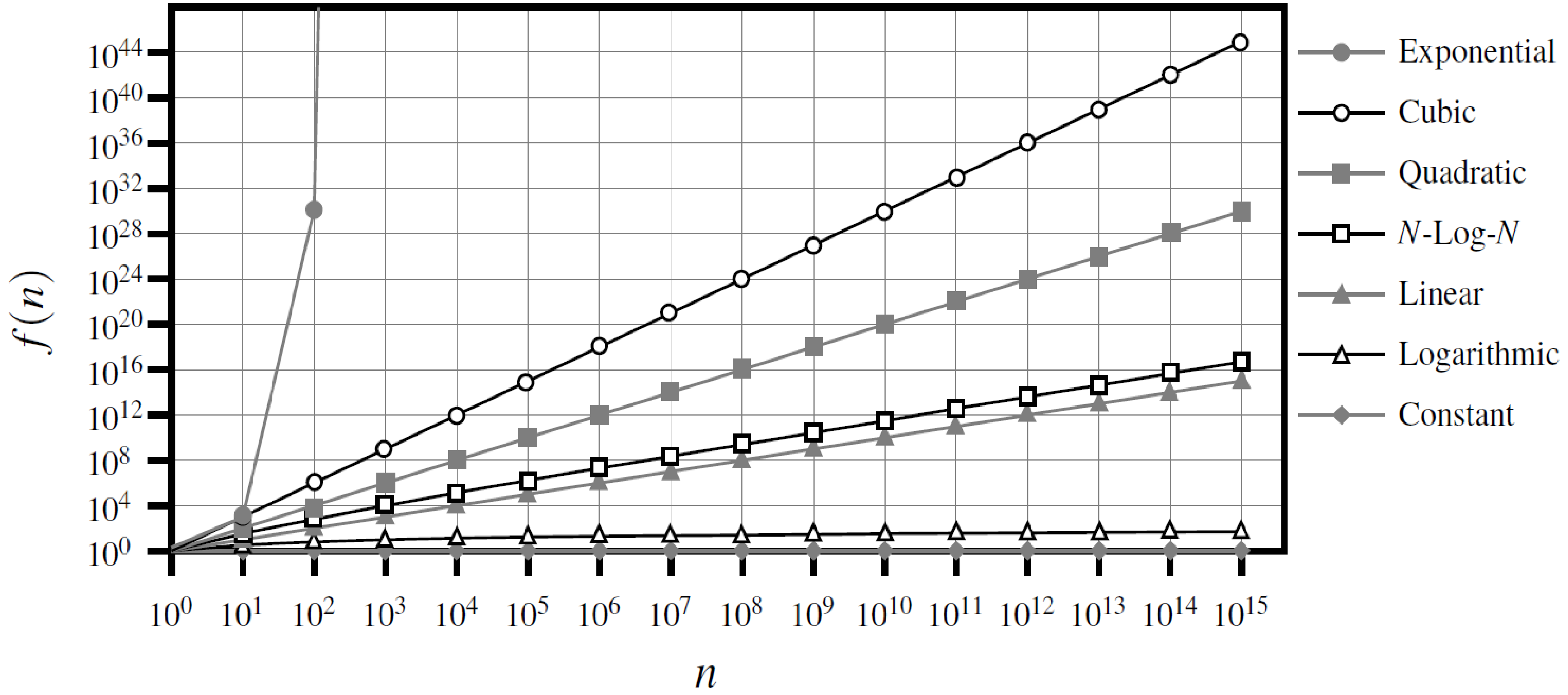
Comparing Growth Rates



constant	**logarithm**	**linear**	***n*-log-*n***	**quadratic**	**cubic**	**exponential**
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 3.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Comparing Growth Rates



- Growth rates for the seven fundamental functions used in algorithm analysis.
- We use base a = 2 for the exponential function.
- The functions are plotted on a **log-log chart**, to compare the growth rates primarily as slopes.
- Even so, the exponential function grows too fast to display all its values on the chart.

Best, Average and Worst case Analysis

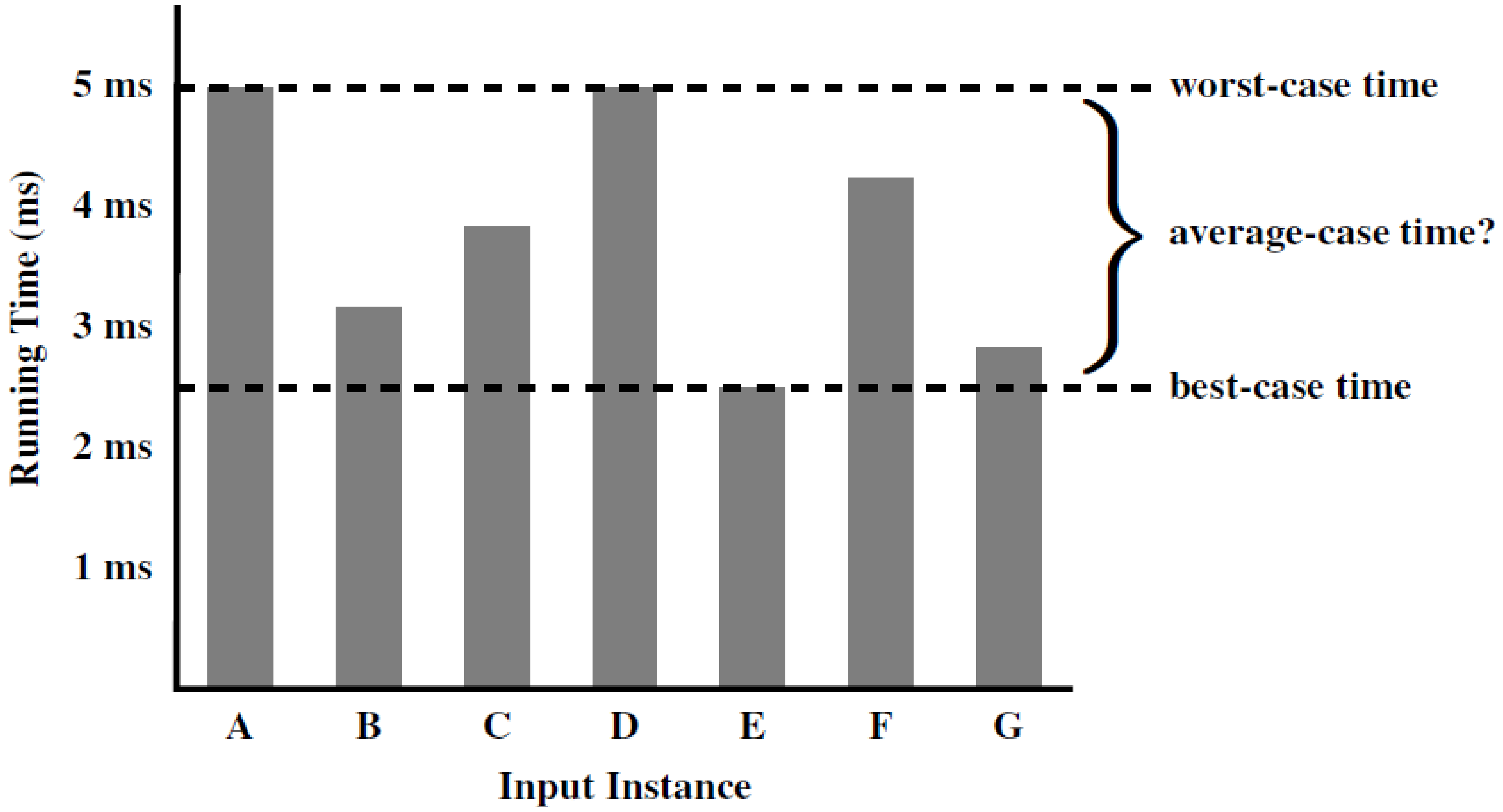
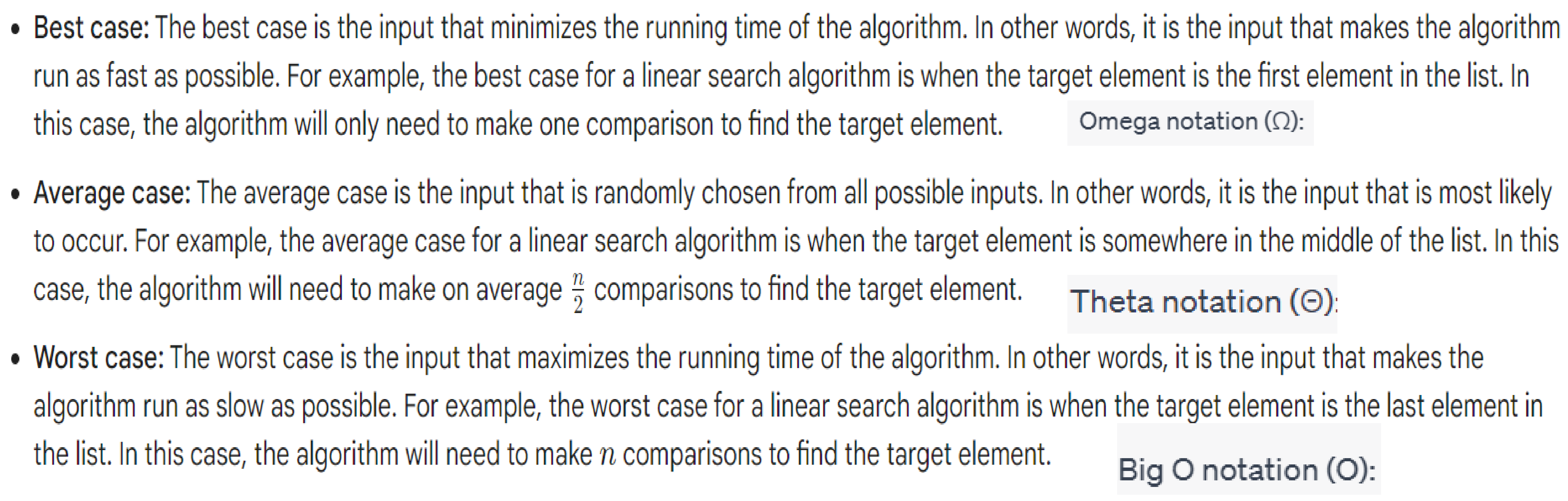


Figure 3.2: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Best, Average and Worst case Analysis

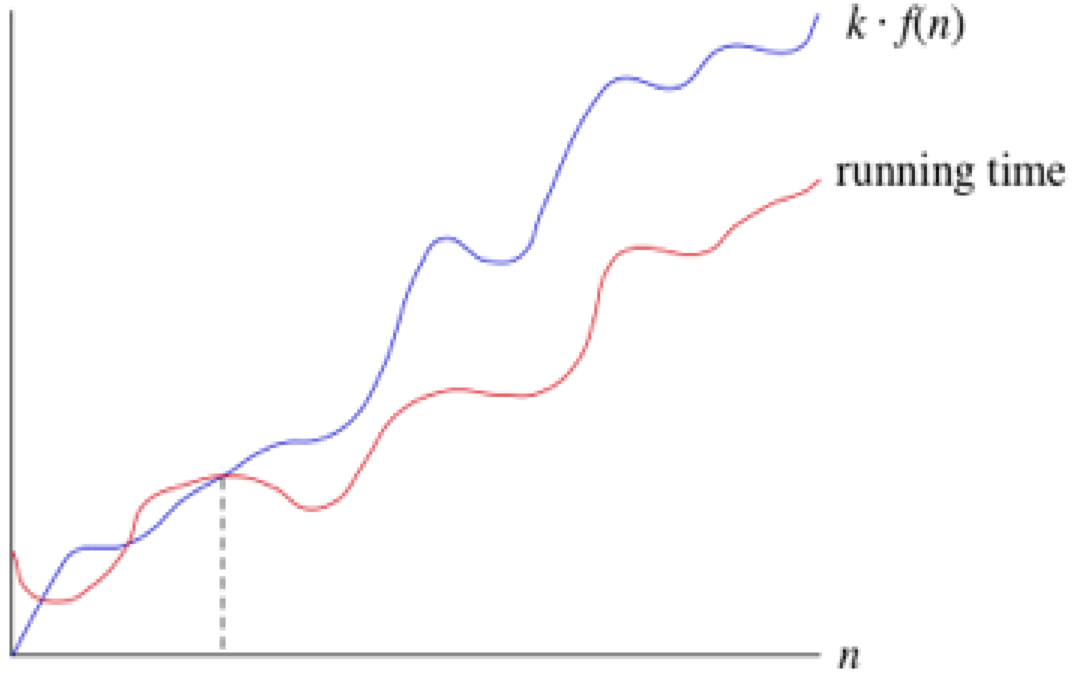


- **Best case:** The best case is the input that minimizes the running time of the algorithm. In other words, it is the input that makes the algorithm run as fast as possible. For example, the best case for a linear search algorithm is when the target element is the first element in the list. In this case, the algorithm will only need to make one comparison to find the target element. Omega notation (Ω):
- **Average case:** The average case is the input that is randomly chosen from all possible inputs. In other words, it is the input that is most likely to occur. For example, the average case for a linear search algorithm is when the target element is somewhere in the middle of the list. In this case, the algorithm will need to make on average $\frac{n}{2}$ comparisons to find the target element. Theta notation (Θ):
- **Worst case:** The worst case is the input that maximizes the running time of the algorithm. In other words, it is the input that makes the algorithm run as slow as possible. For example, the worst case for a linear search algorithm is when the target element is the last element in the list. In this case, the algorithm will need to make n comparisons to find the target element. Big O notation (O):

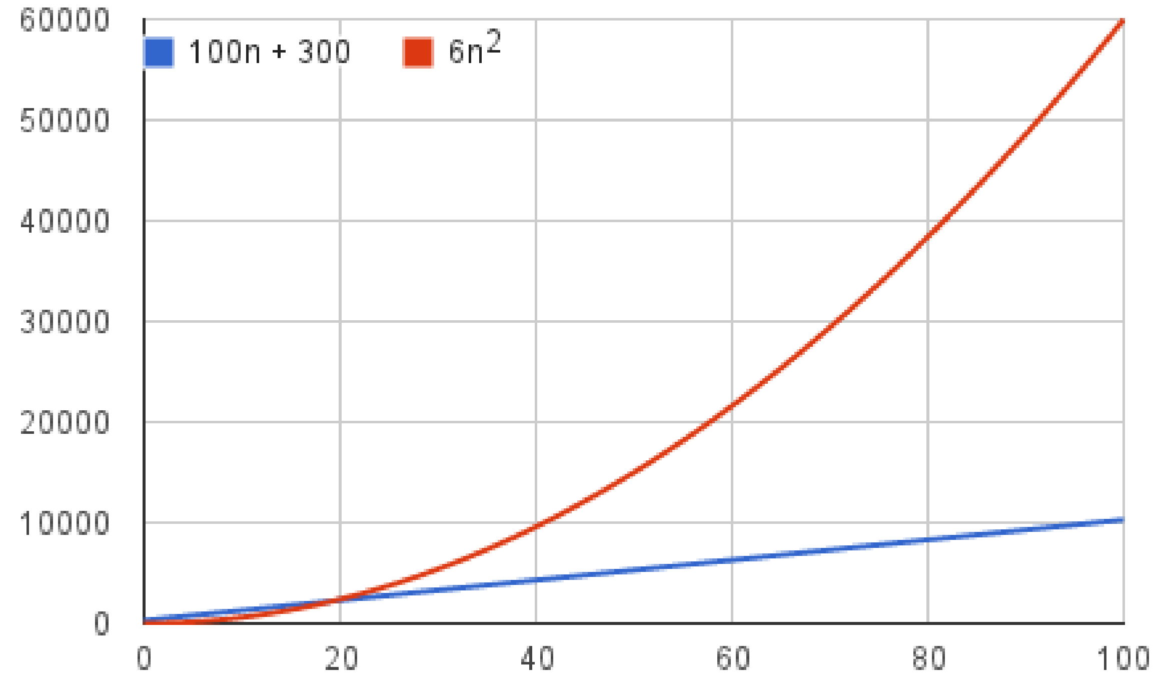
- It is important to note that the best, worst, and average cases are theoretical concepts.
- In practice, the actual running time of an algorithm may vary depending on the specific input. However, the best, worst, and average cases can be used to get an estimate of the performance of an algorithm.

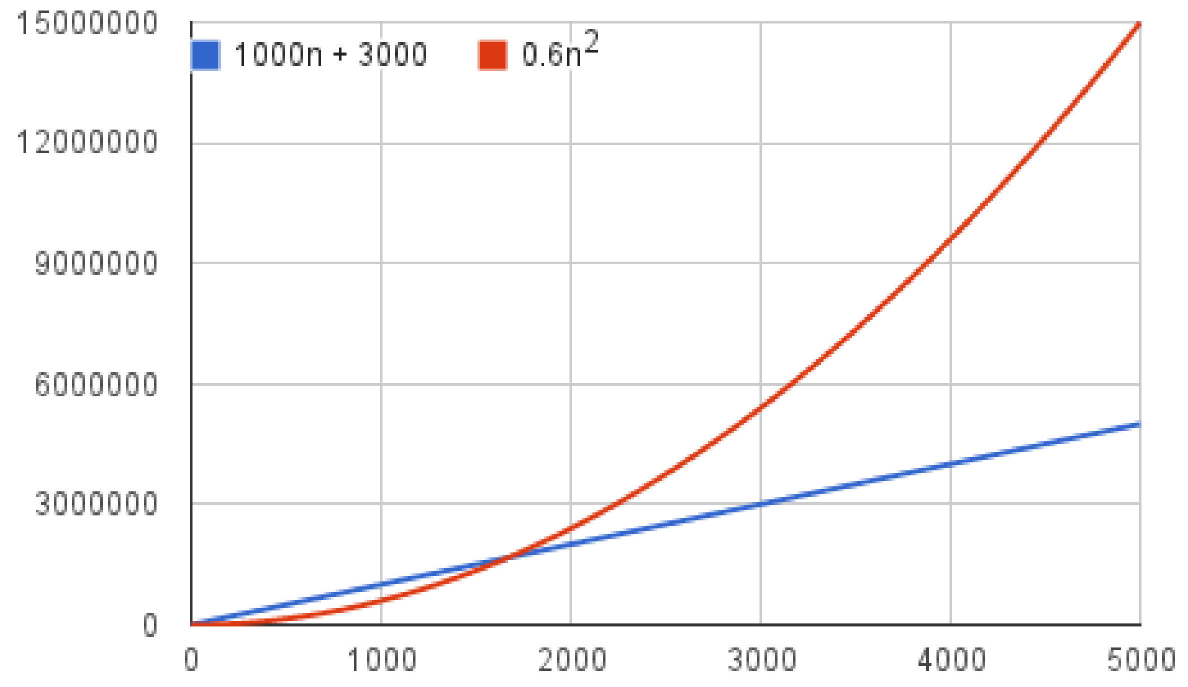
The Big-Oh Notation

If a running time is $O(f(n))$, then for large enough n, the running time is at most $k \cdot f(n)$ for some constant k. Here's how to think of a running time that is $O(f(n))$:

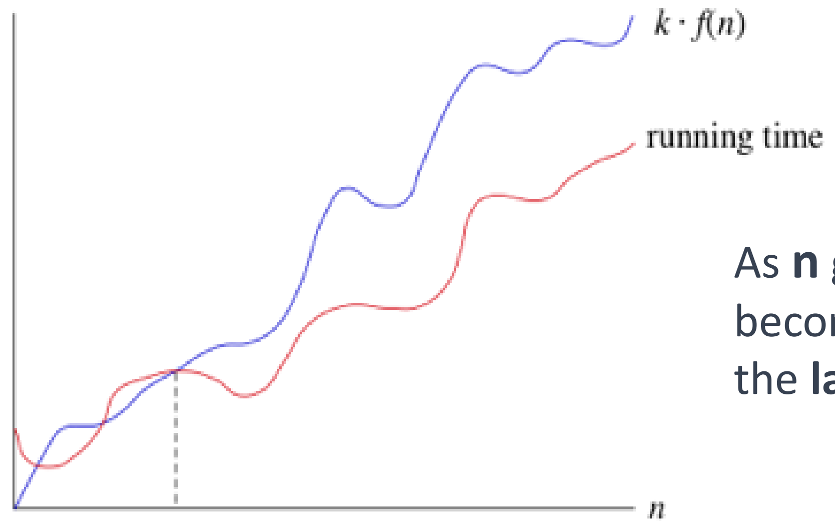


- We use big-O notation for asymptotic upper bounds, since it **bounds the growth of the running time** from above for large enough input sizes.
- When we use big O notation, we are concerned with the behavior of the function as the input size grows infinitely large.





running on an input of size n, takes $6n^2 + 100n + 300$

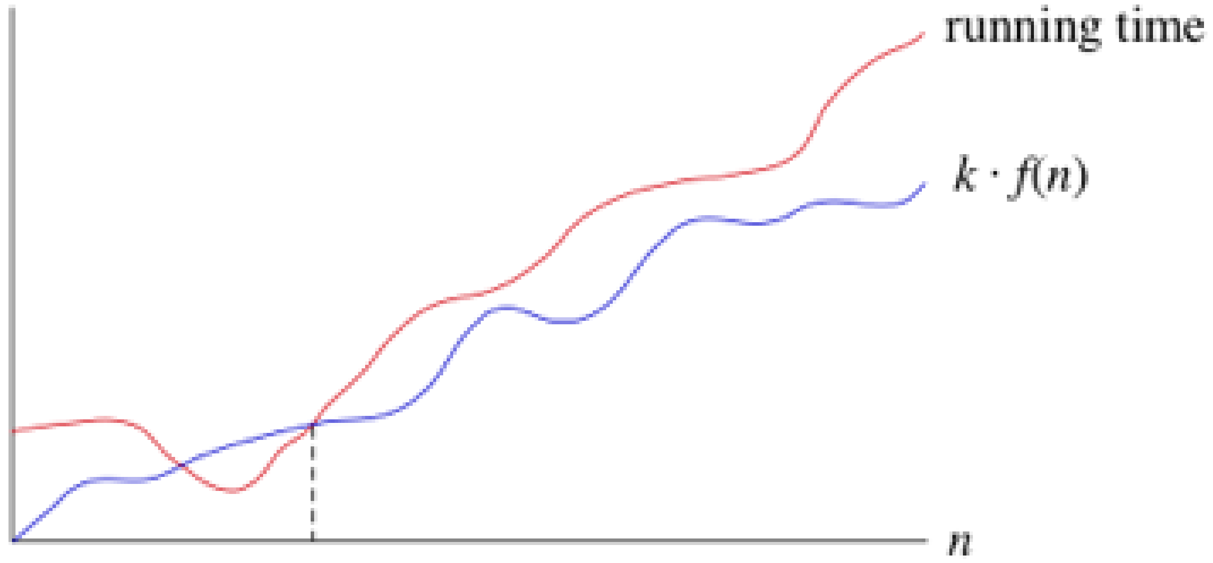


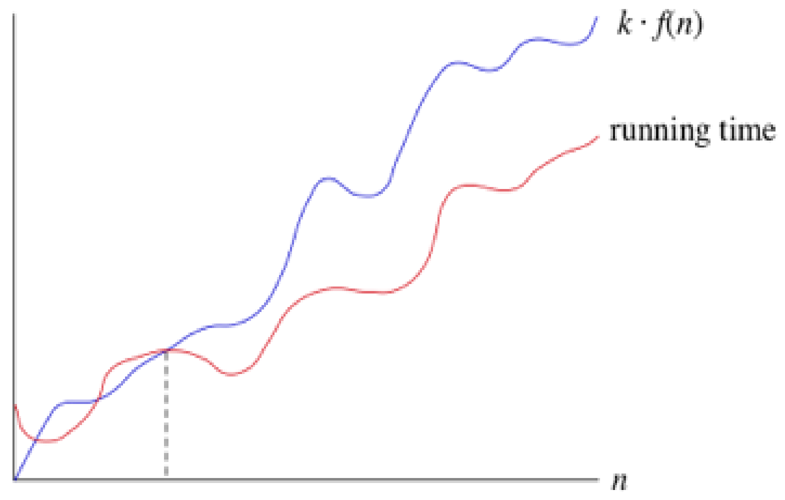
there will always be such a crossover point, no matter what the constants.

As **n** grows larger and larger, the impact of the **constants** and **less significant terms** become less and less significant, and the function is dominated by the term with the **largest exponent**.

Big-Ω (Big-Omega) notation

If a running time is $\Omega(f(n))$, then for large enough n, the running time is at least $k \cdot f(n)$ for some constant k. Here's how to think of a running time that is $\Omega(f(n))$:

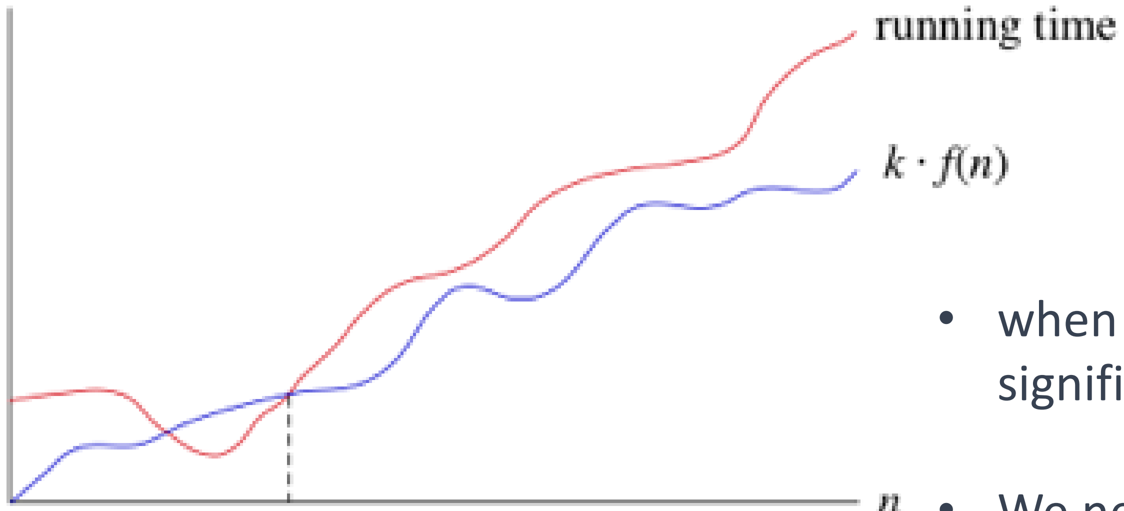




We say that the running time is "big-Ω of $f(n)$." We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

Big-Ω (Big-Omega) notation

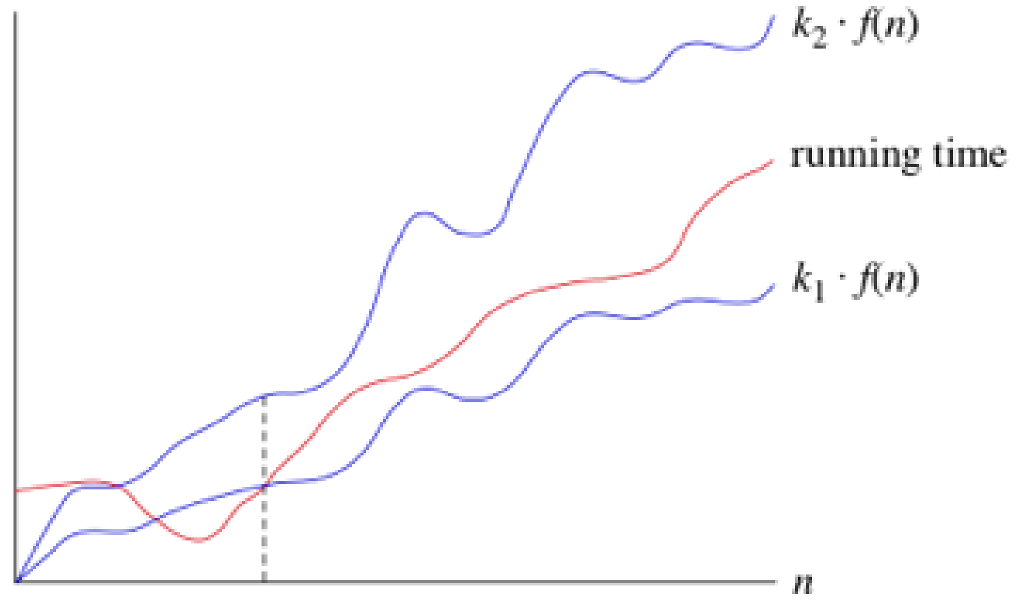
If a running time is $\Omega(f(n))$, then for large enough n, the running time is at least $k \cdot f(n)$ for some constant k. Here's how to think of a running time that is $\Omega(f(n))$:

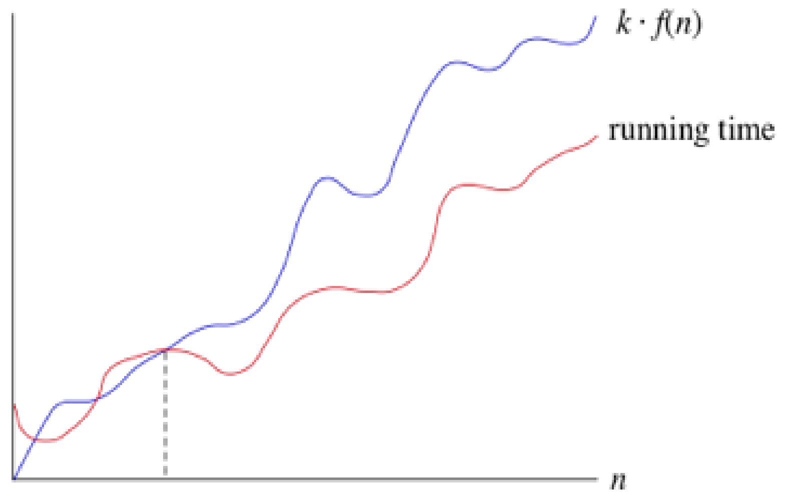


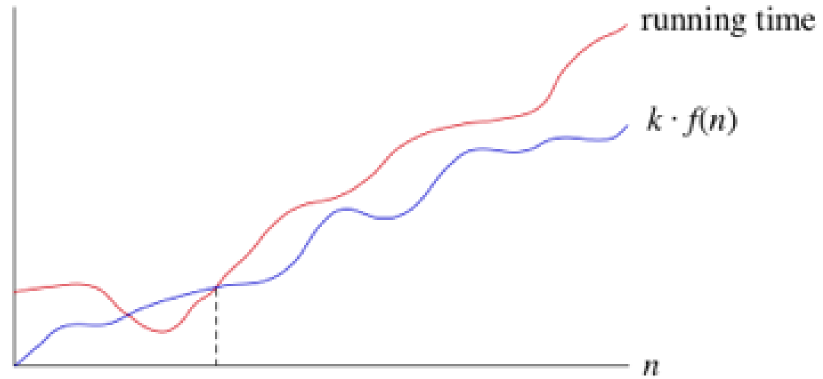
- when calculating big Omega notation, we cannot drop less significant terms or constants.
- We need to find a valid lower bound for the function that takes into account all terms and constants.
- This may require more detailed analysis of the function, including a careful examination of its behavior for small input sizes.

Big-θ (Big-Theta) notation

When we say that a particular running time is $\Theta(n)$, we're saying that once n gets large enough, the running time is at least $k_1 \cdot n$ and at most $k_2 \cdot n$ for some constants k_1 and k_2. Here's how to think of $\Theta(n)$:







- When we use big-Θ notation, we're saying that we have an asymptotically tight bound on the running time.
- "Asymptotically" because it matters for only large values of n.
- "Tight bound" because we've nailed the running time to within a constant factor above and below.

Amortized Analysis

- Designing good algorithms often involves the use of data structures. When we use data structures, we typically execute operations in a sequence rather than individually.
- An operation can only experience its worst-case cost on rare occasions.
  - Simply adding up the worst-case cost of individual operations may grossly over-estimate the actual runtime

Amortized Analysis

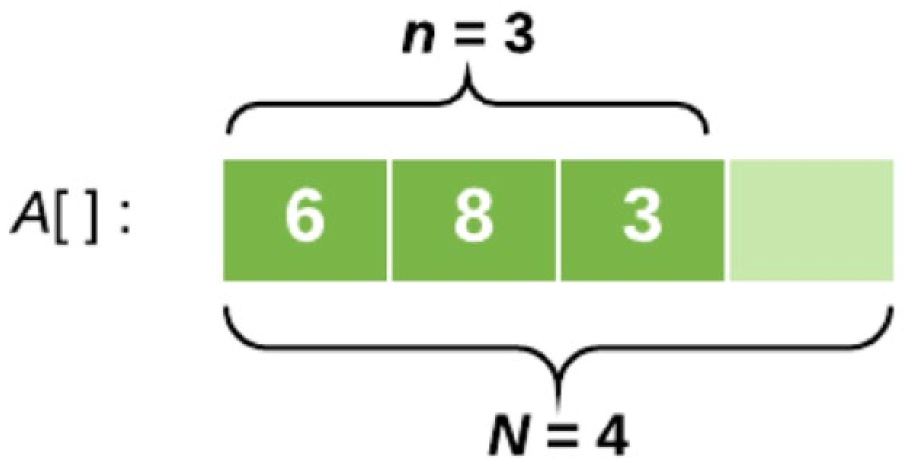
- Designing good algorithms often involves the use of data structures. When we use data structures, we typically execute operations in a sequence rather than individually.
- An operation can only experience its worst-case cost on rare occasions.
  - Simply adding up the worst-case cost of individual operations may grossly over-estimate the actual runtime

➢ Amortization analysis attempts to solve this problem by averaging out the cost of more “expensive” operations across an entire sequence.

➢ More formally, amortized analysis finds the average cost of each operation in a sequence, when the sequence experiences its worst-case.

➢ This way, we can estimate a more precise bound for the worst-case cost of our sequence.

Amortized Analysis



To add an element *x* to the end of our dynamic array, we call an operation called *append*(*x*).

append(*x*) has two steps:

1. Assign *x* to the next available position in the array
2. Check if the array is now full. If it is, we assign some more space by calling the function *copy()*

***copy()* works by copying over each element from our previous array into a new array with twice the length.**

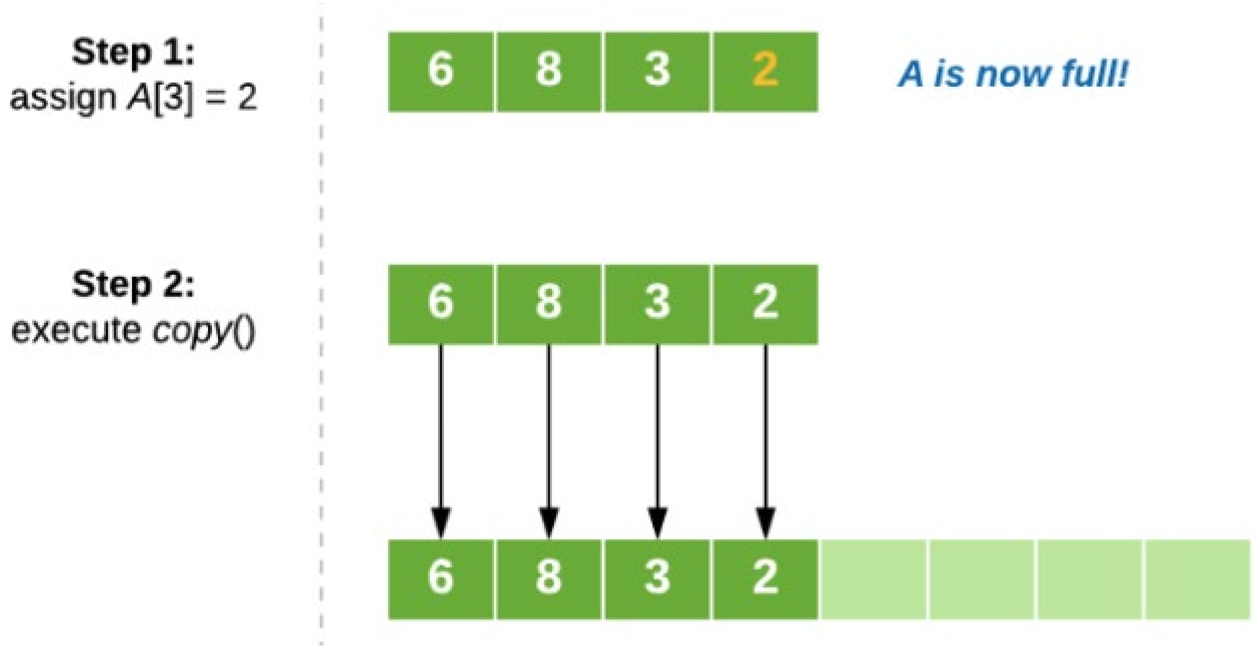
Amortized Analysis

To add an element *x* to the end of our dynamic array, we call an operation called *append*(*x*).

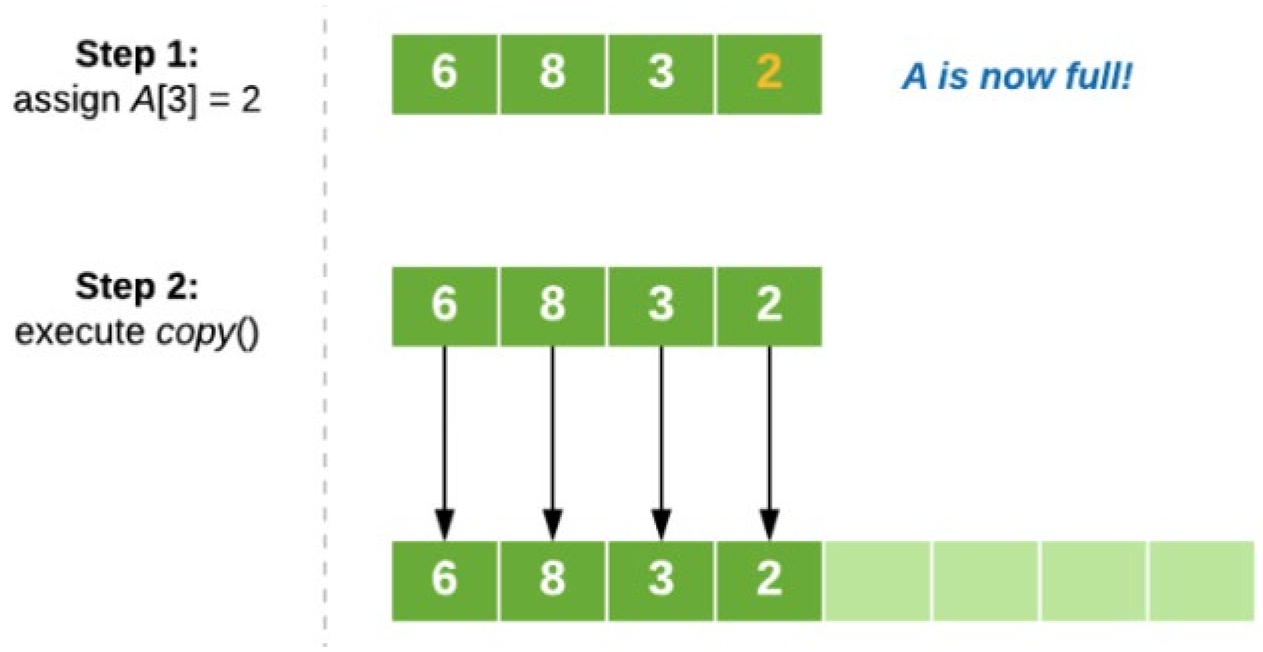
append(*x*) has two steps:

1. Assign *x* to the next available position in the array
2. Check if the array is now full. If it is, we assign some more space by calling the function *copy()*

***copy()* works by copying over each element from our previous array into a new array with twice the length.**



Amortized Analysis



- Worst-case cost of a series of append operations:
    - we could simply sum up the individual worst-case cost of append, **O(n^2),** over the number of operations **n**.
- This will give us a bound of O(n^2), which suggests that our performance will scale quite poorly.

- This estimation is overly pessimistic.

Amortized Analysis: 4. The Aggregate Method

For this method, we apply a sum-and-divide approach, where we:

1. Determine the worst-case cost of our entire sequence of operations, *T*(*n*)
2. Divide this cost by the number of operations in the sequence, *n*

In other words:

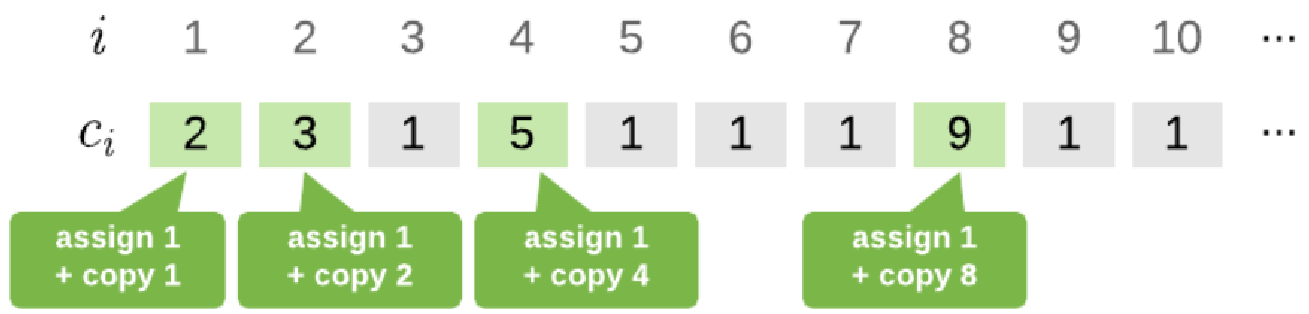
$$Amortized\ operation\ cost = \frac{T(n)}{n}$$

Let's apply this method to our dynamic array.

Amortized Analysis: 4. The Aggregate Method

First, we need to determine the cost of each operation in the sequence. We'll use *C_i* to denote the cost of the *i*th operation and assert that a simple assignment costs 1 unit of time.

We can start off by manually observing C_i for the beginning of our sequence:

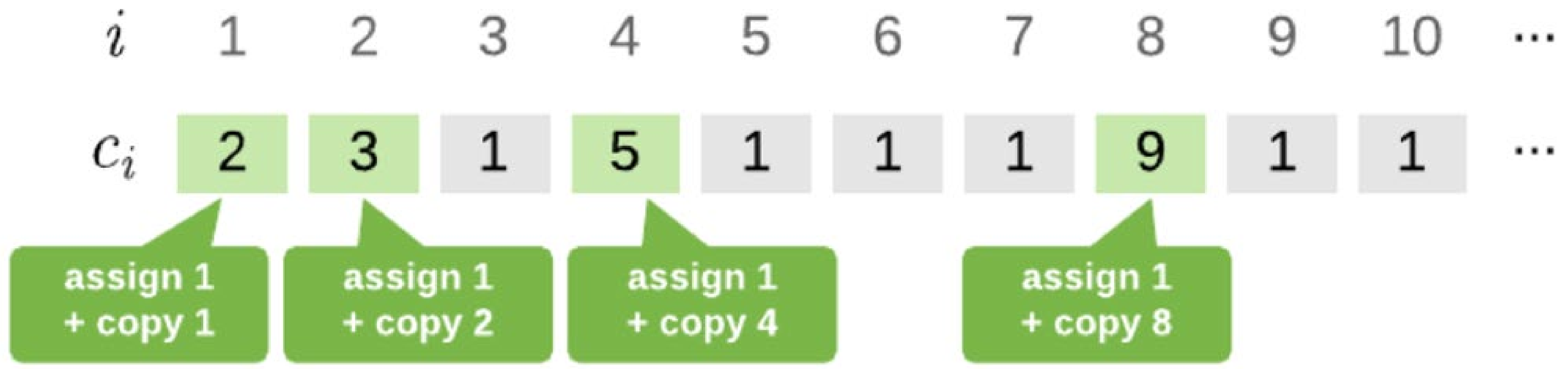


i	1	2	3	4	5	6	7	8	9	10	···
c_i	2	3	1	5	1	1	1	9	1	1	···

A pretty clear pattern is starting to emerge here. ***copy()* is called whenever *i* is a factor of two, that is, when *i* = 1, 2, 4, 8, 16, ...** These operations will cost *i* units of time for *copy*, plus 1 unit of time for our simple assignment.

At every other position in the sequence, *append* will cost 1 unit of time.

Amortized Analysis: 4. The Aggregate Method



$$C_i = \begin{cases} i+1, & \text{when } i \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

Now, we can simplify our last two summations to equal *n*, since together they will occur exactly *n* times.

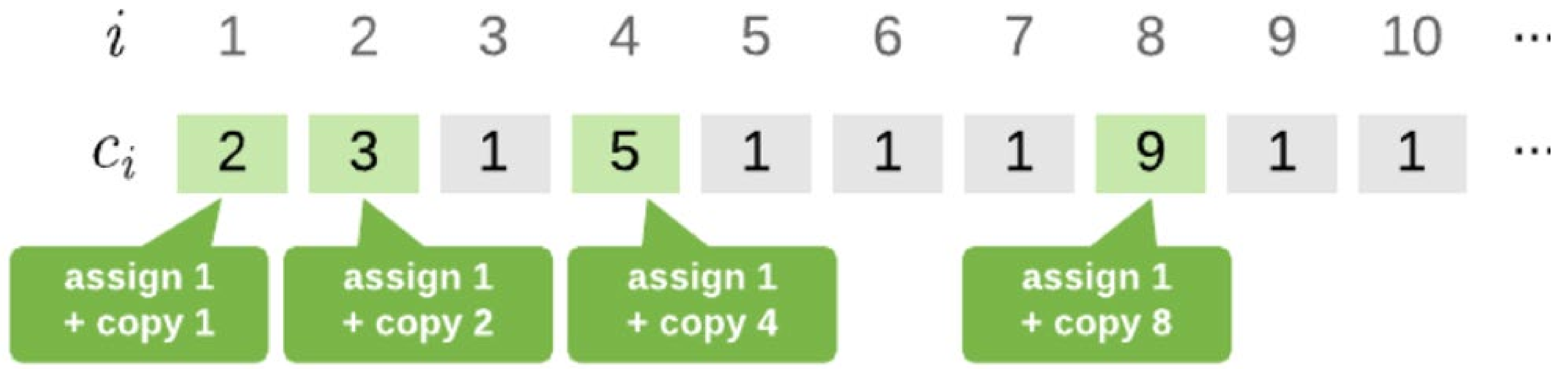
$$T(n) = \sum_{i=1}^{n} C_i$$

$$= \sum_{\substack{i \text{ is a} \\ \text{power of 2}}} (i+1) + \sum_{\substack{i \text{ is not a} \\ \text{power of 2}}} 1$$

$$= \sum_{\substack{i \text{ is a} \\ \text{power of 2}}} i + \sum_{\substack{i \text{ is a} \\ \text{power of 2}}} 1 + \sum_{\substack{i \text{ is not a} \\ \text{power of 2}}} 1$$

$$T(n) = \sum_{\substack{i \text{ is a} \\ \text{power of 2}}} i + n$$

Amortized Analysis: 4. The Aggregate Method



i	1	2	3	4	5	6	7	8	9	10	⋯
c_i	2	3	1	5	1	1	1	9	1	1	⋯

$$C_i = \begin{cases} i+1, & \text{when } i \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{\substack{i \text{ is a} \\ \text{power of 2}}} i + n$$

S = a(1 - r^n) / (1 - r)

S = 1(1 - 2^(log2n+1)) / (1 - 2)
= 1(1 - 2n) / (-1)
= 2n - 1

$$\begin{aligned} T(n) &= \sum_{j=0}^{log_2 n} 2^j + n \\ &\leq 2n + n \\ &= 3n \end{aligned}$$

$$Amortized\ operation\ cost \leq \frac{3n}{n} = 3 = O(1)$$

Amortized Analysis: 4. The Aggregate Method

- this method is very straight forward. However, it only works when **we have a concrete definition** of how much our sequence will cost.

- Also, if our sequence contains **multiple types of operations**, then they will all be treated the same. We can’t obtain different amortized costs for different types of operations in the sequence. This might not be ideal!

- Alternative approaches: **The accounting method and the potential method**